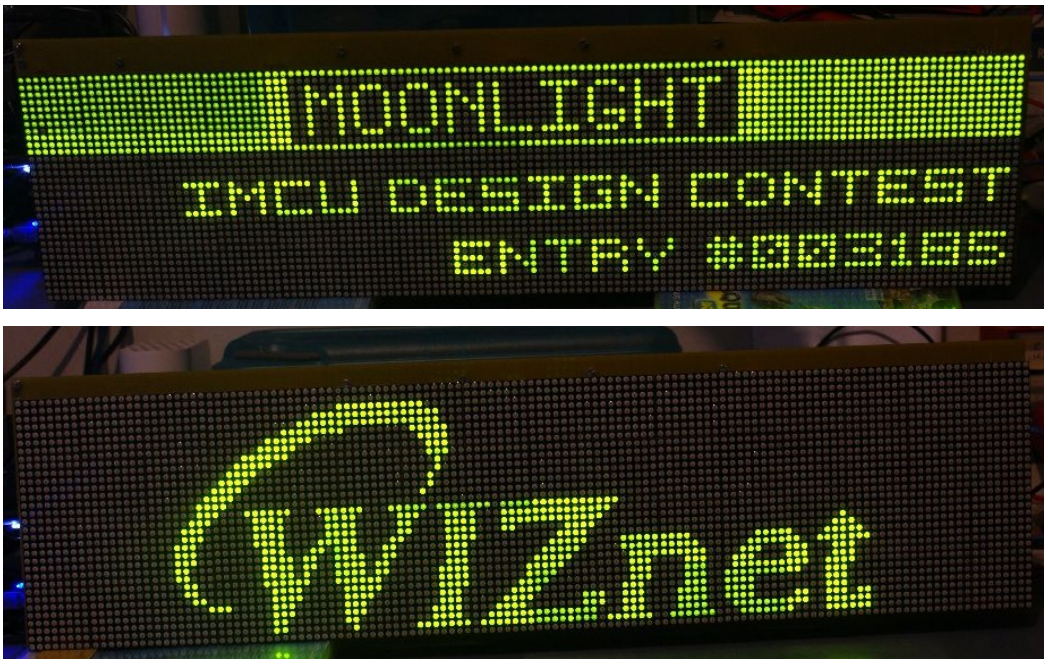
Project Number **003185**

Moonlight

W7100 and eLua based intelligent LED display
with network connectivity



ABSTRACT

NOTE: check *media/moonlight.mp4* for a demo of the project capabilities.

1. Project description

In the microcontroller scene of today, the iMCU W7100 from WIZnet stands up as an interesting, very capable device. Besides implementing a complete network stack in hardware, it also exposes a MCU that is built around the industry standard 8052, which makes it compatible with a lot of tools, libraries and utilities available on the market. The extended amount of on-chip memory (64K of Flash and especially 64K of RAM, quite rare for an 8-bit MCU) makes it suitable for a lot of applications.

However, like any other chip, the W7100 has its shortcomings. Most of them stem from the same 8052 based core that gives it flexibility and extensive tool support: this is an old architecture, with lots of limitations which are clearly visible when enumerating the CPU peripherals. Besides the network controller and the standard GPIO ports, the W7100 CPU exposes only a single UART interface. This might not be enough for a real world application, that often needs other peripherals, such as I2C, SPI, ADCs and DACs, more U(S)ARTs and others. Also, being an 8-bit 8052 core, it might be slow for some applications, and even its quite generous 64k of Flash memory might prove insufficient for medium to high complexity applications.

Starting from the above assumptions, this project (Moonlight) aims to use the W7100 for what he knows to do best: networking. Although this is somewhat of an unusual approach, Moonlight uses the W7100 as a network coprocessor for a 32-bit microcontroller. This microcontroller (the STMicroelectronics STM32F103RE) is built around a modern ARM Cortex-M3 core, and it's naturally much more powerful in terms of resources and processing power than the W7100. It's powerful enough to run eLua (http://www.eluaproject.net), an open source project that aims to bring the Lua programming language (http://www.lua.org) to the embedded MCU world. However, it doesn't "know" anything about networking, which is a serious constraint. This leads directly the idea behind Moonlight: let the 32-bit MCU run eLua (a task that is currently out of reach for any 8-bit CPU on the market) and let the W7100 run the network tasks. The two CPUs communicate through a simple, yet powerful RPC (Remote Procedure Call) mechanism. The STM32 CPU makes network requests to the W7100 using RPC, the W7100 executes them and sends the results back to the STM32.

