Multi I/O for FPGA Development Board (2)

Programming in VHDL

By **Andreas Mokroß, Dominik Riepl, Christian Winkler** and **Professor Thomas Fuhrmann** (Germany)

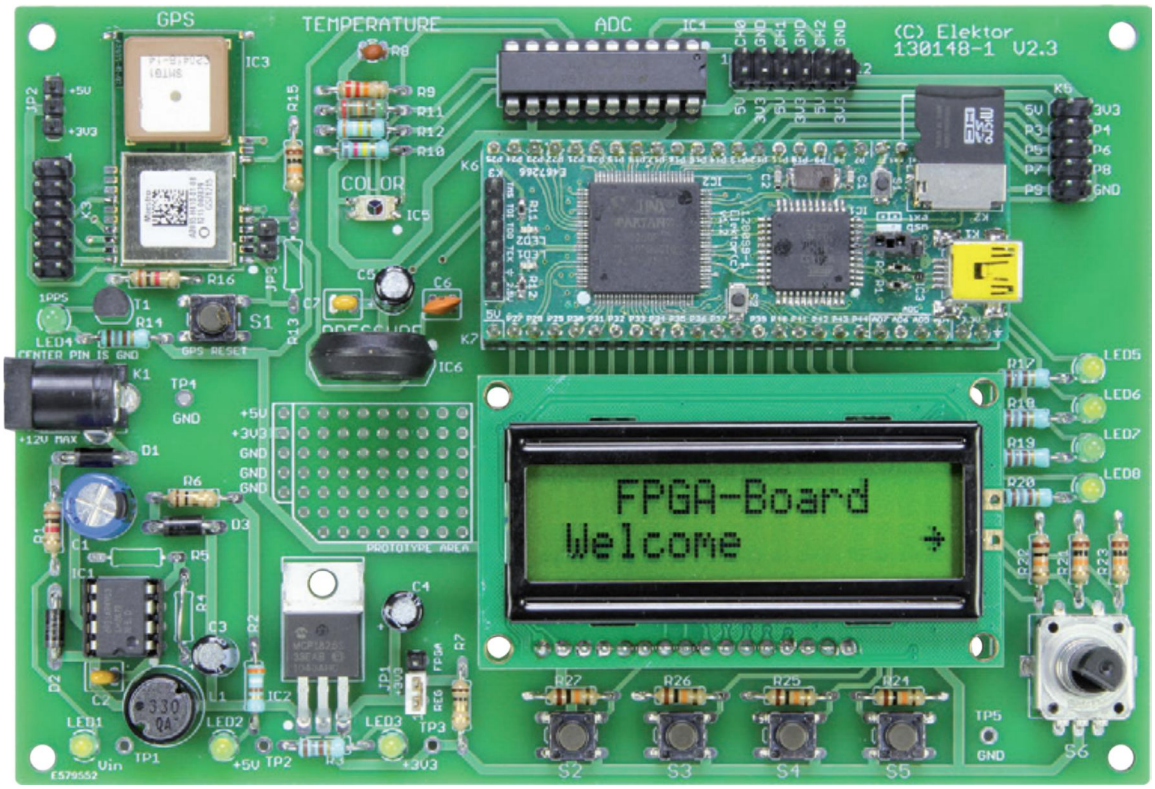

Figure 1.
The board showing the welcome screen.

In the December 2013 edition we described the FPGA development board hardware. With this board it's a simple job to interface the FPGA to real world devices and events. The extensive range of peripherals include sensors and an LC display. The next step is to program the FPGA and get them talking to one another.

The Elektor FPGA board was originally featured in December 2012. It has the part number 120099-91 and can be ordered directly from the Elektor Store. This FPGA board alone does not contain any peripheral chips. To make it more useful as an educational aid an expansion board was developed by the students at the Ostbayerischen Technischen Hochschule in Regensburg, Germany. The expansion board has a number of peripheral sensors and an LCD which interface to the FPGA board. They can be easily configured and controlled using VHDL and put to use in all sorts of applications. The complete development environment with both boards is shown in **Figure 1**. In part 1 [1] of this project the development board hardware was described, in this second part we describe how these peripherals can be controlled using VHDL. This will give you a good grounding in the technology so that you can go on to use it for your own applications.

The complete project has been developed using the XILINX ISE 14.5 Design Suite which is freely available to download from the Internet [2].

