OpenIO Labs - ScriptML

Writing I2C Drivers in C and Python

UG403 (v2017.1) April 3, 2017

Revision History

Date	Version	Revisions
3/4/2017	2017.1	*Initial release.*

Table of Contents

Chapter 1: Introduction

One of the significant aspects of the OpenIO Labs system, is its ability to directly interact with hardware at the very edge of the network. The types of interfaces that the OpenIO Labs supports for the hardware interactions is varied and in this guide we will focus specifically on the I2C interface. A general overview of the OpenIO Labs system is presented in Ref [1]

The I2C interface is described in more detail in REF [2] and its associated references. It is a simple bi-directional serial bus interface between a master and a slave. In the OpenIO Labs system, the Input Output Interface (IOI) device is the master, and, in general, one or more Integrated Circuits (ICs) are the slaves.

The I2C bus interface uses hardware addresses to communicate between different devices attached to the I2C bus. Different classes of devices are allocated part of an address within the design of the IC with the other parts of the address defined by tying pins of the IC to either a logic high level or a logic low level. Typically, the I2C address is 7 bits wide, so many different devices can be connected to the same bus.

Most embedded platforms include some form of I2C interface connectivity and the OpenIO Labs IOIs are no exception. The programming interface to access and control the I2C devices attached to the bus are well defined. If we take, as an example, the Linux operating system, I2C support has been built directly into the Linux kernel. In addition, virtually every device that has been designed to use the I2C bus has a software driver that has been specifically written to support that device, or family of devices.

The writing of I2C device drivers for an experienced programmer is a relatively straightforward task that either involves adding the driver to the Linux kernel, or adapting an existing driver to the Operating System (OS) of choice. To an inexperienced programmer, however, this can be a slightly daunting task.

A central feature of the OpenIO Labs system is the definition and creation of ScriptML. An overview of ScriptML and examples of writing scripts that use ScriptML is presented in Ref [3, 4]. In brief, ScriptML within the OpenIO Labs system, allows the user to write applications that run on the edge devices (IOIs) in a range of languages, such as C, Python, Java etc. The source language is converted into ScriptML tokens, that are sent to the edge device for processing.

In addition to allowing simple application scripts, the ScriptML system also provides direct access to the underlying hardware (such as the I2C bus) from the ScriptML based scripts, by using a simple API.

This User Guide was written to make the task of writing I2C drivers easier to understand and to implement. As most existing I2C bus drivers are written in C, we will start by taking an existing

driver and show the steps that need to be taken to adapt that to the OpenIO Labs system using ScriptML. Although functionally complete, this driver may still be regarded as a relatively sophisticated piece of code. To this end, ScriptML also includes a compact method of accessing I2C drivers by generalising the I2C bus access into three simple functions:

1. Create a device handler (a piece of code that registers the presence of the I2C device and subsequently allows read / write access).
2. Write function that allows the ScriptML script (written in e.g. C or Python) to write to one or more registers.
3. Read function that allows the ScriptML script (written in e.g. C or Python) to read from one or more registers.

With these three simple APIs a user can create a very simple driver that can be adapted to read and write to virtually every I2C device.

In this User Guide, we will start by passing through the steps required to adapt an existing I2C driver written in C to the ScriptML version. We will use an I2C bus EEPROM as an example for this activity. Next we will demonstrate how the ScriptML compact library can be used to create a similar driver, but with much less complexity. Finally, we will then look at how we can create a compact Python I2C driver for our EEPROM device. We do not explore the adaptation of existing Python drivers, as the existing code for drivers written in Python is less common.

Chapter 2: I2C C Driver Creation

We will start this chapter by taking an existing C driver for Linux and going through the stages of modification required to adapt the driver for use in ScriptML.

In pretty much all drivers that can be found on the Internet for devices such as I2C, the driver comprises a number of elements such as:

- Support for multiple devices and device types
- Support to allow the driver to be embedded into an existing OS such as the Linux Kernel
- Code to support all of the functions required for the driver (and in many cases multiple versions to support slight differences in the device).

