Revision: 1.8
2014-06-25

CCpilot XM2 and CrossCore XM2

Programmer's Guide

crosscontrol

www.crosscontrol.com

Contents

Revision history

Rev	Date	Author	Comments
P1.0	2010-07-06		First draft
P1.1	2012-03-26		Minor updates
P1.2	2012-03-27		Corrections
P1.3	2012-05-11		Corrections
P1.4	2012-11-27		Update for CrossCore XM and All-Integrated functionallity.
P1.5	2012-12-03		Updated after review
P1.6	2013-04-09		Updates: Smart and PowerMgr API.
P1.7	2013-05-30		Updates: API v1 -> APIv2 migration information
P1.8	2014-06-24		CrossControl to CrossControl name change and document template update

1. Introduction

1.1. Purpose

Developing applications for CCpilot XM2 and CrossCore XM2 is basically the same as developing any application under Windows or Linux.

This document contains device specific reference information describing application development and APIs used when developing applications for the XM2 device hardware.

These devices are available with both Windows and Linux operating system and this guide is applicable for both operating systems. Operating system specific information is pointed out in the text or addressed in the respective operating system sections.

Several functionalities are available using operating system or de-facto standard APIs. These may be briefly mentioned but not further described within this documentation.

A good prior understanding of Windows and/or Linux programming is needed to fully benefit from this documentation.

1.2. Conventions and defines

CCpilot XM2 and CrossCore XM2 are in most cases identical in functionality and usage. The following definition is used to separate unit specific details. The observe symbol is also used to highlight such difference.

Defines	Use
CCpilot XM	Information that is specific for CCpilot XM
CrossCore XM	Information that is specific for CrossCore XM
XM device	Information that applies to both CCpilot XM and CrossCore XM

The observe symbol is used to highligt information in this document, such as differences between product CCpilot and CrossCore product models.

The A symbol is used to highlight information specific for CCpilot XM All-Integrated and CrossCore XM All-Integrated.

The exclamation symbol is used to highlight important information.

Text formats used in this document.

Format	Use
Italics	Paths, filenames, definitions.
Bold	Command names and important information

1.3. Identification

On the side of the XM device there is a label with version and serial numbers which identify your unique computer. Take note of them. During service and other contact with the supplier it is important to be able to provide these numbers.

1.4. References

For further information on XM device software and the available APIs see the following references.

[1] CCpilot XM and CrossCore XM – Software User Guide

[2] SocketCAN from the Linux kernel,
http://lxr.linux.no/linux+v2.6.31.14/Documentation/networking/can.txt

[3] CCAux API documentation (CC AUX x.x.x.x, available in the CC AUX SDK)

[4] CAN Interface Description (available within the Windows SDK)

1.5. Include files and libraries

The programmers guide may contain references to include files needed when programming the XM device. Note that these files can be downloaded via the CrossControl support web site as development packages. Those packages may also include libraries to use for application development, or in other ways reference to library functions.

2. Interface overview

This section covers basic information on how to access the XM device hardware. Most of the hardware is accessed using the default Windows or Linux interfaces but some XM device specific interfaces, such as CAN and Digital In, require additional interfaces to be accessed.

The table below lists the API used to access each interface. These interfaces can be grouped into four categories. Standard libraries (Standard API), CCAux library (CCAux API), JIDA 32 Library (JIDA 32 API) and Other Libraries (Other API).

Depending on product model, all interfaces may not be present on your specific model. See also the operating system specific sections and additional documentation describing the software API.

Functionality	Standard API	CCAux API	JIDA 32 API	Other API	Comment
CAN	√			√	Depending on operating system.
Ethernet	√				
USB	√				
RS232	√				
Video In		√			
Audio In / Out	√				
Digital In		√			
Status LED		√			
Backlight		√			
Ambient Light sensor		√			
Buzzer		√			
Watchdog			√		
Power management		√			
S.M.A.R.T		√			
User EEPROM			√		
WLAN	√	√			Power management access through CC AUX API required
GPRS/GSM	√	√			Power management access through CC AUX API required
GPS	√	√			Power management access through CC AUX API required
Bluetooth	√	√			Power management access through CC AUX API required

2.1. Standard Libraries

Most interfaces are accessed using standard libraries and access methods. Different access methods can be possible depending on development environment and additional installed frameworks, such as .Net or Qt.

