Product revision: 1.0
2018-03-19

CCpilot VC
Programmer's guide

crosscontrol
www.crosscontrol.com

Contents

1. Introduction

1.1. Purpose

This document contains device specific reference information describing application development and APIs used when developing applications for the *CCpilot VC* products. These devices are available with the Linux operating system.

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 Linux programming is needed to fully benefit from this documentation.

1.2. Conventions and defines

All CCpilot VC devices are in most cases identical in functionality and usage.

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

The exclamation symbol is used to highlight important information.

Text formats used in this document.

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

1.3. Identification

On the side of the 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 the device software and the available APIs see the following references.

[1] CCpilot VC – Software User Guide

[2] SocketCAN in Linux kernel,
http://lxr.linux.no/linux+v2.6.35.14/Documentation/networking/can.txt

[3] CC AUX API documentation

1.5. Include files and libraries

The programmers guide may contain references to include files needed when programming the 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.

Under chapter 4.4 Using correct development headers further attention is given to the include files and libraries.

1.6. Floating point support

The device hardware has an ARM Cortex-A8 processor with a NEON floating point accelerator, which makes it possible to use floating point arithmetic in your applications which is also hardware accelerated for performance. This will require specific attention when building your application and it's further covered under chapter 4.3.2 Floating point usage.

2. Interface overview

This section covers basic information on how to access the device hardware. Most of the hardware is accessed using the default Linux interfaces but some specific interfaces may require additional functions to be accessed.

The table below lists the API used to access each interface. These interfaces can be grouped into two categories; Standard operating system libraries (Standard API) and CC AUX library (CC AUX 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	CC AUX API	Comment
CAN	√		
Ethernet	√		
USB	√		
RS232	√		
Video In	√	√	It's possible to use video4Linux API's for experienced users directly.
PWM Out		√	
Configurable Input		√	
Status indicator		√	
Backlight		√	
Ambient Light sensor		√	
Buzzer		√	
Watchdog	√		
Power management		√	

2.1. Standard Libraries

Most interfaces are accessed using standard libraries and access methods. Various access methods are possible to be used depending on the development environment and additional installed frameworks, such as Qt.

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

2.2. CC AUX library

The CC AUX API gives access to several hardware specific interfaces. The API functions of this library are documented in the CC AUX API reference documentation, called CC AUX, listed as reference [3]. This API is almost equal compared to the CC AUX API in the CCpilot XA devices; it has been designed with portability in mind.

Below is a brief introduction on the API's found therein and their functions. Most API functions can be used from the CCsettings program as well.

2.2.1. About

Contains a hardware information API related to the hardware configurations.

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 versions with display.

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 most settings for CAN usage is available over the SocketCAN interface instead.

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 an API for getting run time information about the device.

2.2.9. CfgIn

Contains an API to get and set the current status of the configurable input signals.

2.2.10. PWMOut

Contains an API to get and set the current status of the PWM Output signals.

2.2.11. FirmwareUpgrade

Contains an API that allows for updating of the System Supervisor (SS) firmware of the device.

! Consider careful usage for these functions, erroneous usage can result in a non-functional device.

2.2.12. FrontLED

Contains an API for overriding the default LED behaviour. In CCpilot VC, which does not have a front LED, this API controls the button backlight LEDs instead. The button backlight can be controlled in the same way as the front LED, except for colour.

2.2.13. LightSensor

Contains an API for reading the light sensor values and getting raw and/or calculated values on CCpilot versions.

2.2.14. Power

Contains an API for reading power status and control functions for advanced shut down behaviour. Unsupported for the VC devices.

2.2.15. TouchScreen

Contains an API for changing the touch screen profile between mouse or touch profile as well as other touch screen related settings on CCpilot versions. Unsupported for the VC devices.

2.2.16. Video

Contains an API for controlling the analogue video streams in terms of location, size, scaling etc.

2.2.17. Telematics

Contains an API for controlling power to the telematics board, as well as some basic initialization functions. Unsupported in the VC devices.

2.2.18. Battery

Contains an API for controlling battery related settings, if a battery is connected.
Unsupported for the VC devices.

2.3. API calling convention

The standard way to call the API functions is shown below. Please adhere to this calling convention. Example code for each function is available in the API documentation.

```
/* Usage in CCaux API 2.x */
#include "Module.h"

MODULEHANDLE pModule = CrossControl::GetModule();

eErr err = Module_function_1(pModule, arg, …);

Module_release(pModule);
```

2.4. Other Libraries

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

3. Guidelines for NAND flash usage

A NAND flash memory is used for persistent storage in the device. NAND 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 premature aging of the NAND flash.