While it can be argued that a 32 bit microcontroller with specialized Ethernet hardware might have been a better choice for this project, this isn't necessarily true for a couple of reasons:

1. the W7100 has a complete **network** stack in **hardware**, not just the hardware needed to interface to an Ethernet network (not to mention that the W7100 can also do ADSL, although this isn't used in Moonlight). In comparison, a 32 bit MCU that only has an Ethernet MAC (and maybe a PHY) still needs a software TCP/IP stack, and this can be very time consuming and expensive in terms of implementation time and maintenance.
2. this solution is generic. The RPC layer is written 100% in ANSI C, and since it uses a serial port it can be adapted to a large number of different CPUs.

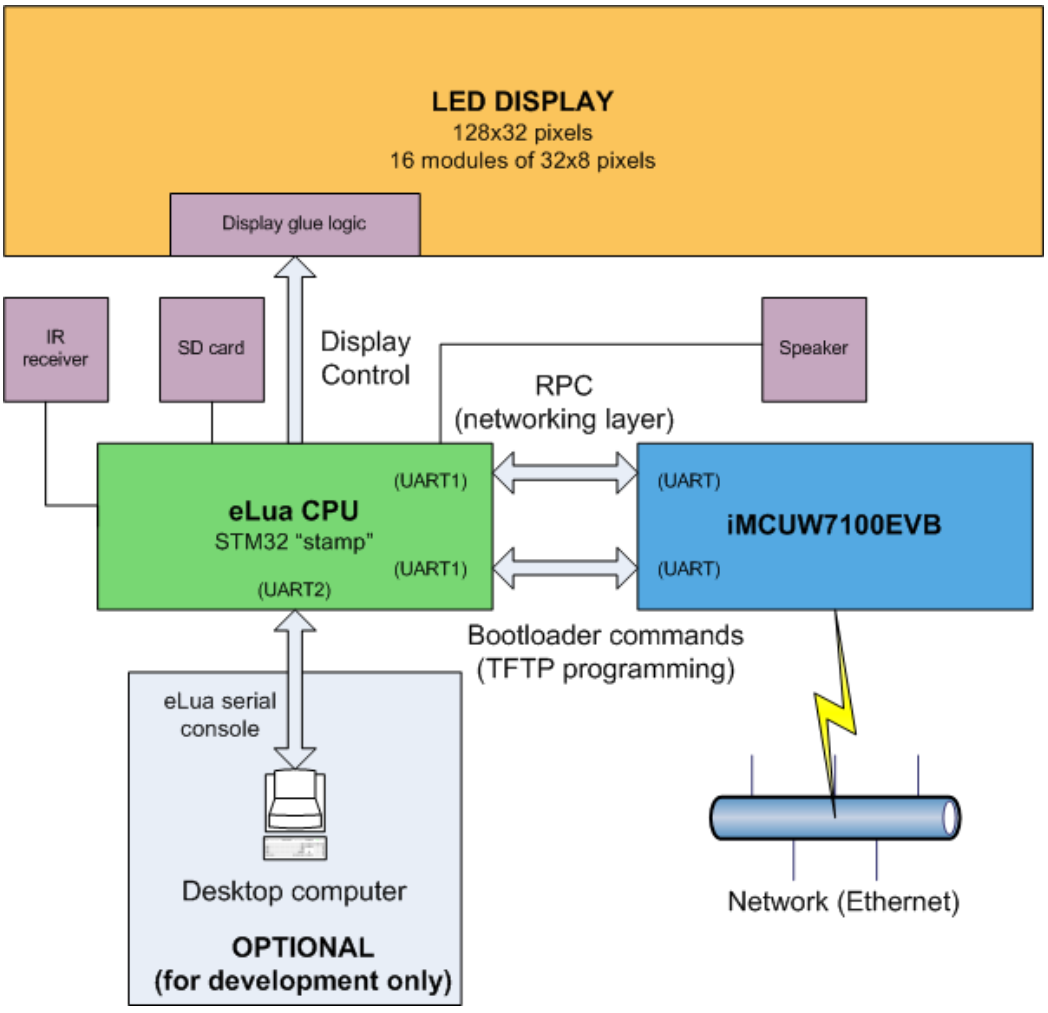
The actual application chosen to illustrate this concept is a large **(**638x176mm) 128x32 LED display, programmable directly in Lua using eLua (which runs on the STM32) and with network connectivity (which is the task of the W7100). The project name comes from "Moon" (which is English for Portuguese "Lua") and "light" (which is a reference to the LED display). The end result is a very versatile (because it can be programmed with Lua directly on the STM32) programmable display, with a huge range of connectivity options (because it uses the W7100). The whole system is controlled using a simple IR remote control, which can also reset the board. A number of applications (all of them written in Lua) are shown that demonstrate this functionality, some of which are listed below (check the complete documentation for the full list). Please note that almost all of them can be configured via a web interface:

- a RSS reader
- a Yahoo! Weather client
- a clock that synchronizes the time with the network
- a remote Winamp visualizer and remote control
- a realtime PC monitor (which shows CPU load and memory consumption amongst other things)
- a configurable network monitor which pings (via ICMP) a number of hosts and show their status on the display

The W7100 takes care of all the networking requirements of the application:

1. **RPC server** for the STM32 board (for basic network functions like **socket, send, recv, readfrom, recvfrom**, but also for more complex services like **lookup** (DNS), **ping** (ICMP), **HTTP client**).
2. **HTTP server** with basic authentication which can be used to configure the network (IP, netmask, gatway, DNS server) via a web interface
3. **Web configuration frontend** for the applications running on the STM32
4. **DHCP client**
5. **TFTP server**. This is used to upload the firmware on the STM32 CPU (using its built-in serial bootloader over the same serial connection used for RPC) with a standard TFTP client running on the PC.
6. **STM32 firmware programmer** (see 5 above)

2. Block diagram



The overall structure of Moonlight is depicted in the previous diagram. The iMCU7100EVB connects to the Internet, and also to a "stamp" board built around the STM32F103 CPU from ST (the board is called ET-STM32 and can be found for example at http://www.futurlec.com/ET-STM32_Stamp.shtml) using the UART. The STM32 board runs eLua and controls the display (a 4x4 array of 32x8 pixels low cost, intelligent LED display modules from Sure Electronics, available at http://www.sureelectronics.net/goods.php?id=903) via some of the CPU PIO lines, with the help of a small glue logic circuit. It can also be connected to a desktop PC via another UART if one wants to write new programs for Moonlight and then upload them to the board (but note that this is optional, Moonlight operates without the need for a PC, unless of course it runs an application that needs a PC connection). It interprets the IR signals (coming from a Sony remote control) which are used to interact with the board, uses a SD card for storage, and drives a small speaker for simple sound effects.

3. Schematics

The hardware is composed of three boards: the connector board, the controller board and the iMCU7100EVB board. The schematics of the first two, as well as the connections for the third are given below.

The schematic of the STM32 board is in *doc/schematics/STM32_SCH.pdf*.