The standard libraries used for Windows and Linux are described in their respective documentation sources, such as MAN pages or MSDN.

2.2. CCAux library

The CCAux API gives access to several hardware specific interfaces. The API is the same for both Windows and Linux. The API functions of this library are documented in the CCAux API reference documentation, called CCAux, listed as reference [3]. Below is a brief introduction on the API's found therein and their function.

All API functions can be used from the CCsettings program as well.

Not all API functions are available in all product instances, and will in those cases return a defined error code.

2.2.1. About

Contains an API for reading hardware configuration, unit data etc.

2.2.2. Adc

Contains an API for reading built in ADC voltage information.

2.2.3. AuxVersion

Contains an API for reading firmware version information.

2.2.4. Backlight

Contains an API for controlling backlight settings as well as configuring automatic backlight functionality on CCpilot XM.

2.2.5. Buzzer

Contains an API for controlling the built-in buzzer.

2.2.6. CanSetting

Contains an API for controlling CAN settings. Note that other or similar CAN-related settings are available through other API's and in the Windows registry.

2.2.7. Config

Contains an API for controlling internal and external power up and power down settings and time configurations, including power button and on/off signal configurations.

2.2.8. Diagnostic

Contains API's for getting run time information about the XM device.

2.3. CCAux library

The CCAux API gives access to several hardware specific interfaces. The API is the same for both Windows and Linux. The API functions of this library are documented in the CCAux API reference documentation, called CCAux, listed as reference [3]. Below is a brief introduction on the API's found therein and their function.

All API functions can be used from the CCsettings program as well.

Not all API functions are available in all product instances, and will in those cases return a defined error code.

2.3.1. About

Contains an API for reading hardware configuration, unit data etc.

2.3.2. Adc

Contains an API for reading built in ADC voltage information.

2.3.3. AuxVersion

Contains an API for reading firmware version information.

2.3.4. Backlight

Contains an API for controlling backlight settings as well as configuring automatic backlight functionality on CCpilot XM.

2.3.5. Buzzer

Contains an API for controlling the built-in buzzer.

2.3.6. CanSetting

Contains an API for controlling CAN settings. Note that other or similar CAN-related settings are available through other API's and in the Windows registry.

2.3.7. Config

Contains an API for controlling internal and external power up and power down settings and time configurations, including power button and on/off signal configurations.

2.3.8. Diagnostic

Contains API's for getting run time information about the XM device.

2.4. Migrating from CCAux 1.x to 2.x libraries

For older CrossControl products, there's a version of the CCAux library that was written a bit differently. For the XM devices, the 1.x interfaces are still available, but there's a new generation of the API also installed, which has some specific differences that one needs to consider, but also some benefits. For instance, the API backwards compatibility in the 2.x series will be much better compared to the 1.x series, and applications shouldn't need to be recompiled when new versions are made available, unless specific usage of such functions is required.

The main difference can be seen in this example, which may apply to previously written code on other CrossControl products:

```
/* Usage in CCaux API 1.x */
#include "Module.h"
Module* pModule = crosscontrol::GetModule();
eErr err = pModule->function_1(arg, …);
pModule->Release();

/* New usage in CCaux API 2.x */
#include "Module.h"
MODULEHANDLE pModule = crosscontrol::GetModule();
eErr err = Module_function_1(pModule, arg, …);
Module_release(pModule);
```

When porting your application to CC AUX API v 2.X, the above differences is what you will need to address for all instances calling the API functions in your code.

2.5. JIDA 32 Library

The JIDA 32 API is a third party library that gives access to CPU board specific functionality such as the watchdog, temperature and user EEPROM access.

2.6. Other Libraries

Custom or third party API may be required to access different interfaces. See the library overview table for information on the specific library.