3.1. Prevention for data loss upon sudden 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. Flush file buffers regularly.
- The device doesn't deviate from the standard OS model in case when using the ON/OFF signaling from hardware. Hence, an application should take care of shut down signals appropriately, and should be started so that the shutdown signals can propagate properly to the application. This is normally done so that the applications can and should listen to operating system power management signals (SIGTERM etc) and/or power status signals via the CC AUX 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 milliseconds to make itself ready for shutdown, including termination of file write operations to storage media. The operating system has an indication method for applications to subscribe to, and the most important task when an event of this kund happens should be to stop writing to the NAND flash (i.e. file system), and mount the file system read-only. The default system will actually do that for you, if your application doesn't write to the disk extensively during a sudden power loss.

3.2. Extend the NAND flash lifetime

- An 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. Note that 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 (i.e. flushing the file buffers, like:

```
Command/script style: # sync
Programming style 1: res = fsync(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.

4. Development Hosts

This section is dedicated to useful tips and hints about Linux development hosts for application development and debugging purposes.

4.1. Retrieval of tool chain

The recommended tool chain is the ARM/Linux cross compiler from Mentor Graphics called *Sourcery CodeBench,* which is a version of the standard gcc compiler. To get the *Sourcery CodeBench* cross compiler tool chain, visit the *Mentor Graphics Sourcery CodeBench* web site http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview. There are several alternatives for the cross-compiler, some with a price penalty but with better support, and one free version, called Lite. Register an account, and download a suitable alternative for you. The recommended version at the time of writing this document is 2011.09-70 (gcc 4.6.1) for **ARM GNU/Linux**.

For support on tool chain issues, please refer to *Mentor Graphics CodeBench* support channels.

4.2. Installing tool chain

The installer binary requires the bash shell to be the default shell, and the downloadable file should have execution rights before the installation begins. If not enabled as default, make sure that */bin/sh* points to bash at least, or perform any other means of getting bash as the default shell. Make sure the installer binary has execution permission, and invoke it with root privileges:

```
# chmod +x arm-2011.09-70-arm-none-linux-gnueabi.bin
# sudo arm-2011.09-70-arm-none-linux-gnueabi.bin
```

The installer will start a Java-based installation GUI. The default settings should be applicable, but it is strongly recommended to install the cross compiler to */opt/codesourcery/*, since it will integrate better with the target development files then.

4.3. Using tool chain

Once installed, the tool chain binaries should be found under */opt/codesourcery/bin/*, all with the **arm-none-linux-gnueabi-** prefix. The tool chain should be added to the path, i.e. add the directory mentioned to the *$PATH* environment variable.

```
user@host:~/$ arm-none-linux-gnueabi-gcc --version
arm-none-linux-gnueabi-gcc (Sourcery G++ 2011.09-70) 4.6.1
Copyright (C) 2011 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

Now the cross compiler should be ready to use.

4.3.1. ARM Cortex-A8 instruction set optimization

The cross-compiler has a specific flag that can be used when building applications. The compiler will then optimize the code with instruction set generation for the specific ARM Cortex-A8 processor over the generic instruction set generation. This is quite simple; just add a flag to the build command line like:

```
user@host:~/$ arm-none-linux-gnueabi-gcc –mcpu=cortex-a8 file.c
```

4.3.2. Floating point usage

The device has floating point optimized libraries installed, so normally floating point arithmetic is partly accelerated already, compared to using standard floating point software libraries. But, it's also possible to optimize the floating point operations a bit further, by adding a couple of additional build flags to the build command. This is done like:

```
user@host:~/$ arm-none-linux-gnueabi-gcc –mcpu=cortex-a8     -mfpu=neon
-funsafe-math-optimizations floatfile.c
```

More information about optimizing flags for the compiler can be found in the compiler documentation that comes with the compiler.

4.4. Using correct development headers

The development package is released together with binary images. This package contains libraries available at the unit and all header files for utilizing them. It is however recommended to use the XA platform development files, but the contents of this section may still apply to those who are interested.

The development package is not installed to the device, it should be installed to the development host used for compiling software for the device. The development package can be extracted to almost any location at the development host, but it's preferred that it's extracted to */opt/VC*, since there are default dependencies for Qt development binaries for that directory as the default directory.

When contents of this package change (not necessary at every release), the new version completely replaces the old release. Therefore, it is recommended that the package is extracted to a dedicated directory, which can be erased and recreated without losing anything else.

E.g.: extracting to */opt/VC/*

```
# mkdir /opt/VC
# cd /opt/VC
# tar xzf ccpilot-vc-devel-n.n.n.tgz
```

For the customer application to utilize the services on the device the cross-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*.

E.g.: if development package is extracted to */opt/VC/*

```
# arm-none-linux-gnueabi-gcc [other_commands]
-I/opt/VC/usr/include -L/opt/VC/lib -L/opt/VC/usr/lib
```

Note that in most cases, the development files for the CCpilot XA or CCpilot XS will suffice when building applications for the CCpilot VC as well. It's recommended that the latest version of XA platform development files is used (version 1.3.0.0 or above).

