Mizar32

Quick Start Guide

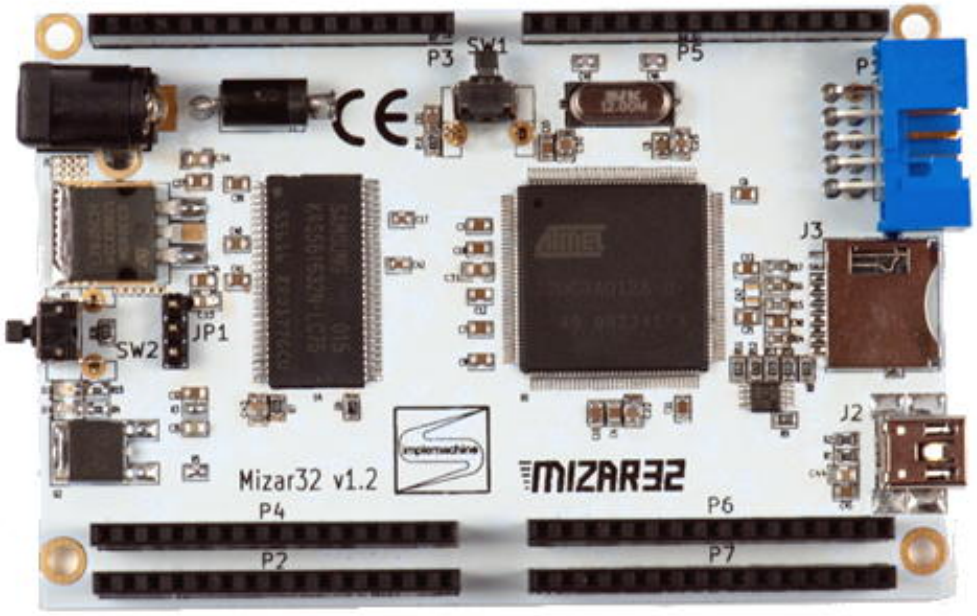
Contents

Articles

References

Article Licenses

Introduction



The Mizar32 project makes it easy to use the hardware and software of a 32 bit microcontroller. The hardware takes the shape of a 9x6cm main board with a 66MHz AVR32 UC3 processor and 32 megabytes of RAM with additional stackable modules to extend its functionality. By default it runs programs written in a modern high level scripting language, Lua, with embedded extensions and is field-programmable: the 128/256/512KB flash memory contains the Lua interpreter that runs your Lua program that is in a file on a micro SD card. This make sit very easy to program in Lua, and the flash can also be reprogrammed with a different interpreter from a PC over USB. Both the software and the hardware are aimed at newcomers to the embedded world who want an easy and powerful environment for rapid application development, for hardware prototyping, for quick production of a solution and for hobbyists to make their own custom devices. The hardware is also offered to manufacturers to avoid a long design cycle and get a new product to market in a short time by simply adding any custom hardware on add-on protoboards and writing a Lua program to do what you want. The hardware design is offered as an open standard and all the project files and manufacturing files are freely available to use, study and modify using open source software tools.

Quick Start

Quick Start Guide

Let's make sure your Mizar32 works OK. You will need:

- A Mizar32 base board (any model: A, B or C)
- A micro SD card
- A PC that can read and write the SD card (you will probably need a micro-to-normal-size SD card adapter or a micro-SD-to-USB adapter to do this)
- A PC-to-Mizar32 micro USB 'mini-B' lead like the ones used with most phones *or* a 7.5V power supply.

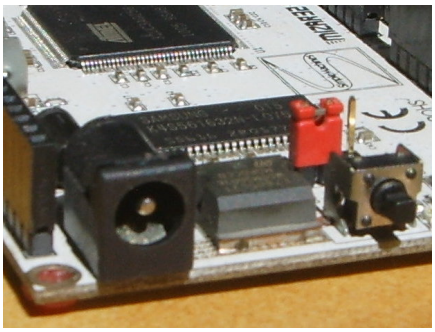
Apply power to the Mizar32

The Mizar32 base board can be powered in two different ways: from its USB socket or from its DC power jack. There is a three-pin header on the main board, "`JP1`", just behind the user button "SW2", which selects the power source that the Mizar32 will run from.

One option is to power the Mizar32 from your PC via a USB cable plugged into the Mizar32's USB connector "`J2`". To take power from here, the jumper on `JP1` needs to be on the two pins furthest from the power jack, pins 1 and 2, which are the pins on the right when you are looking at the board from the side of the power socket and user button.

Alternatively, you can use an external power supply plugged into the power jack "`J1`". In this case, `JP1`'s jumper needs to be connecting the two pins closest to the power jack (pins 2 and 3).

- The external power voltage needs to be at least 7.4 volts but can be anything up to 35V. The centre pin of the power connector needs to be the positive one but the Mizar32 will not be damaged if you accidentally get the positive and negative connections the wrong way round.
- The current needed by the base board is 80 milliAmps. The serial board takes another 5mA, the ethernet board 50mA, the LCD display 7mA and the VGA board 80mA.



The Mizar32 power jack and the power jumper set to use it

When the Mizar32 is correctly powered and the jumper is in the right position, a red light on the main board will be lit next to the user button "`SW2`".

Program the Mizar32 to flash its on-board LED

Let's try running a little program on the Mizar32 to make sure everything is working correctly. This should make it flash the blue light next to the red power light.

On your PC, use a text editor to create a file called "`autorun.lua`" containing:

```
led = pio.PB_29

function delay()
  tmr.delay( 0, 500000 )
end

pio.pin.setdir( pio.OUTPUT, led )
```

```
while true do
  pio.pin.setlow( led )
  delay()
  pio.pin.sethigh( led )
  delay()
end
```

and copy it to a micro SD card. "Safely remove" the SD card and put it in the Mizar32's SD card slot. Now, when you apply power to the Mizar32 (or when you press its Reset button, `SW1`), the blue LED next to the red one should start flashing once per second.

Now you know how to supply electrical power to the Mizar32 and how to write a program for it, load it onto the board and make it run. If this example didn't work for you, please contact us and we'll find out why.

Models and Specifications

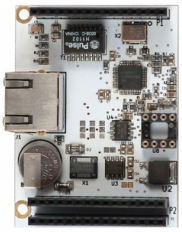
Model	Flash memory	Internal SRAM	SDRAM
Mizar32 A	512KB	64KB	32MB
Mizar32 B	256KB	64KB	32MB
Mizar32 C	128KB	32KB	32MB

- Main processor: AVR32 UC3A0 @ 66 MHz
- Internal fast SRAM: 32KB or 64KB with single-cycle access time
- On-board SDRAM: 32MB with 2-cycle access time
- Internal Flash memory: 128/256/512KB with single-cycle access time
- External Flash memory: up to 4GB on micro SD card.
- Internal operating Voltage: 3.3V with 5V input tolerant I/O
- Digital I/O Pins: 66
- Timer/Counter: 3 channel, 16-bit.
- Analog-to-Digital input pins: 8 with 10-bit resolution measuring 0-3.3v at up to 384,000 samples per second
- Stereo audio bitstream Digital-to-Analog Converter with 16 bit resolution at up to 48kHz
- Pulse Width Modulation channels (PWM): 7
- Universal Sync/Async RX/TX (USART): 2
- Serial Periperal Interface (SPI): 2
- Two-Wire Interface (TWI): 1, I2C-compatible at up to 400kbit/s
- Universal Serial Bus (USB): 1 OTG host with dedicated cable.
- Debug Port: JTAG connector
- Ethernet MAC 10/100: 1 (requires add-on hardware module)
- Oscillators: 2 (12MHz and 32768Hz)
- Buttons: Reset button, user button
- LEDs: Power LED, User LED
- Power supply: 5V USB or 7.5V-35V DC, 80mA (base board) to 222mA (with all add-on modules)
- Dimensions: 96,5mm x 63,5mm
- Weight: 42.5 grams
- Temperature range: -45 to +85°C

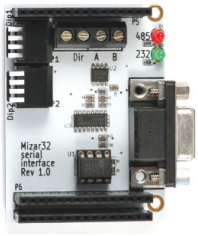
Embedded Hardware interfaces

- MicroSD
- USB
- JTAG
- Add-on bus connectors 1-6 interfaces on the Add-on Bus
- 12 General Purpose I/O pins
- 2 UARTs: one basic, one with modem control signals
- 2 SPI
- I2C interface with 2-way splitter
- 8 ADC inputs
- 3 high-resolution timers
- Ethernet

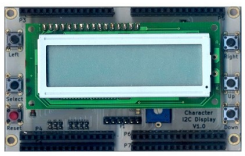
Optional Stacked Modules



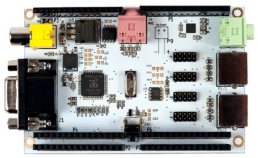
Ethernet and Real Time Clock



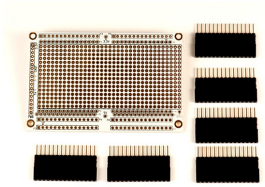
RS232 serial port



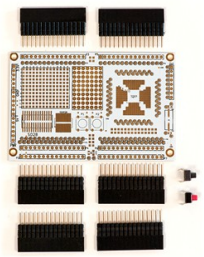
16x2-character LCD display



VGA display, PS/2 keyboard + mouse and audio output



PHT protoboard



SMD protoboard

Hardware subsystems and eLua modules

ADC

Introduction

ADC stands for Analog to Digital Converter. In the real world, the value of signals like light or sound can usually be measured in an analog way but, for these values to be processed by a micro-controller, they must be converted into digital information. The ADC does this work and is embedded in all modern micro-controllers.

So an ADC circuit is the most common way of sensing the outside world but it's not so smart. It can only measure voltage levels at a certain moment in time and you need another piece of hardware, a sensor, in order to convert your analog value into a voltage to work with the ADC and become digital information. Some common types of analog sensors are those for light, sound, temperature, humidity, various type of gas, electricity sensor, and many others.

What the ADC circuit does is to take samples from the analog signal from time to time. Each sample is converted into a number which represents the value at a certain moment in time of the waveform of the analog signal.

To use a well known example, a CD contains digital samples of music, which are the analog to digital conversion of the audio (which in reality is a continuous waveform) with a sampling rate of 44.1Khz. This mean that, 44,100 times a second, its audio waveform is sampled into a number with a resolution of 16 bits: a number from 0 to 65,535. Why 65,535? Because each sample is represented as 16 bits, each of which can take one of two different value (0 and 1) and that gives you 216 = 65,536 different possible values, which are all the numbers from 0 to 65535.

In every analog to digital conversion, we have this two important parameter, the frequency or sampling rate and the number, the sample itself, with a fixed resolution in bits. The resolution is the accuracy of the sampled data.

Mizar32's ADC makes samples at a resolution of 8 or 10 bits, which gives 256 or 1024 different possible values. That's less precise than the high fidelity of an audio CD but it is precise enough for measuring temperatures, pressures, the intensity of light and most other physical signals.

To explain the concept of resolution better, the value of each sampled point represented in our input waveform will be stored in a fixed-length variable. If this variable uses eight bits, this means it can hold values from 0 to 255 (2^8^ = 256). If this variable uses 10 bits, this means it can hold values from 0 to 1023 (2^10^ = 1024). In the case of 10 bits resolution, the number 0 represents the lowest voltage and the number 1023 the highest.

The type of ADC used in AVR32 is a Successive Approximation Register (SAR) ADC, which repeatedly compares the signal it is sampling with its current best estimate until it has the closest value and will find the correct digital value for the sample in 10 clock cycles in the case of 10 bit resolution.

Another advantage of this type of circuit is the use of an output buffer, which allows the circuit that is fed by the ADC to read the digital data while the ADC is already working on the next sample.

Hardware view

The Mizar32 has one analog-to-digital converter that is multiplexed between up to eight inputs, ADC0 to ADC7.

The ADC channels can be set in hardware to perform 8-bit or 10-bit conversions, and the range of input voltages is from 0V to VDDANA (BUS1, pin 9) which is tied to 3.3V on the Mizar32 main board via R3, a 0-ohm resistor.

Signal	GPIO	Bus pin	eLua name	Notes
ADC0	PA21	BUS5 pin 5	`pio.PA_21`	
ADC1	PA22	BUS5 pin 6	`pio.PA_22`	
ADC2	PA23	BUS5 pin 7	`pio.PA_23`	
ADC3	PA24	BUS2 pin 3	`pio.PA_24`	(1)
ADC4	PA25	BUS6 pin 4	`pio.PA_25`	
ADC5	PA26	BUS6 pin 5	`pio.PA_26`	
ADC6	PA27	BUS6 pin 6	`pio.PA_27`	
ADC7	PA28	BUS6 pin 7	`pio.PA_28`	

eLua view

ELua's `adc` module is used to configure and read the ADC inputs. A minimum of two function calls is required: one to start the conversion and another to read the resulting values, which range from 0 (at 0V) to 1023 (at 3.3V).

```
-- Repeatedly measure and print the value on ADC channel 0
adc_channel = 0       -- Measure the first ADC channel
while true do
  adc.sample( adc_channel, 1 )              -- start conversion of one sample
  print( adc.getsample ( adc_channel ) )    -- read the result and print it
end
```

If an ADC pin is not used as an ADC input, it can be used as as a generic PIO pin by calling `pio.pin.setdir()`. For example, to use BUS5 pin 7 (ADC5) as a PIO output instead of an ADC input, you could use:

```
pio.pin.setdir(pio.OUTPUT, pio.PA_24)
```

(1) ADC3, which shares a bus pin with the Ethernet interrupt, can be used as as an ADC input when an ethernet add-on board is not present.

Note that eLua's `adc.setclock()` function is not implemented on AVR32.

Further reading

- The ADC section of the eLua manual [1] for details of all of eLua's `adc.*()` functions;
- The Atmel AT32UC3A datasheet [2] Chapter 33: Analog-to-Digital Converter (ADC).

References

[1] http://www.eluaproject.net/doc/v0.9/en_refman_gen_adc.html

[2] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf#33

CPU

Introduction

The Central Processing Unit is the part of the AT32UC3A chip which controls what the rest of the hardware is doing. It does this by following a program that is stored in its RAM or flash memory.

Hardware view

The Mizar32's CPU is an Atmel AVR32A UC3A RISC processor running at 66MHz with

- 15 general-purpose 32-bit registers
- A 32-bit stack pointer, program counter, link register and status register
- 187 different instructions, most of which take one clock cycle to complete
- Exceptions and interrupts
- User and supervisor modes (not used in eLua)
- Debug mode

The processor implements a load-store architecture, where values are fetched from the memory into registers, worked on there and the results are then stored back into memory. It has 187 different instructions which are:

- MOVE instructions to copy values between the registers or load them with constant values
- Load/store instructions to copy 8-, 16-, 32- or 64-bit data between the registers and the main memory
- Load/store multiple instructions to copy several registers from/to main memory or to the stack at once
- Arithmetic instructions to add, subtract, negate, test and compare values in registers and maximum and minimum to find the highest or the lowest of the values in two registers
- Multiplication: 16x16-bit and 32x32-bit multiply giving 32- or 64-bit results
- DSP instructions: multiply-and-accumulate instructions and saturating or rounding arithmetic
- Logical operations: AND, OR, exclusive OR, one's complement
- Bit operations: Bitfield extraction and assignment, bit set/clear/test, bit reverse, swap bytes/halfwords, count leading zeros
- Shifts and rotates
- Branches and subroutine call and return
- System control instructions

Thanks to its 3-stage pipeline, the processor normally executes one instruction per clock cycle, giving a maximum of 66 million instructions per second.

The CPU is connected to a High Speed Bus Matrix which, in turn communicates with the RAM and Flash memories, the USB and Ethernet hardware and the HSB bridge. The HSB bridge then talks to the rest of the peripheral devices on the chip at a lower speed of 16.5 MHz.

eLua view

The program that the CPU executes is usually the *eLua interpreter*, which either responds to commands that you type on the Mizar32's console, executes them and prints the results, or reads a Lua program from the SD card and does what the program says.

In eLua there are some functions to gain access to low-level features of the CPU:

- `cpu.clock()` returns the CPU's clock frequency (66,000,000Hz)
- `cpu.r32/w32/r16/w16/r8/w8()` to read/write 32-, 16- and 8-bit data from/to the memory or the registers of peripheral devices
- `cpu.sei/cli()` to enable/disable interrupts
- `cpu.set_int_handler/get_int_handler/get_int_flag()` to manage how interrupts are handled

For examples of how to use interrupts from Lua, see the examples in the PIO, Timers and UART sections.

Further reading

- The Atmel AT32UC3A datasheet [1] Chapter 9: Processor and Architecture
- The AVR32 Architecture Manual [2] for a detailed description of the instructions
- The eLua's `cpu` module reference manual [3]

References

[1] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf#9
[2] http://www.atmel.com/Images/doc32000.pdf
[3] http://www.eluaproject.net/doc/v0.9/en_refman_gen_cpu.html

Ethernet

Introduction

The Mizar32 has an add-on hardware module which lets us connect the Mizar32 to the internet.

