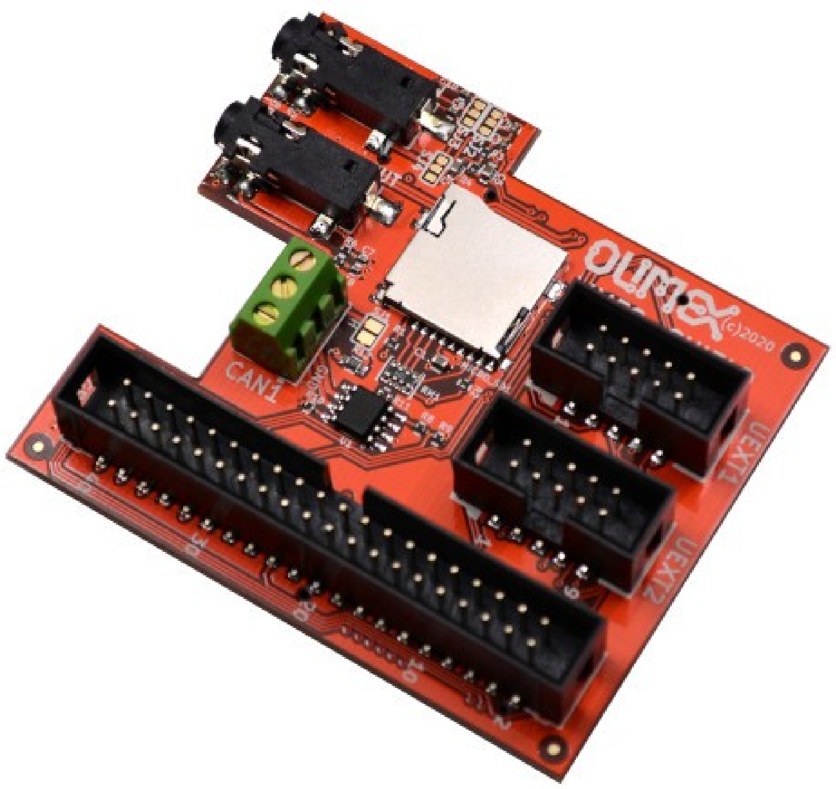


LIME2-SHIELD

User Manual

Rev.1.0 June 2020

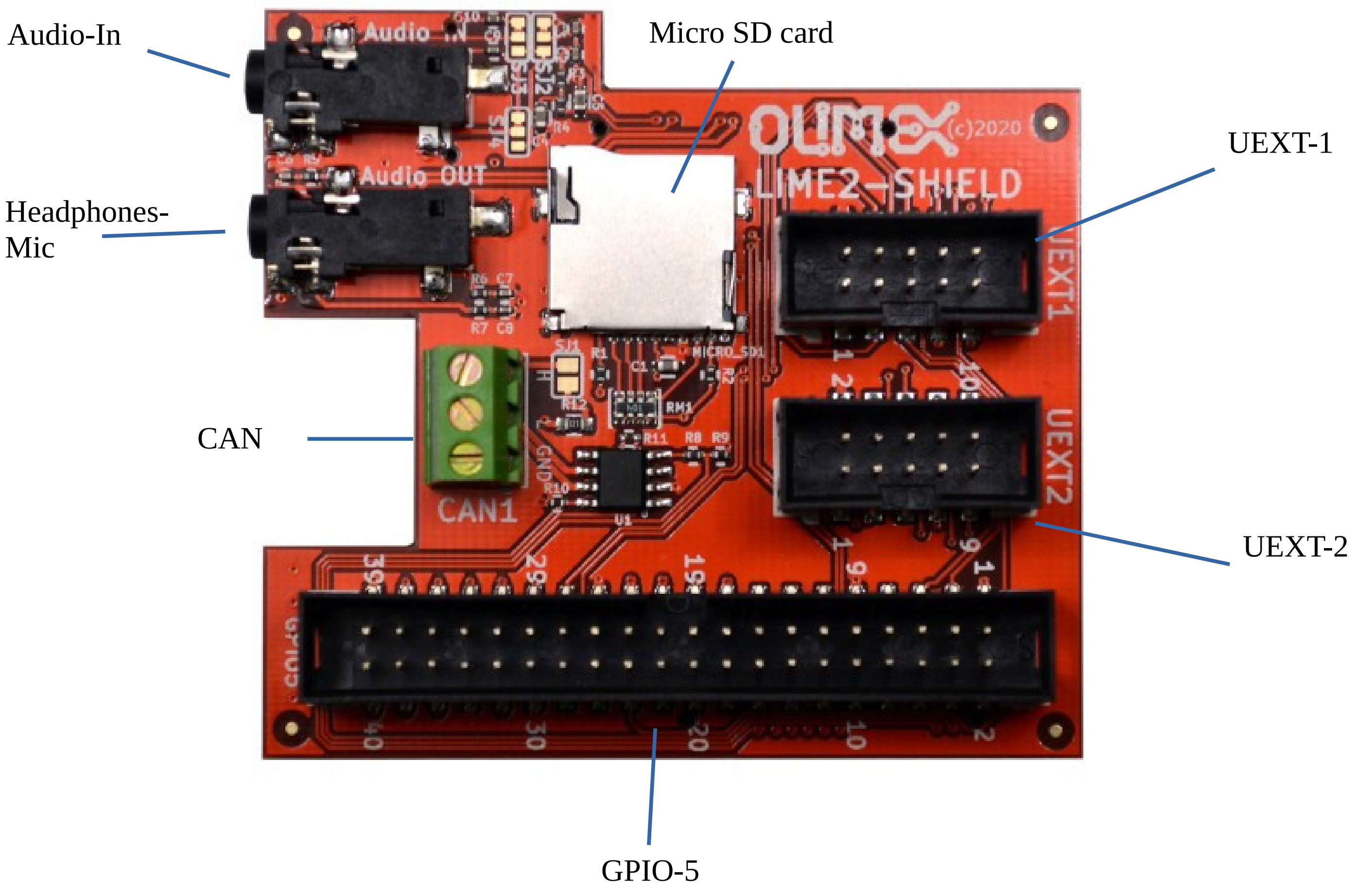
olimex.com

Table of Contents

What is LIME2-SHIELD

LIME2-SHIELD is small board which snaps on top of OLinuXino-LIME2 Linux computers and adds:

- Audio Headphone + microphone 3.5 mm connector
- Audio Line in / out connector
- Second micro SD card for data storage
- CAN driver and connector
- Two UEXT connectors with SERIAL, I2C, SPI available for users to connect UEXT modules
- 40 pin 0.1 breadboard friendly connector which can be used with IDC40 cable or Jumper wires Jwxxx



Available signals and interfaces:

UEXT1

function	Int	Linux	A20			A20	Linux	Int	function
+3.3V				1	2	GND			
UART4-TX		GPIO202	PG10	3	4	PG11	GPIO203		UART4-RX
TWI2-SCL		GPIO52	PB20	5	6	PB21	GPIO53		TWI2-SDA
SPI2-MISO		GPIO86	PC22	7	8	PC21	GPIO85		SPI2-MOSI
SPI2-SCK		GPIO84	PC20	9	10	PC19	GPIO83		SPI2-CS0

UEXT2

function	Int	Linux	A20			A20	Linux	Int	function
+3.3V				1	2	GND			
UART7-TX		GPIO276	PI20	3	4	PI21	GPIO277		UART7-RX
TWI1-SCL		GPIO50	PB18	5	6	PB19	GPIO51		TWI1-SDA
SPI1-MISO	EINT31	GPIO275	PI19	7	8	PI18	GPIO274	EINT30	SPI1-MOSI
SPI1-SCK	EINT29	GPIO273	PI17	9	10	PI16	GPIO272	EINT28	SPI1-CS0

