CrossCore XA

Programmer's guide

crosscontrol

www.crosscontrol.com

Contents

Revision history ... **3**
1. Introduction ... **4**
1.1. Purpose ... 4
1.2. Conventions and defines ... 4
1.3. References ... 4
1.4. Include files and libraries ... 4
1.5. Floating point support ... 4
2. Debug card ... **5**
2.1. J14 - Configuration header ... 5
2.2. J15 – Connection to CrossCore XA ... 5
2.3. J18 – USB Serialport ... 6
2.4. J19 – FPGA ... 6
2.5. J20 – Ethernet JTAG ... 6
2.6. J25 – ARM JTAG ... 6
2.7. J26 – RS232 ... 7
2.8. J28 – I2C ... 7
2.9. J29 – RTC Battery ... 7
2.10. SW1 – Factory Reset ... 7
2.11. SW2 – Bootflash Disable ... 7
3. CAN Communication Interface ... **8**
3.1. Configuration of device interface ... 8
3.2. Summary of data types ... 11
3.3. Interface functions ... 12
4. Digital I/O Device Interface ... **24**
4.1. Configuration of device interface ... 24
4.2. Summary of data types ... 24
4.3. Interface functions ... 24
5. Power Device Interface ... **29**
5.1. Configuration of device interface ... 29
5.2. Summary of data types ... 29
5.3. Interface functions ... 29
5.4. Power signal handling ... 34
6. USB Device Interface ... **35**
6.1. Configuration of device interface ... 35
6.2. Summary of data types ... 35
6.3. Interface functions ... 35
7. Front LED Device Interface ... **38**
7.1. Configuration of device interface ... 38
7.2. Summary of data types ... 38
7.3. Interface functions ... 38
8. Accelerometer Device Interface ... **42**
8.1. Configuration of device interface ... 42

8.2. Summary of data types........42
8.3. Interface functions........43
9. Watchdog Device Interface........46
9.1. Configuration of device interface........46
9.2. Summary of data types........46
9.3. Interface functions........46
10.Serial Number Broadcast interface........49
11.Technical support........50
Trademark, etc. 50

Revision history

Rev	Date	Comments
1.0	2010-07-01	

1. Introduction

1.1. Purpose

This document contains reference information describing driver calls and APIs used when developing applications for the *CrossCore XA* product running the Linux operating system v1.0.0.

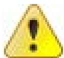
For non-platform specific access to driver calls there is a HALIO-library[3]. For CrossControl specific *canapi* calls there is a CANAPI-library[5].

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

1.2. Conventions and defines

Text formats used in this document.

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



Is used to highlight important information.

Common defines used in this document	
True	Non-zero value
False	Zero value

The term *CrossCore XA* is used to refer both of the device types simultaneously.

1.3. References

[1] CrossCore XA – Software User Guide
[2] Codesourcery Sourcery G++ user manual
[3] CrossCore XA – HALIO
[4] http://openfacts.berlios.de/index-en.phtml?title=Socket-CAN
[5] CrossCore XA – CANAPI

1.4. Include files and libraries

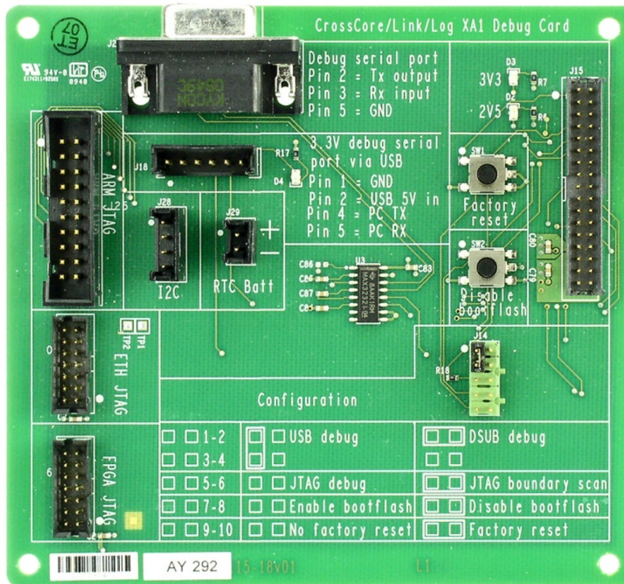
The programmer's guide contains references to include files needed when programming the device. Note that these files can be downloaded separately in a development package from CrossControl web site. This package also includes libraries to use for application development.

1.5. Floating point support

CrossCore XA does not have hardware floating point support, therefore the floating point calculations are done by software. *Codesourcery* cross compiler tool chain [2] supports soft floats and they are also used in *CrossCore XA* as default floats.

2. Debug card

For connecting and debugging software in *CrossCore XA* there exist a debug card. The debug card has several configuration alternatives and connection interfaces accessibility using jumper configuration.



2.1. J14 - Configuration header

Debug card has several different options that are user configurable. They are shown below.

Pins	First option	Second option
1 – 2 3 – 4	USB Serial port	DSUB Serial port
5 – 6	JTAG Debug	JTAG Boundary scan
7 – 8	Enable Bootflash	Disable Bootflash
9 – 10	No factory reset	Factory reset

2.2. J15 – Connection to CrossCore XA

J15 is used to connect debug card to actual *CrossCore XA* hardware with a ribbon cable provided with the debug card.

2.3. J18 – USB Serialport

To connect to debug console, either USB or DSUB9 serial port can be used. J18 is for the USB – serial port. It requires external FT232 (or similar.) USB to RS232 – converter. USB and DSUB9 serial port cannot be used at the same time.

PIN	DESC
1	GND
2	N/C
3	VDD +5V
4	TX
5	RX
6	N/C

2.4. J19 – FPGA

J19 provides interface to reprogram FPGA in *CrossCore XA*.

PIN	DESC.	PIN	DESC.
1	GND	2	2.5V
3	GND	4	FPGA TMS
5	GND	6	FPGA TCK
7	GND	8	FPGA TDO
9	GND	10	FPGA TDI
11	GND	12	N/C
13	GND	14	N/C

2.5. J20 – Ethernet JTAG

The Ethernet JTAG is solely used for service related purposes

2.6. J25 – ARM JTAG

J25 is standard JTAG used for debuging software run by CPU.

PIN	DESC.	PIN	DESC.
1	3.3V	2	3.3V
3	CPU NTRST	4	GND
5	CPU TDI	6	GND
7	CPU TMS	8	GND
9	CPU TCK	10	GND
11	CPU RTCK	12	GND
13	CPU TDO	14	GND
15	NRST	16	GND
17	N/C	18	GND
19	N/C	20	GND

2.7. J26 – RS232

J26 is standard DBUS RS232 – serial port for debug console. It's alternative for J18 and they cannot be used at the same time.