3. Guidelines for Compact Flash usage

A compact flash memory is used for persistent storage in the XM device. Compact flash memories provide a small sized storage that is nearly insensitive for shocks and vibrations. But flash memories also have a limited number of write cycles that must be considered during application design. As with any file system improper shut down of the device may also lead to incomplete file operations and corrupt flash.

Here are some guidelines that can be used to better assure that the file systems are being kept consistent, and to prevent pre-mature aging of the CF card.

3.1. Prevention for device shut down

- To prevent data loss due to sudden power disruption, large files shouldn't be written if there is a risk for abruption. Keep critical windows short, i.e. compress the files before writing to storage media if they are retained in RAM memory prior to the write, or divide the files into smaller pieces.
- The XM device doesn't deviate from the standard OS model in case when using the ON/OFF signaling from hardware. Hence, an application that is able to terminate properly using a normal operating system should be able to use the same solution on the XM device.
- In case of additional prevention, the applications can and should listen to operating system power management signals and/or power status signals via the CCAux API. The shutdown process is a quick process and when shutdown signals occur the application shall terminate

quickly, i.e. be able to quickly abrupt a file write in progress and do not write large files such as log files upon system shut down. A general design guide would be that an application shouldn't need more than a few hundred ms to make itself ready for shutdown

- The CCAux PowerMgr API may be used by an application to delay OS shutdown and OS suspend operations in some use-case scenarios. The application can delay shutdown until it is ready with its operations.

3.2. Extend the compact flash lifetime

- Application should not excessively write to the file system. Better approach is to use RAM-based log files and on regular intervals write files to permanent file system. Although use caution, RAM-files can be lost on sudden power off, so for mission critical data, another approach can be considered.
- Each write to permanent storage should be forced for synchronization, like (Linux perspective):

```
Command/script style: # sync
Programming style 1: res = fsycnc(fd);
Programming style 2: res = system("sync");
```

- It may be possible to add file system locks during startup as a startup script itself, to prevent unnecessary stress on the flash. This approach needs proper usage on several levels, making sure writes can be performed when they are supposed to be. File system locks can also be added as an extra security measure, possibly in combination with file system checks. But this may lead to longer startup times.

- For Linux based system the application should follow startup scripts guidelines to make sure that operating system signals are correctly passed to application, as found in the Software Guide document.
- The CCAux Smart API can be used by applications to monitor the expected lifetime of the compact flash. It can be used to warn the user that a replacement is needed before the compact flash becomes corrupt.

4. Windows specifics programming guide

4.1. Set up the development environment.

Many development environments are available for Windows application development. This section describes setting up the environment for Microsoft Visual Studio, it is recommended to use version 2008 or higher. See the Microsoft documentation for Visual Studio installation procedure information.

4.1.1. SDK needed to start developing (on development computer)

The XM device specific SDK for development is divided into two parts in Windows, the *CC AUX API SDK* for hardware access functions and the *CC Win32 CAN API* for CAN functions. Both should be installed on the development computer for full access to all functions.

4.1.2. Drivers or packages needed to deploy on the XM device

No additional software needs to be installed on the XM device.

4.1.3. Setting up remote debugging

To enable efficient debugging, remote debugging can be set up where the application is executed on the XM device and debugged via the development computer. See the Microsoft Visual studio documentation for set-up information.

4.2. Windows Interfaces specifics

This section covers windows specific details for programming with an XM device.

4.2.1. CAN

In Windows the CAN interfaces is accessed using the proprietary CAN API, see the CAN interface description documentation within the Win32CAN SDK, reference [4]. The CAN drivers are installed per default on the XM device but the SDK is needed in the development environment. Note that settings for CAN are stored in the Windows registry. Some, but not all of these are also available as settings in the CCAux API.

5. Linux specific programming guide

5.1. Development environment setup

For developing software for an XM device, a development PC with the following setup is recommended.

Software	Version	Description
Ubuntu	10.4	Operating system (binary compatible system is OK)
gcc	4.4.3	C – compiler
g++	4.4.3	C++ - compiler
gdb	7.1	Debugger
Eclipse	3.5.2	As example. Other IDE or text editors can also be used.

