**Seven Segment Display (SSD):**

The SSD has seven segments, labeled A-F, which can be individually lit up to display digits/characters, where each segment has its own corresponding pin on the module. There is an additional pin on the module which allows for displaying a second digit, but it requires rapid refreshing otherwise both digits are mirrored across both displays. Since we were not able to incorporate hardware timers in our project in time, we did not implement automatic refreshing of the display, and thus used only one of the two displays.

We connected the module in our design as a GPIO device with the same number of pins. Thus we could easily light up the entire display by sending a byte (eight bits, corresponding to each of the eight input pins) where each bit corresponds to one of the segments A-F. Thus each digit 0-9 and each letter a-f can be represented in code as a single number, allowing for more readable code like

XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_0);

Here is some sample code below:

**#define** DISPLAY\_A (1 << 0)

**#define** DISPLAY\_F (1 << 1)

**#define** DISPLAY\_E (1 << 2)

**#define** DISPLAY\_D (1 << 3)

**#define** DISPLAY\_DIGIT (1 << 4)

**#define** DISPLAY\_G (1 << 5)

**#define** DISPLAY\_C (1 << 6)

**#define** DISPLAY\_B (1 << 7)

**#define** DISPLAY\_OFF 0

**#define** DISPLAY\_CHAR\_0 (DISPLAY\_A | DISPLAY\_B | DISPLAY\_C | DISPLAY\_D | DISPLAY\_E | DISPLAY\_F )

**#define** DISPLAY\_CHAR\_1 ( DISPLAY\_B | DISPLAY\_C )

**#define** DISPLAY\_CHAR\_2 (DISPLAY\_A | DISPLAY\_B | DISPLAY\_D | DISPLAY\_E | DISPLAY\_G)

**#define** DISPLAY\_CHAR\_3 (DISPLAY\_A | DISPLAY\_B | DISPLAY\_C | DISPLAY\_D | DISPLAY\_G)

**#define** DISPLAY\_CHAR\_4 ( DISPLAY\_B | DISPLAY\_C | DISPLAY\_F | DISPLAY\_G)

**#define** DISPLAY\_CHAR\_5 (DISPLAY\_A | DISPLAY\_C | DISPLAY\_D | DISPLAY\_F | DISPLAY\_G)

**#define** DISPLAY\_CHAR\_6 (DISPLAY\_A | DISPLAY\_C | DISPLAY\_D | DISPLAY\_E | DISPLAY\_F | DISPLAY\_G)

**#define** DISPLAY\_CHAR\_7 (DISPLAY\_A | DISPLAY\_B | DISPLAY\_C )

**#define** DISPLAY\_CHAR\_8 (DISPLAY\_A | DISPLAY\_B | DISPLAY\_C | DISPLAY\_D | DISPLAY\_E | DISPLAY\_F | DISPLAY\_G)

**#define** DISPLAY\_CHAR\_9 (DISPLAY\_A | DISPLAY\_B | DISPLAY\_C | DISPLAY\_D | DISPLAY\_F | DISPLAY\_G)

**#define** DISPLAY\_CHAR\_A (DISPLAY\_A | DISPLAY\_B | DISPLAY\_C | DISPLAY\_E | DISPLAY\_F | DISPLAY\_G)

**#define** DISPLAY\_CHAR\_B ( DISPLAY\_C | DISPLAY\_D | DISPLAY\_E | DISPLAY\_F | DISPLAY\_G)

***#define*** *DISPLAY\_CHAR\_C (DISPLAY\_A | DISPLAY\_D | DISPLAY\_E | DISPLAY\_F )*

**#define** DISPLAY\_CHAR\_D ( DISPLAY\_B | DISPLAY\_C | DISPLAY\_D | DISPLAY\_E | DISPLAY\_G)

**#define** DISPLAY\_CHAR\_E (DISPLAY\_A | DISPLAY\_D | DISPLAY\_E | DISPLAY\_F | DISPLAY\_G)

**#define** DISPLAY\_CHAR\_F (DISPLAY\_A | DISPLAY\_E | DISPLAY\_F | DISPLAY\_G)

**void** **displayChar**(**char** c) {

**switch**(toupper(c)) {

**case** '0': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_0); **break**;

**case** '1': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_1); **break**;

**case** '2': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_2); **break**;

**case** '3': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_3); **break**;

**case** '4': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_4); **break**;

**case** '5': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_5); **break**;

**case** '6': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_6); **break**;

**case** '7': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_7); **break**;

**case** '8': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_8); **break**;

**case** '9': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_9); **break**;

**case** 'A': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_A); **break**;

**case** 'B': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_B); **break**;

**case** 'C': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_C); **break**;

**case** 'D': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_D); **break**;

**case** 'E': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_E); **break**;

**case** 'F': XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_CHAR\_F); **break**;

**default**: XGpio\_DiscreteWrite(&DisplayGpio, 1, DISPLAY\_OFF); **break**;

}

delay\_ms(50);

}

**Keypad:**

The keypad we used is built like a number of switches wired up in series row by row. When a button is pressed a connection is made changing the output signal. The module has eight pins, four for one of each column, and four for each row on the keypad. According to the specification, the application should sample each column one at a time by pulling that column's pin low and leaving the other rows high. Each button that is pressed will result in a low signal for its respective column. Thus our code simply "walks" a low signal through each of the columns and monitors how the signals of the rows change.