Hardware view

The Ethernet add-on module is a half-width board which plugs into the left half of the add-on bus connectors, BUS1, BUS2 and BUS3. The hardware module converts the ethernet signals provided by the AVR32UC3A chip into the voltage levels required on an RJ45 connector to connect it to a hub, switch or router.

The board houses a DP83848 Ethernet transceiver, which generates and receives the raw ethernet signals and communicates their contents to the AVR32UC3 on the main board using the RMII protocol, which reduces the number of bus pins necessary to achieve this.

RJ45 pin	Name	Signal
1	TX+	Transmit data
2	TX-	Transmit data
3	RX+	Receive data
6	RX-	Receive data

Signal name	AVR32 pin	Bus pin	Name
ETHERNET	PA24	BUS2 pin 3	Ethernet interrupt
REF_CLK	PB0	BUS1 pin 3	50MHz reference clock
TX_EN	PB1	BUS1 pin 4	Transmit enable
TX0	PB2	BUS1 pin 5	Transmit data
TX1	PB3	BUS1 pin 6	Transmit data
RX0	PB5	BUS2 pin 5	Receive data
RX1	PB6	BUS2 pin 6	Receive data
RX_ER	PB7	BUS2 pin 7	Receive error
MDC	PB8	BUS2 pin 4	MDIO clock
MDIO	PB9	BUS2 pin 8	MDIO data
RX_DV	PB15	BUS1 pin 7	Receive data valid

eLua view

eLua has a `net` module which lets you make TCP connections to other computer and receive incoming TCP connections from them, send and receive data and disconnect.

The following example waits for an incoming connection on port 23, then receives data from the network and prints it on console.

```
-- wait for an incoming connection on the TELNET port
socket, remote, err = net.accept( 23 )
if err ~= net.ERR_OK then
  print( "Error waiting for connection" )
else
  -- print all lines of data until they close the connection
  repeat
    res, err = net.recv( socket, "*l" )  -- That's *L not *One
    if err ~= net.ERR_OK then
      print( res )
    end
  until err ~= net.ERR_OK
end
net.close( socket )
```

IP address assignment

The ethernet software included in the standard firmware for models A and B requests an ethernet address using DHCP. If it cannot find a DHCP server on the local network, it gives up after 60 seconds and assigns itself the address 192.168.1.10 with gateway and DNS server of 192.168.1.1.

You can assign a fixed IP address that will be available immediately by using the Mizar32 Web Builder [1] to create your own firmware: click on "Mizar32 Web Builder", then "Build Now", then select BUILD_UIP, clear BUILD_DHCP and set your required IP address in the fields at the bottom of the page. Instructions for programming the resulting firmware file to the board are on the page "Flashing firmware".

Further reading

- The eLua Reference Manual's page for the net module [2]
- The Atmel AVR32UC3A datasheet [3], section 29: Ethernet MAC
- The DP84838I transceiver datasheet [4]

References

[1] http://builder.simplemachines.it
[2] http://www.eluaproject.net/doc/v0.9/en_refman_gen_net.html
[3] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf#29
[4] http://www.ti.com/lit/ds/symlink/dp83848i.pdf

I2C

Introduction

I2C stands for Inter-Integrated Circuit bus which is used to communicate between different silicon chips. More specifically, a bus is a digital communications channel that can be shared by many devices or peripherals.

I2C has a wide range of application and is usually used to communicate between integrated circuits in the same box, either on the same circuit board or across different boards where high-speed communication speed is not critical.

Philips originally developed I2C for communication between the chips inside a TV set, but with time the system has boomed and from its first release in 1982, it is now used by more than 1000 different integrated circuits.

One reason for its success is that it is a practical and economical way of creating complex circuits by connecting everything on the same bus with just two wires, the data line and the clock line, which significantly reduces the number of electrical signals that must be routed on a circuit board.

Examples of real world applications are connecting device like non volatile memory, Real Time Clock circuits, digital sensors, I/O expanders, digital LEDs, liquid crystal displays and switches.

The I2C specification has been revised several times, always maintaining backward compatibility, from v1.0 in 1992, v2.1 in 2000 and v3.0 in 2007. The Standard mode runs at up to 100 kbit/s, up to 400 kbit/s in Fast mode, up to 1 Mbit/s in Fast-mode Plus, or up to 3.4 Mbit/s in High-speed mode. The higher speed modes were added later in new revisions of the I2C specification, so not all integrated circuits support all the speeds. However, the protocol is backward compatible, so newer, faster devices can still talk to the older, slower devices, and the speed of one device connected to the bus does not affect the other devices on the bus. The hardware of the Mizar32 I2C interface supports Standard mode reaching 100 kbit/s and Fast mode up to 400 kbit/s.

Some manufacturers, like in our case Atmel, use the name TWI (Two Wire Interface) for I2C because it doesn't implement every last detail of full I2C, but they are compatible and are essentially the same thing.

To understand more of I2C, let's take a closer look at how the I2C protocol works.

I2C is an Half Duplex protocol, which means that only one device can talk at a time. The bus has two wires: clock and data.

The Master device supplies the clock to the bus. The most common configuration is one Master device with one or more Slave devices. An alternative to this is Multi master mode, in which more than one Master device can talk on the same bus without causing errors. The Masters have a way of deciding between them who is controlling the bus in each moment. The mechanisms for doing this are called Bus Arbitration and Clocks Synchronization.

In a configuration with a single Master device and many slaves, it is the Master device which supplies the clock signal, but slave devices can make it to slow it down to a lower speed if they some extra time.

Electrically, the two wires are pulled high with one resistor each, and the devices send signals by pulling these lines low. In Standard mode with a maximum bus speed of 100 Kbit/s, the resistor value is 10K ohm and in Fast mode, up to 400 Kbit/s, each resistor is 2.2K ohm.

Each Slave device has a 7-bit address that it responds to on the bus, and every device on the same I2C bus must be set to respond to a different address.

When two devices are communicating, one of them is Master and the other Slave, and one of them is transmitting data on the bus and the other is receiving it, so in any particular moment, a device can be a Master-Transmitter, a Master-Receiver, a Slave-Receiver or a Slave-Transmitter.

Here is the sequence of signals generated by a Master-Transmitter, the device that initiates communication with a Slave:

The first event is when a Master-transmitter generates a start condition, which is a high-to-low transition of the data (SDA) line while the clock (SCL) line is high. The first thing in all I2C messages is always a Start condition.

The Start condition causes all the Slaves to wake up and listen, so the Master sends the Slave-receiver address, composed of 7 bits, followed by a direction bit (0 to send, 1 to receive), then an acknowledgement (ACK) bit is generated by the Slave-Receiver to say that it has recognised its address and is listening. If the direction bit was 0 to say that the Master wants to send data tot he slave, it transmits 8 bits of data, the Slave sends an ACK bit again, the Master sends another 8 bits of data, the Salve an ACK and so on. After every byte of data, the Slave that is receiving the data must send an ACK bit by pulling the Data line low for a moment. If the ACK is missing, this means that nobody received the data and whoever is transmitting stops the transmission. After all the bytes of data are transmitted, the Master closes the communication by generating a Stop condition on the bus, which is a low-to-high transition of the SDA line while the clock (SCL) is high. All I2C messages always end with a Stop condition.

In a more condensed form:

S	Start condition
ADDR	7-bit slave address
R/W	Data direction bit: 1 to read or 0 to write
DATA	8 bits of data
ACK	1 bit of Acknowledgement
P	Stop condition

Sometimes ADDR and R/W are considered as an 8-bit value, with the address in the top 7 bits and the R/W flag in the least significant bit. For example, we might speak of a 7-bit slave address of 42 and a direction bit of 1, while in other moments we might speak of an 8-bit slave address of 85, which means the same thing.

Here is a condensed view of sending two bytes of data from a Master to a Slave:

S	ADDR	W	ACK	DATA	ACK	DATA	ACK	P

Hardware view

The AVR32UC3A chip has one I2C controller, and the Mizar32 provides its signals on the Left bus connector BUS2.

The main board of the Mizar32 also has a PCA9540 two-way I2C multiplexer chip which can connect the I2C signals to either one of two more sets of I2C bus pins, the Left and Right I2C buses, one of which is available on the BUS2 and the other on the BUS5 connector. If you use these instead of the main I2C bus pins, you can have twice as many I2C devices in your system. Furthermore, if your hardware design needs two I2C devices that respond the same I2C address, you can put one on the Left I2C bus and one on the Right.

When the Mizar32 is turned on, the multiplexer is not active and the AVR32's I2C signals are only fed to the main I2C bus pins. To have the I2C bus signals appear on the Left I2C bus, you program the value 5 to the multiplexer's control word, to talk with the Right bus you program the value 4 to the control word. To disable the multiplexer, you program 0.

See the code examples below for how to do this using eLua.

The I2C multiplexer and all the main I2C devices on the Mizar32 add-on modules are on the main I2C bus, so you don't have to program the I2C multiplexer to access them. The only devices that the Mizar32 modules place on the Left and Right I2C buses are the EEPROMS on each add-on module.

Bus pins

Signal	GPIO	Bus pin	Notes
SDA	PA29	BUS2 pin 10	Main I2C bus data
SCL	PA30	BUS2 pin 11	Main I2C bus clock
BUS_SDA_L	-	BUS2 pin 12	Left I2C bus data
BUS_SCL_L	-	BUS2 pin 13	Left I2C bus clock
BUS_SDA_R	-	BUS5 pin 3	Right I2C bus data
BUS_SCL_R	-	BUS5 pin 4	Right I2C bus clock

I2C address assignment

The following tables show the I2C slave addresses used by the chips on the Mizar32 and its add-on modules.

In the following tables, we give both the 7-bit codes in decimal notation, and the corresponding 8-bit command byte values in hexadecimal.

Addresses used by devices on the Mizar32 main I2C bus

Device	7-bit address (decimal)	8-bit (hex)	Notes
LCD display (commands)	`0111110` (62)	`7C/7D`	Reading returns the cursor position
LCD display (data)	`0111111` (63)	`7E/7F`	Reading returns the push-buttons
VGA board 24LC512	`1010000` (80)	`A0/A1`	(1)
PCF8563 RTC on Ethernet board	`1010001` (81)	`A2/A3`	
DS1337 RTC on Ethernet board	`1101000` (104)	`D0/D1`	
I2C multiplexer PCA9540	`1110000` (112)	`E0/E1`	

(1) The VGA's 24LC512 EEPROM contains the program for the Propellor chip that runs the VGA board, which is only connected to the Mizar32's main I2C bus if two pairs of contacts, GS4 and GS5, are joined by solder on VGA board, allowing you to program the Propeller chip from the Mizar32 using the Mizar32 Propeller Programmer [1].

eLua view

eLua has an `i2c` module providing setup, start, address, send/receive byte and stop primitives.

Sending data to a device

As a simple example, we will send a single command byte to the I2C splitter on the main board:

```
-- Send a byte to the I2C multiplexer to tell it to enable the left I2C bus

id = 0                    -- which I2C bus to use (there is only one!)
mux_addr = 112            -- the slave address of the I2C multiplexer
mux_disable = 0           -- control word to disable the multiplexer
mux_left = 5              -- control word to enable left bus
mux_right = 4             -- control word to enable right bus

i2c.start( id )
if not i2c.address( id, mux_addr, i2c.TRANSMITTER ) then
  print "The multiplexer did not reply"
else
  -- enable the multiplexer onto the left bus
  if i2c.write( id, mux_left ) ~= 1 then
    print "The multiplexer did not acknowledge the write"
  end
end
i2c.stop( id )
```

Reading data from a device

To read one byte from an I2C device, instead, the following sequence should be used:

```
-- Retrieve and print the contents of the I2C splitter's control register

id = 0                    -- which I2C bus to use (there is only one!)
mux_addr = 112            -- the slave address of the I2C multiplexer

i2c.start( id )
if not i2c.address( id, mux_addr, i2c.RECEIVER ) then
  print "The multiplexer did not reply"
else
  cr = i2c.read( id, 1 ) -- read the control register (one byte)
  -- print the contents of the control register as a decimal number
  print( "Control register = " .. string.byte( cr ) )
end
i2c.stop( id )
```

Further reading

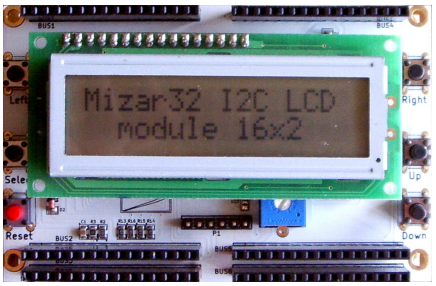
- Wikipedia's I2C article [2]
- The Atmel AT32UC3A datasheet [3] Chapter 24: Two-Wire Interface (TWI).
- Philips Semiconductors PCA9540 2-channel I2C multiplexer's datasheet [4]
- Philips Semiconductors I2C address allocation table [5]
- The I2C module in the eLua reference manual [6].

References

[1] http://github.com/simplemachines-italy/Mizar32-program-propeller
[2] http://en.wikipedia.org/wiki/I2C
[3] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf#24
[4] http://simplemachines.it/Datasheets_mizar/PCA9540_6.pdf
[5] http://simplemachines.it/doc/IC12_97_I2C_ALLOCATION.pdf
[6] http://www.eluaproject.net/doc/v0.8/en_refman_gen_i2c.html

LCD

Introduction



Mizar32 I2C LCD character display module

The Mizar32 LCD Display is an add-on hardware module that plugs into all six of the Mizar32 main board's bus connectors. It has a 16-column by 2-line Liquid Crystal Display, where each position can display one of a range of characters, and it has six push buttons: left, right, up, down, select and reset.

The red reset button resets the on-board PIC microcontroller that controls the display, while the five buttons can be read by the main processor over the I2C bus.

Hardware view

The LCD display board is run by a PIC 16F84 microcontroller, which is a low-power processor running a program written by Simplemachines that talks to the Ampire 162B LCD panel, detects button presses and communicates with the main AVR32 processor over the I2C bus.

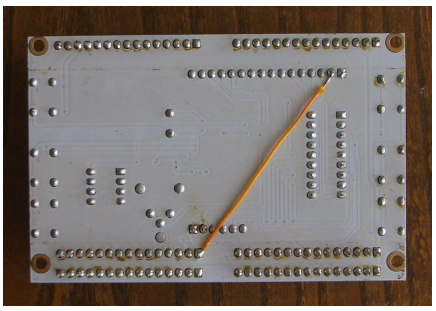
The LCD panel itself hosts another microprocessor, the SED 1278, which receives commands from the PIC, generates characters for the LCD panel and keeps the display refreshed. It has an internal character memory of 40 characters by two lines, of which 16x2 are shown at any one time on the LCD screen. The SED 1278's program is in ROM and cannot be changed.

I2C bus speed

When talking with the LCD display over I2C, the Mizar32's I2C bus must be set to a speed no faster than 50kHz. If the Mizar32 talks with other I2C devices at speeds up to 100kHz, this will not confuse the LCD display: it is able to recognise its own address (and reject all other addresses) at up to 100kHz but can only process data packets at a maximum of 50kHz.

Power supply

The LCD display module needs the Mizar32 to be powered from its 7.2 volt DC power jack. If you wish to run it from the Mizar32's USB power supply, you can solder a wire on the underside of the LCD board from pin 1 of the BUS5 connector (near the centre of the board's edge) to pin 2 of the LCD1 connector (near the "Left" button).



Mizar32 LCD module USB power hack

I2C command set

The board responds to the 8-bit I2C command bytes 0x7C to 0x7F (7-bit slave addresses 62 and 63):

Command	Function
`0x7C`	Send commands to the LCD
`0x7D`	Read the LCD RAM address
`0x7E`	Send data bytes to the LCD
`0x7F`	Read the push-button switches

I2C command 7C: Send LCD commands

Physically the LCD display has 2 rows of 16 characters, but internally it has a character memory which holds 40 characters per row and the 16x2 visible display only ever shows a portion of the internal character memory.

Printing past the end of the 40th character of the first row of the character memory passes automatically to the start of the second row and going past the 40th character of the second row continues at the start of the first row. Similarly, when moving the cursor left from the first character of either row moves it to the 40th position of the other row.

Clear display and return home

Command byte 0: Reset the LCD module

For the SED1278, command byte 0 is a no-op. Here, instead, the PIC recognises it as a special case and resets the SED:

- clear screen
- no cursor displayer
- two rows of characters
- 8x5 pixel character font
- cursor moves right when character data are sent; display does not shift

In eLua, you can do this with `mizar32.lcd.reset()`

Command byte 1: Clear display

Command byte 1 writes spaces into every position of the character memory, moves the cursor to the first row, first column of the character memory and resets the display shift so that the first characters of each line of the internal character memory are aligned to the first characters of the physical display.