To start this section, therefore, we will take elements of an existing driver and highlight the parts that we will work with, and discard the parts that are either unused, or irrelevant. For this example we will be writing a driver for a 24LC32A 32 kbit EEPROM from Microchip (See Ref [5]).

When modifying an existing driver, the best place to start is by searching the Internet for an existing example that can be modified. Typically this can be done by typing a phrase such as '24LC32A Linux driver' into a search engine and review what is returned.

In this example, we will be using an existing Linux driver found in this manner.

Existing Driver for 24LC32A

An existing driver for the 24LC32A can be found in Ref [6]. This driver will support a whole range of different devices using a similar structure. The original driver was written for the AT24 EEPROM family from Atmel, but the driver is also suitable for a range of other devices such as 24LC32A. In the example that we will use here, this driver was written to be embedded within the Linux Kernel. As a consequence the driver is quite a long piece of code, at the time of writing, over 800 lines. We will see, however, that only a few small parts of the code are required to be modified, and it is this process that we will be exploring in this chapter.

At first glance, the driver written to support this type of EEPROM is quite long and complex, however, if we delve into the driver source code, we will see that there are a number of aspects that are being implemented. Many functions to support the driver being inserted into the Linux kernel (such as the Probe functions) and also a range of different methods to read and write to the device.

For the purposes of this ScriptML driver porting we can start by reviewing the code between lines 685 and 700 (in the current version as of the time of writing). We will find the following section of driver source code.

```
if (chip.flags & AT24_FLAG_SERIAL) {
      at24->read_func = at24_eeprom_read_serial;
} else if (chip.flags & AT24_FLAG_MAC) {
      at24->read_func = at24_eeprom_read_mac;
} else {
      at24->read_func = at24->use_smbus ? at24_eeprom_read_smbus
                                 : at24_eeprom_read_i2c;
}

if (at24->use_smbus) {
      if (at24->use_smbus_write == I2C_SMBUS_I2C_BLOCK_DATA)
            at24->write_func = at24_eeprom_write_smbus_block;
      else
            at24->write_func = at24_eeprom_write_smbus_byte;
} else {
      at24->write_func = at24_eeprom_write_i2c;
}
```

In this section, the code is assigning the read and write functions based on the type of device that we are managing. By reviewing the details of the configuration of the device driver based on the device type we will see that the relevant read and write functions are:

```
at24_eeprom_read_i2c
at24_eeprom_write_i2c
```

It is these two functions that will be the basis for the ScriptML driver that we will create.

Original I2C C Write Function - at24_eeprom_write_i2c

We will start with the I2C C write function. The complete original function that we will be modifying is presented below.

```
static ssize_t at24_eeprom_write_i2c(struct at24_data *at24, const char *buf,
                              unsigned int offset, size_t count)
{
      unsigned long timeout, write_time;
      struct i2c_client *client;
      struct i2c_msg msg;
      ssize_t status = 0;
      int i = 0;

      client = at24_translate_offset(at24, &offset);
      count = at24_adjust_write_count(at24, offset, count);

      msg.addr = client->addr;
```

```
        msg.flags = 0;

        /* msg.buf is u8 and casts will mask the values */
        msg.buf = at24->writebuf;
        if (at24->chip.flags & AT24_FLAG_ADDR16)
                msg.buf[i++] = offset >> 8;

        msg.buf[i++] = offset;
        memcpy(&msg.buf[i], buf, count);
        msg.len = i + count;

        loop_until_timeout(timeout, write_time) {
                status = i2c_transfer(client->adapter, &msg, 1);
                if (status == 1)
                        status = count;

                dev_dbg(&client->dev, "write %zu@%d --> %zd (%ld)\n",
                                count, offset, status, jiffies);

                if (status == count)
                        return count;
        }

        return -ETIMEDOUT;
}
```

If we briefly review this source segment, we will start to get a feel for the changes that we need to make. Briefly these changes are as follows:

1. Define a C structure `at24_data` We will use elements of the structure as defined in the original driver, but a number of elements are not required and will not be incorporated.
2. The function `at24_translate_offset` is used to create the client and manage the page offsets that may be required when multiple devices are used in parallel. We will define a slightly different function for this purpose.
3. The function `at24_adjust_write_count` is similarly defined to manage the different memory structures for single and multiple devices and the page over-write issue. We will modify this to assume that we are only using a single device and hence no offset is used.
4. The remainder of the function manages the different address sizes, the main write part and also time-outs. Again we will adapt these to a simpler implementation.