PIN	DESC.
2	TX
3	RX
5	GND
REST	N/C

2.8. J28 – I2C

I2C can be debugged via J28.

PIN	DESC.
1	3.3V
2	SCL
3	SDA
4	GND

2.9. J29 – RTC Battery

RTC battery voltage can be measured here.

PIN	DESC.
1	BATT+
2	GND

2.10. SW1 – Factory Reset

Factory reset can be activated by pressing this switch when powering up the device. After turning power on for the device, the switch can be released. Factory reset will go through and device will boot to normal operation mode once the factory reset is over.

Factory reset will remove all the data from */usr/local/ and /media/cf/.*

Warning: After factory reset device will generate new SSH keys and do other initialization, that if not done correctly may cause device not to function properly, so do not cut the power during this state and wait until unit indicates boot completion by changing status LED to green.

2.11. SW2 – Bootflash Disable

In case of the device software being corrupted so badly that even backup system won't work, the device can be forced to internal Romboot mode where it can be reprogrammed via Serial – or USB – port. Press *Bootflash disable* – switch when powering up the device and device will enter the Romboot mode. After connecting power, the button can be released.

3. CAN Communication Interface

The CAN Communication interface enables the caller to receive messages from and send messages to the CAN interfaces on the *CrossCore XA* device. The CAN communication is handled over the SocketCAN interface, which is a method to use standard socket API functions for CAN communication [4].

3.1. Configuration of device interface

The device node files for the two CAN interfaces are *can0* and *can1*, which should be shown when listing all network interfaces with the **ifconfig** command. The device driver is implemented as loadable kernel modules, *can_dev.ko, xilinx.ko* and *xilinx_platform.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 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**:

```
# 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 second CAN interface by changing *can0* to *can1*.

3.1.1. Selecting CAN driver operation modes

Starting from SW release 0.4.0.0, there are two different operation modes available for the CAN driver *xilinx.ko*: accurate mode and rapid mode.

In accurate mode, received messages are handled one by one, which consumes more CPU time, but ensures accurate timestamps of received CAN frames. This mode is recommended if busload is low and/or there is a need to have as accurate timing as possible.

In rapid mode, received messages are buffered on the CAN controller and handled in groups. This mode reduces CPU load, but since messages are stored for a while on the CAN controller, the resulting timestamps are affected by this period.

Rapid mode is enabled with a module parameter given to the module at load time. Polling period is given in parameter as micro seconds.

Warning: If period is set too long it will cause messages to be lost. What is too long depends on bus speed and bus load.

```
# modprobe xilinx rapidmode=500
```

By default, the driver uses accurate mode. Operation mode cannot be changed during runtime. The same operation mode is always used for both CAN interfaces.

3.1.2. Bus recovery options

There are two options for implementing bus recovery after busoff 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 specified time period.

Automatic bus recovery from busoff state is turned off by default. User can turn it on via sysfs setting wanted restart period in milliseconds into *can_restart_ms* variable. For example 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
```

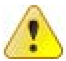
Same commands apply for *can1* by replacing *can0* appropriately. Period is possible to set as needed. Value zero turns automatic bus recovery off.

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

3.1.3. Error interrupt options

Error interrupts are disabled by default. Enabled them by giving module parameter **errorirq=1** during module loading. By enabling error interrupts user can receive error frames. Bus off errors will come through even if the error interrupts are not enabled.

```
# modprobe 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.

3.1.4. Silent mode option

The Xilinx platform module implements a common listen-only mode for both CAN interfaces. In this mode, CAN controller only listens to the CAN receive line without acknowledging the received messages. Sending messages is also disabled. This mode is useful for bus loggers and also needed for undisturbed bus communication under automatic baud rate detection.

Silent mode can be enabled or disabled for each CAN interface using module parameter **silent** with *xilinx_platform* driver.

```
# modprobe xilinx_platform silent=0x3
```

Silent value is an interpret bitwise value for setting each CAN interface to listen-only mode or normal mode. Lowest bit is for *can0* and second bit is for *can1*. Bit value ‘1’ is for listen-only and ‘0’ is for normal operation mode. In example above, both interfaces are set to listen-only mode.

3.1.5. Automatic baud rate detection

Automatic baud rate detection is implemented as a part of *canapi* library. Automatic baud rate detection is started by calling:

```
CanHandle Handle = CanOpen("can1");
CanSetBaudrate(Handle, CCCAN_BAUDRATE_AUTO);
```

Because *canapi* is running on top of SocketCAN and using a socket API, there are three limitations in calling automatic baud rate detection:

1. **CanSetBaudrate** can be called only once with *CCCAN_BAUDRATE_AUTO* after driver modules are loaded.
2. Other access to CAN interface under detection is disturbed by detection (interface is dropped during detection).
3. Both silent mode and error interrupts must be enabled in drivers for successful baud rate detection.

See *CC CANAPI documentation*[5] for more details.

Detection speed depends on actual baud rate as well as bus load. Once the baud rate has been found the baud rate detection is disabled.

3.2. Summary of data types

The following data types are needed to access SocketCAN interface from user application. Note that this list is only partial set of all socket features.

SocketCAN uses its own protocol family PF_CAN coexisting with others like PF_INET. Communication is done analogue to the use of Internet Protocol via Sockets. The protocol family provide the structures to enable different protocols on the bus.

CrossCore XA module support by default raw sockets for direct communication and broadcast manager (bcm) for sending messages periodically or realizing complex message filters.

Definitions for particular protocols of the protocol family PF_CAN:

Define	Description
CAN_RAW	Raw sockets
CAN_BCM	Broadcast manager
CAN_TP16	VAG transport protocol v1.6
CAN_TP20	VAG transport protocol v2.0
CAN_MCNET	Bosch MCNet
CAN_ISOTP	ISO 15765-2

Enum value defining raw socket protocol level options for **setsockopt**:

Define	Description
CAN_RAW_FILTER = 1	Set 0..n can_filter(s)
CAN_RAW_ERR_FILTER	Set filter for error frames
CAN_RAW_LOOPBACK	Set local loopback (echo, default: on)
CAN_RAW_RECV_OWN_MSGS	Set receiving of own messages (default: off)

For socket level options to use with **setsockopt**, see *sys/socket.h*.

Definitions for CAN_ID description flags:

Define	Description
CAN_EFF_FLAG	Extended identifier flag is set
CAN_RTR_FLAG	Remote transmission request
CAN_ERR_FLAG	Error frame

The interface uses following struct to transfer data to and from the driver:

```
struct can_frame {
   canid_t can_id;  /* 32 bit CAN_ID + EFF/RTR/ERR flags */
   __u8    can_dlc; /* data length code: 0 .. 8 */
   __u8    data[8] __attribute__((aligned(8))); /* payload */
};
```

The interface uses following struct to define **sockaddr** structure for CAN sockets:

```
struct sockaddr_can {
   sa_family_t can_family;
   int         can_ifindex;
   union {
      /* transport protocol class address information (e.g. ISOTP) */
      struct { canid_t rx_id, tx_id; } tp;

      /* reserved for future CAN protocols address information */
   } can_addr;
};
```

The interface to the uses following struct to transfer define CAN id base filter:

```
struct can_filter {
   canid_t can_id;
   canid_t can_mask;
};
```

3.3. Interface functions

SocketCAN access is similar to network socket. Only some basic functionality needed for socket access is introduced here. Please see general Linux socket API documentation for further information regarding socket communication.

Function	Description
socket	Creates a communication socket.
bind	Binds socket to a name.
setsockopt	Sets socket options.
select	Tests given socket(s) for accessibility.
read	Reads from a socket.
recvfrom	Receives from a socket (same as read).
write	Writes to a socket.
sendto	Sends to a socket (same as write).
connect	Creates a connection.
listen	Waits for incoming connection.
accept	Accepts a connection from client.
close	Closes a socket.

3.3.1. socket

Description

Create an endpoint for communication.

Include files

#include <sys/socket.h>

Syntax

```
int socket(
   int domain,
   int type,
   int protocol
)
```

Parameters

domain	Specifies communication domain
type	Specifies socket type
protocol	Specifies a particular protocol to be used with the socket

Return value

Returns a non-negative integer, the socket file descriptor, on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

The following example shows how to use SocketCAN for initial scope of CAN_RAW socket communication. Note that this is an example; all possible errors are not handled.

```
#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 );

   /* 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 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) );

/* 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) );
```

3.3.2. bind

Description

Bind a name to a socket.

Include files

#include <sys/socket.h>

Syntax

```
int bind(
   int socket,
   const struct sockaddr *address,
   socklen_t address_len
)
```

Parameters

fd	Socket descriptor
address	Pointer to sockaddr structure containing address to be bound
address_len	Specifies the length of sockaddr structure

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of socket.

3.3.3. setsockopt

Description

Set socket options

Include files

#include <sys/socket.h>

Syntax

```
int setsockopt(
   int socket,
   int level,
   int option_name,
   const void *option_value,
   socklen_t option_len
)
```

Parameters

socket	Socket file descriptor
level	Protocol level (SOL_CAN_RAW or similar)
option_name	Option to set, see definitions in 2.2
option_value	Pointer to new value
option_len	Specifies length of option_value

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

This example shows how to disable filters and loopback feature using setsockopt.

```
   int s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
   setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);

   if (opt == 'B') {
      const int loopback = 0;

      setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK,
              &loopback, sizeof(loopback));
   }
```

3.3.4. select

Description

Performs synchronous I/O multiplexing. Select is used to examine several file descriptors at once to determine if some of them is ready for reading, ready for writing or have an exceptional condition pending.

Include files

#include <sys/select.h>

Syntax

```
int select(
   int nfds,
   fd_set *readfds,
   fd_set *writefds,
   fd_set *errorfds,
   struct timeval *timeout
)
```

Parameters

nfds	Specifies sizes of fd_set type parameters
readfds	Read file descriptor list
writefds	Write file descriptor list
errorfds	Error file descriptor list
timeout	Specifies maximum wait time for select to block

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

This example shows how to use select to determine readability of sockets and how data is received from sockets using **recvfrom()**.

```
fd_set rdfs;
int s[MAXDEV];
int currmax = 1; /* we assume at least one can bus */
int running = 1
int nbytes, i;
struct sockaddr_can addr;
struct can_filter rfilter;
struct can_frame frame;

/* ... Initialize sockets and set currmax to correct value in here ... */

while (running) {
   FD_ZERO(&rdfs);
   for (i=0; i<currmax; i++)
      FD_SET(s[i], &rdfs);

   if ((ret = select(s[currmax-1]+1, &rdfs, NULL, NULL, NULL)) < 0) {
      perror("select");
      running = 0;
      continue;
   }

   for (i=0; i<currmax; i++) {  /* check all CAN RAW sockets */

      if (FD_ISSET(s[i], &rdfs)) {

         socklen_t len = sizeof(addr);
         int idx;

         if ((nbytes = recvfrom(s[i], &frame,
                   sizeof(struct can_frame), 0,
                   (struct sockaddr*)&addr, &len)) < 0) {
            perror("read");
            return 1;
         }
```

```
            if (nbytes < sizeof(struct can_frame)) {
                fprintf(stderr, "read: incomplete CAN frame\n");
                return 1;
            }
        }
    }
}
```

3.3.5. read

Description

Read from file.

Include files

#include <unistd.h>

Syntax

```
ssize_t read(
   int fd,
   void *buf,
   size_t nbyte
)
```

Parameters

fd	File descriptor
buf	Points to buffer where the data should be stored
nbyte	Specifies buffer length in bytes

Return value

Returns a non-negative integer indicating the number of bytes actually read. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of socket.

3.3.6. recvfrom

Description

Receive a message from a socket. If no messages are available, it shall block until a message arrives.

Include files

#include <sys/socket.h>

Syntax

```
ssize_t recvfrom(
   int socket,
   void *buffer,
   size_t length,
   int flags,
   struct sockaddr *address,
   socklen_t *address_len
)
```

Parameters

socket	Socket file descriptor
buffer	Points to buffer where the message should be stored
length	Specifies the length of receive buffer
flags	Specifies the type of message reception
address	A null pointer or points to a sockaddr structure in which the sending address is to be stored
address_len	Specifies the length of address structure

Return value

Returns length of the message in bytes on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of select.

3.3.7. write

Description

Write on file.

Include files

#include <unistd.h>

Syntax

```
int write(
   int fd,
   const void *buf,
   size_t nbyte
)
```

Parameters

Fd	File descriptor
Buf	Points to buffer containing data to write
Nbyt	Specifies write size in bytes

Return value

Returns non-negative integer indicating the number of bytes actually written. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of socket.

3.3.8. sendto

Description

Send a message to a socket.

Include files

#include <sys/socket.h>

Syntax

```
ssize_t sendto(
   int socket,
   const void *buffer,
   size_t length,
   int flags,
   const struct sockaddr *to,
   socklen_t tolen
)
```

Parameters

socket	Socket file descriptor
buffer	Points to a buffer containing message to send
length	Specifies message length in bytes
flags	Specifies the type of message transmission
to	Specifies address of the target
tolen	Specifies address size

Return value

Returns non-negative integer indicating bytes sent on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

This example shows how to operate bcm socket and send data to it.