In eLua, `mizar32.lcd.clear()`

Command byte 2: Return home

Command byte 2 moves the cursor to the first row, first column of the character memory and resets the display shift. It's the same as "clear display" without clearing the contents of the character memory.

In eLua, `mizar32.lcd.home()`

Data entry modes

Command byte 4: when printing, move cursor right

After command byte 4, printing a character to the LCD display makes the cursor move one position to the right: if the cursor goes off the right edge of the physical display, it continues to write to the internal character memory but the cursor, and these characters, are not shown on the physical display.

Command byte 5: when printing, shift display right

This is another way to print text backwards, this time keeping the cursor in the same position in the display and shifting the text one place to the right each time a character is printed. If you are in this mode with the cursor at (1,1) and print "`Hello`", you'll end up with the `H` in the fifth position of the first row and "`olle`" in the first four positions of the second, because going off either end of one row of the 40-column character memory always continues at the opposite end of the other line.

Command byte 6: when printing, move cursor left

When command byte 6 has been sent, printing a character makes the cursor move one place to the left, so if you print "Hello", the writing comes out backwards on the display "`olleH`" leaving the cursor to the left of the last character. To see this, you would first have to move the cursor to the right of the display, otherwise it would immediately go off the left edge of the display, leaving only an "H" visible and printing the "`olle`" at the end of the second line (off the screen).

Command byte 7: when printing, shift display left

After command byte 7, each time a character is sent to the display, the cursor remains in the same physical position and all the displayed characters move left by one place. Another way to see it is that the 16x2 physical display moves one place to the right on the virtual display so that, for example, if it was displaying columns 1-16, it then displays columns 2-17. To see the text you print, you would first have to move the cursor to the right side of the display, otherwise everything you display would immediately be shifted off the left edge of the display, and would only re-appear on the right edge when you had printed another 24 characters (24 because the virtual display is 40 characters wide and the physical one 16 wide).

In eLua, these four options can be selected with `mizar32.lcd.setup( display_shift, right_to_left )` where the two parameters are either `true` or `false`. By default, the display doesn't shift and characters are printed left-to-right.

Cursor modes

Turning the display on and off and saying what type of cursor you want displayed are done together.

Command byte 8: turn display off

This turns the display off, showing a blank panel. However, it remembers what data are in the 40x2 character memory, the user-defined characters, the cursor position and which part of the character memory was being displayed on the physical display.

Command byte 12: turn display on with no cursor

If the display was off, it is turned back on, but no cursor will be displayed.

Command byte 13: turn display on and show blinking block cursor

If the display was off, it is turned back on, and at the cursor position a blinking black block will be displayed.

Command byte 14: turn display on with underline cursor

If the display was off, it is turned back on, and at the cursor position, the bottom line of the character cell will be all black.

Command byte 15: turn display on with underline and blinking block cursor

In this case, you get both the constant underline and the blinking block. I'm not sure why you would want that...

In eLua, the cursor type can be selected using `mizar32.lcd.cursor( what )`, where `what` is one of `"none"`, `"line"` or `"block"`, and you can turn the display off and on using `mizar32.lcd.display( what )` where `what` is `"off"` or `"on"`. When you turn it back on, eLua remembers the type of cursor and restores it.

Cursor or display shift

There is another way to shift the visible part of the 40-column memory left or right, and to move the cursor one place left or right.

Command byte 16: Move cursor one place left

This moves the cursor one character place left both in the memory and on the display. If it moves off the visible part of the display, the cursor is not visibile, and it if moves left from column 1, it goes to column 40 of the other line.

Command byte 20: Move cursor one place right

This moves the cursor one character place right both in the memory and on the display. If it moves off the visible part of the display, the cursor is not visibile, and it if moves right from column 40, it goes to column 1 of the other line.

In eLua, these can be done using `mizar32.lcd.cursor( "left" )` and `"right"`.

Command byte 24: Shift the display one place left

This changes which part of the virtual display is visible on the physical display. It shifts all the visible characters to the left by one place, revealing a new column of characters from the virtual display at the right edge.

When shifting the display, the cursor remains in the same place in the internal character memory, so on the physical display it is also seen to move one place to the left.

Command byte 28: Shift the display one place right

The reverse effect is obtained with command 28, which moves the displayed characters and cursor one place to the right and reveals two new characters on the left side. Again, the cursor stays with the character that it was sitting on before.

In eLua, these can be done using `mizar32.lcd.display( "left" )` and `"right"`.

Set display operating mode

Command bytes 32-63

Bits 4, 8 and 16 of these commands set the display operating mode:

Bit	Meaning	Values	
		0	1
4	Display font type	5x8	5x11
8	Number of display lines	1 line	2 lines
16	Interface data length	4 bits	8 bits

For the Mizar32 display module, you need 5x8, 2 lines, 4 bits, which is the default setting made on power-up.

Set RAM addresses

Command bytes 64-127: Set CG RAM address

Command bytes 64 to 127, move the cursor from the display to the character-generator RAM. The following data bytes do not display anything on the screen, but instead they say which dots are clear and which black of the eight user-definable characters with codes 0 to 7 (and 8 to 15).

The first user-definable character's definition is in locations 64 to 71, the second from 72 to 79, and so on. Each byte defines one row of the character working from top to bottom, with the five least significant bits defining the five pixels of each row in order 16, 8, 4, 2, 1 from left to right. A "one" bit is displayed black.

For example, to define characters 0 and 1 as left-pointing and right-pointing black triangles:

16	8	4	2	1	Sum	16	8	4	2	1	Sum
				*	1	*					16
			*	*	3 = 2 + 1	*	*				24 = 16 + 8
		*	*	*	7 = 4 + 2 + 1	*	*	*			28 = 16 + 8 + 4
	*	*	*	*	15 = 8 + 4 + 2 + 1	*	*	*	*		30 = 16 + 8 + 4 + 2
		*	*	*	7 = 4 + 2 + 1	*	*	*			28 = 16 + 8 + 4
			*	*	3 = 2 + 1	*	*				24 = 16 + 8
				*	1	*					16
					0						0

you would send command byte 64 followed by 1, 3, 7, 15, 7, 3, 1, 0, 16, 24, 28, 30, 28, 24, 16, 0 and to return to regular character-writing mode to display them, you need to issue a "Set DD RAM address" command byte (see the next entry). If the characters are already displayed on the screen, their shape changes instantly when they are redefined.

In eLua, you can do all of this with the function calls:

```
mizar32.lcd.definechar(0, 1, 3, 7, 15, 7, 3, 1)
mizar32.lcd.definechar(1, 16, 24, 28, 30, 28, 24, 16)
```

Unlike the command byte interface, eLua's `mizar32.lcd.definechar()` function switches you back to normal character-writing mode and leaves the text cursor where it was.

Command bytes 128-255: Set DD RAM address

If the top bit of a command byte is set, the lower 7 bits set the position of the cursor within the data memory, which is where the next character will be stored in the display's character memory (and shown on the screen when the display is shifted so as to show that part of the character memory).

Codes	Where the cursor moves to
128-167	The first row of 40 characters
192-231	The second row of 40 characters

I don't know what happens if you set a DD RAM address of 168-191 or 232-255.

In eLua you can use `mizar32.lcd.goto( row, column )` where `row` is 1 or 2 and `column` is from 1 to 40.

I2C command 7D: Read Address Counter

I2C command `0x7D` reads the current position of the cursor within the character memory and sends it back over the I2C bus.

This is the internal memory address of where the next character will be stored if you send a data byte. The byte that is returned is a value from 0 to 127 that corresponds to lower 7 bits of the codes used in the "Set DD RAM address" command above.

In eLua, you can use `row, column = mizar32.lcd.getpos()`, which returns values 1 or 2 and from 1 to 40.

I2C command 7E: Send LCD data bytes

I2C command `0x7E` sends data to the LCD display.

Each data byte you send usually displays one character on the LCD display and moves the cursor right one position.

Codes	What they display
0-7	Eight user-defined characters
8-15	The same eight user-defined characters
16-31	Blank
32-126	Standard ASCII characters
127	
128-159	Blank
160-255	Chinese characters and special symbols

The exception is when you have just sent one of command bytes 64-127, which make the following data bytes set the pixels of a user-definable character (see above, "Set CG RAM address").

In eLua, you can use `mizar32.lcd.print( date )`, where `data` is a list of one or more strings and numbers, separated by commas. Strings are the normal way to display messages of ASCII text, while a numerical datum should have a value from 0 to 255 and displays a single character from the above table. To print a whole number in decimal, instead, you need to format it first. For example if the variable `gen` contains a whole number that fits in 3 characters (0-999), you could use:

```
mizar32.lcd.print(string.format("%3d", gen))
```

I2C command 7F: Read push-buttons

I2C command `0x7F` fetches the current state of the LCD module's push-button switches as a byte with some combination of the lower 5 bits set to indicate which buttons are currently held down.

Key	Select	Left	Right	Up	Down
Bit	1	2	4	8	16

The display can reliably detect whether the Select button is held down and any two of the other four buttons. If three of the other buttons are held down, the module returns a byte saying that all four are pressed (this is a consequence of the way the buttons are connected on the circuit board to give you five buttons with only four wires).

In eLua, you can use `buttons = mizar32.lcd.buttons()`, which returns a string containing a selection of the characters A string containing a selection of the characters `L`, `R`, `U`, `D` and `S` to say whether the Left, Right, Up, Down and Select buttons are currently held down. If none are pressed, an empty string is returned. The hardware allows Select to be detected reliably and up to two of the other four, but if three of Left, Right, Up and Down are being held, all four are returned.

eLua view

If you are feeling brave, you can talk to the LCD Display module using the above codes and eLua's `i2c` platform module primitives. For example, to display "`Hello`" you can say:

```
i2c.setup(0, 50000)
i2c.start(0)
ack = i2c.address(0, 63, i2c.TRANSMITTER)
if ack then
  i2c.write(0, "Hello")
end
i2c.stop(0)
```

Simplemachines has added a platform module to eLua, `mizar32.lcd.*()`, documented above, which does all the I2C magic and special timing for you. Using this module to achieve the above, you can just say:

```
mizar32.lcd.print("Hello")
```

It provides the following functions

mizar32.lcd.reset()

Initialises the display, resetting everything to as initial state: clear screen, no cursor, displaying columns 1-16 of the 40-column memory, ready to print at (1,1), writing text from left to right and moving the cursor one place right after each character. You don't have to call reset at the start of your program, but doing so does will ensure that your program still works if the display has been left in a funny state by some previous run.

mizar32.lcd.setup(display_shift, right_to_left)

This can be used to set some of the stranger operating modes of the LCD display. Both parameters are optional and if you omit them, they default to false, which sets sensible mode.

`display_shift`: If true, then with each character you subsequently print, the cursor will move by one place in the character memory as usual but the display's contents will also move by one position horizontally in the opposite direction so that the cursor remains in the same column of the physical display. This can be used to achieve

"scrolling text" effects. Note, however, that when the cursor passes from column 40 to column 1 or vice versa, it flips over to the other row.

`right_to_left`: If true, text will be printed right-to-left: the cursor will move one position to the left in the character memory and, if display shifting is also enabled, the contents of the display will shift to the right so that the cursor stays in the same column on the screen.

mizar32.lcd.clear()

Clears the display, moves the cursor to the top left (position 1,1) and resets the display shift to show columns 1 to 16.

mizar32.lcd.home()

Moves the cursor to the top left (position 1,1) and resets the display shift.

mizar32.lcd.goto(row, column)

Moves the cursor to the specified row and column.

- `row`: A number (1 or 2) giving the row you want to move to.
- `column`: A number (1 to 40) giving the position within that row in the character memory.

row, column = mizar32.lcd.getpos()

Returns the current cursor position.

- `row`: A number (1 or 2) giving the current row.
- `column`: A number (1 to 40) giving the current column in the character memory.

mizar32.lcd.print([data1] [, data2] ... [datan])

Writes into the LCD character memory starting at the current cursor position. The cursor will advance by one position for each character printed. When it goes past column 40, it moves to column 1 of the other line, (and vice versa when printing right-to-left).

Each item of data can be a string or an integer. Strings are the normal way to display messages of ASCII text. An integer parameter should have a value from 0 to 255 to display a single character, which can be one of the user-defined characters 0-7, the regular ASCII characters 32-125 plus 126 and 127 for right- and left-pointing arrows and the chinese, greek and mathematical symbols with codes 160-255.

mizar32.lcd.cursor(what)

Sets the type of cursor that is displayed at the cursor position or move the cursor left or right.

`what` is string to say what should be done:

- "none", "line" or "block" will display, respectively, no visible cursor, a constant underline or a blinking solid block at the cursor position.
- "left" or "right" move the cursor one position left or right in the character memory and on the display without changing the underlying characters. The display never shifts in this case and, as usual, the cursor wraps between column 40 of one row and column 1 of the other.

mizar32.lcd.display(what)

Turns the physical display on or off, or shifts the displayed characters left or right.

`what` is string to say what should be done:

- "off" and "on" turn the physical display off or back on again. While the display is off it appears blank but the contents of the character memory, the position and type of cursor, user-defined characters and setup mode are all remembered and you can write to the character memory and perform all other operations while the display is off. This allows you to update the display without the viewer seeing too much flickering.
- "left" or "right" shift the displayed characters one place left or right. For example, if it was displaying the usual columns 1-16 and you say mizar32.lcd.display("left"), it will then display columns 2-17: the visible characters move left but the window onto the character memory moves right.

mizar32.lcd.definechar(code, glyph)

Programs one of the eight user-definable characters whose codes are 0 to 7. When it has been defined, a character can be displayed using mizar32.lcd.print(n), where n is a number from 0 to 7. If the character in question is already being displayed, its visible form will change immediately on the display. At power-on, the 8 characters are defined as random garbage.

- `code`: A number (0 to 7) saying which of the characters you wish to redefine.
- `glyph`: A table of up to eight numbers giving the bit-patterns for the eight rows of the character, in order from top to bottom. Each of these number is a value from 0 to 31, to define which of the 5 bits in the row should be black. The pixels' values from left to right are 16, 8, 4, 2 and 1. For example, { 1, 3, 7, 15, 31, 15, 7, 3, 1, 0 } would define a left-pointing solid triangle in the top 7 rows. Extra rows are ignored, and missing rows are blanked.

buttons = mizar32.lcd.buttons()

Tells which of the five user buttons are currently pressed.

- `buttons` is a string containing up to five of the characters L, R, U, D and S to say whether the Left, Right, Up, Down and Select buttons are currently held down. If none are pressed, an empty string is returned. The hardare allows Select to be detected reliably and up to two of the other four: if three of Left, Right, Up and Down are being held, all four are returned.

For example, a fragment of code used in a game could be:

```
pressed = mizar32.lcd.buttons()
if pressed:find("S") then do_select() end
if pressed:find("L") then do_move_left() end
if pressed:find("R") then do_move_right() end
```

Further reading

- The circuit diagram for the LCD module [1]
- The datasheet for Ampire's 162B LCD module [2]
- Interfacing LCD with 8051 [3], an article by Parveen Kumar about the 162B
- The source code for the Mizar32 Display Module's firmware [4]
- The manual for eLua's `mizar32.lcd()` module [5]

References

[1] https://mizar32.googlecode.com/files/Display_module_1.1.1_schematic_file.pdf
[2] http://simplemachines.it/doc/AC-162B.pdf
[3] http://embeddedtutorial.com/2010/01/interfacing-lcd-with-8051
[4] http://github.com/simplemachines-italy/Mizar32_Display_Module
[5] http://www.eluaproject.net/doc/v0.9/en_refman_ps_mizar32_lcd.html

PIO

Introduction

PIO means Programmable Input/Output and this is the simplest way of controlling and measuring digital voltage levels on the pins of the AVR32 processor pins connected to the bus connectors.

To use a GPIO pin as a PIO, you must first set the pin to be an input or an output. If you set it as an input, you can then check the input voltage to see whether it has a low or a high value coming into it, for example to check the position of a switch. If you set it to be an output, you can program it to output a low voltage or a high voltage to control lights, motors or other circuits.

For PIO pins that are inputs, you can also ask that, when the voltage on that pin changes from 0 to 1 or from 1 to 0, this will generate an interrupt. When this happens, the processor will stop what it is doing, run a special piece of code called an interrupt routine, and when it has finished doing that it will go back and continue what it was doing when the interrupt happened.

Lastly, each pin has an optional pull-up resistor that can be enabled so that, if nothing is connected to a pin that is an input, it will float up to logic "1" instead of waving up and down at random. This is the usual way to connect switches or pushbuttons: you program a pull-up resistor on the pin, then connect the switch between the pin and zero volts so that when contact is made you will read a value of 0, and when the contact is open you will read a value of 1.