C struct at24_data

If we examine the C structure in the original file, there are a number of elements that are not required. These elements include pointers to the read/write functions, mutex locks and the storage

for the different sets of devices supported. The elements that we will retain are presented below.

```
/* Reduced structure from the original.
 * The writebuf is replaced with an explicit array as the original
 * used a kernel malloc. We know that the maximum we can write is
 * fixed by the EEPROM page size which we have #defined here.  */
struct at24_data {
      uint8_t *writebuf;
      unsigned write_max;
      struct i2c_client client;
};
```

Within the structure, the elements that we will keep are the write buffer, a variable that defines the maximum number of write elements per write and a structure that is used to define the details of the specific I2C device that we will be utilising., which we have below.

```
/* Reduced version of the Linux I2C client structure, taking only the
 * parts that we need here */
struct i2c_client {
      int addr; /* chip address - NOTE: 7bit    */
      sml_i2c_t *adapter; /* Modification to match SML syntax */
};
```

Similarly we have trimmed this structure, for the i2c_client struct we only need storage for the address and also the adapter structure. The adapter is used to define the device address. In this instance we have changed the structure type from the i2c_adapter to sml_i2c_t which is a ScriptML structure that links to the underlying I2C device drivers in the IOI that we will be using. When we go through a completed example, we will explore this in more detail.

struct i2c_msg

Next (from line 6 of the write driver) we need to define the structure that we will need to transport the I2C message. In this case, we will retain the same structure that is defined in the original driver and presented below.

```
/* This is the same structure as used in the Linux I2C driver.
 * The inline #defines are left where they are, but could be put somewhere
 * neater if desired */
struct i2c_msg {
    uint16_t  addr;     /* slave address                        */
    uint16_t  flags;
#define I2C_M_RD                0x0001  /* read data, from slave to master */
                                        /* I2C_M_RD is guaranteed to be 0x0001!
*/
```

```
#define I2C_M_TEN               0x0010  /* this is a ten bit chip address */
#define I2C_M_RECV_LEN          0x0400  /* length will be first received byte
*/
#define I2C_M_NO_RD_ACK         0x0800  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_IGNORE_NAK        0x1000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_REV_DIR_ADDR      0x2000  /* if I2C_FUNC_PROTOCOL_MANGLING */
#define I2C_M_NOSTART           0x4000  /* if I2C_FUNC_NOSTART */
#define I2C_M_STOP              0x8000  /* if I2C_FUNC_PROTOCOL_MANGLING */
    uint16_t  len;              /* msg length                               */
    uint8_t  *buf;              /* pointer to msg data                      */
};
```

Modified I2C C Write Function - at24_eeprom_write_i2c

The resulting modifications are presented in the code sample below. The comments include additional details on the fields and reasons for any specific changes.

```
/* So, the modified write function will look like this. Some slight type changes
 * may have been made. Also the comments have been added to improve readability */
int at24_eeprom_write_i2c(struct at24_data *at24, const char *buf,
                unsigned int offset, size_t count) {
        struct i2c_msg msg[1];
        int status = 0;
        int i = 0;

        /* Trim the write count to fit within a page write limit */
        count = at24_adjust_write_count(at24, offset, count);

        /* Set the I2C address msg field for the device */
        msg[0].addr = at24->client.addr;

        /* Set the access flags, for Write this is set to zero */
        msg[0].flags = 0;

        /* Create memory for the write buffer. Add two bytes for the EEPROM address */
        at24->writebuf = (uint8_t *) malloc(count + 2);

        /* Assign msg.buf to the write buffer that we will use to transfer the data to write */
        msg[0].buf = at24->writebuf;

        /* The device we are using requires 2 bytes for the EEPROM address field */
        msg[0].buf[i++] = offset >> 8;
        msg[0].buf[i++] = offset;

        /* Copy the message to the buffer */
        memcpy(at24->writebuf + i, buf, count);

        /* Set the message length. This will include the address field (2 Bytes) */
        msg[0].len = i + count;

        /* Now send this to the EEPROM via the I2C bus */
        sml_i2c_transfer(*(at24->client.adapter), msg, 1);

        /* Wait for the write */
```

```
        sleep(1);

        /* Free the memory allocated */
        free(at24->writebuf);

        return 0;
}
```