```
   int sl, sa, sc;
   struct sockaddr_can caddr;
   socklen_t caddrlen = sizeof(caddr);
   struct ifreq ifr;

   struct {
      struct bcm_msg_head msg_head;
      struct can_frame frame;
   } msg;
```

```
if((sl = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("inetsocket");
    exit(1);
}

/* open "regular" network socket */
saddr.sin_family = AF_INET;
saddr.sin_addr.s_addr = htonl(INADDR_ANY);
saddr.sin_port = htons(PORT);

while(bind(sl,(struct sockaddr*)&saddr, sizeof(saddr)) < 0) {
    printf(".");fflush(NULL);
    usleep(100000);
}

/* Listen socket for a connection */
if (listen(sl,3) != 0) {
    perror("listen");
    exit(1);
}

while (1) {
    sa = accept(sl,(struct sockaddr *)&clientaddr, &sin_size);
    if (sa > 0 ){
        if (fork())
            close(sa);
        else
            break;
    }
}

/* open BCM socket */
if ((sc = socket(PF_CAN, SOCK_DGRAM, CAN_BCM)) < 0) {
    perror("bcmsocket");
    return 1;
}

/* Connect socket */
if (connect(sc, (struct sockaddr *)&caddr, sizeof(caddr)) < 0) {
    perror("connect");
    return 1;
}

caddr.can_family = PF_CAN;
caddr.can_ifindex = ifr.ifr_ifindex;

while(1){
    FD_ZERO(&readfds);
    FD_SET(sc, &readfds);
    FD_SET(sa, &readfds);
```

```
    ret = select((sc > sa)?sc+1:sa+1, &readfds, NULL, NULL, NULL);

    if (FD_ISSET(sc, &readfds)) {
        ret = recvfrom(sc, &msg, sizeof(msg), 0,
                (struct sockaddr*)&caddr, &caddrlen);
    }

    if (FD_ISSET(sa, &readfds)) {
        if (read(sa, buf+idx, 1) < 1)
            exit(1);
    }

    if (!ioctl(sc, SIOCGIFINDEX, &ifr)) {
        /* Send message to BCM socket */
        sendto(sc, &msg, sizeof(msg), 0,
            (struct sockaddr*)&caddr, sizeof(caddr));
    }
}
```

3.3.9. connect

Description

Attempt to make a connection on socket.

Include files

#include <sys/socket.h>

Syntax

```
int connect(
  int socket,
  const struct sockaddr *address,
  socklen_t address_len
)
```

Parameters

Socket	Socket file descriptor
address	Points to a sockaddr structure containing the peer address
address_len	Specifies the length of the address argument

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of send.

3.3.10. listen

Description

Listen for socket connections and limit the queue of incoming connections.

Include files

#include <sys/socket.h>

Syntax

```
int listen(
   int socket,
   int backlog
)
```

Parameters

socket	Socket file descriptor
backlog	Provides a hint to use to limit the number of connections

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of send.

3.3.11. accept

Description

Accept a new connection on a socket.

Accept may be called to a socket that was created with **socket()**, has been bound to an address with **bind()**, and has issued a successful call to **listen()**.

Include files

#include <sys/socket.h>

Syntax

```
int accept(
   int socket,
   struct sockaddr *address,
   socklen_t *address_len
)
```

Parameters

socket	Socket file descriptor
address	Either a null pointer, or a pointer to a sockaddr structure where the address of the connecting socket shall be returned.
address_len	Points to a socklen_t structure which on input specifies the length of the supplied sockaddr structure, and on output specifies the length of the stored address.

Return value

Returns the non-negative file descriptor of the accepted socket on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of send.

3.3.12. close

Description

Closes the file descriptor.

Include files

#include <unistd.h>

Syntax

```
int close(
  int fd
)
```

Parameters

Fd	File descriptor

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

```
  close(socket);
```

4. Digital I/O Device Interface

There are eight digital input signals that can be monitored through **ioctl()** calls or the read function, and two digital output signals that can be controlled through **ioctl()** call or **write** function.

4.1. Configuration of device interface

The device node file for the I/O interface is */dev/digio*, which is the entry point to the I/O driver.

4.2. Summary of data types

The interface to the digital I/O device does not need to use an explicit data type; it can simply use an unsigned integer. **ioctl**-commands use the parameter a little differently. Here is the definition of the parameters:

Define	Description
CROSS_DIG_IOC_SET	16-bit unsigned integer. Upper 8-bit is the output-pin number, 1-8. Lower 8-bit is the state value of the pin, 0 or 1.
CROSS_DIG_IOC_GET	8-bit unsigned integer. Number of input-pin to get, 1-8.
CROSS_DIG_IOC_GET_ALL	Returns 16-bit unsigned integer. Upper 8-bits contain values for connector 1. Lower 8-bits contain values for connector 2. In both 8 bits, lowest 4 bits are values of input pins, and bit 5 is the value of the output pin in that connector.

4.3. Interface functions

open	Opens the I/O device
close	Close the I/O device
read	Reads from the I/O device
write	Writes to the I/O device
ioctl	Used for reading and writing from/to device

4.3.1. open

Description

Opens the I/O device. The call will return a file descriptor used in subsequent calls such as **read()** and **ioctl()**.

Include files

#include <fctnl.h>

Syntax

```
int open(
   const char *pathname,
   int flags
)
```

Parameters

pathname	Path to the device file
flags	Parameter flag must include one of the following access modes: O_RDONLY, O_WRONLY or O_RDWR

Return value

The new file descriptor if successful. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **ioctl**.

4.3.2. close

Description

Closes the file descriptor.

Include files

#include <unistd.h>

Syntax

```
int close(
   int fd
)
```

Parameters

fd	File descriptor

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **ioctl**.

4.3.3. read

Description

Reads data from the digital I/O-device. Note that the returned data is in a textual format that shows the status of all the digital-in signals described in the I/O map section. For example of data output, look at the last example listing in this chapter.

Include files

#include <unistd.h>

Syntax

```
ssize_t read(
   int fd,
   void *buf,
   size_t count,
)
```

Parameters

fd	File descriptor
buf	Pointer to the buffer where received data is stored
count	Number of bytes to read from the I/O-device

Return value

On success, the number of read bytes is returned. In case of errors, -1 is returned and *errno* is set appropriately.

Example

This example shows how to open a device (*/dev/digio*) and read data. Note that this example is intended to show how to make the different function calls. All possible errors are not handled.