We had originally implemented the devices as an two four-bit GPIOs in our design, so that we can write to all the columns and read from all the rows with a single write/read call, respectively. Unfortunately we were not able to get accurate readings with the combined row GPIO and we were not able to figure out why exactly the issue occurred. When we separated all the row pins to individual one-bit GPIOs, the issue went away.

Sample code for reading key presses is below:

**int** **readKey**(**char** \*name) {

**int** col = 0;

**for**(col = 0; col < 4; col++) {

**int** row = 0;

u32 colMask = ~(1 << col);

XGpio\_DiscreteWrite(&KeypadColGpio, 1, colMask);

**if**(!XGpio\_DiscreteRead(&KeypadRowGpio\_1, 1)) {

\*name = KEY\_NAMES[0][col];

**return** 1;

} **else** **if**(!XGpio\_DiscreteRead(&KeypadRowGpio\_2, 1)) {

\*name = KEY\_NAMES[1][col];

**return** 1;

} **else** **if**(!XGpio\_DiscreteRead(&KeypadRowGpio\_3, 1)) {

\*name = KEY\_NAMES[2][col];

**return** 1;

} **else** **if**(!XGpio\_DiscreteRead(&KeypadRowGpio\_4, 1)) {

\*name = KEY\_NAMES[3][col];

**return** 1;

} **else** {

**continue**;

}

}

**return** 0;

}

**int** **readKeyAndSetDisplay**(**char** \*name) {

**int** ret = readKey(name);

**if**(ret) {

displayChar(\*name);

delay\_ms(50);

}

**return** ret;

}

**Bluetooth:**

The Bluetooth module is simply represented as a UART interface in the application. The original Microblaze project comes with two UART modules synthesized (since the Virtex5 boards come with two native UART devices). We simply mapped the pins of the second device from an unused port to the pins of the Bluetooth module. Thus the application can easily send and receive data from the Bluetooth module as a UART device, much like doing IO through the serial port.

The Bluetooth module can also go into command mode, which is useful if the application wants to change its behavior. We used this functionality to disconnect from any devices which do not supply the correct password.

Sample code for forcing a Bluetooth disconnect:

BTSendCommandAndEcho("$$$");

BTSendCommandAndEcho("K,\r\n");

BTSendCommandAndEcho("---\r\n");

BTReadFlush();

Sample code for reading/writing data to/from Bluetooth module:

**int** **BTSend**(**char** \*buf, uint size) {

**char** \*offset = buf;

**int** written = 0;

**while**(offset < (buf+size)) {

uint toSend = min(BT\_MAX\_SIZE, (buf+size)-offset);

XUartNs550\_Send(&BT\_UART, (u8 \*)offset, toSend);

offset += toSend;

written += toSend;

delay\_ms(50);

}

**return** written;

}

**int** **BTRead**(**char** \*buf, uint size) {

**memset**(buf, 0, size);

**char** \*offset = buf;

uint read = 0;

uint totalRead = 0;

**do** {

uint toRead = min(BT\_MAX\_SIZE, (buf+size)-offset);

read = XUartNs550\_Recv(&BT\_UART, (u8 \*)offset, toRead);

totalRead += read;

offset += read;

} **while**(read != 0 && offset < (buf+size));

**return** totalRead;

}

**Authentication:**

The application authenticates another device by reading the first four bytes it sends. If those bytes match the password input from the user during start up, the connection is authenticated and data is allowed to flow in both directions. If the password is wrong or not enough data is sent, the application terminates the connection and waits for another connection to occur. This all happens in the body of the main loop, allowing the application to continuously connect to and disconnect from different devices.

Sample code:

// Check authentication

**if**(!authenticated) {

**char** otherpw[PASS\_SIZE];

**int** read = BTRead(otherpw, **sizeof**(otherpw));

// Other device sent us its password

**if**(**sizeof**(otherpw) == read) {

**int** i;

**for**(i = 0; i < **sizeof**(otherpw); i++) {

otherpw[i] = toupper(otherpw[i]);

}

**printf**("got: \"%c%c%c%c\"\r\n", otherpw[0], otherpw[1], otherpw[2], otherpw[3]);

**printf**("pass is: \"%c%c%c%c\"\r\n", pass[0], pass[1], pass[2], pass[3]);

**if**(**memcmp**(pass, otherpw, PASS\_SIZE) == 0) {

authenticated = 1;

print("authenticated\r\n");

BTSendWithoutNull("authenticated!\r\n");

**continue**;

} **else** {

// Wrong password, disconnect

BTSendWithoutNull("wrong password\r\n");

BTReadFlush();

BTSendCommandAndEcho("$$$");

BTSendCommandAndEcho("K,\r\n");

BTSendCommandAndEcho("---\r\n");

BTReadFlush();

delay\_ms(1500);

**continue**;

}

} **else** **if**(read != 0 && connected){

BTSendWithoutNull("password must be 4 characters!\r\n");

BTReadFlush();

BTSendCommandAndEcho("$$$");

BTSendCommandAndEcho("K,\r\n");

BTSendCommandAndEcho("---\r\n");

BTReadFlush();

delay\_ms(1500);

}

**continue**;

}

**Authenticated connection/Data transfer stage:**

Once the connection is authenticated, the application simply keeps polling the keypad for input and the Bluetooth module for external data. Any keypad input is sent over to the other device, while any incoming data is displayed on the SSD.

Sample code:

// Main loop when connected

**if**(connected) {

**char** name;

**if**(BTReadLastKeyChar(&name)) {

displayChar(name);

}

**if**(readKey(&name)) {

BTSend(&name, 1);

BTSendWithoutNull("\r\n");

}

}