Original I2C C Read Function - at24_eeprom_read_i2c

Having considered the write function, we can now consider the read function. We will follow a similar presentation style as previously, but will not repeat an items that have already been considered.

The code from the original driver is shown below.

```
static ssize_t at24_eeprom_read_i2c(struct at24_data *at24, char *buf,
                                    unsigned int offset, size_t count)
{
        unsigned long timeout, read_time;
        struct i2c_client *client;
        struct i2c_msg msg[2];
        int status, i;
        u8 msgbuf[2];

        memset(msg, 0, sizeof(msg));
        client = at24_translate_offset(at24, &offset);

        if (count > io_limit)
                count = io_limit;

        /*
         * When we have a better choice than SMBus calls, use a combined I2C
         * message. Write address; then read up to io_limit data bytes. Note
         * that read page rollover helps us here (unlike writes). msgbuf is
         * u8 and will cast to our needs.
         */
        i = 0;
        if (at24->chip.flags & AT24_FLAG_ADDR16)
                msgbuf[i++] = offset >> 8;
        msgbuf[i++] = offset;

        msg[0].addr = client->addr;
        msg[0].buf = msgbuf;
        msg[0].len = i;
```

```
        msg[1].addr = client->addr;
        msg[1].flags = I2C_M_RD;
        msg[1].buf = buf;
        msg[1].len = count;

        loop_until_timeout(timeout, read_time) {
                status = i2c_transfer(client->adapter, msg, 2);
                if (status == 2)
                        status = count;

                dev_dbg(&client->dev, "read %zu@%d --> %d (%ld)\n",
                                count, offset, status, jiffies);

                if (status == count)
                        return count;
        }

        return -ETIMEDOUT;
}
```

Modified I2C C Read Function - at24_eeprom_read_i2c

The modified code for the driver is shown below. In comparing the two, we can see that the modifications made for the read case are similar to the write case. In the read driver, we must first write the address to read to the EEPROM device, and then in a second message, we will read back the contents of the EEPROM that we have requested.

```
/* The modified read file. Very similar to the original, but with a few changes and comments.
 changes mainly to map to the ScriptML I2C driver */
int at24_eeprom_read_i2c(struct at24_data *at24, char *buf, unsigned int offset,
                size_t count) {

        /* Define some variables and structures to retain the data and configuration */
        struct i2c_msg msg[2];
        int status, i;
        uint8_t msgbuf[2];

        /* Set the read limit to reduce load on the I2C bus */
        static unsigned io_limit = IO_LIMIT;

        if (count > io_limit)
                count = io_limit;

        i = 0;

        /* Set the read address, which for this device is two bytes */
        msgbuf[i++] = offset >> 8;
        msgbuf[i++] = offset;

        /* Read address must be written first, then followed by the read */
        msg[0].addr = at24->client.addr;
```

```
        msg[0].buf = msgbuf;
        msg[0].len = i;

        /* Read the data from offset */
        msg[1].addr = at24->client.addr;
        msg[1].flags = I2C_M_RD;
        msg[1].buf = buf;
        msg[1].len = count;

        /* Request the ScriptML I2C driver to perform the read and return the data */
        sml_i2c_transfer(*(at24->client.adapter), msg, 2);

        return 0;
}
```

Additional Functions

Two additional functions are also used in the main application. These are `at24_adjust_write_count` and `at24_eeprom_create`. The first is a slight modification of the similar function in the original driver. The second is in place of the probe function in the Linux kernel driver and which instantiates the EEPROM driver.