```
int fd, result;
unsigned char buffer[1000];

/* Open device */
fd = open("/dev/digio", O_NONBLOCK | O_RDWR)
if(fd >= 0)
{
        printf("open OK\n");
}
else
{
        printf("open FAILED!\n");
}

/* Read data from the device */
result = read(fd, &buffer, 512);
if(result < 0)
{
        printf("read ERROR!\n");
}
else
{
        printf("read OK!\n");
}
```

Format of read data contents:

```
$ cat /dev/digio
 1 = 1
 2 = 0
 3 = 1
 4 = 0
 5 = 1
 6 = 0
 7 = 1
 8 = 0
```

4.3.4. write

Description

Writes a value to a digital I/O-device.

Include files

#include <unistd.h>

Syntax

```
ssize_t write(
   int fd,
   void *buf,
   size_t count,
 )
```

Parameters

fd	File descriptor
buf	Code of the request
count	Number of bytes to write to the I/O device

Return value

On success, the number of written bytes is returned. In case of errors, -1 is returned and *errno* is set appropriately.

Example

Note that this example is intended to show how to make the different function calls. All possible errors are not handled.

```
int fd, retval;
unsigned char buffer[1000];

fd = open("/dev/digio", O_WRONLY);
if (fd < 0)
{
        perror("Unable to open device");
        return -1;
}

/* Set the output 1 to 0 */
memset(buffer, 0, 100);
memcpy(buffer,"1 0", 4);
retval = write(fd, buffer, 4);
if (retval < 0)
{
        perror("Unable to set IO-pin");
        return -1;
}

close(fd);
```

4.3.5. ioctl

Description

The **ioctl** function calls are used for accessing the Digital I/O - device. The available **ioctl** operations

Define	Description
CROSS_DIG_IOC_SET	Set Digital output-pin state
CROSS_DIG_IOC_GET	Get Digital input-pin state
CROSS_DIG_IOC_GET_ALL	Get state of all IO-pins at once

Include files

#include <sys/ioctl.h>
#include “dig-io.h”

Syntax

```
int ioctl(
   int fd,
   int request,
   void *argp
)
```

Parameters

Fd	File descriptor
Request	Code of the request
Argp	Used for transferring data in the ioctl call. In this case it is an integer.

Return value

Returns a positive value on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

This example shows a range of different configuration settings by calling **ioctl()**. All possible errors are not handled, since the purpose of the example is to explain how the calls are made.

```
int fd, result;
int data;

/* Open device */
fd = open(“/dev/digio”, O_RDWR | O_NONBLOCK);

/* Read from device - Get input 3 state*/
data = 3;

result = ioctl(fd, CROSS_DIG_IOC_GET, data);

/* Write to device – Set Output pin 2 to 1 */
data = (2 << 8) | 1;

result = ioctl(fd, CROSS_DIG_IOC_SET, data);
```

5. Power Device Interface

There is a set of built-in modules, which have their supply power controlled by **ioctl()** calls or the **write** function. Also the device power supply state has a couple of state signals, which can be accessed through this interface.

5.1. Configuration of device interface

The device node file for the I/O interface is */dev/cross_pwr_io*, which is the entry point to the PWR driver.

5.2. Summary of data types

The interface to the PWR IO device does not need to use an explicit data type, simply use an unsigned integer. **ioctl**-commands use the parameter a little differently. Here is the definition:

Define	Description
CROSS_PWR_IOC_SET_PWR_STATE	16-bit unsigned integer. Upper 8 – bit is PWR_PIN number. Lower 8-bit is the state of the pin.
CROSS_PWR_IOC_GET_PWR_STATE	8-bit unsigned integer, stating the number of PWR_PIN to get.
CROSS_PWR_IOC_GET_PWRFAIL_STATUS	Return either true if the pwr fail is active or false if the pwr fail status is not active
CROSS_PWR_IOC_GET_OVERVOLTAGE_STATUS	Return either true if the overvoltage is active or false if the overvoltage is not active
CROSS_PWR_IOC_GET_SIGNAL	Return status of which event caused the signal. Power down is bit 0 and over voltage is bit 2.
CROSS_PWR_IOC_SET_SIGNAL	Set the signal handler that is called when power signals change state. Argument is pointer to set_pwr_signal_struct. typedef struct { uint32 sig; /* Signal number to be used */ uint32 pid; /* Process ID of the handler */ } set_pwr_signal_struct;

PWR_STATE_PINS enum; the *PWR_PIN* number used by **read/write/ioctl**:

PWR_PIN_ETH = 0	Ethernet adapter
PWR_PIN_WLAN = 1	WLAN card
PWR_PIN_ADDON = 2	Addon card
PWR_PIN_CAN = 3	CAN module
PWR_PIN_USB = 4	USB module
PWR_PIN_GPRS = 5	GSM/GPRS modem

5.3. Interface functions

open	Opens the I/O device
close	Closes the I/O device
read	Reads from the I/O device
write	Writes to the I/O device
ioctl	Used for reading and writing from/to device

5.3.1. open

Description

Opens the PWR device. The call will return a file descriptor used in subsequent calls such as **read()** and **ioctl()**.

Include files

#include <fctnl.h>

Syntax

```
int open(
   const char *pathname,
   int flags
 )
```

Parameters

Pathname	Path to the device file
Flags	Parameter flag must include one of the following access modes: O_RDONLY, O_WRONLY or O_RDWR

Return value

The new file descriptor if successful. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **ioctl**.

5.3.2. close

Description

Closes the file descriptor

Include files

#include <unistd.h>

Syntax

```
int close(
  int fd
)
```

Parameters

fd	File descriptor

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **ioctl**.

5.3.3. read

Description

Reads data from the PWR-device. Note that the returned data is in a kind of textual format that shows the status of all the PWR-states.

Include files

#include <unistd.h>

Syntax

```
ssize_t read(
   int fd,
   void *buf,
   size_t count,
 )
```

Parameters

fd	File descriptor
buf	Pointer to the buffer where received data is stored
count	Number of bytes to read from the PWR-device

Return value

On success, the number of read bytes is returned. In case of errors, -1 is returned and *errno* is set appropriately.

Example

This example shows how to open a device (*/dev/cross_pwr_io*) and read data. Note that this example is intended to show how to make the different function calls. All possible errors are not handled.

```
int fd, result;
unsigned char buffer[1000];

/* Open device */
fd = open("/dev/cross_pwr_io", O_NONBLOCK | O_RDWR)
if(fd >= 0)
{
        printf("open OK\n");
} else {
        printf("open FAILED!\n");
}

/* Read data from the device */
result = read(fd, &buffer, 512);
if(result < 0)
{
        printf("read ERROR!\n");
} else {
        printf("read OK!\n");
}
```

5.3.4. write

Description

Writes a value to a PWR-device. Values can be set on or off independently.

Include files

#include <unistd.h>

Syntax

```
ssize_t write(
   int fd,
   void *buf,
   size_t count,
 )
```

Parameters

fd File descriptor
buf Code of the request
count Number of bytes to write to the front LED device

Return value

On success, the number of written bytes is returned. In case of errors, -1 is returned and *errno* is set appropriately.

Example

Note that this example is intended to show how to make the different function calls. All possible errors are not handled.