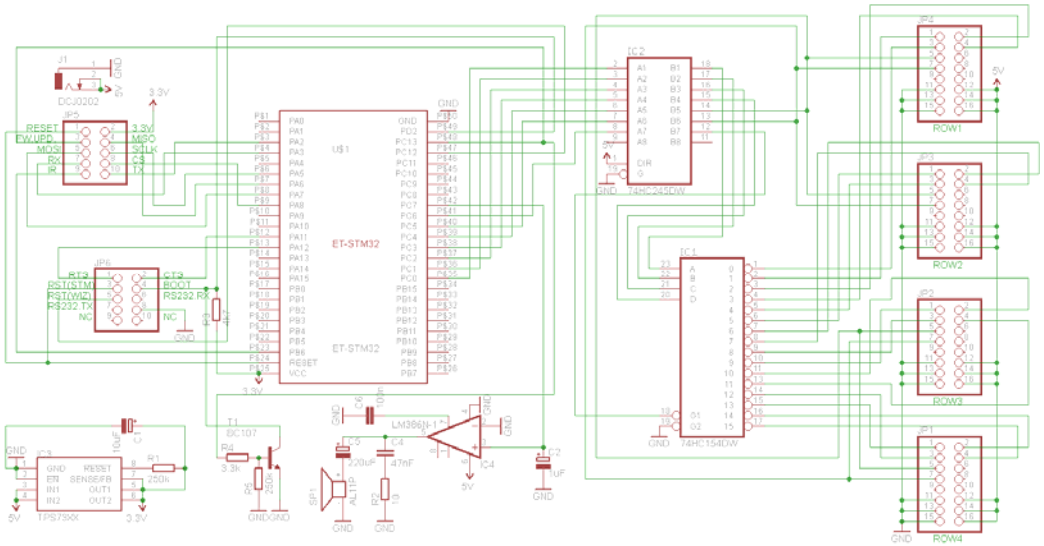


Figure 1 – The controller board

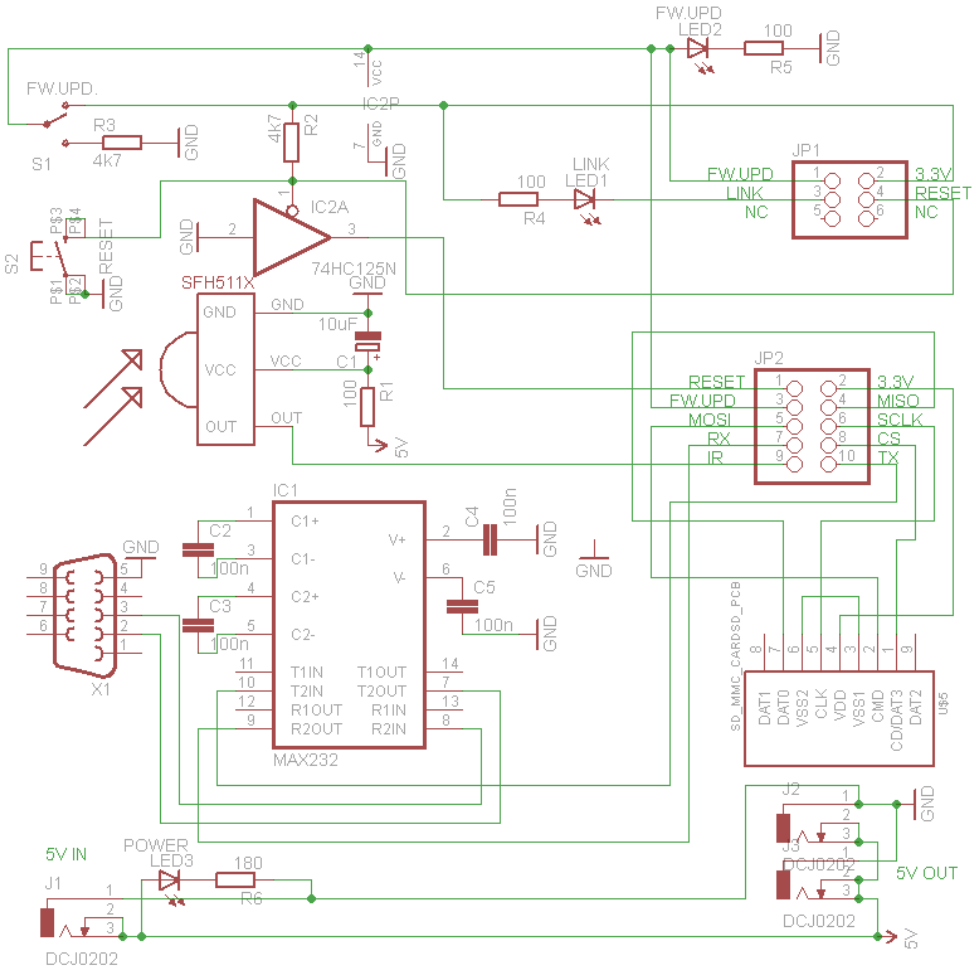