The details for these two functions along with comments in the source code can be found in the appendix.

Chapter 3: Compact I2C C Driver

In the previous section we have seen how to take an existing I2C driver and adapt it to work with the OpenIO Labs system using ScriptML. The intention of that section was to guide the user through the driver creation process based on existing code. The driver that we created is fully functional.

In many cases, the I2C drivers that are needed are simply either sending or receiving a sequence of bytes from the I2C device. In these instances ScriptML provides a simple method for writing a driver to access these devices.

In this chapter, we will review the steps and the code needed to access the I2C device. As before, we will use the 24LC32A EEPROM device as the example.

myCompactDriver.c

The complete listing for the example driver is presented in the Appendix, here we will consider the key elements.

To ease in the burden of writing the I2C drivers in C, a helper library has been created and accessed using the i2c_regmap.h header file. This helper library manages most of the detail required to create an i2c device handler, write to the I2C device and then read from the I2C device.

If you review the two driver examples in the Appendix, you will see that the ported driver is approximately 200 lines long (with comments) whereas the compact driver is around 60 lines long. The savings are facilitated by this helper library.

To review the code, there are now three main functions that we should consider:

1. Create the device handler
2. Write to the device
3. Read from the device

We will look at each of those in turn

Create I2C Device Handler

The function to create the I2C device handler is shown below.

```
sml_i2c_regmap_t at24  = sml_i2c_regmap_create( sml_i2c_open( "/dev/i2c-1" ), BASE_ADDR |
addr_offset, MAX_REG_SIZE );
```

The functions of this are:

- Create the device
- Specify the physical hardware address
- Define the maximum register size for the I2C device.

Withthis function, we open the I2C device using its location within the system

Write to Device

Similar to the create function, the write function takes the device handler, a buffer for the write data, the register address offset, the address size in bytes and the number of bytes to write. This is seen below.

```
sml_i2c_regmap_write( at24, buf, &offset, ADDR_SIZE, count );
```

Read from Device

Similar to the write function, the read function takes the device handler, a buffer for the read data, the register address offset, the address size in bytes and the number of bytes to read. This is seen below.

```
sml_i2c_regmap_read( at24, rd_buf, &offset, ADDR_SIZE, count);
```

Chapter 4: Compact I2C Python Driver

If we now turn to the case of writing a Python driver. In this case, we will go straight to the compact driver using the equivalent to the I2C_regmap C library, but implemented in Python.

The Python I2C-regmap package implements all of the necessary functions that will allow the user to create drivers that read and write to multiple registers, either a register at a time, or as a sequence.

Similar to the C case, the EEPROM will only allow the writing of a single page at a time (32 bytes in the case of the 24LC32A

```
import i2c
import i2c_regmap

_24LC32_I2C_ADDR = 0x50 # I2C hardware address for the EEPROM
MAX_I2C_REGISTERS = 4096 # Maximum number of registers inthe EEPROM (32kbit)
REG_START_ADDR = 0 # Start address for the read and write operations
BYTES_TO_WRITE = 32 # Number of bytes to write
BYTES_TO_READ = 32 # Number of bytes to read

# Create the EEPROM device handler with the I2C device location, EEPROM I2C
address and maximum number of registers
eeprom = i2c_regmap.regmap( i2c.device("/dev/i2c-1"), _24LC32_I2C_ADDR,
size=MAX_I2C_REGISTERS )

# Define some test data for the write
data =
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,2
9,30,31]

# Write the test data to the EEPROM
eeprom.write(data, REG_START_ADDR, BYTES_TO_WRITE)

# Read the data back from the EEPROM
read_data = eeprom.read(REG_START_ADDR, BYTES_TO_READ)

# Print the read data
print(read_data)
```

Examining the structure of the Python program, apart from the definition of a few constants and test data, there are three main functions that are used as follows:

1. The creation of the EEPROM device in which the software address to the device is defined,

the hardware address of the I2C device and also the maximum number of registers that the device supports is also defined. It should be noted that the setting of this value is important and it is used to define the length (in bytes) of the register address within the I2C device. An incorrect size can set the incorrect register address length and hence case an error when reading and writing to the device. As an example, the 24LC32 EEPROM has 4096 registers which requires an address of 12 bits and hence two bytes for the register address.