GPIO5

rPi	function	Int	Linux	A20			A20	Linux	Int	function	rPi
+3.3V					1	2	+5V				
GPIO2	TWI2-SDA		GPIO53	PB21	3	4	+5V				
GPIO3	TWI2-SCL		GPIO52	PB20	5	6	GND				
GPIO4			GPIO271	PI15	7	8	PG7	GPIO199		UART3-RX	GPIO14
GND					9	10	PG6	GPIO198		UART3-TX	GPIO15
GPIO17		EINT23	GPIO267	PI11	11	12	PI3	GPIO259		PWM1	GPIO18
GPIO27		EINT22	GPIO266	PI10	13	14	GND				
GPIO22			GPIO263	PI7	15	16	PE6	GPIO136			GPIO23
+3.3V					17	18	PE5	GPIO135			GPIO24
GPIO10	SPI2-MOSI		GPIO85	PC21	19	20	GND				
GPIO9	SPI2-MISO		GPIO86	PC22	21	22	PE4	GPIO132			GPIO25
GPIO11	SPI2-SCK		GPIO84	PC20	23	24	PC19	GPIO83		SPI2-CS0	GPIO8
GND					25	26	PH14	GPIO238	EINT14		GPIO7
GPIO0	TWI1-SDA		GPIO51	PB19	27	28	PB18	GPIO50		TWI1-SCL	GPIO1
GPIO5			GPIO129	PE1	29	30	GND				
GPIO6			GPIO136	PE8	31	32	PH13	GPIO237	EINT13		GPIO12
GPIO13			GPIO137	PE9	33	34	GND				
GPIO19			GPIO138	PE10	35	36	PH12	GPIO236	EINT12		GPIO16
GPIO26			GPIO139	PE11	37	38	PH11	GPIO235	EINT11		GPIO20
GND					39	40	PH10	GPIO234	EINT10		GPIO21

Linux images for SD card booting:

Olimex has it own image build scripts available at GitHub.

The pre-built and tested images can be found at: images.olimex.com. There are release and testing folders, use the stabile images from release only. Testing images are not tested completely and you can use on your own risk.

We support Ubuntu and Debian minimal and basic images. The list of the packages is in the .lst file.

For all examples below we use Ubuntu minimal image .

To make your own SD card download the image and un-archive it with 7z archiver.

Do not use DD for image write to SD-card, use balenaEtcher to write the image to minimum 8GB Class 10 SD card like this one.

We strongly suggest you to use Olimex official images as they are 100% tested and know to work with all board features.

If you have USB cable you can insert it to USB-OTG connector of A20-OLinuXino-LIME2 and run any serial terminal:

```
$ sudo minicom -D /dev/ttyACM0
```

Initial login is user: root or olimex, password: olimex

When new Linux image is released you do not need to download the image and write to the card, but only to run:

```
$ sudo apt update
$ sudo apt upgrade
$ sudo reboot now
```

This way you will have always the latest version of Linux kernel and uboot.

Boot from eMMC:

If you have A20-OLinuXino-LIME2 with eMMC memory you can make your boot from eMMC instead of SD-card to do this you have to boot from SD-card then execute:

```
$ olinuxino-sd-to-emmc
```

after the script finish the SD-card image is copied to eMMC, then remove the SD-card and reboot.

Boot from SATA:

This is possible only for the boards which has SPI Flash where uboot to be hold, as A20 can't boot from SATA directly.

When you boot from SD-card you run:

```
$ olinuxino-sd-to-sata
```

This script will write uboot to SPI Flash and Linux file system to SATA drive.

Linux overlays:

To enable the different interfaces use:

```
$ olinuxino-overlay
```

This will give you access to all overlays for the board.

Note that some interfaces are multiplexed with other functions and if you enable may distrub other functionality.

All overlays do not fit on single screen so you have to scroll down to see all options.

Display overlays:

You can use olinuxino-display script to select your default LCD display.

```
$ olinuxino-display
```

Installing C:

The minimal Linux image comes with no C compiler installed so you need to install yourself:

```
$ sudo apt install gcc
```

then you can compile your c code with

```
$ gcc test.c -o program
```

and allow program to be executed:

```
$ chmod +777 program
```

then run it with

```
$ ./program
```

Installing pyA20Lime2:

First you need to install Python

```
$ sudo apt install python3
$ sudo apt install python3-venv python3-pip
$ python3 -m pip install --upgrade pip setuptools wheel
$ sudo su
# pip3 install pyA20Lime2
```

Now you have access to GPIO, SPI, I2C and Serial ports of A20-OLinuXino-LIME2 via Python package pyA20Lime2.