Hardware view

Any of the AT32UC3A chip's peripheral pins can be read as a digital input or set as an output and programmed to 0 or 1 logic level.

When a pin is set to be an output, a 0 logic output connects the pin to 0 volts, while a logic 1 ("high") value puts 3.3 volts on it with a maximum current supply or drain of 4 milliamperes for both states.

When they are set to be inputs, a voltage level from 0.0 to 0.8 volts reads as a "0" (low) input, and a voltage level from 2.0 volts to 5.0 volts reads as a "1" (high) input. Values from 0.8 to 2.0 volts may read as high or low and are not certain.

Some pins of the Mizar32 can only be used as programmable I/O pins because they are not used for anything else. Others carry signals to various peripheral devices but, if those devices are not being used, the pins can be used as

PIO pins instead.

Other pins are critical to the correct functioning of the processor, for example those used to access the SDRAM, oscillators and other on-board circuitry; if you use those as PIO pins the board will probably crash and need its reset button pressing.

Dedicated PIO pins

Pin	Name	Bus pin	eLua name
PA2	GPIO2	BUS5 pin 11	`pio.PA_2`
PA7	GPIO7	BUS5 pin 12	`pio.PA_7`
PB17	GPIO49	BUS5 pin 8	`pio.PB_17`
PB18	GPIO50	BUS5 pin 9	`pio.PB_18`
PB29	GPIO61	On-board LED	`pio.PB_29`
PB30	GPIO62	BUS6 pin 9	`pio.PB_30`
PB31	GPIO63	BUS6 pin 10	`pio.PB_31`
PX16	GPIO88	User button	`pio.PX_16`
PX19	GPIO85	BUS6 pin 12	`pio.PX_19`
PX22	GPIO82	BUS6 pin 11	`pio.PX_22`
PX33	GPIO71	BUS5 pin 10	`pio.PX_33`

Optional PIO pins

Unused ADC, PWM, SPI or UART pins can also be used as PIO pins. See those sections for the relevant pin names. This brings the total number of usable PIOs to 66.

eLua view

The `pio` eLua module allows you to set any pin as a logic input or to set it as an output and put 0v or 3.3V on it and you can enable a pull-up resistor on any pin.

Driving a pin as an output

This example lights the on-board LED.

```
led = pio.PB_29
pio.pin.setdir( pio.OUTPUT, led )
pio.pin.setlow( led )
```

Note that, straight after a reset, the LED lights up at line 2 because the reset state is that all pins are low and the onboard LED lights when the signal is low. To set a pin as an output whose value starts high, you need to call `pio.pin.sethigh()` before calling `pio.pin.setdir()`.

Reading the voltage on a pin as an input

Here is an example of reading one PIO pin and driving another in response to it. We will read the pin connected to the onboard user button and, as long as it is pressed down, we will make the on-board LED flicker.

```
-- Make the Mizar32 on-board LED flicker as long as the user button is held
led = pio.PB_29
button = pio.PX_16

-- A function to give a short delay of 1/10th of a second
function delay()
  tmr.delay( 0, 100000 )
end

-- Main program

-- First, make sure the LED starts in the "off" position
pio.pin.sethigh( led )
-- Now enable input/output pins
pio.pin.setdir( pio.OUTPUT, led )
pio.pin.setdir( pio.INPUT, button )

while true do
  -- If the button is pressed down...
  if pio.pin.getval( button ) == 0 then
    -- ... turn the on-board LED on and off again
    pio.pin.setlow( led )
    delay()
    pio.pin.sethigh( led )
    delay()
  end
end
```

Programmable pull-up resistors

The user button's electrical circuit is quite simple: when the button is pressed, it connects its PIO pin to zero volts, giving a low input value, and there is a resistor connected between the PIO pin and 3.3 volts so that if the button is not pressed the PIO pin is gently pulled up to 3.3V and reads as a high value.

Other PIO pins that are not connected to anything, if you program them as inputs, will pick up random noise from the surrounding environment and give values that are sometimes high and sometimes low. The programmable pull-up resistors are a way to ensure that, if no signal is connected to an input pin, it will gently be pulled up to a high value instead of floating randomly.

This example turns one of the unused GPIO pins into a PIO input, but ensures that, if nothing is physically connected to it, it will always return a high value of `1`.

If you remove the `pio.pin.setpull()` line from the following code (on my test board, at least), it prints mostly zeroes but if you touch the underside of the Mizar32 board the value flickers between 0 and 1. With the `pio.pin.setpull()` line included, the input value is always 1 unless you connect the pin it to a GND pin (e.g. BUS5 pin 14) with a piece of wire.

```
pin = pio.PA_2    -- Stabilise GPIO2 (connector BUS5 pin 11)
pio.pin.setdir( pio.INPUT, pin )
pio.pin.setpull( pio.PULLUP, pin )
while true do
  io.write( pio.pin.getval( pin ) )
end
```

Although eLua also has a similar primitive `pio.PULLDOWN`, the AVR32 chip used in the Mizar32 does not have programmable pull-down resistors in its hardware, so using this will provoke an error message.

To disable the pull-up resistor again, you use

```
pio.pin.setpull( pin, pio.NOPULL )
```

Open-collector outputs

Another use for the programmable pull-up resistors is to implement "open-collector outputs". In this scheme of things, a pin can be in one of two states: either it is being driven as a 0 output or it being read as an input. The pull-up resistor ensures that, if no one is driving it as an output, everyone will read it as a high value. This is used when several computers need to communicate over a single signal wire in such a way that any computer can talk with any other one without needing a master-slave relationship or a way to negotiate who is controlling the bus. Using this system, any computer can read the wire to see if its value is high or low, and any computer can drive the wire to a low value, to be read by all the others. This is different from driving a wire as a high or low output because if one computer is driving it high and another is driving it low at the same time, that could damage the computers in question and would certainly result in garbled communication.

A simple example would be a system to turn a house light on and off in a way that can be activated by any one of several switch units. The I2C bus, instead, is an advanced example that uses open-collector outputs on its signal wires so that any one of the computers on an I2C bus to talk with any other one without ever causing conflicting signals on the bus wires.

The following code implements an open-collector output on a PIO pin, giving one function to configure it as an OC pin, one function to drive it as a low output and one to set it as an input and tell you what value is on the wire at the moment:

```
-- Turn a PIO pin into an open-collector output.
-- Call this function once before using the other two to drive and read the pin.
function oc_setup( pin )
  -- OC output starts as an input, not driving the bus wire
  pio.pin.setdir( pio.INPUT, pin )
  -- arrange that, when it is an input and no one is driving it, it will float high
  pio.pin.setpull( pio.PULLUP, pin )
  -- and that when we set it as an output, it will drive a low value
  pio.pin.setlow( pin )
end

-- Drive a low output value onto the pin.
-- The low output value is already programmed during setup()
-- so we only need to enable it as an output.
function oc_drive_low( pin )
  pio.pin.setdir( pio.OUTPUT, pin )
end
```

```
-- Make the pin an input and return the value on the bus wire
function oc_read( pin )
  pio.pin.setdir( pio.INPUT, pin )
  return pio.pin.getval( pin )
end
```

Lua interrupts

You can arrange for your own Lua function to be called whenever the logic level changes on a GPIO pin, without having to keep checking the pin all the time.

You can ask for your interrupt function to be called either when the logic level on a GPIO pin goes from `0` to `1`, when it goes from `1` to `0` or both.

The following example creates an interrupt function and arranges for it to be called every time the user button is pressed. The circuitry of the user button is such that its GPIO pin is high when the button is not pressed and low when it is pressed, so we ask for our interrupt routine to be called when the logic level of the pin goes from high to low.

```
-- Example program to show the use of Lua interrupts on PIO edges.
-- This flashes the on-board LED every time the user button is depressed.

-- Which PIO pins are the button and the LED connected to?
button = pio.PX_16
led    = pio.PB_29

function when_pressed( resnum )
  -- flash the onboard LED
  pio.pin.setlow(led)
  for i=1,10000 do end  -- for about 1/100th of a second
  pio.pin.sethigh(led)
end

-- Enable the LED as an output, starting in the "off" state.
-- and the button as an input.
pio.pin.sethigh( led )    -- off
pio.pin.setdir( pio.OUTPUT, led )
pio.pin.setdir( pio.INPUT, button )

-- Set our interrupt handing function and make it sensitive to a
-- change from high to low of the PIO pin connected to the user button.
cpu.set_int_handler( cpu.INT_GPIO_NEGEDGE, when_pressed )
cpu.sei( cpu.INT_GPIO_NEGEDGE, button )

-- Do something for about ten seconds while the test runs
for i=1,10000000 do end

-- disable the interrupt
cpu.cli(  cpu.INT_GPIO_NEGEDGE, button )
```

```
-- and remove our interrupt handler function.
cpu.set_int_handler( cpu.INT_GPIO_NEGEDGE, nil )
```

Any GPIO pin can be made to react to edge-based interrupts, whether they are inputs, outputs or have some completely different function.

You can enable the positive edge interrupt function for a pin, the negative edge interrupt function, or both. If you enable both, you can either make both kinds of interrupt be handled by the same function, or by two different functions.

Decoding pin numbers

Functions like `pio.PB_29` return numbers that are used internally in eLua to identify the GPIO pins. You do not need to know anything about the values of these numbers except in one circumstance: when you have several edge-triggered interrupts enabled on different GPIO pins at the same time.

You see, you can only tell `cpu.set_int_handler()` to call a single function for all positive edge interrupts and one other function (or the same one) for all negative-edge interrupts, but you can find out which GPIO pin caused the interrupt by inspecting the parameter that is passed to your interrupt-handling function, which we have called `resnum` in the code examples above.

This is the "resource number", the internal number that eLua uses to represent the pin, and it is the same as the value returned by `pio.PX_16` and so on.

So you can say, in your interrupt function,

```
if resnum == button ...
```

or

```
if resnum == pio.PA_3 ...
```

however, if you have enabled many pin interrupts and wish to decode it into a port number and a pin number, you can do so using

```
port, pin = pio.decode( resnum )
```

For pins on Port A, `port` will be `0` and `pin` will be from 0 to 31.
For pins on Port B, `port` will be `1` and `pin` will be from 0 to 31.
For pins on Port C, `port` will be `2` and `pin` will be from 0 to 5.
For pins on Port X, `port` will be `23` and `pin` will be from 0 to 39.

Further reading

- The PIO section of the eLua manual [1] for details of all eLua `pio.*` functions
- The Atmel AT32UC3A datasheet [2] Chapter 22: General-Purpose Input/Output Controller (GPIO).
- Open collector [3] article on Wikipedia

References

[1] http://www.eluaproject.net/en_refman_gen_pio.html
[2] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf#22
[3] http://en.wikipedia.org/wiki/Open_collector

PWM

Introduction

PWM stands for Pulse Width Modulation, which is a way of generating square-wave output signals with different frequencies and a different proportion of each waveform cycle being a low output or a high output.

When you set up a PWM output, you specify the frequency of the output (how many times it goes high and low per second) and its duty cycle (what percentage of each cycle is spent as a high output).

The most common way that PWM outputs are used is with a fixed frequency and varying the duty cycle, thereby creating a crude kind of digital-to-analog converter to control the brightness of a light or the power delivered by a motor. If the output signal is amplified to control high-power devices, this results in a more efficient system because if an amplifier is fully on for half the time and fully off for the other half, this wastes less energy than being on at half power all the time.

Another use is to produce simple sound effects or organ notes by using a fixed duty cycle and changing the frequency to generate sounds with a square waveform. In this case, changing the duty cycle changes the tonal quality of the sound that is produced.

Lastly, the PWM system can be used to generate interrupts at fixed time intervals by enabling the PWM interrupt. This way, each time one cycle of PWM output is completed, the processor stops what it is doing, runs a special piece of code called an interrupt routine and when this is done it goes back to continue what it was doing before the interrupt occurred. Though the hardware is capable of this, PWM interrupts are not implemented in eLua yet.

Hardware view

The Mizar32 has seven independent PWM outputs, though on the circuit diagram only PWM0 to PWM5 are labelled as such. PWM6, if enabled, appears on the pin labelled "GPIO50".

Bus pins

PWM	AVR32 pin	Bus pin	eLua name	Notes
PWM0	PB19	BUS4 pin 7	`pio.PB_19`	
PWM1	PB20	BUS4 pin 8	`pio.PB_20`	Also connected to JTAG pin 7 "EVT0"
PWM2	PB21	BUS1 pin 8	`pio.PB_21`	
PWM3	PB22	BUS6 pin 1	`pio.PB_22`	
PWM4	PB27	BUS6 pin 2	`pio.PB_27`	
PWM5	PB28	BUS6 pin 3	`pio.PB_28`	
PWM6	PB18	BUS5 pin 9	`pio.PB_18`	On Mizar32 bus, the pin is called "GPIO50"

eLua view

eLua's `pwm` module [1] is used to program the PWM pins.

PWM6 is not available to eLua users as it it used internally to provide the system timer `tmr.SYS_TIMER`.

Output frequency and duty cycle

Two functions are used get some PWM output on a pin.

```
pwm.setup( channel, frequency, duty_cycle )
```

sets the output frequency and duty cycle, where `channel`, from 0 to 6, is the PWM channel you wish to use, `frequency`, from 1 to 1000000, determines the frequency of the output waveform in cycles per second and `duty_cycle`, a value from 0 to 100, determines what percentage of each cycle the output value of the waveform will spend at at the high level, and

```
pwm.start( channel )
```

sets the oscillator running to produce a cyclic output waveform.

If `pwm.setup()` is called before `pwm.start()`, as we have just described, the corresponding pin on the bus becomes an output pin outputting zero volts when setup() has completed and then, when `pwm.start()` is called, the output waveform goes high, remains high for the specified percentage of the cycle, then goes low for the rest of the cycle, repeating until `pwm.stop()` is called for the same channel.

Alternatively, if `pwm.start()` is called first, the pin remains an input until `pwm.setup()` is called, at which point it becomes an output and starts producing the waveform.

When `pwm.stop()` is called, the PWM output always completes the current cycle; the "stopped" state means that when the current cycle completes, it will not start a new one, so you can get exactly one complete cycle of output like this:

```
pwm.setup( 0, 10, 50 )
pwm.start( 0 )
pwm.stop( 0 )
```

This program code will complete before the whole cycle has been output, and note that, when it is "stopped", the pin continues outputting 0 volts.

Clock frequency

A further function

```
pwm.setclock( id, freq )
```

allows you to set the PWM clock frequency, which is a higher frequency than the output frequency and determines the time-granularity of the output waveform. In effect, the PWM hardware only decides whether to change the output value of each PWM pin once every `1/freq` of a second, so a higher clock frequency gives better precision in time and frequency.

The PWM clock frequency also detemines the lowest and highest possible frequencies of the PWM output waveforms: a lower clock frequency allows the output frequency to be lower.

Possible values for the clocking frequency on Mizar32 are from 63Hz to 16500000Hz. If you ask for values outside this range, it will set the lowest or highest of these two, accordingly. Within this range, not all frequencies are available; of the available frequencies it sets the one that is closest to what you asked for, and both

`pwm.setclock()` and `pwm.getclock()` return an integer which is the actual frequency that has been set into the PWM clock.

Although eLua's `pwm.setclock()` and `pwm.getclock()` accept a first parameter to say which channel you want to set the clock for, in reality the hardware only has one clock for all the channels, so setting the clock frequency for any one channel will change the clock frequency for all of them, as well as changing the current output frequency of any that are running. For this reason `pwm.setclock()` is usually only called once before setting up the individual channels.

Range and precision of PWM output frequencies

In eLua, `pwm.setup()`'s parameters only take notice of the whole part of numbers, ignoring any fractional part, so the lowest frequency you can ask for is one cycle per second and the maximum precision is one hertz.

Furthermore, the hardware can only generate certain frequencies (the ones that are exact divisors of the clock frequency) and `pwm.setup()` returns an integer value which is the closest whole number to the actual output frequency that was set, which may be different from the frequency that you asked for.

For example, with the default clock frequency of one megahertz, you can set any integer frequency from 1 to 1037, but 1038 gives you 1039, then more and more values become more and more inaccurate until we come to the highest available frequencies of 250000Hz, 333333Hz, 500000Hz and 1000000Hz, which are the clock frequency divided by 4, 3, 2 and 1.

At the highest clock rate of 16500000Hz instead, every integer frequency from 16Hz to 4105Hz can be obtained and above this frequency a wider range of values are available, which would be more suitable for applications such as an electric organ, where accuracy in the frequency of high notes is more important than generating extremely low notes.