2. The next main function is the write function. This uses three parameters as follows:
    1. data – an array of data that we want to write to the EEPROM (note that the write must be on the same EEPROM page due to device hardware limitations.
    2. REG_START_ADDR – this is the start register address that we want to write to.
    3. BYTES_TO_WRITE – this is the number of bytes to write to the EEPROM. This must be less than or equal to the EEPROM page size (32 bytes for the 24LC32) and also needs to account for the register start address, i.e. we should not write across a page boundary, even if the number of bytes is less than or equal to 32. In addition to being able to write to a full page, we can also write to a single register, in which case we would set BYTES_TO_WRITE to 1.
3. The final function is the read function. This uses the following parameters:
    1. REG_START_ADDR – the register start address from where we want to start reading the EEPROM.
    2. BYTES_TO_READ – the number of bytes that we want to read. Unlike the write operation, the read function can read across page boundaries and can also read multiple pages. Equally, we can also read a single register. In practice, it is recommended that we restrict the length of an I2C read to a few pages as a time, as the use of long reads can block the access to the I2C bus by other devices attached to the bus.

Appendix

Complete Code Listing for myFirstDriver.c

The complete code listing for the myFirstDriver.c is presented below. The driver will pull in some ScriptML header files that are not shown here.

```
#include <scriptml/i2c.h>
#include <stdint.h>
#include <test_print.h>
#include "stdio.h"
#include "string.h"

#define BASE_ADDR 0x50
#define MAX_PAGE_SIZE 32 /* Maximum page size and hence max bytes per write */
#define IO_LIMIT 128 /* Limit to reads to prevent excessive blocking of I2C bus */

/* Reduced version of the Linux I2C client structure, taking only the
 * parts that we need here */
struct i2c_client {
        int addr; /* chip address - NOTE: 7bit    */
        sml_i2c_t *adapter; /* Modification to match SML syntax */
};

/* Reduced structure from the original.
 * The writebuf is a pointer to memory that Malloc will allocate later.
 * We know that the maximum we can write is fixed by the EEPROM page
 *size which we have #defined here.  */
struct at24_data {
        uint8_t *writebuf;
        unsigned write_max;
        struct i2c_client client;
};

/* New function to create instance of eeprom device.
 * This is in-place of the linux function at24_probe which would
 * perform a lot of the instantiation when the driver is loaded
 * into the Kernel */
struct at24_data at24_eeprom_create(sml_i2c_t i2c, int addr_offset) {
        struct at24_data at24;

        /* Save the I2C device handle */
        at24.client.adapter = &i2c;

        /*  Set the EEPROM I2C address and any offset (hardware defined */
        at24.client.addr = BASE_ADDR | addr_offset;

        /* Set the maximum page write size */
        at24.write_max = MAX_PAGE_SIZE; /* Maximum number of bytes to write based on page size */

        return at24;
}

/* This function manages the count variable with respect to the
 * page address (offset) and the page length. */
size_t at24_adjust_write_count(struct at24_data *at24, unsigned int offset,
                size_t count) {
```

```
        unsigned write_max = at24->write_max;

        /* Check if we will write across a page and if so reduce count */
        if ((offset + count - 1) >= ((offset / write_max) + 1) * write_max) {

                count = ((offset / write_max) + 1) * write_max - offset;
        }

        return count;
}

/* So, the modified write function will look like this. Some slight type changes
 * may have been made. Also the comments have been added to improve readability */
int at24_eeprom_write_i2c(struct at24_data *at24, const char *buf,
                unsigned int offset, size_t count) {
        struct i2c_msg msg[1];
        int status = 0;
        int i = 0;

        /* Trim the write count to fit within a page write limit */
        count = at24_adjust_write_count(at24, offset, count);

        /* Set the I2C address msg field for the device */
        msg[0].addr = at24->client.addr;