```
int fd, retval;

fd = open("/dev/cross_pwr_io", O_WRONLY);
if (fd < 0)
{
        perror("Unable to open cross_pwr_io");
        return -1;
}

/* Set the output 0(Eth) to 0 */
memset(resp_buf, 0, 100);
memcpy(resp_buf,"0 0", 4);
retval = write(fd, resp_buf, 4);
if (retval < 0)
{
        perror("Unable to set PWR-state");
        return -1;
}

close(fd);
```

5.3.5. ioctl

Description

The **ioctl** function calls are used for accessing the I/O device at different offsets. The available **ioctl** operations are summarized in the table below.

Define	Description
CROSS_PWR_IOC_GET_PWRFAIL_STATUS	Get low power – pin state
CROSS_PWR_IOC_GET_OVERVOLTAGE_STATUS	Get over voltage – pin state
CROSS_PWR_IOC_GET_SIGNAL	Get source what caused signal
CROSS_PWR_IOC_SET_SIGNAL	Register signal handler. Only one process can register signal handler. Note! Before closing the driver, remove handler by setting both pid and signal number to zero.
CROSS_PWR_IOC_GET_PWR_STATE	Get power state of specific subsystem
CROSS_PWR_IOC_SET_PWR_STATE	Set power state of specific subsystem

Include files

#include <sys/ioctl.h>
#include “pwr-io.h”

Syntax

```
int ioctl(
   int fd,
   int request,
   void *argp
)
```

Parameters

fd — The file descriptor
request — The code of the request
argp — Used for transferring data in the ioctl call.

Return value

Returns a positive value on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

This example shows a range of different configuration settings by calling **ioctl()**. All possible errors are not handled, since the purpose of the example is to explain how the calls are made.

```
int fd, result;

/* Open device */
fd = open("/dev/cross_pwr_io", O_RDWR | O_NONBLOCK);

/* Read from device - Get power state for Eth */
result = ioctl(fd, CROSS_PWR_IOC_GET_PWR_STATE, PWR_PIN_ETH);

/* Write to device – Set power state for Wlan */
data = (PWR_PIN_WLAN << 8) | 0;

result = ioctl(fd, CROSS_PWR_IOC_SET_PWR_STATE, data);
```

5.4. Power signal handling

There are two signals that user applications can use for monitoring status of the power supply voltage. When either one of them activates, the application should call the power interface to turn off all unnecessary built-in peripherals, in order to preserve power.

Then program should start writing any modified data to file, close all open files as soon as possible and exit. As a last action, the program should initiate system shutdown by calling **post-powerfailure**. System shutdown will kill all other tasks and remount file systems as read-only.

5.4.1. Configuring signal handler

Signal handler is registered through */dev/cross_pwr_io* – device's **ioctl** call *CROSS_IOC_SET_SIGNAL*. The handler receives reason for the signal as an argument.

Argument bitfield

Define	Description
POWER_DOWN_SIGNAL	Power down occurred
OVERVOLTAGE_SIGNAL	Overvoltage occurred

It is possible to have only one or both reasons active at the same time, especially in case of overvoltage.

Example

This example shows a very simple signal handler. All possible errors are not handled, since the purpose of the example is to explain how the calls are made.

```
int fd, result;
set_pwr_signal_struct pwrsig;
int pwr_state = 0;

void power_failure_handler(int state)
{
 pwr_state = state;
}

pwrsig.pid = getpid();
pwrsig.sig = CROSS_PWR_SIG_INTERFACE;

fd = open("/dev/cross_pwr_io", O_RDWR | O_NONBLOCK);

/* Register power failure signal handler */
result = ioctl(fd, CROSS_PWR_IOC_SET_SIGNAL, &pwrsig);

/* Wait for signal */
while(!pwr_state) {
 sleep(1);
}

if (pwr_state) {
 system("/usr/bin/post-powerfailure");
}
```

6. USB Device Interface

Whether or not a USB device is connected to the USB host can be detected using **ioctl()** calls.

6.1. Configuration of device interface

The device node file for the USB interface is */dev/cross_usb_io*, which is the entry point to the USB driver.

6.2. Summary of data types

The interface to the USB IO device does not need to use an explicit data type, simply use an unsigned integer. **ioctl**-commands use the parameter a little differently. Here is the definition:

Define	Description
CROSS_USB_IOC_USB_STATUS	Return either true if USB is connected or false if USB is not connected.
CROSS_USB_IOC_GET_SIGNAL	Return status of which event caused the signal. USB connected is bit 0.
CROSS_USB_IOC_SET_SIGNAL	Set the signal handler that is called when power signals change state. Argument is pointer to set_usb_signal_struct. typedef struct { uint32 sig; /* Signal number to be used */ uint32 pid; /* Process ID of the handler */ } set_usb_signal_struct;

6.3. Interface functions

open	Opens the I/O device
close	Closes the I/O device
ioctl	Used for reading and writing from/to device

6.3.1. open

Description

Opens the I/O device. The call will return a file descriptor used in subsequent calls such as **read()** and **ioctl()**.

Include files

#include <fctnl.h>

Syntax

```
int open(
   const char *pathname,
   int flags
)
```

Parameters

pathname	Path to the device file
flags	Parameter flag must include one of the following access modes: O_RDONLY, O_WRONLY or O_RDWR

Return value

The new file descriptor if successful. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **ioctl**.

6.3.2. close

Description

Closes the file descriptor

Include files

#include <unistd.h>

Syntax

```
int close(
   int fd
)
```

Parameters

fd	File descriptor

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **ioctl**.

6.3.3. ioctl

Description

The ioctl function calls are used for accessing the I/O device at different offsets. The available **ioctl** operations are summarized in the table below.

Define	Description
CROSS_USB_IOC_GET_USB_STATUS	Get USB connected – pin state
CROSS_USB_IOC_GET_SIGNAL	Get source what caused signal
CROSS_USB_IOC_SET_SIGNAL	Register signal handler. Only one process can register signal handler. Note! Before closing the driver, remove handler by setting both pid and signal number to zero!

Include files

#include <sys/ioctl.h>
#include "usb-io.h"

Syntax

```
int ioctl(
   int fd,
   int request,
   void *argp
)
```

Parameters

fd	File descriptor
request	Code of the request
argp	Used for transferring data in the ioctl call

Return value

Returns a positive value on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

This example shows the different configuration settings by calling **ioctl()**. All possible errors are not handled, since the purpose of the example is to explain how the calls are made.

