'TDDY'
Traffic Data Display

General Description

Atmega128 with matrix of 512 bi-color LEDs organized as a map of major LA freeways

Abstract

This document outlines the principle of operation of a digital traffic data display (TDDY) meant to be used in public service vehicles, like public buses, emer-gency response vehicles, taxi-vehicles (cabs), delivery vehicles and/or similar.

Introduction

Collisions between emergency vehicles and other vehicles are common events. Many of these collisions occur at inter-sections. Public vehicles approaching at 90 degrees to the direction of travel of emergency vehicles are very difficult to see. Using sensors mounted on the vehicle, or in a fixed location near the intersection, or both, in conjunction with two-way radio-transceiver, GPS device or two-way pager could alert emergency vehicle drivers (and other motorists) of approaching traffic (cross traffic). In such a scenario, sensors sends information to some sort of dispatching center which process incoming data and, in return, sends data to interested parties in form of positional information to traffic warning devices mounted in vehicles.

This design presents an early implement-tation of vehicle-mounted receive-only traffic warning device. Conventionally, it uses server software hosted on PC and one-way pager which serves as client to display position of vehicles on city map.

Concept description

The idea behind the concept is to have LED diodes (mechanically) placed on PCB to reassemble the city map and to have receiving device capable to receive ASCII characters which alphanumerical pager normally does. Light-emitting diodes (LED) organized to form major freeways would give to driver"quick-to-recognize" visual information about current situation on the roads and/or freeways. The very early 3D concept drawing is shown on Figure 1.

Figure 1: The early 3D concept drawing of TDDY

Above LEDs is thought to be mounted plastic screen whose purpose is threefold: to protect LEDs, to serve as mounting base for adhesive membrane keyboard and the city map printed on adhesive plastic folia. One of the first designs of such a "LED map" (Los Angeles, CA) is shown on Fig. 2.

Figure 2: The early design of Los Angeles LED map

The block description

This application, in its heart, incorporates a 5V version of ATmega128 due to its ability to operate at higher clock rate (up to 16MHz). It uses 14.7456 MHz crystal whose frequency is internally divided by integer number to provide a zero-error system clock for UARTs. The same clock frequency is used to generate timer interrupt, whose ISR is used to transfer and in turn, supply row and column latches inside the CPLD with the data. Internally CPLD contains hold registers whose purpose is to hold data while being displayed using LEDs. Individual bits in such formatted bit-stream controls bi-color LEDs. The simplified concept diagram is shown on Figure 3.

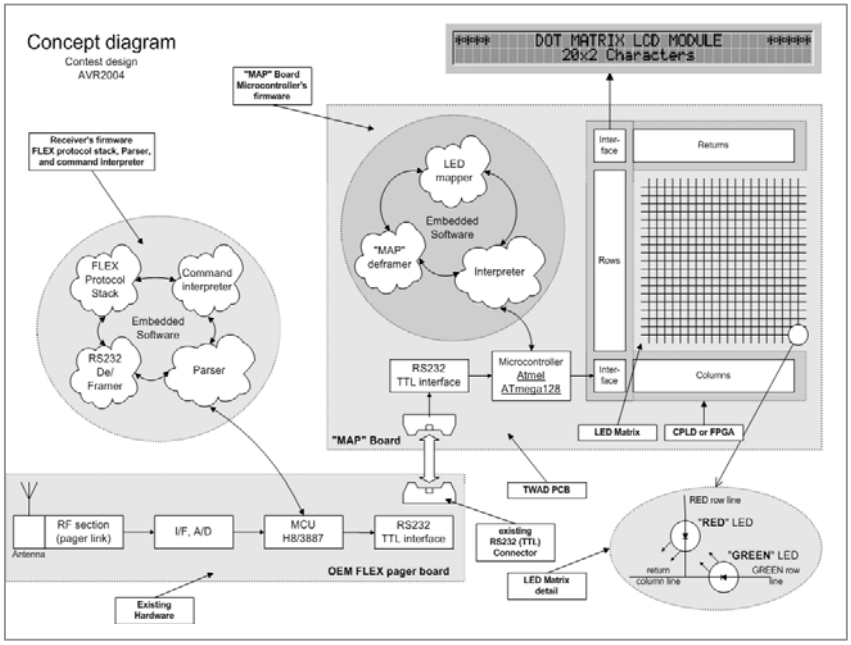

Figure 3: The block diagram, concept

The LED matrix is organized into 16 rows and 32 columns. Row and column decoders along with control logic are internally implemented in CPLD. A bit stream represent on/off condition for each of 32 LEDs per row, with 1 bit for both red and green color, thus 32 bits per column per color, 16 times per second for each row, yields 64 bits is transferred from CPU to CPLD each 1/16 second. In another words, the "screen" is being refreshed 16 times per second, having LED diodes work in impulse regime, utilizing 1:16 duty-cycle. In such a regime, the power is significantly conserved while having LED's brightness adequate for this purpose.