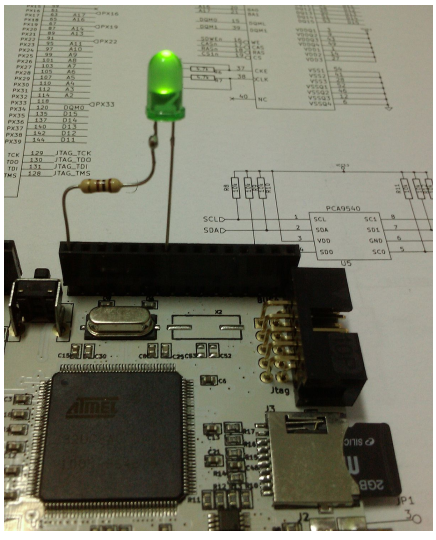
Using PWM pins as PIO pins

If a particular PWM output is not being used for PWM output, you can use it as a generic PIO pin simply by calling the functions in the `pio` module on that pin.

You would also use this if you wanted a pin that you had used for PWM output to stop outputting a voltage at all: you would call

```
pio.pin.setdir( pio.INPUT, pio.PB_xx )
```

where `xx` is the pin number of that PWM channel in the table above.



Example circuit for the Mizar32 PWM LED fader

Example code

```
-- Make a LED slowly fade up and down forever

-- Connect a LED in series with a
-- 330 ohm resistor between the PWM0 pin
-- (BUS4 pin 7) and GND (BUS4 pin 1)

local pwmid = 0     -- Which channel to use?
local speed = 3000  -- PWM frequency in Hz
local fadetime = 1  -- How many secs to fade up?
local tmrid = 0     -- Which timer for the delay?
local nsteps = 100  -- How many steps in the fade?

-- Calculate the delay for each steps, in microseconds
local delay = pwm.getclock( tmrid ) * fadetime / nsteps

pwm.start( pwmid )

while true do
  -- Fade the LED up
  for duty = 0, nsteps do
    pwm.setup( pwmid, speed, duty )
    tmr.delay( tmrid, delay )
  end

  -- Fade the LED back down again
  for duty = nsteps, 0, -1 do
    pwm.setup( pwmid, speed, duty )
    tmr.delay( tmrid, delay )
  end
end
```

Further reading

- The eLua PWM module reference manual [1];
- The Atmel AT32UC3A datasheet [2] Chapter 32: Pulse Width Modulation Controller (PWM).

References

[1] http://www.eluaproject.net/en_refman_gen_pwm.html
[2] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf#32

RTC

Introduction

The Real Time Clock chip on Mizar32's add-on ethernet board keeps track of the time of day and the date, whether the Mizar32 is powered on or not. It does this by having a small rechargeable battery and a crystal that vibrates at 32768 cycles per second, which it divides down to get a pulse that beats exactly once per second and uses this signal to keep the seconds, minutes and hours up to date, as well as the date, month and year.

Hardware view

The ethernet board has a 32768 Hz crystal and a DS1337 or a PCF8563 chip which is connected on the I2C bus. The two different chips have the same pin-out and almost identical functions; the reason we support both is due to a production error by one of our suppliers who printed DS1337 on the casing of a batch of PCF8563 chips.

Which actual chip is present is identified by the slave address that it responds to in the main I2C bus: the DS1337 responds to 7-bit decimal address 104, while the PCF8563 responds to slave address 81.

Both chips respond up to a speed of up to 400kHz on the I2C bus. We recommend using 100kHz, the default I2C that is set in eLua.

For the register layout and how to address the devices using the raw I2C protocol, please see their datasheets.

eLua view

Simplemachines has added a module to eLua to set the time and to read it.

This eLua module is included from Mizar32's 2013 firmware release of eLua 0.9.

Setting the time

You set the time using the function `mizar32.rtc.set(t)` where `t` is a table which can have any of the following fields present:

Field	Value	Meaning
`sec`	0-59	Seconds
`min`	0-59	Minutes
`hour`	0-23	Hours (24-hour clock)
`day`	1-31	Day of month
`month`	1-12	Calendar month
`year`	1900-2099	Year
`wday`	1-7	Day of week

For example,

```
now = {year=2013, month=9, day=16, hour=0, min=32, second=59}
mizar32.rtc.set( now )
```

When some fields are present but not others, `mizar32.rtc.set()` sets those parts of the time and date and leaves the other parts with the same value as they had before.

The day of the week field, `wday`, uses the following values to represent the seven days of the week:

Value	1	2	3	4	5	6	7
Meaning	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday

It is up to you to write the correct value into this field if you wish to use it; it is not set automatically from the date.

Reading the current time and date

The function `mizar32.rtc.get()` returns a table with the seven fields described above.

The following code example makes a clock appear on the LCD display.

```
mizar32.lcd.reset()
while true do
  local t = mizar32.rtc.get()
  mizar32.lcd.goto( 1, 5 )
  mizar32.lcd.print( string.format("%02d:%02d:%02d",
                     t.hour, t.min, t.sec ) )
  mizar32.lcd.goto( 2, 4 )
  mizar32.lcd.print( string.format("%02d/%02d/%04d",
                     t.day, t.month, t.year ) )
end
```

Further reading

- The manual for the `mizar32.rtc` eLua module [5]
- The DS1337 datasheet [1]
- The PCF8563 datasheet [2]

References

[1] http://datasheets.maxim-ic.com/en/ds/DS1337-DS1337C.pdf
[2] http://www.nxp.com/documents/data_sheet/PCF8563.pdf

SPI

Introduction

Another popular bus that is used to connect chips on the same board or in a modular design with more than one board inside the same box, is the SPI bus, which stands for Serial Peripheral Interface, sometimes also referred to as a “four wire” interface.

SPI is a synchronous serial data link that transmits and receives data at the same time and can achieve much higher data rates than I2C. It is ideally suited to streaming data applications such as reading/writing SD cards or sending/receiving audio. Normal frequencies are in the order of tens of Megahertz, for example a common 8 pin 32Mbit Atmel dataflash connected to SPI bus can operate at 66MHz.

There is no formal SPI standard. The bus was introduced for the first time by Motorola in 1985 and well documented the first time with the M68HC11 micro-controller. Since then it has been adopted by many other silicon manufacturer and is widely used in the embedded industry for many different applications. For example, all SD memory cards can be used with a SPI interface.

Other examples of real world applications of the SPI bus are to connect with Analog to Digital Converters, Digital to Analog Converters, digital sensors, EEPROMs, Flash memories, touch screen controllers, digital potentiometers, Real Time Clocks, switches, serial port controllers and USB controllers or simply to establish point to point communication between different micro-controllers in the same board.

SPI is a very simple communication protocol with almost no high-level protocol, which means that there is very little overhead and data can be transmitted at high rates in both directions at once.

Like I2C, SPI devices also communicate using a Master-Slave relationship, but instead of selecting a slave device by sending its address in the data, it selects it with an extra wire that goes to each slave device, which is called the Chip Select signal.

Unlike I2C, there can only be one Master device on each SPI bus.

Like I2C, the Master is the one that sends the clock pulses, but at each pulse, 1 bit data is sent from Master to Slave and one bit is sent from Slave to Master.

SPI uses 4 wires to communicate between a master and a slave device:

- The Clock (SCLK)
- Master Output Slave Input (MOSI)
- Master Input Slave Output (MISO)
- Chip Select (CS)

The first three wires are common to all devices on the bus and there is a separate Chip Select wire for each Slave device on the bus.

For example, a Master device with a single Slave needs four wires: three for SCLK, MOSI, MISO and a one for the Chip Select.

Every additional Slave, in an Independent Slave configuration, requires another GPIO pin as Chip Select to select another Slave from the Master.

In addition to the clock frequency in MHz, which corresponds to the bit rate, you must also defined the Clock Polarity (CPOL) and the Clock Phase (CPHA) with respect to the data.

There are 4 modalities for SPI communication and the modality that is chosen must correspond between the Master and Slave devices that are communicating in any one moment.

- Mode 0 with CPOL=0 and CPHA=0

- Mode 1 with CPOL=0 and CPHA=1
- Mode 2 with CPOL=1 and CPHA=0
- Mode 3 with CPOL=1 and CPHA=1

Communication is initiated by the Master device. It configures its clock frequency to be equal or less than the maximum frequency supported by the slave that it wants to communicate with. Then the desired Slave is selected with Chip Select line, by pulling it to a low state, then the Master starts to issue clock pulses.

A full duplex data transmission on two shift registers occur during each clock cycle. That means the master sends a bit on the MOSI line; the slave reads it from that same line and the slave sends a bit on the MISO line and the Master reads it from that same line.

If more data needs to be exchanged, the shift registers are loaded with new data and the process is repeated. Then, when no more data needs to be transmitted, the master stops the clock.

This is basically all that is defined for the SPI protocol. SPI does not define any maximum data rate, nor any particular addressing scheme; it does not have an acknowledgement mechanism to confirm receipt of data and does not offer any flow control.

Hardware view

The AT32UC3A system-on-chip on the main board includes two SPI interfaces, SPI0 and SPI1, with SPI0 on the left bus and SPI1 on the right. Each interface has four chip-select lines, allowing a maximum of eight SPI devices; one of these (SPI1 CS0) is used to talk to the SD/MMC card and three of them (SPI0 CS0 and SPI1 CS1 and CS2) are available on the bus connectors.

Bus pins

Signal	GPIO	Bus pin	eLua name
SPI0_CS0	PA10	BUS1&3 pin 12	`pio.PA_10`
SPI0_MISO	PA11	BUS1&3 pin 13	`pio.PA_11`
SPI0_MOSI	PA12	BUS1&3 pin 14	`pio.PA_12`
SPI0_SCK	PA13	BUS1&3 pin 15	`pio.PA_13`
SPI1_CS1	PA18	BUS3 pin 11	`pio.PA_18`
SPI1_CS2	PA19	BUS4 pin 12	`pio.PA_19`
SPI1_MISO	PA17	BUS4 pin 11	`pio.PA_17`
SPI1_MOSI	PA16	BUS4 pin 10	`pio.PA_16`
SPI1_SCK	PA15	BUS4 pin 9	`pio.PA_15`

eLua view

eLua reads and writes the SD card over SPI and provides access to the files on the card using Lua's `io` module using filenames that start with "`/mmc/`".

An `spi.*()` module is included in eLua to talk low-level SPI protocol and is presumed to work but is untested.

Further reading

- *Introduction to Serial Peripheral Interface* [1], David Kalinsky and Roee Kalinsky, in Embedded's electronics blog, 1 Feb 2002.
- EEHerald's Online course on Embedded Systems, Module 12: SPI Bus interface [2]
- The Atmel AT32UC3A datasheet [3], section 23: "Serial Peripheral Interface"
- Wikipedia - Serial Peripheral Interface [4]

References

[1] http://www.embedded.com/electronics-blogs/beginner-s-corner/4023908/Introduction-to-Serial-Peripheral-Interface
[2] http://www.eeherald.com/section/design-guide/esmod12.html
[3] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf#23
[4] http://en.wikipedia.org/wiki/SPI

Timers

Hardware view

The AT32UC3A has three 16-bit countdown timers which can run at three different frequencies, independent of each other. When they are set to run at a high frequency, the timing accuracy is higher but the longest possible delay is shorter.

Only a small set of clock frequencies is available, which are divided down from the PBA bus frequency of 16.5MHz:

Divisor	Clock frequency	Longest delay
PBA/2	8.25 MHz	7.94 ms
PBA/8	2.0625 MHz	31.77 ms
PBA/32	515.625 kHz	0.127 second
PBA/128	128.906 kHz	0.508 second

Apart from these four selections, the chip also has provision for running any of the timers from an external 32768Hz real-time precision crystal. The Mizar32 board from version 1.3.2 has a space for this component (X2) where it may be mounted.

eLua view

eLua provides a `tmr` library to access the real-time counters.

Three types of timer

Hardware timers

The first two hardware timers are directly accessible, using timer IDs `0` and `1`. eLua uses a PBA frequency of 16.5MHz and by default the clock rates are set to lowest available frequency of 128906Hz, giving a timing precision of about a hundred thousandth of a second and a maximum delay of just over half a second.

```
tmr.delay( 0, 100000 )  -- Wait for 1/10th of a second using the first timer
```

These can be used to achieve short delays of high accuracy.

For either of the first two timers, you can set a higher clock rate than the default of 129kHz. However, only four values are supported as shown in the table above with PBA frequency = 16.5MHz. Other values will set the arithmetically nearest available frequency.

```
-- Set the highest possible timing precision for timer 1,
-- giving a maximum delay of 7.94ms
freq = tmr.setclock( 1, 10000000 )
print( freq )
```

prints `8250000`

Virtual timers

The third hardware timer cannot be accessed directly, but instead is used to generate four "virtual timers" whose timer ids are `tmr.VIRT0` to `tmr.VIRT3`. These have a lower tick frequency and accuracy - ten times per second - but can be used to create delays of up to 35 minutes in integer eLua or 142 years in floating point eLua.

```
tmr.delay( tmr.VIRT0, 5000000 )  -- Wait for five seconds
```

These are used to achieve longer delays of lower accuracy but the clock rate of the virtual timers cannot be changed.

System timer

From the 20120123 firmware release, there is a third timer mechanism, the system timer `tmr.SYS_TIMER` which has an accuracy of one millionth of a second and can be used to give high-precision delays and timings up to 35 minutes with integer Lua and up to 142 years in floating point Lua, but you cannot change the clock frequency of the system timer, and it cannot be used to generate interrupts (see below).

Timer operations

Delays

All three types of timer can be used to make your program wait for a specified length of time, as shown in the examples above. The precision of the delay and the maximum delay available depend on the type of timer used. In general, the system timer is the best for all types of delay, as it has high precision and can perform long delays.

Measuring time

Sometimes it can be useful to know how much time has elapsed since some previous moment, for example to measure the speed of your code or when you need to take some decision after a certain amount of time has passed but also need to do something else while you are waiting.

This example measures people's reaction time by printing "Go!" on the console and then seeing how long it takes them to press a key in response. We will use the system timer for this.

```
print "Welcome to the reaction timer. When I say Go!, press [Enter]."
print "Press q [Enter] to quit."
repeat
  timer = tmr.SYS_TIMER
  print( "Ready?" )
  -- Wait for a random time from 2 to 5 seconds
  tmr.delay( tmr.SYS_TIMER, 2000000 + math.random( 3000000 ) )
  print( "Go!" )
  start_time = tmr.read( timer )
  answer = io.read()  -- wait for them to press Enter
  end_time = tmr.read( timer )
  print( "You reacted in " .. tmr.gettimediff( timer, start_time, end_time ) .. " microseconds" )
until answer == "q"
```

Of course, if you press Enter before it says Go!, it will say you reacted incredibly quickly...

Timer interrupts

You can arrange that a Lua function be called either regularly or after a certain time has elapsed - you can then go and do other things while this happens. The following example shows how to generate an interrupt once every half second using hardware timer 0. Each time the timer causes an interrupt, a Lua function of ours, called `irq_handler` here, is called to flash the on-board LED.

```
-- Test timer interrupts handled in Lua.
-- Flash Mizar32's onboard LED twice a second under Lua interrupt control.

led = pio.PB_29   -- Which PIO pin is the LED connected to?
timer = 0         -- which timer to use to generate the interrupts?
period = 500000   -- how often, in microseconds, should it make an interrupt?

function int_handler( resnum )
  -- flash the onboard LED
  pio.pin.setlow( led )
  tmr.delay( nil, 10000 )  -- on for 1/100th of a second
  pio.pin.sethigh( led )
end

pio.pin.sethigh( led )             -- prepare the LED as starting "off"
pio.pin.setdir( pio.OUTPUT, led )  -- Make the LED pin an output

-- tell eLua which function it should call every time the timer times out
cpu.set_int_handler( cpu.INT_TMR_MATCH, int_handler )
```

```
-- enable that Lua interrupt
cpu.sei( cpu.INT_TMR_MATCH, 0 )
-- and start the timer to cause an interrupt once every half second
tmr.set_match_int( timer, period, tmr.INT_CYCLIC )

-- Busy-wait for about ten seconds while the test runs
for i=1,10000000 do end

-- disable the interrupt-generating timer
tmr.set_match_int( timer, 0, tmr.INT_CYCLIC )
-- disable the Lua interrupt
cpu.cli( cpu.INT_TMR_MATCH, timer )
-- and remove our interrupt handler function
cpu.set_int_handler( cpu.INT_TMR_MATCH, nil )
```

To generate a single interrupt after a certain time has elapsed instead, you use

```
tmr.set_match_int( timer, period, tmr.INT_ONESHOT )
```

instead of

```
tmr.set_match_int( timer, period, tmr.INT_CYCLIC )
```

Note that timer interrupts only work with hardware timers (0 and 1) and virtual timers (`tmr.VIRT0` to `tmr.VIRT3`); the system timer cannot generate interrupts. The choice of which kind of timer to use depends on the time-precision that you require and the length of time you need to deal with. The hardware timers have a maximum period of half a second but are precise to 100,000th of a second, while the virtual timers are only precise to 1/10th of a second but can deal with periods up to 35 minutes in integer eLua or 142 years in floating point eLua.