A hierarchical approach

For large projects in VHDL it is sensible to approach the design in the way you would a software project by breaking down the program into small manageable parts. Unlike software VHDL doesn't use functions or classes, instead we use descriptions of hardware, the so-called Module. Each module is described by its own file and has its own defined interface to the outside world. A module functions as a self-contained unit and can be simulated with the help of a Test-Bench. It can be seen as a functional black box which integrates into the complete project. The use of modules greatly simplifies the process of system debugging.

Based on the sensors, display, control elements and their control, the VHDL project is divided into the following areas (see **Figure 2**):

- `menu_control`: The central module; this is where all the data comes together and the menu functions are taken care of.
- `taster`: Interfaces to the pushbuttons with debounce logic.
- `lcd`: Drives the LCD and loads the display data.
- `gps_control`: Controls the GPS module and receives Global position information and time of day.
- `ADC_control`: Controls the A/D converter.

At the top level the individual modules are combined and no logic process is described. The connections are made in the sub-modules defined in the `top_level` description using the `component` key word. The compiler is thus informed which components are used and what inputs and outputs are available. Connections to the module are defined in the `port map`, where the input and output of the sub modules are linked to the signals defined in the top level.

While VHDL is a Hardware Description Language, they get translated directly into gates and because the modules work in parallel it is not necessary to consider the order of the modules. All modules have a connection to the 8 MHz clock so that every process is synchronous with the clock. Apart from this the module's data output has an enable signal which goes to a logic '1' state for one clock period when all the processing in the module is completed. It indicates that output data is stable and can be used elsewhere.

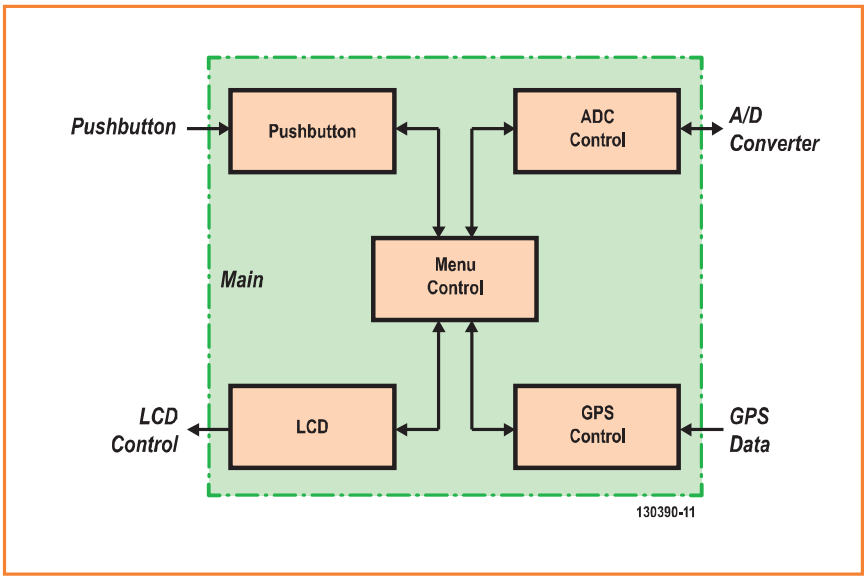

Figure 2.
The project's block diagram.

Package: self_defined_types

This Package contains the global definitions for the complete project and frequently used data types and conversions to make them more readable.

It defines the following data types:

- `BYTE`: An 8-bit wide array of type `std_logic`
- `BYTE_ARRAY`: An arbitrary width array of type `BYTE`

In addition the function `HEXtoASCII` is defined which converts a 4-bit wide `std_logic_vector` with a hexadecimal value into a displayable ASCII character.

The packages are linked into this and any other library using the command:

```
library work;
use work.self_defined_types.all;
```

The ADC Module

The first module described here is used to control the A/D converter and read the output data. The module uses a 3-bit long `std_logic`-input vector `in_channel` to select the analog channel and an 8-bit long `std_logic`-output vector `out_adval` to output the data. The other inputs and

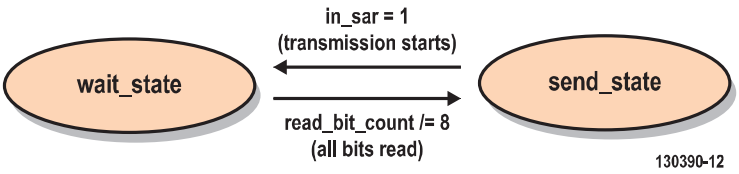

Figure 3.
State diagram of the ADC read processes.

outputs are necessary to communicate with the IC (`in_sar`, `in_do`, `out_clk`, `out_di`, `out_not_cs`, `out_not_se`).