```
int fd, result;

/* Open device */
fd = open("/dev/cross_usb_io", O_RDWR | O_NONBLOCK);

/* Read from device - Get power state for Eth */
result = ioctl(fd, CROSS_USB_IOC_GET_USB_STATUS, NULL);
```

7. Front LED Device Interface

The *CrossCore XA* device has user controllable LED in the front. It can be set to flash at 1-50 Hz, or to be constantly on or off. Device also has two CAN leds that can be controlled through **ioctl()** calls.

7.1. Configuration of device interface

The front and CAN LED driver is accessed through device node */dev/frontled*.

7.2. Summary of data types

The interface to the front LED device doesn't need to use an explicit data type, simply use an unsigned integer. The unsigned integer values are explained in the following table:

Define	Description
CROSS_DIG_IOC_SET_STATUS_LED	Set the state of the front LED. 16-bit unsigned integer. Upper 8 – bit contains the LED color to control (0 = RED, 1 = GREEN, 2 = AMBER). Lower 8 - bit containing the LED flash interval in Hertz (0 = off, 1-50 = valid Hertz, >50 = constantly on)
CROSS_DIG_IOC_SET_CAN0_LED	Set state of the CAN0 LED. 16-bit unsigned integer. Upper 8 – bit contains the LED color to control (0 = RED, 1 = GREEN, 2 = AMBER). Lower 8 - bit containing the LED flash interval in Hertz (0 = off, 1-50 = valid Hertz, >50 = constantly on)
CROSS_DIG_IOC_SET_CAN1_LED	Set state of the CAN1 LED. 16-bit unsigned integer. Upper 8 – bit contains the LED color to control (0 = RED, 1 = GREEN, 2 = AMBER). Lower 8 - bit containing the LED flash interval in Hertz (0 = off, 1-50 = valid Hertz, >50 = constantly on)

7.3. Interface functions

open	Opens the front LED device
close	Closes the front LED device
write	Writes to the front LED device
ioctl	Writes to the front LED device through a specific function call

7.3.1. open

Description

Opens the front LED device. The call will return a file descriptor used in subsequent calls such as **write()** or **close()**.

Include files

#include <fctnl.h>

Syntax

```
int open(
   const char *pathname,
   int flags
)
```

Parameters

pathname	Path to the device file
flags	Parameter flag must include one of the following access modes: O_RDONLY, O_WRONLY or O_RDWR

Return value

The new file descriptor if successful. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **write**.

7.3.2. close

Description

Closes the file descriptor.

Include files

#include <unistd.h>

Syntax

```
int close(
   int fd
)
```

Parameters

fd	File descriptor

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **write**.

7.3.3. write

Description

Writes a value to the front LED device. It can be set to off, on or flashing in the range (1-50 Hz). Writing the value “0” will turn the front LED off. A value of “51” or higher will turn on the front LED constantly on. String values in the range “1” to “50” will make the front LED flash.

Include files

#include <unistd.h>

Syntax

```
ssize_t write(
   int fd,
   void *buf,
   size_t count,
 )
```

Parameters

fd	File descriptor
buf	Code of the request
count	Number of bytes to write to the front LED device

Return value

On success, the number of written bytes is returned. In case of errors, -1 is returned and *errno* is set appropriately.

Example

Note that this example is intended to show how to make the different function calls. All possible errors are not handled.

```
int fd, retval;

fd = open("/dev/frontled", O_WRONLY);
if (fd < 0)
{
        perror("Unable to open frontled");
        return -1;
}

/* Configure the RED frontled to flash in 10 Hz */
memset(resp_buf, 0, 100);
memcpy(resp_buf,"10 0", 5);
retval = write(fd, resp_buf, 5);
if (retval < 0)
{
        perror("Unable to set front led");
        return -1;
}

close(fd);
```

7.3.4. ioctl

Description

An **ioctl** function call can be used for setting the state of the front LED. It has only one data type which was covered in section 7.2. The available **ioctl** operations are listed in the table below.

Define	Description
CROSS_DIG_IOC_SET_STATUS_LED	Set the state of the front LED
CROSS_DIG_IOC_SET_CAN0_LED	Set the state of the CAN0 LED
CROSS_DIG_IOC_SET_CAN1_LED	Set the state of the CAN1 LED

Include files

#include <sys/ioctl.h>
#include <dig-io.h>

Syntax

```
int ioctl(
   int fd,
   int request,
   void *argp
)
```

Parameters

fd	File descriptor
request	Code of the request
argp	Used for transferring data in the ioctl call. In this case it is an integer.

Return value

Returns a positive value on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

This example shows how to operate the frontled by calling **ioctl()**.Note that this example is intended to show how to make the different function calls. All possible errors are not handled.

```
hertz = 2; /* Slow flashing */
data = (LED_RED << 8) | hertz;
fd = open("/dev/frontled", O_RDWR);

if(!fd)
{
   printf("Open /dev/frontled failed\n");
   return 0;
}
res = ioctl(fd, CROSS_DIG_IOC_SET_STATUS_LED, data);
if(res < 0)
{
   printf("Frontled set error: %d\n", res);
} else {
   printf("Frontled set to %d\n", hertz);
}
close(fd);
```

8. Accelerometer Device Interface

The *CrossCore XA* device can have built-in accelerometer. It can used to detect sudden changes in acceleration. It can be controlled through **ioctl()** calls.

Note: The accelerometer is available as an optional module and may not be available on your unit.

8.1. Configuration of device interface

The accelerometer driver is accessed through device node */dev/accelerometer*.

8.2. Summary of data types

The interface to the uses following struct to transfer data to and from the driver:

```
typedef struct smb380face_tag {
  /////////  triggering interface
  uint16 trigger_count;
  // this is a counter, incremented upon every detected trigger. One way to
  // notify trigger is to periodically poll this value and see if it has changed.

  int32 trigger_client_pid;
  // If this is set nonzero, it is taken as a pid to send a signal to when
  // acceleration surveillance triggers.

  int32 trigger_signo;
  // If trigger_client_pid != 0, this is the signal to send

  uint8 trigger_signal_latch;
  // Everytime signal should be sent, this latch is examined.
  // If the value is zero, signal is sent and this flag is raised.
  // In order to receive signal again, this flag has to be set zero again,
  // by the signal handler for example. This arrangement is solely for
  // avoiding unwanted signal flooding.

  //////// Update operational settings

  uint8 update;
  // This is a flag that driver monitors periodically. If flag is nonzero,
  // operational parameters (defined below) will be taken as new settings
  // and the chip is adjusted accordingly.

  uint8 update_count;
  // After chip is armed with new settings, this counter is incremented
  // to denote succesful update.