Software access to board resources:

GPIO in command line:

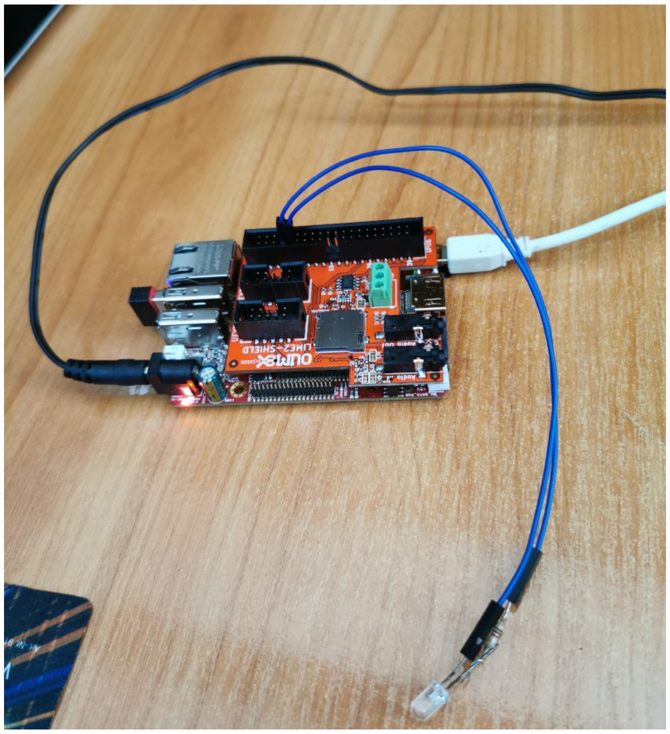
GPIOs are digital inputs or outputs which are available for the user. By default none of them is enabled. You can see the available GPIOs with the command:

```
$ls sys/class/gpio
```

to enable GPIO you have to export it and to assign it as input or output

Let for example attach LED to GPIO5 connector pin.7 A20 port PI15 and GPIO5 connector pin.9 (GND). We need to connect negative (cathode) lead of the LED to GND (GPIO5.pin9) and the positive LED positive (anode) lead through 470 ohm resistor to A20.PI15 (GPIO5.pin7). The 470 ohm resistor is necessary to limit the current which will flow through LED.

Here is picture of our setup:



Now to light on the LED we have to export A20.PI15 then to make it output and write 1 in it which will bring 3.3V to the A20.PI15 pin which voltage will light the LED on.

To deal with the GPIOs we need to have superuser rights, so we start with:

```
$ sudo su
```

we will be asked for the password, then the prompt will change to #

```
# echo 271 > /sys/class/gpio/export
```

now we can check if gpio271 is already available with

```
# ls sys/class/gpio
```

and we see the gpio271 is already listed

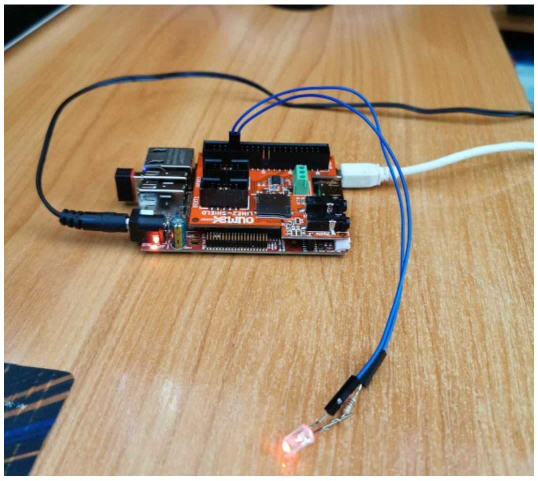
now we have to define it as output:

```
# echo out > /sys/class/gpio/gpio271/direction
```

now we have GPIO271 defined as output we can change it state with

```
# echo 1 > /sys/class/gpio/gpio271/value
```

And the LED is lighting:



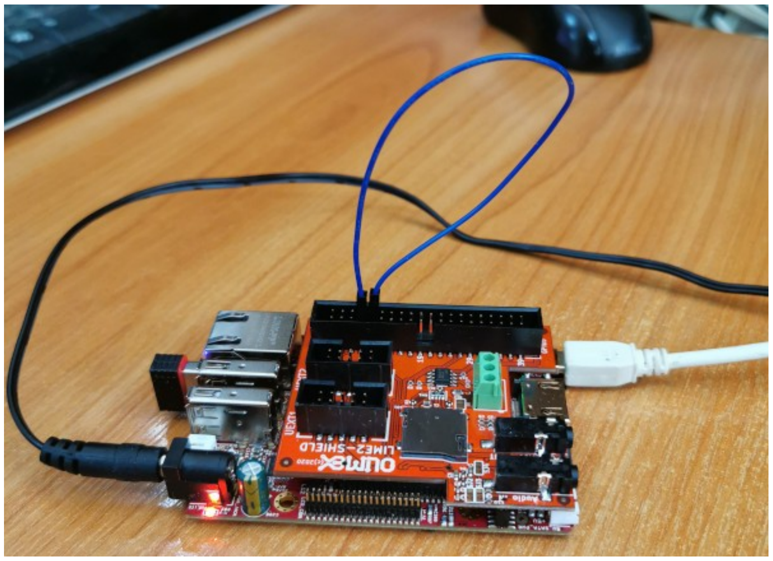
to turn if off we type

```
# echo 0 > /sys/class/gpio/gpio271/value
```

Let now make GPIO271 as input

```
# echo in > /sys/class/gpio/gpio271/direction
```

let connect A20.PI15 GPIO5.pin7 to GND GPIO5.pin9 :



now we check the value with:

```
#cat /sys/class/gpio/gpio271/value
```

we see 0. If we connect A20.PI15 GPIO5.pin7 to 3.3V GPIO5.pin17 and execute again cat command we see 1

Where the number 271 comes from? It’s Linux numbering the number corresponding to the port is calculated as PA starts from 0, PB starts from 32,... PI is staring from 8*32 = 256 so PI15 is 256+15 = 271

GPIO with C code:

Here below is example how to drive GPIOs in C code. The sources are available on GitHub.

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

int main() {

char s_out[] = "out";
char s_gpio[] = "271";
char s_0[] = "0";
char s_1[] = "1";
int fd;

//export GPIO271

if((fd=open("/sys/class/gpio/export",O_WRONLY))<0) {
   printf("can't open export GPIO\r\n");
   return(1);
} else {
   if(write(fd,s_gpio,strlen(s_gpio))<0) {
      printf("can't write to export GPIO\r\n");
      return(1);
   }
   printf("GPIO271 is exported\r\n");
   if(close(fd)<0) {
      printf("can't close export GPIO\r\n");
      return(1);
   }
}

//set as output GPIO271

if((fd=open("/sys/class/gpio/gpio271/
direction",O_WRONLY))<0) {
   printf("can't open GPIO direction\r\n");
   return(1);
} else {
   if(write(fd,s_out,strlen(s_out))<0) {
      printf("can't write GPIO direction\r\n");
      return(1);
   }
   printf("GPIO271 is set as output\r\n");
   if(close(fd)<0) {
      printf("can't close GPIO direction\r\n");
      return(1);
   }
}

//toggle GPIO271

if((fd=open("/sys/class/gpio/gpio271/value",O_WRONLY))<0) {
   printf("can't open GPIO value\r\n");
   return(1);
} else {

   while(1) {

      if(write(fd,s_1,strlen(s_1))<0) {
         printf("can't write GPIO value\r\n");
         return(1);
      }

      if(write(fd,s_0,strlen(s_0))<0) {
         printf("can't write GPIO value\r\n");
         return(1);
      }
   };

   if(close(fd)<0)
      return(1);
}

}
```

GPIO with Python:

Let use same setup LED connected to GPIO5 connector pin.7 A20 port PI15 and GPIO5 connector pin.9 (GND). We need to connect negative (cathode) lead of the LED to GND (GPIO5.pin9) and the positive LED positive (anode) lead through 470 ohm resistor to A20.PI15 (GPIO5.pin7). The 470 ohm resistor is necessary to limit the current which will flow through LED.