The Clock Process

According to the data sheet of the A/D converter [3] it should be provided with a clock signal in the range from 10 kHz to 400 kHz. For this application we use 100 kHz. The clock is generated a dedicated process, see **Listing 1**.

The counter `cnt_clock` is incremented on each rising edge of the 8 MHz input clock `in_clk` and the internal clock `adc_clk` at `(const_divider - 1) / 2` and `(const_divider - 1)` which is inverted, so that the time that the timer runs from 0 to reset corresponds exactly to a clock period of the generated clock.

Listing 1

```
constant const_divider: integer := 80;
signal cnt_clock: integer range 0 to const_divider - 1 := 0;
signal rising_clk: std_logic;
signal falling_clk: std_logic;
signal adc_clk: std_logic := '0';

process(in_clk)
begin
  if rising_edge(in_clk) then
    if cnt_clock = const_divider - 1 then
      adc_clk <= '1';
      rising_clk <= '1';
      cnt_clock <= 0;
    elsif cnt_clock = (const_divider - 1)/2 then
      adc_clk <= '0';
      falling_clk <= '1';
      cnt_clock <= cnt_clock + 1;
    else
      rising_clk <= '0';
      falling_clk <= '0';
      cnt_clock <= cnt_clock + 1;
    end if;
  end if;
end process;
```

The constant `const_divider` is the ratio of the clock frequency of the external oscillator and the required converter clock frequency:

$$const\_divider = \frac{oscillator\ clock\ frequency}{converter\ clock\ frequency}$$

Apart from this the Enable signal `rising_clk` and `falling_clk` are generated for state machine communication with the converter.

The reading process: implementing a State Machine

For the sake of clarity control of the FPGA functions are implemented in modules using state machines. The following implementation will be for the A/D converter. The state transition diagram of the machine is given in **Figure 3**. Each state is represented by a circle and transition to another state is indicated by an arrow labeled with the transition condition.

Two states are required to read data from the ADC: In `wait_state` the state machine waits until data is available indicated by the SARS output from the ADC going to a logic '1'. The state machine then jumps to the `send_state` which reads the 8-bits from the converter.

To describe the state diagram in **Figure 3** using VHDL a separate data type `state_type_read` is defined with all the possible states and then a `signal` that is this type. The compiler converts this construct into a timer. For a developer the approach used here is more easily understandable.

It is written as a process using a case statement, in which all the possible states of the machine are described, see **Listing 2**.

The two separate states of the machine can now be described:

- `wait_state`: When the A/D converter outputs a new word the SAR status output is set to a logic '1'. This makes the machine state change to `read_state`.
- `read_state`: This is where the ADC serial

output value is read bit by bit and starting with the MSB, written into a shift register. When 8-bits have been read the Value-Enable-Bit `int_val_en` is set to logic '1' and returns to the `wait_state`. After the change this will be reset to logic '0' again.

The ADC outputs a new bit on every falling clock edge of the 100 kHz clock. Each bit is read on the rising edge of the clock with help from the Enable signal from the clock process.

This implementation is used in the same way by all the other state machines so only the individual states and transitions will be described and not the principle itself.

The sending process

A/D conversion is initiated by sending a telegram to the A/D converter. This is performed using a separate process. According to the ADC data sheet (data sheet [3], Figure 20) it reads control signals from the FPGA on rising clock edges. The A/D converter outputs a new bit at rising clock edges. To ensure that the data is stable the implemented state machine reads the value of these bits on the falling clock edge.
The state diagram showing the sending process structure is given in **Figure 4**. The process is implemented as a state machine with three states:

- `wait_state`: Waits for the Channel_Enable signal to start the conversion. When this signal is logic '1' the chip select signal is set to logic '0' and the first bit of the telegram is sent. The state machine changes to the `send_state` state and decodes the remaining bits in the telegram.
- `send_state`: In this state bits are sent one after another on the data line to the A/D converter. The number of sent bits are counted until all the bits have been sent then the state changes to `wait_for_rec_ready`.
- `wait_for_rec_ready`: In this state the wait for the conversion process of the analog voltage is finished. This is done with help of the `int_val_en` signal. When reading out is finished communication with the A/D converter is ended by switching the chip select signal to logic '1'. The machine returns to the output state `wait_state` and the next request can be processed.