4.5. Debugging remotely

4.5.1. gdbserver

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

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

Then start the host GDB and connect to the server:

```
# arm-none-linux-gnueabi-gdb testApplication
# (gdb) target remote Y.Y.Y.Y:10000
```

Above Y.Y.Y.Y is the IP address of the 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 to make 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. Device specifics

This section is dedicated to device specifics that require extra attention when programming.

5.1. CAN

In Linux CAN is interfaced using SocketCAN which 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 [2].

5.1.1. Configuration of the device interface

The device node files for the CAN interfaces are *can0* and *can1*. The interfaces should be shown when listing all network interfaces with the **ifconfig** command. The device drivers are implemented as loadable kernel modules. The modules are *can_dev.ko* and *flexcan.ko*. In addition, there are at least two CAN protocol modules providing access to the CAN protocol interface. Startup scripts handle the loading of the kernel modules upon start-up.

When the 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 | grep can
can_raw                 7552       0
can                     23656      1 can_raw
flexcan                 2848       0
can_dev                 15616                1 flexcan
```

Since the drivers are 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
```

5.1.2. Configuring the CAN socket transmission buffer

By default, the CAN driver is configured with a transmission buffer that can hold 10 CAN frames. As each frame is sent on the bus, the buffer is cleared. However, it is possible to write frames to the socket faster than the frames are sent, especially if your messages are low-priority frames on a high-traffic bus. If your application needs to send more than 10 CAN frames in bursts, it might be a good idea to increase the size of the transmission buffer:

```
# ifconfig can0 txqueuelen 100
# ifconfig can1 txqueuelen 100
```

5.1.3. 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*:

```
# 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:

```
# ifconfig can0 down
# echo 100 > /sys/class/net/can0/can_restart_ms
# ifconfig can0 up
```

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

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

5.1.4. SocketCAN Example

Below is an example of a SocketCAN application code. This example is not a complete compliable 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
VC
                               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. Analogue video

There are some design constraints on the usage of analog video and CC AUX API that is important to highlight, to create a better understanding of what can be done and what's necessary to do within the applications. Below is a brief video API description for the developers to consider.

In the Linux system there is a sample video application, *ccvideo* for displaying video channel 1. This application is for test purposes, we recommend implementing the video functionality into your application for correct behavior.

The most important CC AUX Video API functions are as follows:

Initialize (will open file handles, setup basic settings and request frame buffers), select deviceNr=1 for input channel 1:

```
Video_init(VIDEOHANDLE pObj, unsigned char deviceNr)
```

Select the active channel 1-4, corresponds to the physical port number (only channel 1 supported on VC devices):

```
Video_setActiveChannel(VIDEOHANDLE pObj, VideoChannel channel)
```

Set the area where video is shown:

```
Video_setVideoArea(VIDEOHANDLE
pObj, unsigned short topLeftX, unsigned short topLeftY, unsigned short bottomR
ightX, unsigned short bottomRightY)
```

Enable or disable (horizontal) mirroring of the video image:

```
Video_setMirroring(VIDEOHANDLE pObj, CCStatus mode)
```

Show (or hide) video image:

```
Video_showVideo(VIDEOHANDLE pObj, bool show)
```

Further and more detailed API information can be found in [3].

5.3. Graphics

The device uses a frame buffer as the graphics rendering part. It's fast and efficient, and has a well-known and standardized interface for applications. There are a lot of different frame buffer applications that can be ported to the device, but the most prominent that's used is the Qt libraries with the QWS extension. With those, any cross-compiled Qt application that runs on the device can draw it's graphics via the built-in QWS enabled Qt libraries to the frame buffer. It doesn't exclude the usage for direct frame buffer applications.

5.3.1. Qt specifics

The first Qt application that requires the usage of the QWS extension must be started with the –qws flag, i.e:

```
# ./Application -qws
```

That is valid for the first Qt application that starts the graphics on the frame buffer. Any following applications on top of the first one draws to the same screen widget, and will act as a top application with the first one in the background. They do NOT require the –qws flag in that case.

The frame buffer also is depending on the color bit depth. By default, it uses 32 bits for colors. Some Qt applications receive a slight performance penalty due to the large amount of color bits. It's possible to increase the performance then on behalf of a smaller color bit depth. Before starting any frame buffer graphics application, set the frame buffer into 16 bit mode:

```
# fbset –depth 16
```

It's possible to set back to 32 bit mode later with the same method. The bit depth change isn't permanent, when the device is restarted, the color bit depth is reset to default mode again, unless a startup script performs these changes on every startup.

5.4. Serial Number Broadcast interface

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

6. Examples

In the development package, there are a couple of example applications that can be used as templates or starter points for your applications.

Technical support

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

- The part number and serial number of the 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
- Status indicator patterns.
- The VC Device log files (if possible)
- Prepare a system report on the device, from within *CCsettings* (if possible).
- Description of external equipment which is connected to the device.

Trademark, etc.