```
#!/usr/bin/env python
from pyA20Lime2.gpio import gpio
from pyA20Lime2.gpio import port
from pyA20Lime2.gpio import connector

gpio.init() #Initialize module. Always called first

gpio.setcfg(port.PI15, gpio.OUTPUT)  #Configure LED1 as 
output
gpio.setcfg(port.PI15, 1)            #This is the same as 
above

gpio.output(port.PI15, gpio.HIGH)    #Set PI15 High (3.3V) 
and Light on LED
gpio.output(port.PI15, 1)            #same as above

gpio.output(port.PI15, gpio.LOW)     #Set PI15 Low (0V) 
Light off
gpio.output(port.PI15, 0)            #same as above

gpio.setcfg(port.PI15, gpio.INPUT)   #Configure PI15 as 
input
gpio.setcfg(port.PI15, 0)            #Same as above

gpio.pullup(port.PI15, 0)            #Clear pullups
gpio.pullup(port.PI15, gpio.PULLDOWN) #Enable pull-down
gpio.pullup(port.PI15, gpio.PULLUP)   #Enable pull-up

if gpio.input(port.PI15) == 1:
    # this will be executed when input is HIGH i.e. 3.3V
else:
    # this will be executed when input is LOW i.e. 0V
```

I2C in command line

You can detect where your device is connected by using i2cdetect command. For instance if you have connected MOD-IO to UEXT1 connector and run i2cdetect you will see it’s address 0x58 active:

```
$ sudo i2cdetect -r -y 2
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- 58 -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

```
Then you can drive relays on and off:
```

```
$ sudo i2cset -y 2 0x58 0x10 0x01
```

this will swtich on RELAY1, to switch it off:

```
$ sudo i2cset -y 2 0x58 0x10 0x00
```

if you want to read MOD-IO status you can use i2transfer:

```
$ sudo i2ctransfer -y 2 w1@0x58 0x20 r1
0x00
```

Now connect wire from IN1 left side to GND (for instance AIN-2 connector has AGND) and right side to +3.3-12V for instance AIN-2 connector has 3.3V)

if you run again i2ctransfer command the returned value will be 0x01:

```
$ sudo i2ctransfer -y 2 w1@0x58 0x20 r1
0x01
```

I2C in C code

Here is example how to use I2C with C. The source is available on GitHub.

main.c

```
#include <stdio.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <string.h>
#include "i2c.h"

int main(int argc, char **argv)
{

//switch on MOD-IO relay 1

   int file;
   unsigned char buffer[2], address;

   address = 0x58;
   buffer[0] = 0x10;
   buffer[1] = 0x01;
   I2C_Open(&file, address);
   I2C_Send(&file, buffer, 2);
   I2C_Close(&file);

   return 0;
}
```

i2c.c

```
 #include <stdio.h>
 #include <linux/i2c.h>
 #include <linux/i2c-dev.h>
 #include <fcntl.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <sys/ioctl.h>
```

```
#include <string.h>

 void I2C_Open(int *file, unsigned char address)
{
 *file = (int)open("/dev/i2c-2", O_RDWR);
 if(*file < 0)
 {
   perror("Failed to open I2C");
   exit(1);
 }
 else
 {
   if(ioctl(*file, I2C_SLAVE, address) < 0)
   {
     perror("Failed to access I2C bus");
     exit(1);
   }
 }
}

void I2C_Close(int *file)
{
  close(*file);
}

void I2C_Send(int *file, char *buffer, int num)
{
 int bytes;
 bytes = write(*file, buffer, num);
 if(bytes != num)
 {
   perror("Failed to send data");
   exit(1);
 }
}

void I2C_Read(int *file, unsigned char *buffer, int num)
{
 int bytes;
 bytes = read(*file, buffer, num);
```

```
  if(bytes != num)
  {
    perror("Failed to read data");
    exit(1);
  }
}
```

i2c.h

```
#ifndef I2C_H
#define I2C_H

void I2C_Open(int *file, unsigned char address);
void I2C_Send(int *file, unsigned char *buffer, int num);
void I2C_Read(int *file, unsigned char *buffer, int num);
void I2C_Close(int *file);

#endif
```

makefile

```
CC = gcc
CFLAG = -c -Wall

all: i2c-demo

i2c-demo: i2c.o main.o
	$(CC) i2c.o main.o -o mod-io

main.o: main.c
	$(CC) $(CFLAG) main.c

i2c.o: i2c.c
	$(CC) $(CFLAG) i2c.c

clean:
	rm -rf *.o i2c-demo
```

I2C in Python

Next code switch on and off MOD-IO relays. MOD-IO should be connected to UEXT1. The code is available on GitHub.

```
from pyA20Lime2 import i2c
i2c.init("/dev/i2c-2")

i2c.open(0x58)
i2c.write([0x10, 0x01]) # switich ON relay1
i2c.close()

i2c.open(0x58)
i2c.write([0x10, 0x00]) # switich OFF relay1
i2c.close()

i2c.open(0x58)
i2c.write([0x20])       # read IN1..4 state
value = i2c.read(1)
i2c.close()
```