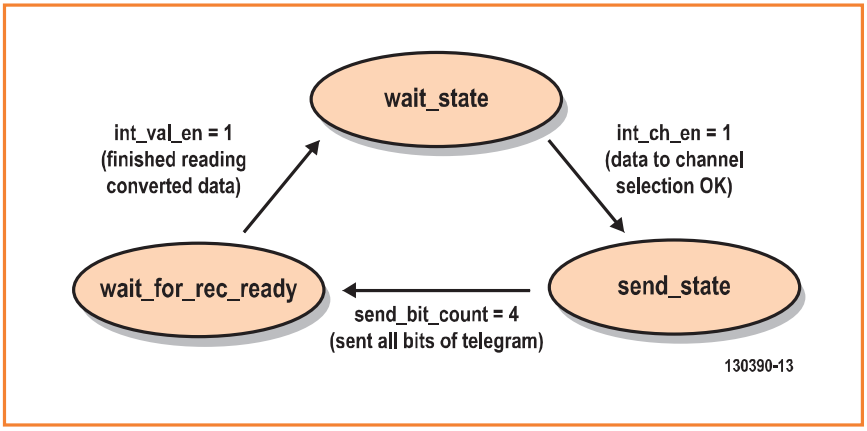

Figure 4.
State diagram of the ADC send processes.

Listing 2

```
type state_type_read is (wait_state, read_state);

signal read_state_machine : state_type_read := wait_state;

process(in_clk) begin
  if rising_edge(in_clk) then
    if rising_clk = '1' then
      case read_state_machine is
      when wait_state =>
        read_bit_count <= 0;
        int_val_en     <= '0';

        if in_sar = '1' then
          read_state_machine <= read_state;
        end if;

      when read_state =>
        if read_bit_count /= 8 then
          int_adval (7 downto 1)   <= int_adval (6 downto 0);
          int_adval (0) <= in_do;
          read_bit_count <= read_bit_count + 1;
        else
          read_state_machine <= wait_state;
          int_val_en <= '1';
        end if;
      end case;
    end if;
  end if;
end process;
```

The pushbutton input module

The process which reacts to pushbutton activity is contained in a separate VHDL module. Its main function is to perform contact debouncing (see article on the board hardware [4]).
To achieve debounce a counter is started when the signal level from the pushbutton input changes state. The counter is used to produce a delay so that the signal level is only valid once this counter has finished counting. A long press would result in the counter timing out several times and registering several presses. To avoid this situation the active pushbutton is polled at every rising clock edge to check if the press has already been registered. The pushbutton input signal state is compared with its state stored when the counter last elapsed. When the two states are the same (long press detected) the counter is reset otherwise it runs until the debounce delay time finishes. Any contact bounce will be finished before the counter reaches the end of its counting period. The signal level is now stored to temporary memory and a short pulse is output.

Listing 3

```
generic (WAIT_40MS: integer := 320100;
         WAIT_4_1MS: integer := 32900;
         WAIT_1_52MS: integer := 12200;
         WAIT_100US: integer := 900;
         WAIT_38US: integer := 400;
         WAIT_450NS: integer := 10);
```

Listing 4

```
when send_data =>
  out_enable <= '1';
  if wait_counter = WAIT_450NS - 1 then
    state <= wait_state;
    wait_counter   <= 0;
  else
    wait_counter <= wait_counter + 1;
  end if;

when wait_state =>
  out_enable <= '0';
  if wait_counter = wait_time - 1 then
    if prev_state = '0' then
      state <= init;
    else
      state <= write_data;
    end if;
    wait_counter <= 0;
  else
    wait_counter <= wait_counter + 1;
  end if;
```

The LCD Module

The LCD has a parallel interface so all control and data information is sent in the form of parallel words. After each word it is necessary to introduce a wait period to allow the LCD board controller to process the information. It is therefore necessary to generate some wait periods. This is achieved with the `generic` command where constants valid in the module are placed. These are defined in the Entity declaration, as in **Listing 3**. The wait period is defined by the integer value which defines the maximum value of the counter clocked at 8 MHz.