5.1.1. Using correct development headers

For the customer application to utilize the services on the XM device the compiler has to find the correct development headers and libraries. Paths of those files must be added to the search path in project settings or the *Makefile*.

5.1.2. SDK needed to start developing (on development computer)

If the CC Aux API is needed, the following packages should be installed on the target XM device:

```
# dpkg –i libccaux-dev_x.x.x.x_i386.deb libccaux_x.x.x.x_i386.deb
```

This can be considered an SDK for the CC AUX API in Linux, and it installs the header files to the system include file directory, i.e. under /usr/include.

5.1.3. Drivers or packages needed to deploy on the XM device

A set of standard Ubuntu libraries installed, so no additional installation is needed by default. If additional packages are needed, see the CCpilot XM and CrossCore XM – Software User Guide for more details on how to perform installation.

For XM device installed packages and libraries, use this command to list all installed packages:

```
# dpkg -l
```

Development versions of the appropriate libraries can easily be installed on the development computer. See the standard Ubuntu documentation for more details.

5.1.4. Setting up remote debugging

To use gdb to debug an application running on the XM device, the application must have been compiled with the -g flag. Start **gdbserver** on the XM device:

```
~# gdbserver :10000 testApplication
```

Then start the host gdb and connect to the server:

```
# gdb testApplication
# (gdb) target remote Y.Y.Y.Y:10000
```

Above Y.Y.Y.Y is the IP address for the XM device. You can now debug the application normally, except that rather than to issue the run command one should use continue since the application is already running on the remote side.

Note that it is possible to fully debug the application but not the system calls made by the application. Such system calls include calls to the soft float library, like divide, add or multiply on floating point variables. It is therefore recommended to use next rather than step when such system calls are being made.

5.2. Linux Interfaces specifics

5.2.1. CAN

In Linux CAN is interfaced using SocketCAN. SocketCAN is a widely used CAN communication interface for Linux environments, and is a standard used in the Linux kernel.

Usage of SocketCAN requires knowledge of some system specific settings and details described herein, for additional SocketCAN information see the official SocketCAN documentation.

5.2.1.1. Configuration of the XM device interface

The XM device node files for the four CAN interfaces are *can0* to *can3*, which should be shown when listing all network interfaces with the **ifconfig** command. The XM device driver is implemented as loadable kernel modules, *can_dev.ko, xilinx.ko* and *xilinx_platform_pci.ko*. In addition, there are at least two CAN protocol modules providing access to the CAN protocol interface. A script handles the loading of the kernel modules upon start-up.

When XM device has finished its start-up, the CAN driver modules are loaded as a part of the kernel. This can be checked via terminal access using **lsmod** command:

```
# lsmod | egrep "can|xilinx"
can_raw         7552  0
can            23656  1 can_raw
xilinx_platform   2848  0
xilinx          6080  1 xilinx_platform
can_dev        15616  1 xilinx
```

Since the driver is compiled as modules, unnecessary protocols may be removed or new modules inserted according to user needs.

The CAN bus itself is not initialized during start-up, it only loads the drivers. Before any communications can be executed, user must set correct bus speed (as an example 250kbit/s) by first writing value into bitrate parameter:

```
# echo 250000 > /sys/class/net/can0/can_bittiming/bitrate
```

and then setting interface up with **ifconfig**:

```
# sudo ifconfig can0 up
```

After this, **ifconfig** should show *can0* as a network interface:

```
# ifconfig
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00
          UP RUNNING NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:49
```

Same applies to the other CAN interfaces by changing *can0* to *can1, can2* or *can3*.

5.2.1.2. Bus recovery options

There are two options for implementing bus recovery after bus off has occurred: **manual** and **automatic**.

Manual recovery is initiated by writing a non-zero value to *can_restart* variable under *sysfs*:

```
sudo sh –c "echo 1 > /sys/class/net/can0/can_restart"
```

Bus restart is then scheduled through kernel and implemented through can-core.

In automatic bus recovery, can-core detects state changes and re-initializes controller after the specified time period.