CAN in command line

LIME2-SHIELD has CAN driver so you can connect to CAN bus.

```
$ ifconfig -a
can0: flags=128<NOARP>  mtu 16
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-
00-00  txqueuelen 10  )
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0
collisions 0
        device interrupt 59
```

To use CAN interface you can install can-utils and setup the CAN interface:

```
$ sudo apt-get install can-utils
$ ip link set can0 down
$ ip link set can0 type can bitrate 100000 triple-sampling
on loopback off
$ ip link set can0 up
```

Now conect A20-CAN to the CAN network two wire interface.

To send a packet over CAN use :

```
cansend <can_interface> <packet>
```

```
For instance:

$ cansend can0 5AA#10.10.10
```

To sniff for CAN network messages you can use candump :

```
$ candump can0
```

Now you can log your car CAN networking messages and interpret them. There is plenty of info on the web about the different CAN messages which are exchanged on car CAN bus.

CAN in C code

Kernel code documentation https://www.kernel.org/doc/Documentation/networking/can.txt

C code Examples from GitHub

cansend.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

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

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

int main(int argc, char **argv)
{
        int s;
        struct sockaddr_can addr;
        struct ifreq ifr;
        struct can_frame frame;

        printf("CAN Sockets Demo\r\n");

        if ((s = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0) {
                perror("Socket");
                return 1;
        }

        strcpy(ifr.ifr_name, "can0" );
        ioctl(s, SIOCGIFINDEX, &ifr);

        memset(&addr, 0, sizeof(addr));
        addr.can_family = AF_CAN;
        addr.can_ifindex = ifr.ifr_ifindex;

        if (bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
                perror("Bind");
                return 1;
        }
        frame.can_id = 0x555;
        frame.can_dlc = 5;
        sprintf(frame.data, "Hello");

        if (write(s, &frame, sizeof(struct can_frame)) !=
sizeof(struct can_frame)) {
                perror("Write");
                return 1;
        }

        if (close(s) < 0) {
                perror("Close");
                return 1;
        }

        return 0;
}
```

canreceive.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

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

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

int main(int argc, char **argv)
{
        int s, i;
        int nbytes;
        struct sockaddr_can addr;
        struct ifreq ifr;
        struct can_frame frame;

        printf("CAN Sockets Receive Demo\r\n");

        if ((s = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0) {
                perror("Socket");
                return 1;
        }

        strcpy(ifr.ifr_name, "can0" );
        ioctl(s, SIOCGIFINDEX, &ifr);
        memset(&addr, 0, sizeof(addr));
        addr.can_family = AF_CAN;
        addr.can_ifindex = ifr.ifr_ifindex;

        if (bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
                perror("Bind");
                return 1;
        }

        nbytes = read(s, &frame, sizeof(struct can_frame));

        if (nbytes < 0) {
                perror("Read");
                return 1;
        }

        printf("0x%03X [%d] ",frame.can_id, frame.can_dlc);

        for (i = 0; i < frame.can_dlc; i++)
                printf("%02X ",frame.data[i]);

        printf("\r\n");

        if (close(s) < 0) {
                perror("Close");
                return 1;
        }

        return 0;
}
```

SPI in Python

SPI communication in Python:

```
#!/usr/bin/env python

from pyA20Lime2 import spi

spi.open("/dev/spidev2.0")
#Open SPI device with default settings
# mode : 0
# speed : 100000kHz
# delay : 0
# bits-per-word: 8

#Different ways to open device
spi.open("/dev/spidev2.0", mode=1)
spi.open("/dev/spidev2.0", mode=2, delay=0)
spi.open("/dev/spidev2.0", mode=3, delay=0,
bits_per_word=8)
spi.open("/dev/spidev2.0", mode=0, delay=0,
bits_per_word=8, speed=100000)

spi.write([0x01, 0x02]) #Write 2 bytes to slave device
spi.read(2) #Read 2 bytes from slave device
spi.xfer([0x01, 0x02], 2)   #Write 2 byte and then read 2
bytes.

spi.close() #Close SPI bus
```

Revision History

Revision 1.0 June 2020