Figure 2 – The connector board

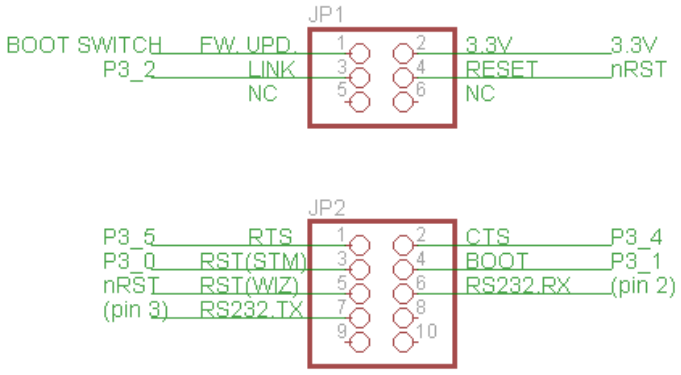
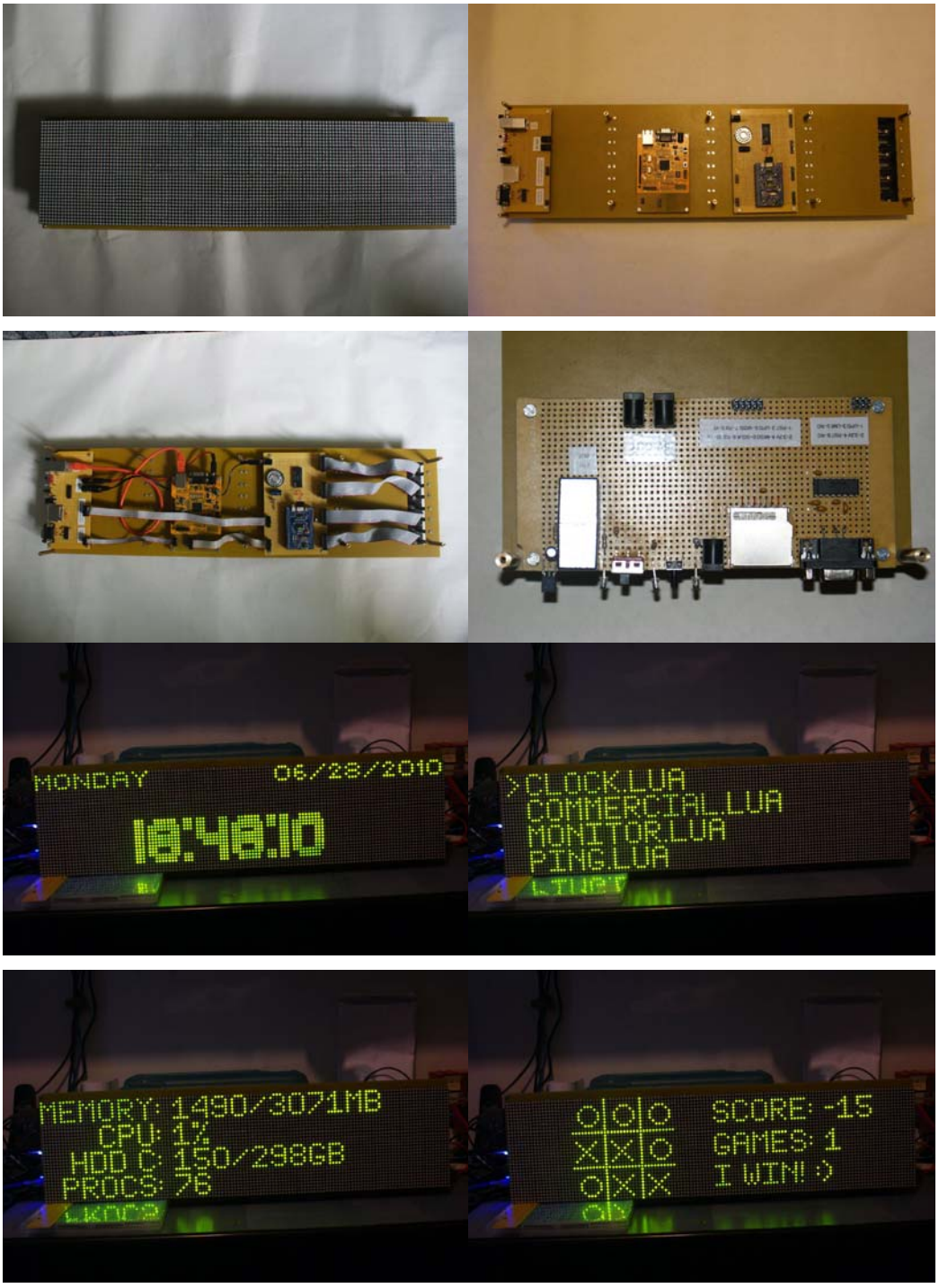