Further reading

- The tmr module in the elua reference manual [1]
- Virtual timers in the eLua reference manual [2].
- The Atmel AT32UC3A datasheet [3] Chapter 31: Timer/Counter (TC)

References

[1] http://www.eluaproject.net/en_refman_gen_tmr.html
[2] http://www.eluaproject.net/en_arch_platform_timers.html#virtual_timers
[3] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf#31

UART

Introduction

The Universal Asynchronous Receiver Transmitter circuit or simple UART (pronounced "you art") is one of the more common interfaces used to communicate serially. Some computers, such as the IBM PC, used an integrated circuit called a UART to convert characters to and from asynchronous serial form. By ‘serial’, we mean that data is transmitted one bit at a time.

This circuit was at the base of Recommended Standard - 232 (RS-232 in short) that also defined the physical external COM port in IBM PC. The latest revision to RS-232 is the EIA RS-232 that was done in 1997.

In recent years, the PC has lost the RS-232 port (maybe only externally), replacing it in favour of the USB, but RS-232 is still in wide use including many variants, from RS-485 used in the industry to SpaceWire used in the Space Station. In its simpler forms, it is currently widely adopted in the embedded industry.

To understand a bit more the UART circuit and the RS-232 let’s take a look at the past. Serial transmission is very old, born with the first Teleprinters (TTYs). The teleprinters were in use from 1914 to recent times, with the last company manufacturing TTY closing in 1990, while their last use is reported for air lines bulletin in recent years and most of the terms used come from the TTY world. Mark and Space for example are terms describing logic levels in teleprinter circuits. The baud rate, or Symbol rate, is the speed of a serial connection and is based on multiples of the rates for electromechanical teleprinters.

Today, although it is becoming less common in personal computers, it remains one of the most common peripherals found on micro-controllers (MCU) and is used for communication with external devices and systems, to talk with on-board serial devices or to make links between boards, between boxes or between embedded boards and PCs with an RS-232 port.

RS-232 globally specifies:

- wiring
- signal voltages
- signal functions
- signal timing
- protocol for information exchange
- UART configuration

With a micro-controller oriented view we will go now to examine all the above points.

The UART is a full duplex communication channel, in asynchronous mode each line is indipendent of the others in terms of its main function. The RX pin can receive data regardless of the TX pin’s activity and vice versa.

Usually the UART wires from the micro-controller are labelled TX, RX, GND (for transmit, receive and ground) in 3 wire configuration (without flow control implemented) and 5 wires TX, RX, GND, RTS, CTS, with hardware flow control, where RTS stay for Request To Send and CTS for Clear To Send.

The signal is described as a positive voltage to communicate the logic value 0 called a “Mark”, and a negative voltage to communicate the logic value 1, called a “Space”.

This signals are usually too weak in current and voltage as get out from the micro-controller and the standard specify that should be used voltages from ±5V to ±15V with length of wire of 10 meters, so how to interface to the UART lines, that output ±3V with very low current?

The MAX232 chip on the Mizar32's Serial UART add-on board is a TTL to RS-232 level converter which pumps the voltage from ±3V to ±15V.

The signal timing is measured in Baud, where one baud in binary communication corresponds to one bit per second, so at 9600 baud rate, we have 9600 bits per second with a time frame to describe each bit of 104 μs 1/9600. Often the clock will run at 16 times the baud rate to allow the receiver to do centre sampling.

The data exchange protocol is very simple.

The packet begins with start bit, which is a logic 0, being transmitted/received first. In the software side, this bit is important as we can poll the RX pin for this bit to signal that a packet of data is coming. The data bits or the payload may or may not have a parity bit, then the packets end with a logic 1, one or two stop bit.

0	start)
XXXXXXX	(7 or 8 bit of data)
X	1 optional bit of parity
1	one or two stop bits

Note that the least significant bit of the byte is sent first, whereas we normally write numbers with the LSB on the right, so we should read it from right to left.

Regarding configuration parameters, a common configuration (usually stored in a register) is 9600/8n1 that means for the serial port: 9600 baud, 8 bits data, no parity bit, 1 stop bit. Obviously both UART should be configured in the same way in order to communicate.

Hardware view

The Mizar32 has two serial ports available on the bus connectors, UART0 on the right bus and UART1 on the left bus. UART0 has just data (TXD, RXD) and hardware flow control (CTS, RTS) signals, while UART1 also has modem control signals (DSR, DTR, DCD, RI).

In the Atmel documents, these are called "USART"s because they can also be programmed into synchronous mode to work as additional SPI ports.

Bus pins

Signal	GPIO	Bus pin	eLua name
UART0_RX	PA0	BUS4 pin 3	`pio.PA_0`
UART0_TX	PA1	BUS4 pin 4	`pio.PA_1`
UART0_RTS	PA3	BUS4 pin 5	`pio.PA_3`
UART0_CTS	PA4	BUS4 pin 6	`pio.PA_4`
UART1_RX	PA5	BUS3 pin 3	`pio.PA_5`
UART1_TX	PA6	BUS3 pin 4	`pio.PA_6`
UART1_DCD	PB23	BUS3 pin 5	`pio.PB_23`
UART1_DSR	PB24	BUS3 pin 6	`pio.PB_24`
UART1_DTR	PB25	BUS3 pin 7	`pio.PB_25`
UART1_RI	PB26	BUS3 pin 8	`pio.PB_26`
UART1_CTS	PA9	BUS3 pin 9	`pio.PA_9`
UART1_RTS	PA8	BUS3 pin 10	`pio.PA_8`

RS232/RS485 serial add-on board

The add-on serial board has two banks of switches, DIP1 and DIP2 to select between RS232 and RS485 modes.

RS232 mode

If all switches of DIP1 are up and all of DIP2 down, it converts the bus signals to RS232 levels on its female DB9 connector J7. This connector is configured as DCE equipment, which is the opposite of a PC serial port, so a cable to communicate with a PC should connect the same pins at each end; a null modem cable is not required. Connecting other DCE equipment to it, like a modem or GPS receiver, requires interchanging of TX and RX, for example with a null modem cable.

Note that in the 1.1.1 version of the serial port the CTS and RTS pins were swapped over by mistake, so to get the right connections here you need to modify either the board or the cable. However, hardware flow control is not yet working in eLua so it makes no difference; see issue #29 [1].

Signal	Bus pin	UART module rev. 1.0 DB-9F pin	UART module rev. 1.1.1 DB-9F pin
UART0_RX	P5 pin 3	Pin 3 (input)	Pin 3 (input)
UART0_TX	P5 pin 4	Pin 2 (output)	Pin 2 (output)
UART0_RTS	P5 pin 5	Pin 8 (output)	Pin 7 (output)
UART0_CTS	P5 pin 6	Pin 7 (input)	Pin 8 (input)
GND	Various	Pin 5	Pin 5

RS485 mode

If all switches of DIP1 are down and all of DIP2 up, the board's DB9 connector is disabled and RS485 signals appear on the four screw terminals.

This interface allows up to 32 RS485 devices to be connected to the same wires with a cable length of up to 1200m at 100 kbit/sec.

Currently, RS485 mode is not supported in eLua; see issue #77 [2].

eLua view

Simple I/O

Depending on which firmware you have, UART0 may be used for the Lua console (configured at 115200 baud, 8 data bits, 1 stop bit, no parity) and Lua's default input and output files are the console, so functions like "`print()`" and "`io.write()`" can be used to output characters on the serial port (with, and without, a trailing CR-LF newline sequence, respectively)

```
-- Greet the user
io.write( "What's your name? " )     -- Issue a prompt (with no trailing newline)
name = io.read()                     -- Read a line of input and store it in "name"
print( "Hello, " .. name .. "!" )    -- Salute them
```

Low-level I/O

UART0 can also be accessed (and UART1 must be accessed) using the lower-level `uart` eLua module, which gives a higher degree of control over the UART's behaviour.

The following example sets a different baud rate on UART0 and spits out a prompt character twice a second until a character is received in reply. To do this, it uses the `uart.setup()` function and the optional timeout parameter of the `uart.getchar()` function.

```
-- Prompt a 9600 baud serial device until we receive a character in reply
uartid = 0          -- Which UART should we be talking on?
timeout = 500000    -- Prompt once every half second
timerid = 0         -- Use timer 0 to measure the timeout
prompt = "U"        -- The prompt character (0x55 : binary 01010101)

uart.setup( uartid, 9600, 8, 0, 1 )    -- Configure the UART
repeat
  uart.write( uartid, prompt )
  reply = uart.getchar( uartid, timeout, timerid )
until reply ~= ""
```

Hardware flow control

Note that enabling hardware flow control with

```
uart.set_flow_control( uartid, uart.FLOW_RTS + uart.FLOW_CTS )
```

does not work yet. See issue #29 [1].

Input buffer

When a UART receives a character it will remember it until you ask for its value using `uart.getchar()`. However, if a second character arrives before you have read the first one, the first one will be forgotten.

You can get round this by enabling a UART buffer, for example:

```
uart.setup( 1, 115200, 8, 0, 1 )    -- Configure UART 1
uart.set_buffer( 1, 1024 )          -- and give it an input buffer
```

and this will allow the UART to receive up to 1024 characters and remember them all even if you haven't read the first one yet (the 1025th character will provoke an error message and will be forgotten.

UART buffer sizes must be a power of two, i.e. 1, 2, 4, 8, 16 and so on up to a maximum of 32768 characters.

Some firmware uses UART0 as the eLua console. When this is the case, a buffer is always enabled on this UART.

Lua interrupts

When input buffering is enabled on a UART, an interrupt is generated every time a character is received. This interrupt saves the character in the buffer you asked for, until your program is ready to read it.

If you use firmware with Lua interrupts (included in the Mizar A and B firmware from the 20120123 elua 0.8 release) you can also arrange for your own piece of code to be called every time a character is received.

The following example code quickly flashes the on-board LED every time a character is received:

```
-- Test UART interrupts handled in Lua.
-- Should flash the onboard LED each time a character is received.

led = pio.PB_29         -- Which PIO pin is the LED connected to?

function uart_handler( resnum )
  -- flash the onboard LED
  pio.pin.setlow( led )
  for i=1,10000 do end  -- for about 1/100th of a second
  pio.pin.sethigh( led )
end

pio.pin.sethigh( led )    -- off
pio.pin.setdir( pio.OUTPUT, led )

uart.setup( 0, 115200, 8, uart.PAR_NONE, 1 )
uart.set_buffer( 0, 1024 )  -- buffer must be enabled for UART IRQs to happen

-- tell eLua which function it should call every time the UART receives
cpu.set_int_handler( cpu.INT_UART_RX, uart_handler )
-- and enable that Lua interrupt
cpu.sei( cpu.INT_UART_RX, 0 )

-- Wait for about ten seconds while the test runs
for i=1,10000000 do end

-- disable the Lua interrupt
cpu.cli( cpu.INT_UART_RX, 0 )
-- and remove our handler function
cpu.set_int_handler( cpu.INT_UART_RX, nil )
```

The character is received from the UART and placed in the buffer before the Lua interrupt routine is called, so you can read it in your Lua interrupt routine with `uart.getchar( 0, 0 )` and act on it immediately.

Baud rate accuracy

The following table gives the actual baud rates set by eLua for the most commonly used ones:

Want	Get	Error
300	300	0%
600	600	0%
1200	1200	0%
2400	2400	0%
4800	4799	-0.02%
9600	9604	+0.04%
19200	19186	-0.07%
31250	31250	0%
38400	38372	-0.07%
57600	57692	+0.07%
115200	114583	-0.5%

Hijacking the serial board's TX LED

If UART0 is not used, the LED on the serial board can toggled by using `pio.PA_1` as a generic Mizar32/PIO output: a low output value turns this LED off and a high value turns it on.

```
-- Turn the serial board's TX LED on (a low output lights the LED)
txled = pio.PA_1
pio.pin.setlow( txled )                -- Prepare "off" as the output value
pio.pin.setdir( pio.OUTPUT, txled )    -- Make the pin a GPIO output, disabling serial port 0
```

Further reading

- The UART section of the eLua manual [3] for details of all eLua `uart.*` functions;
- The Atmel AT32UC3A datasheet [4] Chapter 26: Universal Synchronous/Asynchronous Receiver/Transmitter (USART).

References

[1] http://code.google.com/p/mizar32/issues/detail?id=29
[2] http://code.google.com/p/mizar32/issues/detail?id=77
[3] http://www.eluaproject.net/en_refman_gen_uart.html
[4] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf#26

Advanced topics

GPIO

Introduction

GPIO means General Purporse Input/Output and is the generic name for all the pins of the AVR32UC3A microcontroller. Every pin of the microcontroller is a GPIO pin except for the power supply pins, the pin that resets the processor and the USB JTAG signal connections. This includes the connections to the SDRAM and every other input or output signal.

Each of the GPIO pins can be programmed to have one of three functions, and the three functions are different for every pin. When the Mizar32 is turned on, it is deaf and blind and the first thing that the on-board software must do is to program the correct function for each of the GPIO pins that are necessary for it to be able to talk to the rest of the world.

For example, eLua does the following when it starts up:

- it programs the pins that are connected to the oscillator crystals to function as oscillator inputs. This increases its processing speed from 115kHz (its internal oscillator) to 66MHz (the external oscillator). It also programs the pins connected to the 32768Hz crystal as oscillator inputs so that it can measure time accurately;
- it programs the pins connected to the SDRAM memory to work as SDRAM memory pins so that, as well as the 64Kbyte (Models A and B) or 32Kbyte (Model C) internal RAM, it can also access the 32Mbyte SDRAM memory chip;
- if you have configured the eLua console to be on the serial port, it programs UART0's RX and TX pins to talk RS232;
- if your eLua firmware includes MMC/SD card support, the pins of the second SPI port, `SPI1`, are programmed to talk the MMC/SD card.

The GPIO pins for all other devices are programmed to perform their function the first time you call them from Lua. If you are not using some device you are free to use its GPIO pins as simple input/output pins: see the **PIO** section for further details on how to do this.

Pin functions

On the Mizar32, the following functions are assigned to the GPIO pins:

Legend:

- **X** = Pin is not connected to anything
- **Foo/Bar** = signal is used for more than one thing

Port A

PIN	SIGNAL	Description	Bus pin
PA00	UART0_RX	Right bus serial port	BUS4 pin 3
PA01	UART0_TX		BUS4 pin 4
PA02	GPIO2	Bus GPIO, 4mA max, VGA SRAM /HOLD pin	BUS5 pin 11
PA03	UART0_RTS	Right bus serial port	BUS4 pin 5
PA04	UART0_CTS		BUS4 pin 6
PA05	UART1_RX	Left bus serial port	BUS3 pin 3
PA06	UART1_TX		BUS3 pin 4
PA07	GPIO7	Bus GPIO, 4mA max	BUS5 pin 12
PA08	UART1_RTS	Left bus serial port	BUS3 pin 10
PA09	UART1_CTS		BUS3 pin 9
PA10	SPI0_CS0	Left bus SPI	BUS1 pin 12
PA11	SPI0_MISO		BUS1 pin 13
PA12	SPI0_MOSI		BUS1 pin 14
PA13	SPI0_SCK		BUS1 pin 15
PA14	SPI1_CS0	SD card chip select	None
PA15	SPI1_SCK	Right bus SPI/SD card clock	BUS4 pin 9
PA16	SPI1_MOSI	Right bus SPI/SD card data in (host->card)	BUS4 pin 10
PA17	SPI1_MISO	Right bus SPI/SD card data out (card->host)	BUS4 pin 11
PA18	SPI1_CS1	"FREE"	None
PA19	SPI1_CS2	Right bus SPI chip select	BUS4 pin 12
PA20	EXT_INT	?	BUS5 pin 13
PA21	ADC0	ADC	BUS5 pin 5
PA22	ADC1		BUS5 pin 6
PA23	ADC2		BUS5 pin 7
PA24	ETHERNET	Ethernet interrupt	BUS2 pin 3
PA25	ADC4	ADC	BUS6 pin 4
PA26	ADC5		BUS6 pin 5
PA27	ADC6		BUS6 pin 6
PA28	ADC7		BUS6 pin 7
PA29	SDA	I2C	BUS2 pin 10
PA30	SCL		BUS2 pin 11