  //////// Operational parameters

  // physical characteristics
  uint8 range;  // SMB380_RANGE_xG
  uint8 bandwidth; // SMB380_BANDWIDTH_xHZ
```

```
  uint8 surveillance_strategy;  // bit-or'ed surveillance strategie(s) to arm

  // high gain threshold mode stuff (see SMB380 datasheet for more details)
  uint16 HG_thres;
  uint16 HG_hyst;
  uint16 HG_dur;
  uint16 HG_debounce_mode;

  // any motion mode stuff (see SMB380 datasheet for more details)
  uint8 any_motion_dur;
  uint8 any_motion_thres;
} __attribute__((packed)) smb380face;
```

8.3. Interface functions

open	Opens the accelerometer device
close	Closes the accelerometer device
mmap	Maps accelerometer device into memory

8.3.1. open

Description

Opens the accelerometer device. The call will return a file descriptor used in subsequent calls such as **mmap()**, **close()**.

Include files

#include <fctnl.h>

Syntax

```
int open(
   const char *pathname,
   int flags
 )
```

Parameters

pathname	Path to the device file
flags	Parameter flag must include following access modes: O_RDWR and O_SYNC

Return value

The new file descriptor if successful. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **mmap**.

8.3.2. close

Description

Closes the file descriptor.

Include files

#include <unistd.h>

Syntax

```
int close(
   int fd
)
```

Parameters

fd	File descriptor

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **mmap**.

8.3.3. mmap

Description

Maps accelerometer device into a memory for user access.

Include files

#include <sys/mman.h>
#include <smb380face.h>

Syntax

```
void* mmap(
   void *start,
   size_t length,
   int prot,
   int flags,
   int fd,
   off_t offset
)
```

Parameters

start	Start of the virtual address. NULL=Let kernel choose
length	Length of the mapped memory
prot	Desired memory protections
flags	Flags to determne whether updates to the mapping are visible to other processes mapping the same region
fd	File descriptor
offset	Offset of the file where mapping should start
count	Number of bytes to write to the front LED device

Return value

On success, the number of written bytes is returned. In case of errors, -1 is returned and *errno* is set appropriately.

Example

Note that this example is intended to show how to make the different function calls. All possible errors are not handled.

```
int fd, retval;
int mmapsize = sizeof(smb380face);
smb380face* modules = 0;

#define TRIGGER_SIGNAL SIGUSR2

void handle_trigger(int sno) {
 printf("Trigger signal received!\n");
}

fd = open("/dev/accelerometer", O_RDWR|O_SYNC);

modules = mmap(0, mmapsize, PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_LOCKED, fd, 0);

signal(TRIGGER_SIGNAL,handle_trigger);

modules->update = 1;
modules->trigger_client_pid = getpid();
modules->trigger_signo = TRIGGER_SIGNAL;
modules->trigger_signal_latch = 0;

sleep (30);

modules->trigger_client_pid = 0;
close(fd);
```

9. Watchdog Device Interface

The device has a watchdog device, which can be used to reset the device in case if the software does not respond in predefined time. The watchdog device can be controlled through **ioctl()** calls.

9.1. Configuration of device interface

The device node file for the Watchdog device is */dev/watchdog*, which is the entry point to the Watchdog driver.

By default the watchdog is taken care by specific watchdog kicker software called *wdt*. If user software wants to take over the watchdog handling, then *wdt* task should be killed first (for example **killall wdt**). If the user software does not want to handle the watchdog anymore, then the *wdt* should be restarted. *Wdt* is started at start-up every time if the watchdog timeout is something other than zero.

Watchdog timeout is 16 seconds by default. Timeout can be changed from a file */etc/watchdog_timeout.conf*. Value should be between 0 (=disabled) and 16.

9.2. Summary of data types

The interface to the Watchdog device doesn't use parameters in **ioctl**-calls.

9.3. Interface functions

open	Opens the watchdog device
close	Closes the watchdog device
ioctl	Used for reading and writing from/to device

9.3.1. open

Description

Opens the watchdog device. The call will return a file descriptor used in subsequent calls such as **read()** and **ioctl()**.

Include files

#include <fctnl.h>

Syntax

```
int open(
   const char *pathname,
   int flags
)
```

Parameters

pathname	Path to the device file
flags	Parameter flag must include one of the following access modes: O_RDONLY, O_WRONLY or O_RDWR

Return value

The new file descriptor if successful or -1 if an error occurred. In case of error, *errno* is set appropriately.

Example

See example of **ioctl**.

9.3.2. close

Description

Closes the file descriptor

Include files

#include <unistd.h>

Syntax

```
int close(
   int fd
)
```

Parameters

fd File descriptor

Return value

Returns zero on success. In case of errors, -1 is returned and *errno* is set appropriately.

Example

See example of **ioctl**.

9.3.3. ioctl

Description

The **ioctl** function calls are used for accessing the Watchdog. The available **ioctl** operations are summarized in the table below.

Define	Description
WDIOC_KEEPALIVE	Pat the dog to keep it from waking
WDIOC_GETTIMEOUT	Get the current watchdog timeout

Include files

#include <sys/ioctl.h>
#include <linux/watchdog.h>

Syntax

```
int ioctl(
   int fd,
   int request,
   void *argp
)
```

Parameters

fd	File descriptor
request	Code of the request
argp	Used for transferring data in the ioctl call.

Return value

Returns a positive value on success. On error, -1 is returned and *errno* is set appropriately.

Example

This example shows a way for kicking watchdog by calling **ioctl()**. All possible errors are not handled, since the purpose of the example is to explain how the calls are made.

```
int fd, result,dummy;

/* Open device */
fd = open("/dev/watchdog", O_RDWR | O_NONBLOCK);

/* Pat the watchdog */
result = ioctl(fd, WDIOC_KEEPALIVE, &dummy);
```

10. Serial Number Broadcast interface

Device has 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 *CrossCore XA* 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 Device type. The sender 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 *CrossCore XA* development package *in example_src/snb/snb_reader.c*

11. Technical support

Contact your reseller or supplier for help with possible problems with your *CrossCore XA* device. In order to get the best help, you should have access to your *CrossCore XA* 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
- LED indicator flash patterns.
- *CrossCore XA* device log files (if possible)
- Description of external equipment which is connected to the *CrossCore XA*

Trademark, etc.

© 2006-2010 CrossControl AB

All trademarks sighted in this document are the property of their respective owners.

Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

CrossControl AB is not responsible for editing errors, technical errors or for material which has been omitted in this document. CrossControl is not responsible for unintentional damage or for damage which occurs as a result of supplying, handling or using of this material. The information in this handbook is supplied without any guarantees and can change without prior notification.

cr•ssc•ntr•l

CrossControl AB
P.O. Box 83 • SE-822 22 Alfta • Sweden
Phone: +46 271 75 76 00• info@crosscontrol.se • www.crosscontrol.com