        /* Set the access flags, for Write this is set to zero */
        msg[0].flags = 0;

        /* Create memory for the write buffer. Add two bytes for the EEPROM address */
        at24->writebuf = (uint8_t *) malloc(count + 2);

        /* Assign msg.buf to the write buffer that we will use to transfer the data to write */
        msg[0].buf = at24->writebuf;

        /* The device we are using requires 2 bytes for the EEPROM address field */
        msg[0].buf[i++] = offset >> 8;
        msg[0].buf[i++] = offset;

        /* Copy the message to the buffer */
        memcpy(at24->writebuf + i, buf, count);

        /* Set the message length. This will include the address field (2 Bytes) */
        msg[0].len = i + count;

        /* Now send this to the EEPROM via the I2C bus */
        sml_i2c_transfer(*(at24->client.adapter), msg, 1);

        /* Wait for the write */
        sleep(1);

        /* Free the memory allocated */
        free(at24->writebuf);

        return 0;
}

/* The modified read file. Very similar to the original, but with a few changes and comments.
 changes mainly to map to the ScriptML I2C driver */
int at24_eeprom_read_i2c(struct at24_data *at24, char *buf, unsigned int offset,
                size_t count) {

        /* Define some variables and structures to retain the data and configuration */
        struct i2c_msg msg[2];
```

```
        int status, i;
        uint8_t msgbuf[2];

        /* Set the read limit to reduce load on the I2C bus */
        static unsigned io_limit = IO_LIMIT;

        if (count > io_limit)
                count = io_limit;

        i = 0;

        /* Set the read address, which for this device is two bytes */
        msgbuf[i++] = offset >> 8;
        msgbuf[i++] = offset;

        /* Read address must be written first, then followed by the read */
        msg[0].addr = at24->client.addr;
        msg[0].buf = msgbuf;
        msg[0].len = i;

        /* Read the data from offset */
        msg[1].addr = at24->client.addr;
        msg[1].flags = I2C_M_RD;
        msg[1].buf = buf;
        msg[1].len = count;

        /* Request the ScriptML I2C driver to perform the read and return the data */
        sml_i2c_transfer(*(at24->client.adapter), msg, 2);

        return 0;
}

/* A small test application to first write count bytes to the EEPROM at some start
 address (offset), and then read count elements back. Note that the number of
 bytes read back will, in general, be more than the write, as the write
 will truncate at a page boundary, but read will read across the boundary. */
int main() {
        /* Define a device handle for the I2C device */
        sml_i2c_t *i2c;

        /* We will use the base I2C address for the EEPROM */
        int addr_offset = 0;

        /* Define the memory start location within the EEPROM */
        unsigned int offset = 0;

        /* Define how many bytes to send. This may be truncated if it extends over a page */
        int count = MAX_PAGE_SIZE;

        /* Define some test data to write to the EEPROM. Use a full page */
        char buf[MAX_PAGE_SIZE] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                        14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
                        31 };

        /* Buffer to store data read from the EEPROM */
        uint8_t rd_buf[IO_LIMIT];

        /* Open the I2C device and save the device handle */
        i2c = &(sml_i2c_open("/dev/i2c-1"));

        /* Create a device manager object to store local device parameters */
        struct at24_data at24 = at24_eeprom_create(*i2c, 0);

        /* Write to the EEPROM */
```

```
        int BytesSent = at24_eeprom_write_i2c(&at24, buf, offset, count);

        /* Wait for write to complete */
        sleep(1);

        /* Read the contents of the EEPROM. Need to allow for two bytes for the EEPROM start
address.
         Note if the count length goes over a page boundary, write will truncate, but read will
         not, so we will end up reading more elements than the write has written, in most cases. */
        at24_eeprom_read_i2c(&at24, rd_buf, offset, count + 2);

        /* Print the contents read from the EEPROM */
        int i;
        for (i = 0; i < count; i++) {
                printint(rd_buf[i]);
        }

        return 0;
}
```