The LCD state machine

A state machine with six states is implemented to control the LCD. **Figure 5** shows a simplified state diagram for the LCD. The state machine starts with `start_up`. Firstly there is a 40 ms delay introduced to allow for the LCD to power up. Next is the `init` state to initialize the LCD.

After initialization it automatically jumps to the `wait_for_data` state and stays here until `in_data_en` (an external input) is logic '1'. This indicates that display data is available to be written to the display.

Next it jumps to the `write_data` state, where all the data is written to the display. For every character written the LCD interface requires a '1' of at least 450 ns on its Enable input. This is taken care of after each character is sent out in the `send_data` state. In here the `out_enable` is set followed by a 450 ns wait.

The display requires a processing time of 38 µs after each character is sent to the display. This is generated by using a `wait_state` before the next character is sent.
After each `wait_state` elapses it returns to `write_data` state until there are no more characters left to send to the display.
Listing 4 shows the relevant section of VHDL code which handles `send_data` and its `wait_state`.

GPS

The VHDL description of the GPS module is made up of four individual modules. The top module for GPS control is `gps_control` which contains three sub-modules `gps_serial_parallel`, `gps_checksum` and `gps_parser`.

GPS control (gps_control)

In the top module `gps_control` the other sub-modules referenced above are declared as `components` and linked to the corresponding ports. The process to turn the GPS module off and on is in this module. In addition a short process flashes an LED each time a valid GGA-type sentence is read.

The Conversion Process (gps_serial_parallel)

In the `gps_serial_parallel` module the serial UART protocol data from the GPS module is converted into a one byte wide parallel signal. Using a previously calculated divider constant the communication speed with the GPS module (here we use 4,800 bit/s) can be adapted as necessary.

It is important that the sampling points of the received GPS data stream are synchronized to the data rate. To achieve this, the falling edge of the start bit at the beginning of every byte is detected, the GPS data `gps_data` input signal is shifted into the vector `data_shift` using the internal 8 MHz clock and compared with the bit sequence '1110'.

Once the falling clock edge is detected the following GPS data will be sampled one half of a bit width later i.e. mid-bit, and then shifted into the `int_data` shift register until a complete byte has been received.
The valid data is now stored in `int_data` and written to the parallel data output `out_data` and the Enable-signal set.

Checksum calculation (gps_checksum)

The `gps_checksum` module calculates the checksum on all the transmitted data bits and compares it with the checksum value sent from the GPS module. This ensures that there are no errors in the received sentence. When an error is detected the corrupted sentence is discarded.
A state machine with five states is used to read-in, calculate, validate and them output the result:

- `reset`: All of the signals used for these calculations are first reset to zero. When valid data from the serial/parallel converter (`out_data_enable` = 1) is available the state of `zeichen_in` changes, as soon as a '$' symbol is detected in the data. This symbol is the GPS sentence start character.
- `zeichen_in`: This detects where the checksum begins in the received sentence and changes to the `checksum_in_1` state. An 'if' condition is used to detect an asterisk which marks the end of the sentence data. The two characters following the asterisk are the two-byte sentence checksum.
- `checksum_in_1`: This reads in the first checksum character. This is XOR'ed with the sum of the input characters. When it is not valid the signal `int_checksum_err` will be assigned logic '1'.
- `checksum_in_2`: The second checksum character is read in here. Otherwise identical to `checksum_in_1`.
- `output`: When the checksum is valid then the signal `int_checksum_ok` is given the value logic '1' otherwise it has the value '0'.