Port B

PIN	SIGNAL	Description	Bus pin
PB00	REF_CLK	Ethernet	BUS1 pin 3
PB01	TX_EN		BUS1 pin 4
PB02	TX0		BUS1 pin 5
PB03	TX1		BUS1 pin 6
PB04	X		None
PB05	RX0		BUS2 pin 5
PB06	RX1		BUS2 pin 6
PB07	RX_ER		BUS2 pin 7
PB08	MDC		BUS2 pin 4
PB09	MDIO		BUS2 pin 8
PB10	SDCK	SDRAM	None
PB11	SDCKE		None
PB12	RASn		None
PB13	CASn		None
PB14	SDWEn		None
PB15	RX_DV	Ethernet	BUS1 pin 7
PB16	SDA10	SDRAM A10	None
PB17	GPIO49	Bus GPIO, 4mA max	BUS5 pin 8
PB18	GPIO50/PWM6	Bus GPIO, 4mA max/PWM channel 6	BUS5 pin 9
PB19	PWM0	PWM	BUS4 pin 7
PB20	PWM1		BUS4 pin 8
PB21	PWM2		None
PB22	PWM3		BUS6 pin 1
PB23	UART1_DCD	Left bus serial port	BUS3 pin 5
PB24	UART1_DSR		BUS3 pin 6
PB25	UART1_DTR		BUS3 pin 7
PB26	UART1_RI		BUS3 pin 8
PB27	PWM4	PWM channel 4	BUS6 pin 2
PB28	PWM5		BUS6 pin 3
PB29	GPIO61/LED	On-board LED (0=lit)	None
PB30	GPIO62	Bus GPIO, 4mA max	BUS6 pin 9
PB31	GPIO63		BUS6 pin 10

Port C

PIN	SIGNAL	Description
PC00	Xin32	32768Hz crystal "X2" (*)
PC01	Xout32	
PC02	Xin0	12MHz crystal "X1"
PC03	Xout0	
PC04	Xin1	Not used
PC05	Xout1	

-) On circuit pads C47 before Mizar32 v1.3.2

Port X

PIN	SIGNAL	Description	Bus pin
PX00	D10	SDRAM	None
PX01	D9		
PX02	D8		
PX03	D7		
PX04	D6		
PX05	D5		
PX06	D4		
PX07	D3		
PX08	D2		
PX09	D1		
PX10	D0		
PX11	DQM1		
PX12	X		
PX13	X		
PX14	CS1n		
PX15	X		
PX16	GPIO88/BUTTON	Bus GPIO/Push button on main board (low when pressed)	BUS6 pin 13
PX17	A17	SDRAM BA1	None
PX18	A16	SDRAM BA0	
PX19	GPIO85	Bus GPIO, 4mA max	BUS6 pin 12
PX20	A14	SDRAM A12	None
PX21	A13	SDRAM A11	
PX22	GPIO82	Bus GPIO, 4mA max (SDRAM A10 is on PB16)	BUS6 pin 11

PX23	A11	SDRAM A9	None
PX24	A10	SDRAM A8	
PX25	A9	SDRAM A7	
PX26	A8	SDRAM A6	
PX27	A7	SDRAM A5	
PX28	A6	SDRAM A4	
PX29	A5	SDRAM A3	
PX30	A4	SDRAM A2	
PX31	A3	SDRAM A1	
PX32	A2	SDRAM A0	
PX33	GPIO71	Bus GPIO, 4mA max	BUS5 pin 10
PX34	DQM0	SDRAM	None
PX35	D15		
PX36	D14		
PX37	D13		
PX38	D12		
PX39	D11		

Bus renaming:
In Mizar32 v1.3.2, the bus connectors were renamed

Old	New	Also known as
P3	BUS1	BUS_HALF_LEFT_UP
P4	BUS2	BUS_HALF_LEFT_DOWN
P2	BUS3	BUS_HALF_LEFT_EXT
P5	BUS4	BUS_HALF_RIGHT_UP
P6	BUS5	BUS_HALF_RIGHT_DOWN
P7	BUS6	BUS_HALF_RIGHT_EXT

Further reading

- The **PIO** section explains further how these pins are used.
- There is a summary table of the possible functions that can be assigned to each GPIO pin in the Atmel AVR32UC3A Datasheet [1] in section 12.7 "Peripherals: Peripheral Multiplexing on I/O lines", table 12-9.
- The last part of the file in the AVR32 GNU toolchain `/usr/avr32/include/avr32/uc3a0128.h` gives a complete definition of what functions can be assigned to each pin.

References

[1] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf

Memory map

Mizar32 memory map

The Mizar32's memory map is the same as that of the AT32UC3A chip, with the addition of the SDRAM on CS1:

Memory map

0x00000000-0x00007FFF 0x00000000-0x0000FFFF	32KB internal RAM (Model C) 64KB internal RAM (Models A & B)
0x80000000-0x8001FFFF 0x80000000-0x8003FFFF 0x80000000-0x8007FFFF	128KB internal Flash memory (Model C) 256KB internal Flash memory (Model B) 512KB internal Flash memory (Model A)
0x80800000-0x808001FF	512 bytes Flash User Page (DFU boot loader config word is at 0x808001FC)
0xD0000000-0xD1FFFFFF	32MB external SDRAM
0xE0000000-0xE000FFFF	64KB USB configuration
0xFFFE0000-0xFFFEFFFF	HSB-PB Bridge A
0xFFFF0000-0xFFFFFFFF	HSB-PB Bridge B

Device control registers

0xFFFE0000	USBB
0xFFFE1000	HMATRIX
0xFFFE1400	Flash memory
0xFFFE1800	MACB
0xFFFE1C00	SMC
0xFFFE2000	SDRAM
0xFFFF0000	PDCA
0xFFFF0800	Interrupt controller
0xFFFF0C00	PM (Power manager)
0xFFFF0D00	RTC
0xFFFF0D30	WDT
0xFFFF0D50	FREQM
0xFFFF0D80	EIC
0xFFFF1000	GPIO and PIO controller
0xFFFF1400	USART0
0xFFFF1800	USART1
0xFFFF1C00	USART2

0xFFFF2000	USART3
0xFFFF2400	SPI0
0xFFFF2800	SPI1
0xFFFF2C00	TWI
0xFFFF3400	SSC
0xFFFF3800	TC
0xFFFF3C00	ADC
0xFFFF3000	PWM
0xFFFF4000	ABDAC

Flash memory

Mizar32 comes in three versions with different amounts of flash memory and static RAM:

- Model A: 512KB flash / 64KB SRAM / 32MB SDRAM
- Model B: 256KB flash / 64KB SRAM / 32MB SDRAM
- Model C: 128KB flash / 32KB SRAM / 32MB SDRAM

The flash memory is built into the `AT32UC3A0128/256/512` processor chip and responds from address 0x80000000.

The first 8KB (`0x80000000-0x80001FFF`) contain the "Atmel AT32UC3 USB DFU bootloader", which is able to download and write new contents to the rest of the flash memory over the USB port. When the chip resets or powers up, it starts executing at `0x80000000`, which is the USB DFU bootloader. This checks whether the user button is pressed and, if it is, gets ready to download and program the rest of the flash memory. If the button is not pressed, it jumps to address `0x80020000` which contains the first word of the executable code for the eLua interpreter.

Alternatively, the flash memory from `0x80020000` can be programmed with **emBLOD**, the embedded boot loader, which loads the executable code from a file "`autorun.bin`" on the SD card into SDRAM and executes it there. In SDRAM code runs about 6 times slower that a program in flash memory but you can load and run programs up to 32MB in size.

If you have a JTAG programming device, you can program the entire 128/256/512KB by overwriting the USB DFU boot loader.

There is also a second, 512-byte area of flash at `0x80800000 - 0x808001FF`, the "Flash user page", which keeps a configuration word for the USB DFU bootloader in the last word, but is otherwise free.

Further reading

- the AVR32 UC3 USB DFU Bootloader manual [1].

References

[1] http://www.atmel.com/Images/doc7745.pdf

Flashing firmware

Flashing firmware images

There are several ways to write new firmware images to the Mizar32:

With the USB DFU bootloader

The first 8KB of the Mizar32 flash memory comes pre-programmed with Atmel's USB DFU boot loader, which is able to write to the rest of the flash memory. To talk to it, you can use Atmel's closed-source 'flip' and 'batchisp3' tools, which are awful, or the open-source 'dfu-programmer' which is OK.

Using dfu-programmer

`dfu-programmer` is an open-source program to talk to the USB DFU boot loader. It is included in Debian and Ubuntu, for which the installation step is (as root):

```
apt-get install dfu-programmer
```

Fetch the firmware:

```
wget http://simplemachines.it/files/mizar32-firmware-latest.tgz
tar xfz mizar32-firmware-*.tgz
cd mizar32-firmware-*
```

Now

- Connect the Mizar32 to your PC with a USB cable
- Power the Mizar32 on (or press its reset button) while holding the user button (SW2) depressed
- On the PC, issue the commands:

```
sh program-128.sh    # If you have the Mizar32 model C
```

or

```
sh program-256.sh    # For Mizar32 model A of B for the integer firmware
```

or

```
sh program-256fp.sh  # For Mizar32 model A of B for floating point firmware
```

If it says `dfu-programmer: no device present.`, try running it as root. If so, and you want anyone to be able to run it, you can go, as root:

```
chown root $(which dfu-programmer)
chmod 4755 $(which dfu-programmer)
```

though this opens a security hole so, if you may have malicious users logged into your system, it might be better to add yourself to group "`admin`" in `/etc/group` and go:

```
chown root:admin $(which dfu-programmer)
chmod 4750 $(which dfu-programmer)
```

Note that the Debian/Ubuntu program "dfutool", included in the "bluez" package, is something completely different.

Bugs in old versions of dfu-programmer

There is a bug in dfu-programmer v0.5.1 which very occasionally misprograms the flash. The symptom is

```
$ dfu-programmer at32uc3a0256 flash elua_lualong_at32uc3a0256.hex
Validating...
Image did not validate.
```

The bug is fixed in dfu-programmer-0.5.2 and later.

As if that weren't enough, the Mizar32A carries the automotive-quality ultra-robust version of the AT32UC3A chip with 512KB of flash memory, which takes a few seconds longer to erase its flash memory than `dfu-programmer` expects (about 14 seconds instead of 10). This makes the `program-256.sh` script in the firmware distribution fail because it tries to start programming the chip before the erasure is complete. A workaround is to add a line to the `program-256.sh` script:

```
  echo Erasing...
  dfu-programmer at32uc3a0256 erase
+ sleep 5
  echo Programming...
  dfu-programmer at32uc3a0256 flash elua_lualong_at32uc3a0256.hex
```

This is fixed in dfu-programmer-0.5.5 and later but Debian and Ubuntu still have 0.5.4. You can check the version of your installed `dfu-programmer` by saying

```
dfu-programmer --version 2>&1 | head -1
```

and you can install a more recent version by compiling from source:

```
apt-get install libusb-dev build-essential
# Visit http://sourceforge.net/projects/dfu-programmer/files/latest/download
# and save the file it gives to your browser
tar xfz dfu-programmer-*.tar.gz
cd dfu-programmer-*
./configure
make
sudo make install                     # Installs under /usr/local
sudo apt-get purge dfu-programmer     # Remove the old version
```

Further reading

- The Atmel USB DFU Programmer project [1] at sourceforge

Using batchisp3

Atmel publishes closed-source programs to talk to the USB DFU boot loader: the graphical "flip" and the command-line "batchisp3". They are both of poor quality and only the second one is currently usable with AT32UC3 parts.

Note that the Debian/Ubuntu package "flip" is something completely different.

Installation on Ubuntu (as root) is (adapted from eLua's AVR32 platform info [2]):

```
apt-get install openjdk-6-jre
cd /usr/local
wget http://www.atmel.com/dyn/resources/prod_documents/flip_linux_3-2-1.tgz
```

```
tar xfz flip_linux_3-2-1.tgz
rm flip_linux_3-2-1.tgz

cat > bin/batchisp3 << \EOF
#! /bin/sh
FLIP_HOME=/usr/local/flip.3.2.1/bin
JAVA_HOME=/usr/lib/jvm/java-6-openjdk/jre
USB_DEVFS_PATH=/dev/bus/usb
export FLIP_HOME JAVA_HOME USB_DEVFS_PATH
exec /usr/local/flip.3.2.1/bin/batchisp3.sh "$@"
EOF

chmod 755 bin/batchisp3
```

(on Red Hat systems, the `USB_DEVFS_PATH` runes should be omitted).

To get `flip` to work at all, you have to `cd /usr/local/flip.3.2.1/bin` first, and it doesn't yet support AT32UC3 parts so we can only use the command-line `batchisp3` program:

- Connect the Mizar32 to your PC with a USB cable
- Power the Mizar32 on (or press its reset button) while holding the user button (SW2) depressed.
- On the PC, issue the command

```
batchisp3 -hardware usb -device at32uc3a0128 -operation erase f memory
flash blankcheck loadbuffer $PWD/elua_lualong_at32uc3a0128.elf program
verify start reset 0
```

Note that you have to explicitly give the full pathname of the firmware file (the `$PWD/` trick here). Otherwise it looks for the firmware file in `/usr/local/flip.3.2.1/bin/`.

Further reading

- The Atmel USB DFU Bootloader Datasheet [3]
- The FLIP home page [4]

Using Batchisp under Windows Vista/7 32bit

- Download the latest version of Flip [5] for Windows from Atmel's web site (`BATCHISP.EXE` is inside the Flip installer) and follow the instructions for installation. At the moment, only Batchisp supports AT32UC3A microprocessors while Flip does not support its yet, so you must install Flip but can't use it.
- Activate the DFU Bootloader: connect the Mizar32 to your PC with a USB cable, connect the power plug and press and hold the reset button while holding the user button (SW2), then release the reset button, then release the user button.
- Go to the Windows Control Panel, right-click on Computer → Properties → Device Manager → right-click on 'AT32UC3A DFU' → Update Driver Software → 'Browse my computer for driver software' and select the path Flip\usb (here, it's `c:\Program Files (x86)\Atmel\Flip 3.4.3\usb\`) and click OK. Now Windows tells you 'Windows can't verify the publisher of this driver software' click on 'Install this software anyway'. Now your Mizar32 driver for Batchisp is installed.
- Open your Windows Command Processor: Start → type 'cmd' in 'Search command and file' → Right-click on cmd.exe → Start as Administrator. Type 'PATH' followed by entire path of your Batchisp.exe directory; on our machine the command is:

```
PATH c:\Program File (x86)\Atmel\Flip 3.4.3\bin
```

- Restart Windows. Now Windows is able to find the `Batchisp.exe` program
- Download [6] and decompress this file in some directory. Run your Windows Command Processor: Start → type 'cmd' in 'Search command and file' → Right-click on cmd.exe → Start as Administrator. Type the following command (this command is case sensitive):

```
batchisp -device at32uc3aXXXX -hardware usb -operation erase f memory
flash blankcheck loadbuffer
\Mizar32_firmware_directory\elua_lualong_at32uc3aXXXX.elf program
verify start reset 0
```

where:

- `at32uc3aXXXX` is your Atmel device that can be: `at32uc3a0128`, `at32uc3a0256` or `at32uc3a0512`.
- `\Mizar32_firmware_directory\` is the entire PATH where you stored the Mizar32 firmware.
- `elua_lualong_at32uc3aXXXX.elf` is the firmware version.

For example, you can flash a Mizar32 B (the 256Kb version) with this command line:

```
batchisp -device at32uc3a0256 -hardware usb -operation erase f memory
flash blankcheck loadbuffer
C:\Users\Simplemachines\Desktop\project\elua_firmware\0256\elua_lualong_at32uc3a0256.elf
 program verify start reset 0
```

If batchisp can't run because "MSVCR71.dll is missing"

- download the file msvcr71.dll [7] into your `Flip\bin` directory (on this PC it's `c:\Program Files (x86)\Atmel\Flip 3.4.3\bin\`)
- Re-type the batchisp command above to update Mizar32 firmware.

Further reading

- The Atmel USB DFU Bootloader Datasheet [3]
- The FLIP home page [4]

Please report any feedback or suggestions on this procedure to `support@simplemachines.it`

Using Batchisp under Windows Vista/7 64bit

- Download the latest version of Flip [5] for Windows from the Atmel web site (Batchisp.exe is inside Flip installation). Follow the video instructions for installation. At the moment only Batchisp supports AT32UC3A microprocessors; Flip does not support it yet, so you have to install Flip but can't use it.
- Download and unzip this USB driver [8], because Atmel's original drivers are unsigned and Windows Vista/7 64bit does not let you install unsigned driver. Unpack the .zip file downloaded in the `Flip\usb` folder; on our machines the folder is:

```
c:\Program Files (x86)\Atmel\Flip 3.4.3\usb
```

- Activate the DFU Bootloader: connect the Mizar32 to your PC with a USB cable Connect the power plug and press and hold the reset button while holding the user button (SW2), release the reset button, then release the user button.
- Go to the Windows Control Panel with: right-click on Computer --> Properties --> Device Manager --> right-click on 'AT32UC3A DFU' --> Update Driver Software --> 'Browse my computer for the driver software' and select the path where you copied the new USB signed driver (on this machine `c:\Program Files (x86)\Atmel\Flip 3.4.3\usb\atmel-flip-3.4.2-signed-driver`) and click OK. Now your Mizar32 driver for Batchisp is installed.