Complete Code Listing for myCompactDriver.c

The complete code listing for the myCompactDriver.c is presented below. The driver will pull in some ScriptML header files that are not shown here.

```
#include <scriptml/i2c.h>
#include <stdint.h>
#include <test_print.h>
#include "stdio.h"
#include "string.h"
#include <assert.h>
#include "i2c_regmap2.h"

#define BASE_ADDR 0x50 /* This is the hardware address for the I2C device.
                          This can be a fixed number with some HW pins to change it. */
#define MAX_PAGE_SIZE 32 /* Maximum page size and hence max bytes per write */
#define IO_LIMIT 128 /* Limit to reads to prevent excessive blocking of I2C bus */
#define MAX_REG_SIZE 4096 /* This is a 32kbit device with one byte per register */
#define ADDR_SIZE 2 /* The number of bytes used to store the address */

int main(){


    /* We will use the base I2C address for the EEPROM */
    int addr_offset = 0;

    /* Define the memory start location within the EEPROM */
    unsigned int offset = 0;

    /* Define how many bytes to send. This may be truncated if it extends over a page */
    int count = MAX_PAGE_SIZE;

    /* Define some test data to write to the EEPROM. Use a full page */
    char buf[32] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                     14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
```

```
                    31 };

    /* Buffer to store data read from the EEPROM */
    uint8_t rd_buf[IO_LIMIT];

    /* Create a device manager object to store local device parameters */
    sml_i2c_regmap_t at24  = sml_i2c_regmap_create( sml_i2c_open( "/dev/i2c-1" ), BASE_ADDR |
addr_offset, MAX_REG_SIZE );

        /* Write to the EEPROM */
    sml_i2c_regmap_write( at24, buf, &offset, ADDR_SIZE, count );

    /* Wait for write to complete */
    sleep(1);

    /* Read the contents of the EEPROM. Need to allow for two bytes for the EEPROM start address.
       Note if the count length goes over a page boundary, write will truncate, but read will
       not, so we will end up reading more elements than the write has written, in most cases. */
    sml_i2c_regmap_read( at24, rd_buf, &offset, ADDR_SIZE, count);

    /* Print the contents read from the EEPROM */
    int i;
    for(i=0; i < count; i++){
        printint(rd_buf[i]);
    }

    return 0;
}
```

Complete Code Listing for myCompactDriver.py

The complete code listing for the myCompactDriver.py is presented below. The driver will pull in some ScriptML package files that are not shown here.

```
import i2c
import i2c_regmap

_24LC32_I2C_ADDR = 0x50 # I2C hardware address for the EEPROM
MAX_I2C_REGISTERS = 4096 # Maximum number of registers inthe EEPROM (32kbit)
REG_START_ADDR = 0 # Start address for the read and write operations
BYTES_TO_WRITE = 32 # Number of bytes to write
BYTES_TO_READ = 32 # Number of bytes to read

# Create the EEPROM device handler with the I2C device location, EEPROM I2C
address and maximum number of registers
eeprom = i2c_regmap.regmap( i2c.device("/dev/i2c-1"), _24LC32_I2C_ADDR,
size=MAX_I2C_REGISTERS )

# Define some test data for the write
data =
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,2
```

```
9,30,31]

# Write the test data to the EEPROM
eeprom.write(data, REG_START_ADDR, BYTES_TO_WRITE)

# Read the data back from the EEPROM
read_data = eeprom.read(REG_START_ADDR, BYTES_TO_READ)

# Print the read data
print(read_data)
```

Additional Resources

References

The following documents and references are cited within this guide:

1. UG101 – OpenIO Labs System Architecture
2. https://en.wikipedia.org/wiki/I2C
3. UG401 – ScriptML System Overview
4. UG402 – ScriptML Examples in C and Python
5. http://ww1.microchip.com/downloads/en/DeviceDoc/21713M.pdf
6. https://github.com/torvalds/linux/blob/master/drivers/misc/eeprom/at24.c