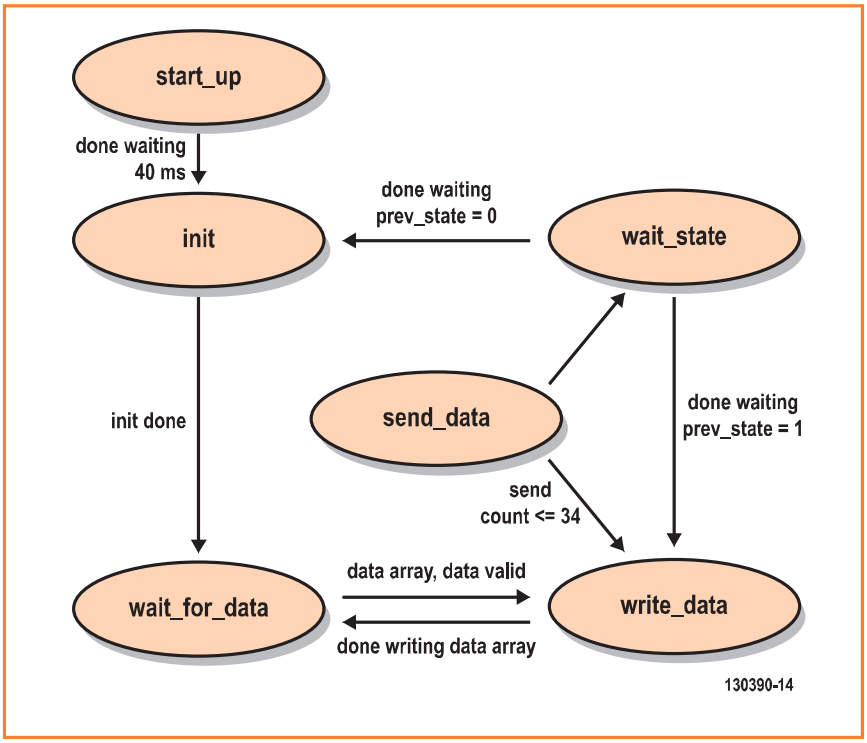

Figure 5.
State diagram of the LC display.

Reading GPS data (gps_parser)

The `gps_parser` module filters out the relevant information from the received GPS data stream and prepares it for further processing. The GPS module sends all its data sentences sequentially

according to the NMEA protocol. It is necessary to identify the sentence of interest (for our purposes the GGA sentence containing positional fix information) and recover it from the data stream. A '$' character identifies the start of every new data sentence. A state machine checks when this occurs. Following this character is the 'GPGGA' sequence which is the preamble to the GPS data of interest to us. There is a state for each data sentence of interest in which the data is read in. Each data field in the sentence is separated by a comma and this is used to change the state of the machine. One after another all the data is read in and sent to the corresponding output.