- Open your Windows Command Processor: Start --> type 'cmd' in 'Search commands and files' --> Right-click on cmd.exe --> Start as Administrator. Type 'Path' followed by entire path of your `Batchisp.exe` directory; on our machine the command is:

Path c:\Program File (x86)\Atmel\Flip 3.4.3\bin

- Restart Windows. Now Windows is able to find the `Batchisp.exe` program.
- Download [6] and unpack this file, then run your Windows Command Processor: Start --> type 'cmd' in 'Search commands and files' --> Right-click on cmd.exe --> Start as Administrator and type the following command (this command is case sensitive):

```
batchisp -device at32uc3aXXXX -hardware usb -operation erase f memory
flash blankcheck loadbuffer
\Mizar32_firmware_directory\elua_lualong_at32uc3aXXXX.elf program
verify start reset 0
```

where:

- `at32uc3aXXXX` is your Atmel device that can be: at32uc3a0128, at32uc3a0256, at32uc3a0512.
- `\Mizar32_firmware_directory\` is the entire PATH where you stored the Mizar32 firmware.
- `elua_lualong_at32uc3aXXXX.elf` is the firmware file version.

We flash our Mizar32 B (256KB version) with this command line:

```
batchisp -device at32uc3a0256 -hardware usb -operation erase f memory
flash blankcheck loadbuffer
C:Users\Simplemachines\Desktop\project\elua_firmware\0256\elua_lualong_at32uc3a0256.elf
 program verify start reset 0
```

Further reading

- The Atmel USB DFU Bootloader Datasheet [3]
- The FLIP home page [5]

With a JTAG programmer

A JTAG programmer is a device that lets you do more than the USB DFU bootloader, including updating the USB DFU bootloader itself, and requires special software on your PC to drive it.

If you need to do this, please contact `support@simplemachines.it`

References

[1] http://sourceforge.net/projects/dfu-programmer
[2] http://www.eluaproject.net/en_installing_avr32.html
[3] http://www.atmel.com/dyn/resources/prod_documents/doc7618.pdf
[4] http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3886
[5] http://www.atmel.com/tools/FLIP.aspx
[6] http://simplemachines.it/files/mizar32-firmware-latest.tgz
[7] https://docs.google.com/viewer?a=v&pid=explorer&chrome=true&srcid=0B3KeO9iN5LY-OGMxOGY3MWEtYjFlNC00NWE0LTlmYzAtNmUzYzI3M2RhMjFl&hl=it
[8] https://docs.google.com/viewer?a=v&pid=explorer&chrome=true&srcid=0B3KeO9iN5LY-NjQ2ZmU3MTEtYWFhOS00MjYyLWJkZTUtMzUyNmZjNGU2NDhl&hl=it

Compiling eLua

Use the Web Builder

The Mizar32 Web Builder at http://builder.simplemachines.it [1] lets you customise the firmware in several ways and will build you a new firmware image without you having to do any of this horrible stuff.

If that doesn't do enough for you, or if you just enjoy this kind of thing, then take a deep breath and...

Install an AVR32 cross-compiler

On Linux

Atmel AVR 32-bit Toolchain 3.4.2

The cross-compiler for Linux is the Atmel AVR Toolchain for Linux [1].

At the time of writing, April 2013, the latest version is Atmel AVR Toolchain 3.4.2, based on the GNU C Compiler version 4.4.7.

To skip the Atmel registration form, you can use these quick download links:

- Atmel AVR 32-bit Toolchain 3.4.2 - Linux 32-bit [2]
- Atmel AVR 32-bit Toolchain 3.4.2 - Linux 64-bit [3]

You will also need

- Atmel AVR Toolchain 3.4.2 - Header Files [4]

and, if you are interested in building the toolchain yourself, the source code is here:

- Atmel AVR 32-bit Toolchain 3.4.2 - Source code [5]

Fetching and unpacking the toolchain

Here we show the steps to install the 32-bit version:

```
cd
wget http://www.atmel.com/Images/avr32-gnu-toolchain-3.4.2.435-linux.any.x86.tar.gz
tar xfz avr32-gnu-toolchain-3.4.2.435-linux.any.x86.tar.gz
# Fetch, unpack and install the header files
wget http://www.atmel.com/Images/atmel-headers-6.1.3.1475.zip
unzip atmel-headers-6.1.3.1475.zip
mv atmel-headers-6.1.3.1475/avr32 avr32-gnu-toolchain-linux_x86/avr32/include/
# Clean up
rm -r atmel-headers*
rm avr32-gnu-toolchain-*.gz
```

then, to use it, once per session:

```
PATH=$HOME/avr32-gnu-toolchain-linux_x86/bin:$PATH
export PATH
```

For the 64-bit version, replace every "`x86`" in the above with "`x86_64`"

Installing it as a Debian/Ubuntu package

To make a Debian/Ubuntu package out of this, do the above, then go:

```
sudo apt-get install alien fakeroot
cd avr32-gnu-toolchain-linux_x86
mkdir usr
mv [a-s]* usr/
# It includes a lot of files that it shouldn't, so select the toolchain ones.
tar cfz avr32-gnu-toolchain-3.4.2.tgz usr/avr32 usr/bin/avr32-* \
    usr/lib/gcc/avr32 usr/libexec usr/share/man/man1
fakeroot alien --keep-version avr32-gnu-toolchain-3.4.2.tgz
mv avr32-gnu-toolchain_3.4.2-1_all.deb avr32-gnu-toolchain_3.4.2-1_i386.deb
rm avr32-gnu-toolchain-3.4.2.tgz
rm -r usr
```

or, for the 64-bit version, do the same thing in `avr32-gnu-toolchain-linux_x86_64` and rename the `.deb` file to `..._amd64.deb`

If you had installed their old 2.4.2 Ubuntu packages, you will need to remove those first:

```
sudo apt-get purge avr32-binutils avr32-buildroot-essentials avr32-gcc-newlib \
 avr32-gdb avr32gdbproxy avr32headers avr32parts avr32program avr32trace \
 avrfwupgrade libavr32ocd libavr32sim libavrtools libelfdwarfparser
```

then you can install the new toolchain with

```
sudo dpkg -i avr32-gnu-toolchain_3.4.2-1_*.deb
```

(to remove it again, `sudo apt-get purge avr32-gnu-toolchain`)

Build a toolchain from source code

ct-ng [6], a fork of crosstool-ng [7], builds better AVR32 cross compilers for you:

```
sudo apt-get install git autoconf bash gawk g++ libncurses-dev
git clone http://anonymous@spaces.atmel.com/git/ct-ng
cd ct-ng
./bootstrap
./configure --enable-local
make
./ct-ng build
```

By default, it installs the cross-toolchain under your home directory, so before building eLua you need to say:

```
PATH=$HOME/x-tools/avr32-unknown-none/bin:$PATH
export PATH
```

On Windows

Atmel also provide the Atmel AVR Toolchain for Windows [8] as a `setup.exe`, for which the quick download link is:

- Atmel AVR Toolchain 3.4.2 for Windows [9]

Install the eLua build system

The eLua build system uses "`scons`" and "`gcc`", while the eLua sources are under `git`, so install those

On Debian or Ubuntu:

```
sudo apt-get install scons gcc git
```

Download the eLua sources

Stable release

The latest stable source release is elua0.9 [10], which fully supports the Mizar32 boards.

```
wget http://download.berlios.de/elua/elua0.9.tgz
tar xfz elua0.9.tgz
cd elua0.9
```

An alternative is:

The current development version

```
git clone https://github.com/elua/elua
cd elua
```

though its build system has changed, so the instructions using `scons` will need changing.

Compile the eLua interpreter

For Mizar32 A and B

The 256KB and 512KB flash/64MB internal RAM versions of Mizar use the same eLua firmware. The command:

```
scons board=mizar32 cpu=AT32UC3A0256
```

will create a file `elua_lua_at32uc3a0256.elf` which can be programmed into the Mizar32 boards in various ways: see the chapter Flashing firmware.

To build a version of the interpreter where all numeric variables are integers instead of floating point values, which is smaller and slightly faster, use:

```
scons board=mizar32 cpu=AT32UC3A0256 target=lualong
```

For Mizar32 C

In 128KB of flash memory you can fit the 8KB DFU bootloader and a 120K integer version of eLua. Standard eLua for the Mizar C, compiled with

```
scons board=mizar32 cpu=at32uc3a0128 target=lualong optram=0
```

produces a firmware with most of the eLua modules omitted, leaving just the PIO, Timer and UART modules.

The `Mizar32` branch of eLua [11] has a series of modification to reduce the code size so as to be able to include all the eLua modules except for the Ethernet module. The features that it removes from eLua are:

- the eLua shell (it runs the Lua interpreter directly);
- Lua interrupts (an option available with the GPIO, Timer and UART modules);
- the Lua `debug` module, which is of little use in an embedded system;
- `dump()`, used to write compiled code or Lua data into files;
- `undump()`, used to read the same back in, or to load precompiled Lua bytecode from `*.lc` files;
- `collectgarbage()` and `gcinfo()`, used to monitor and fine-tune eLua's use of RAM;
- the Emergency Garbage Collector, used to improve the performance of systems with very little RAM.

To use this branch, fetch a copy of it:

```
git clone http://github.com/elua/elua
cd elua
git checkout Mizar32
```

and compile it thus:

```
scons board=mizar32 cpu=at32uc3a0128 target=lualong optram=0
```

Program the firmware to the board

When the compilation is finished, it should have created a file called something like `elua_lualong_at32uc3a0256.elf` which can be programmed into the Mizar32 board in various ways: see the chapter on Flashing firmware.

If you will be using `dfu-programmer` to do this, you will first need to convert your `.elf` file to a `.hex` file. For example:

```
elua=elua_lualong_at32uc3a0256
avr32-objcopy -O ihex $elua.elf $elua.hex
```

References

[1] http://www.atmel.com/tools/ATMELAVRTOOLCHAINFORLINUX.aspx
[2] http://www.atmel.com/images/avr32-gnu-toolchain-3.4.2.435-linux.any.x86.tar.gz
[3] http://www.atmel.com/Images/avr32-gnu-toolchain-3.4.2.435-linux.any.x86_64.tar.gz
[4] http://www.atmel.com/Images/avr-headers-6.1.3.1475.zip
[5] http://distribute.atmel.no/tools/opensource/Atmel-AVR-Toolchain-3.4.2/avr32/
[6] http://spaces.atmel.com/gf/project/ct-ng/
[7] http://crosstool-ng.org
[8] http://www.atmel.com/tools/ATMELAVRTOOLCHAINFORWINDOWS.aspx
[9] http://www.atmel.com/images/avr-toolchain-installer-3.4.2.1573-win32.win32.x86.exe
[10] http://prdownload.berlios.de/elua/elua0.9.tgz
[11] http://github.com/elua/elua/tree/Mizar32

emBLOD

Overview

Normally, the boot sequence is:

- the USB DFU bootloader at `0x8000000-80001FFF` (8KB), which checks if the user button is pressed and, if it isn't, runs:
- the eLua interpreter at `0x80002000`

emBLOD is an embedded boot loader that replaces eLua at 0x80002000 and loads a modified version of the eLua interpreter code from a file on a FAT-formatted SD card into the start of the 32MB SDRAM then executes it there. It is fast to start up (a fraction of a second) and gets round the 120KB code size limit of the Mizar32 model C.

The downside is that when loaded into SDRAM instead of Flash, the eLua interpreter runs at one sixth of the speed. However, if you need floating point or ethernet support on a Mizar32 model C, this is the only way to do it.

Compiling emBLOD from source

emBLOD is an open source code project hosted at http://github.com/cmp1084/emBLOD

To fetch and build the source on Ubuntu you will need to install:

```
apt-get install git dfu-programmer
```

and the avr32 GCC cross-compiler, for which instructions are at the start of the Compiling eLua page.

The boot loader can be built from source as follows:

```
git clone https://github.com/cmp1084/emBLOD
cd emBLOD
make
```

which creates two versions of the same object file: `emblod.elf` and `bin/emblod.bin`. However, `dfu-programmer` requires a `.hex` file, so convert it:

```
avr32-objcopy -O ihex emblod.elf emblod.hex
```

then connect the Mizar32 to your PC over USB, reset or power-on the board while holding SW2 depressed and issue these commands on the PC:

```
dfu-programmer at32uc3a0128 erase
dfu-programmer at32uc3a0128 flash emblod.hex
dfu-programmer at32uc3a0128 start
```

If you have the serial port connected at 115200-8-N-1, you will see messages issued by emBLOD.

Compiling eLua for emBLOD

You should already be familiar with Compiling eLua.

Use eLua 0.9 [1], add

```
bootloader=emblod
```

to the `scons` compilation command. This will create an ELF file, which you should convert to a BIN file using

```
avr32-objcopy -O binary *.elf autorun.bin
```

and copy `autorun.bin` to the root of a FAT-formatted micro SD card for the Mizar32.

References

[1] http://download.berlios.de/elua/elua0.9.tgz

Article Sources and Contributors

Introduction *Source*: https://en.wikibooks.org/w/index.php?oldid=2557199 *Contributors*: Martinwguy2, 1 anonymous edits

Quick Start *Source*: https://en.wikibooks.org/w/index.php?oldid=2550456 *Contributors*: Martinwguy2, 8 anonymous edits

Models and Specifications *Source*: https://en.wikibooks.org/w/index.php?oldid=2557201 *Contributors*: Martinwguy2, 6 anonymous edits

ADC *Source*: https://en.wikibooks.org/w/index.php?oldid=2557206 *Contributors*: Martinwguy2, 4 anonymous edits

CPU *Source*: https://en.wikibooks.org/w/index.php?oldid=2557209 *Contributors*: Martinwguy2, 1 anonymous edits

Ethernet *Source*: https://en.wikibooks.org/w/index.php?oldid=2557219 *Contributors*: Martinwguy2, 1 anonymous edits

I2C *Source*: https://en.wikibooks.org/w/index.php?oldid=2557225 *Contributors*: Martinwguy2, 5 anonymous edits

LCD *Source*: https://en.wikibooks.org/w/index.php?oldid=2557241 *Contributors*: Dadosutter, Martinwguy, Martinwguy2, 12 anonymous edits

PIO *Source*: https://en.wikibooks.org/w/index.php?oldid=2557243 *Contributors*: Martinwguy2, 2 anonymous edits

PWM *Source*: https://en.wikibooks.org/w/index.php?oldid=2557248 *Contributors*: Martinwguy2

RTC *Source*: https://en.wikibooks.org/w/index.php?oldid=2557152 *Contributors*: Martinwguy2

SPI *Source*: https://en.wikibooks.org/w/index.php?oldid=2557188 *Contributors*: Martinwguy2

Timers *Source*: https://en.wikibooks.org/w/index.php?oldid=2557154 *Contributors*: Martinwguy2

UART *Source*: https://en.wikibooks.org/w/index.php?oldid=2557191 *Contributors*: DavidCary, Martinwguy2

GPIO *Source*: https://en.wikibooks.org/w/index.php?oldid=2557128 *Contributors*: Martinwguy2

Memory map *Source*: https://en.wikibooks.org/w/index.php?oldid=2557132 *Contributors*: Martinwguy2

Flash memory *Source*: https://en.wikibooks.org/w/index.php?oldid=2557133 *Contributors*: Martinwguy2

Flashing firmware *Source*: https://en.wikibooks.org/w/index.php?oldid=2557249 *Contributors*: Martinwguy, Martinwguy2, 4 anonymous edits

Compiling eLua *Source*: https://en.wikibooks.org/w/index.php?oldid=2557141 *Contributors*: Martinwguy, Martinwguy2, 3 anonymous edits

emBLOD *Source*: https://en.wikibooks.org/w/index.php?oldid=2557143 *Contributors*: Martinwguy2

Image Sources, Licenses and Contributors

File:Mizar32 front small2.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_front_small2.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy2

Image:Mizar32 power jack.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_power_jack.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy2

File:Mizar32 Ethernet RTC Module.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_Ethernet_RTC_Module.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy2

File:Mizar32 RS 232 485 Module.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_RS_232_485_Module.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy2

File:Mizar32 I2C LCD Module.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_I2C_LCD_Module.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy2

File:Mizar32 VGA Propeller Module.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_VGA_Propeller_Module.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy2

File:Mizar32 Protoboard PHT.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_Protoboard_PHT.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy2

File:Mizar32 Protoboard SMD.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_Protoboard_SMD.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy2

File:Mizar32 I2C LCD module 16x2.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_I2C_LCD_module_16x2.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy

File:Mizar32_LCD_module_USB_power_hack.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_LCD_module_USB_power_hack.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy2

File:Mizar32_PWM_LED_fade_example.jpg *Source*: https://en.wikibooks.org/w/index.php?title=File:Mizar32_PWM_LED_fade_example.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Martinwguy2

License