Automatic bus recovery from bus off state is by default turned off. It can be turned on via *sysfs* setting, where the wanted restart period in milliseconds is set into using the *can_restart_ms* variable. For example, a 100ms restart period for *can0* is set from command line like this:

```
sudo ifconfig can0 down
sudo sh –c "echo 100 > /sys/class/net/can0/can_restart_ms"
sudo ifconfig can0 up
```

Same commands apply for *can1* by replacing *can0* appropriately. Period is possible to set as needed from application perspective. Value zero turns automatic bus recovery off.

Warning: Enabling automatic bus recovery may disturb other nodes on bus, if CAN interface is incorrectly initialized.

5.2.1.3. Error interrupt options

Error interrupts are disabled by default. By enabling error interrupts user can receive error frames. Bus off errors will come through even if the error interrupts are not enabled.

Enable them by giving module parameter **errorirq=1** during module loading

```
# sudo modprobe xilinx errorirq=1
```

Or, by editing xilinx.conf under /ro/etc/modprobe.d/ directory.

```
#xilinx.conf: Add new options to end of next line
options xilinx errorirq=1
```

Warning: Enabling error interrupts and sending frames when module is not connected to active bus may cause CAN acknowledge errors to overload CPU. User caution required. It is recommended to avoid sending until one frame is received.

5.2.1.4. SocketCAN Example

Below is an example of a SocketCAN application code. This example is not a complete and may or may not compile as an application but it shows the nature of SocketCAN programming and the usage of standard socket programming.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>

#include <linux/can.h>
#include <linux/can/raw.h>
#include <string.h>

/* Define constants, if not defined in the headers */
#ifndef PF_CAN
#define PF_CAN 29
#endif

#ifndef AF_CAN
#define AF_CAN PF_CAN
#endif

/* ... */

/* Somewhere in your app */
```

```
   /* Create the socket */
   int skt = socket( PF_CAN, SOCK_RAW, CAN_RAW );
   const int loopback = 0;

   /* Locate the interface you wish to use */
   struct ifreq ifr;
   strcpy(ifr.ifr_name, "can0");
   ioctl(skt, SIOCGIFINDEX, &ifr); /* ifr.ifr_ifindex gets filled
                                    * with that XM device's index */

/* Select that CAN interface, and bind the socket to it. */
   struct sockaddr_can addr;
   addr.can_family = AF_CAN;
   addr.can_ifindex = ifr.ifr_ifindex;
   bind( skt, (struct sockaddr*)&addr, sizeof(addr) );

/* Disable filters and loopback feature. */
   setsockopt(skt, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);
   setsockopt(skt, SOL_CAN_RAW, CAN_RAW_LOOPBACK,
              &loopback, sizeof(loopback));

/* Send a message to the CAN bus */
   struct can_frame frame;
   frame.can_id = 0x123;
   strcpy( &frame.data, "foo" );
   frame.can_dlc = strlen( &frame.data );
   int bytes_sent = write( skt, &frame, sizeof(frame) );

   /* Read a message back from the CAN bus */
   int bytes_read = read( skt, &frame, sizeof(frame) );
```

5.2.2. Serial Number Broadcast interface

The XM device has Serial Number Broadcast service. *SNB* does not have programming interface at the XM device end, but the broadcasted data output can be handled elsewhere, even in another XM device if required.

The message sent is a multicast UDP datagram to address 224.0.0.27. The message contains a char array with three values separated by tabs; Serial number, Firmware version and XM Device type. The sender's IP address is available in datagram headers.

Example data contents (without quotes):

```
"PR01<tab>0.3.0<tab>0"
```

An example implementation of the data listener is available in development package in *example_src/snb/snb_reader.c*

Technical support

Contact your reseller or supplier for help with possible problems with your XM device. In order to get the best help, you should have access to your XM device and be prepared with the following information before you contact support.

- The part number and serial number of the XM device, which you find on the brand label
- Date of purchase, which is found on the invoice
- The conditions and circumstances under which the problem arises
- LED indicator flash patterns.
- The XM Device log files (if possible)
- Prepare a system report on the XM device, from within *CCsettings* (if possible).
- Description of external equipment which is connected to the XM device.

Trademark, etc.