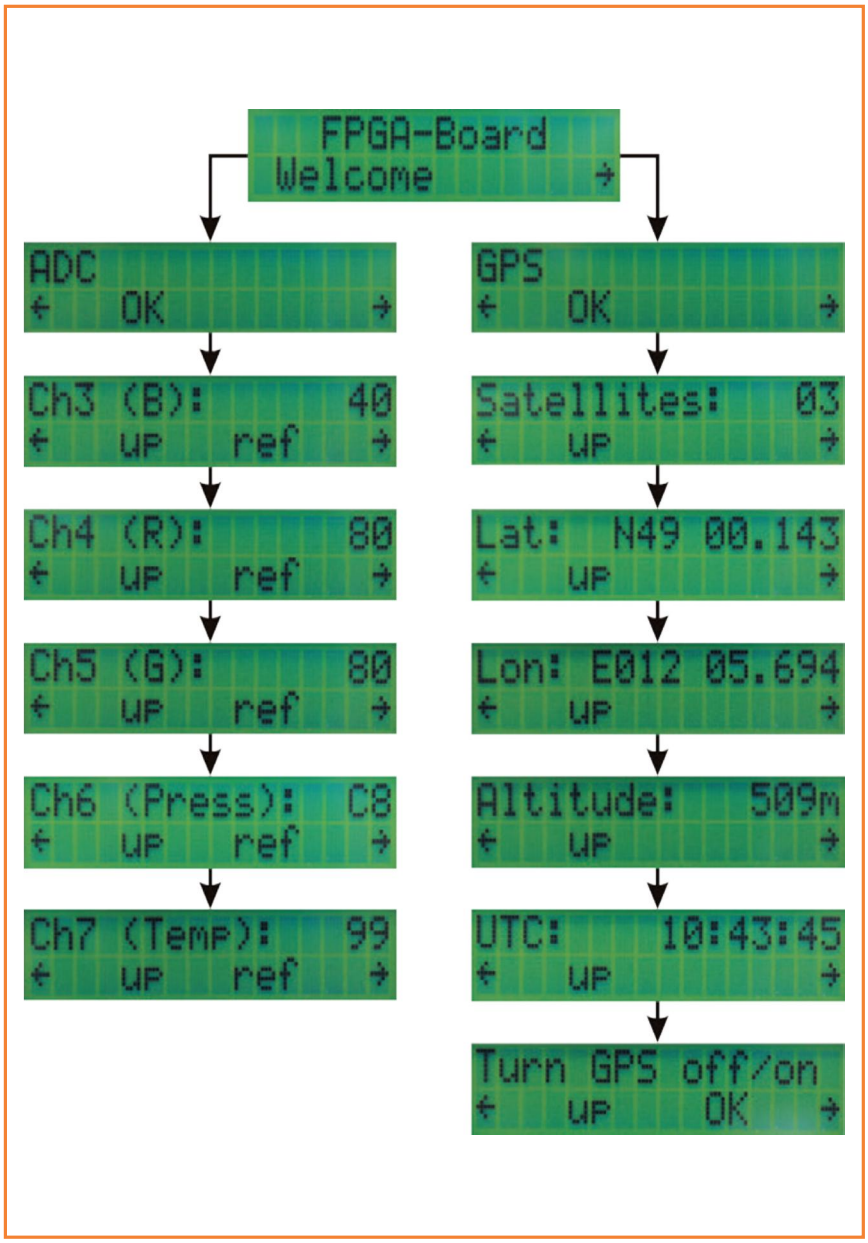

Figure 6.
The menu options.

Menu control in VHDL

A menu has been implemented on the display to allow user control of the GPS module and A/D converter. Pushbuttons under the display allow intuitive interaction with displayed menu options. **Figure 6** shows the menu layout. After the start screen there is an option to select sub menus 'GPS' or 'ADC'.

The GPS sub-menu firstly gives you the option to turn it off or on. Other pages give you the option to view additional information such as your current longitude and latitude. The command 'up' returns you to the next level up in the menu structure.

Selecting 'ADC' from the menu allows you to select a channel of the A/D converter. The measured values are displayed on a page in the sub-menu. Pressing 'ref' (refresh) causes the ADC to make another measurement of the displayed channel and update the display with the new value.

The GPS menu

The menu control is also built with a state machine and can be easily restructured (by changing the state transition diagram) or expanded (add new states). The menu structure and associated state machines are described using the GPS menu as an example.

Each page in the menu has a corresponding state of the state machine for control of the menu. The current state is stored in the signal `state`. The machine starts in the `state_init` state and then changes to the `state_welcome` state.

Now the welcome screen is shown. A press of the pushbutton on the right changes to the `state_gps` state. This state builds the highest level of the GPS menu. In the lower line of the display are arrows pointing to the left and right. Pressing the button beneath the arrow changes the state to `state_adc` and now the A/D converter menu options are displayed.

Staying in the GPS menu you can press OK to get to the first GPS sub-menu, here you have the option to switch the GPS module on and off (`state_GPS_toggle`). When powering down the GPS module it is important to observe the correct power-down sequence to avoid any possible internal memory data corruption. On one level with the state to switch on and off there are display options for GPS data such as longitude and latitude which can be selected using the left and right pushbuttons.

As an example we can show how the longitude is displayed on the LCD. In the `state_longitude` state it will (automatically when valid data is available) assign to the vector elements of `out_lcd_line` the display data elements. In the VHDL description, for example, the data element `in_lon_pre` is assigned to the vector element `out_lcd_line1(5)`. At the fifth position on the first display line is the value of `in_lon_pre` which in this case will be either the letter 'E' or 'W' i.e. east or west of the central meridian. In accordance to this principle each position of the display will be assigned the character to be displayed. After this process `refresh_lcd` refreshes the display and displays the characters.
The principles of state changing and display of data described above also operate in the same way for the other menu pages.

To sum up

As an example project we have demonstrated a menu driven control of the FPGA expansion board. All the Modules consist of systematically implemented state machines. This project used up 40 % of the gates and look up tables in the FPGA. There is still enough in reserve for additional applications. It would be fairly easy, for example to add a function to convert the A/D output values into a temperature reading or a voltage level.

(130390)

Internet Links

[1] Multi I/O for FPGA Development Board, Part 1: www.elektor-magazine.com/130148

[2] Xilinx ISE: www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools/v2012_4---14_5.html

[3] Data sheet for the A/D converter: www.ti.com/lit/ds/snas531b/snas531b.pdf

[4] www.elektor.com/130390