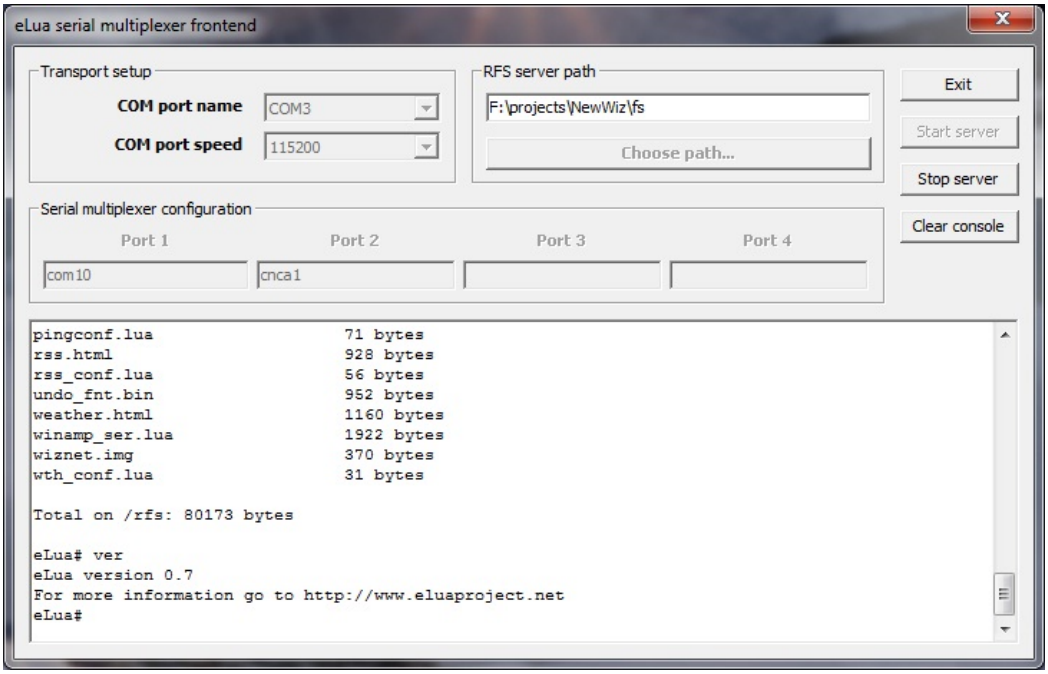
Figure 3 – The iMCU7100EVB connections

The schematic of the iMCU7100EVB is in *doc/schematics/ iMCU7100EVB_SCH.pdf*.

4. Project pictures (board and applications)



A screenshot of the PC application used to access Moonlight in interactive mode is given below.



5. Sample code

The code of the clock application (which features a TIME client and a web configuration interface) is given below.

```
local d = stm32.dp105
local r = stm32.sircs
local tmrid = 1
local server = "129.6.15.29"
local s
local ip
local delta = 2208988800 -- conversion from NIST epoch to Unix epoch
local h_delta = 3600 -- hour delta
local gmt_delta = 3
local t, ts
local total_time
local resync_time = false
local f = d.load_font( "/rom/h14_fnt.bin" )
local sync_timer_indication = -1
local tx, ty = 30, 15
local cdict = {}

local function create_default_config()
  local cfile = io.open( "/mmc/clk_conf.lua", "wb" )
  cfile:write( 'return { "129.6.15.29", "3" }' )
  cfile:close()
end

local function send_config()
  local f = io.open( "/mmc/clock.htm", "rb" )
  if not f then
    print "Unable to open HTML file"
    return
  end
  wiz.app_web_set( "clock", f, cdict )
  f:close()
```

```
end

local function write_config()
  local f = io.open( "/mmc/clk_conf.lua", "wb" )
  if f then
    f:write( string.format( 'return { "%s", "%s" }', cdict[ 1 ], cdict[ 2 ]
) )
    f:close()
  end
end
local function get_config()
  collectgarbage( 'collect' )
  if not cdict[ 1 ] then
    local cfile = io.open( "/mmc/clk_conf.lua", "rb" )
    if not cfile then
      create_default_config()
      cfile = io.open( "/mmc/clk_conf.lua", "rb" )
    else
      cfile:close()
    end
    local fconf = assert( loadfile( "/mmc/clk_conf.lua" ) )
    cdict = fconf()
  end
  ip = wiz.ip( cdict[ 1 ] )
  gmt_delta = tonumber( cdict[ 2 ] )
  print( wiz.unpackip( ip, "*s" ), gmt_delta )
end
local function acquire_time( timeout )
  wiz.sendto( s, ip, 37, "A" )
  local datestr = wiz.recvfrom( s, 4, timeout )
  if #datestr > 0 then
    local dummy, v = pack.unpack( datestr, ">L" )
    return v - delta + h_delta * gmt_delta
  end
end
d.clrscr( 0 )
get_config()
send_config()
-- Initial time acquire
for i = 1, 5 do
  s = wiz.socket( wiz.SOCK_UDP )
  t = acquire_time( 500 )
  if t then break end
  wiz.close( s )
end
if not t then
  wiz.close( s )
  d.putstr( 0, 0, "CANNOT READ TIME" );
  d.putstr( 0, 8, "FROM SERVER" );
  d.draw()
  r.getkey( true )
  d.clrscr( 0 )
  d.draw()
  return
end
total_time = 0
while true do
  ts = tmr.start( tmrid )
  -- Check remote key
  local dummy, code = r.getkey( false )
  if code then break end
  -- Read configuration
  if wiz.app_has_cgi() then
    print "CGI!"
    local pdict = wiz.app_get_cgi()
    print "AFTER CGI"
    cdict[ 1 ], cdict[ 2 ] = pdict.server, pdict.delta
    send_config()
```

```
    write_config()
    get_config()
    total_time = 5 * 60 - 1
  end
  -- Show time
  d.filled_rectangle( tx, ty, tx + 80, ty + 16, 0 )
  d.set_font( f )
  d.putstr( tx, ty, elua.strftime( "%H:%M:%S", t ), 1 )
  -- Show date
  d.set_font( d.FONT_5x5_ROUND )
  d.filled_rectangle( 0, 0, 127, 8, 0 )
  local datestr = elua.strftime("%m/%d/%Y", t )
  local w = d.get_text_extent( datestr )
  d.putstr( 128 - w, 0, datestr, 1 )
  datestr = elua.strftime( "%A", t )
  d.putstr( 0, 0, datestr, 1 )
  d.draw()
  -- Increment time
  t = t + 1
  total_time = total_time + 1
  -- Check if we need to acquire time from the network again
  if total_time == 5 * 60 then
    total_time = 0
    resync_time = true
  end
  if resync_time then
    local temp = acquire_time( wiz.NO_TIMEOUT )
    if temp then
      t = temp
      resync_time = false
      d.filled_rectangle( 126, 30, 127, 31, 1 )
      d.draw()
      sync_timer_indication = 2
    end
  end
  if sync_timer_indication >= 0 then
    if sync_timer_indication == 0 then
      d.filled_rectangle( 126, 30, 127, 31, 0 )
      d.draw()
    end
    sync_timer_indication = sync_timer_indication - 1
  end
  while tmr.gettimediff( tmrid, tmr.read( tmrid ), ts ) < 1000000 do end
end
wiz.close( s )
```

The code of the clock application features a web configuration interface which is shown below, exactly as it appears in the browser.

Clock configuration

NIST server	129.6.15.29
GMT delta	3

Update Back