The custom formatted message is being transferred via pager communication link. The message is made upon simple STX/ETX protocol with commands to turn red, green, yellow or none (black) color. The message is prepared on host using off-the-shelf mapping software (MS MapPoint) which outputs the regular geo coordinates (longitude/latitude), further-on coordinates are calculated and encrypted into a map position using lookup-table, upon which regular email is being formatted and generated containing LED light turn-on/off commands and finally aired via pager infrastructure equipment to reach the pager receiver in a form of regular page, similar, if not the same, to SMS message. Once received, the parser decrypts the message, and demapper converts commands into an actual light being displayed on LED matrix. LED's red and green light combined give off yellow light. Similarly, if needed, email text can be displayed on dot matrix LCD module.

Hardware: FLEX™ pager description

The FLEX™ Pager Controller "FPC" is an off-the-shelf pager receiver manufactured by Automated Engineering Corp., Tampa, FL. Figure 4 shows a physical dimension (scale 1:1) of such a board.

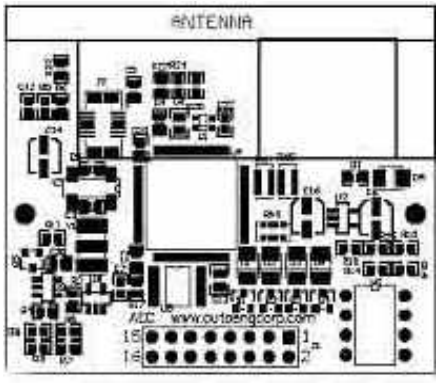

Figure 4: The pager board outline

The pager communication protocol is the standard Motorola's FLEX™ protocol with a microcontroller, interface and power supply module, all integrated into single board. The pager receives pages to control output functions or to run one of ten user programmed sequences. The module contains four open collector output channels, capable of driving relays or other output devices. The four 5V digital inputs along with the 2 analog inputs may used by the user programs to alert the output sequences. The inputs may interface to sensors or switches. These I/Os are left unused. A full duplex serial channel is provided to output all incoming paging messages and to control the module's Operating System (OS). A serial data are fed to Atmel CPU's serial channel directly (TTL), no RS232 level shifters required. The pager's OS provides basic commands similar to paging messages. Ten programs of 100 statements can be stored and controlled by the OS and paging messages. When interfaced to the PC or microcontroller, the OS can provide TRACE information about the running user program. The FPC programming language consists of three-dozen distinct commands. The internal Atmel CPU's parser recognizes and uses a subset of these commands to exchange received messages and control the pager.

Hardware: The CPU

This application, in its heart, incorporates a 5V version of ATmega128 due to its ability to operate at higher clock rate (up to 16MHz). It uses 14.7456 MHz crystal whose frequency is internally divided by integer to provide a zero-error system clock feeding UARTs. The same clock frequency is used to generate 1/16 s interrupt, whose ISR is used to supply row and column latches data to the CPLD. Atmel ATMega series of microcontrollers comes stuffed with a wide variety of commonly used peripherals and host of them are used in this design. For that reason, a minimal number of external passive and active components are being used. The only active components attached to CPU are CPLD which contains the rest of the logic, and 3.3V, 5V tolerant buffer chip used for interfacing to the pager receiver. Pager receiver's serial port is connected to CPU's UART0 port, and UART1 is used for debug purpose and future expansion, connecting GPS receiver in particular. Since pager has only one port available, this second, UART1 port is also used for redirecting incoming pager messages to UART1, further on being collected on development PC for verbose/debug purpose. Second serial port is also used for communication between Atmel CPU and development PC.

I tried to use 6 and ½ available 8-bit ports as gently as possible, always having in mind future extensions. Port F is sparsely used, only for programming and ICE purpose. Port G is left unused. Port A (analog, lower address/data bus) and Port C (higher address byte bus) are used to carry control bits between CPU and CPLD. Port C is normally higher address byte so I use it to address different "slots" and "rows" of LED matrix (more about later). Port A is used to transfer data bits from CPU to CPLD. Having extensive number of available I/O lines on CPLD, Port A and C are meant to be buffered inside CPLD and used for connecting external SRAM (which normally require a latch) to the CPU, thus keeping components count on minimum. External SRAM is thought to be used for storing messages, in a form of large ring-buffer. For this design entry I used only on-chip RAM space (4 kb), which, for this purpose, turned out to be more than sufficient. Port D's I/Os are used as input for keyboard scanner, RX/TX I/Os of UART1 as well as keyboard interrupt and low battery indication inputs. Part of Port B is used as outputs for indicators, while rest of the Port B and Port E carries CPLD control bits as well as auxiliary signals.

Hardware: The CPLD

The CPLD, complex programmable logic device, Xilinx's 5V part XC95216 in PC160 package contains the rest of digital logic used in this application. In its essence CPLD contains simplified graphic controller: the column decoder, the row decoder, hold and shift registers control logic and I/O registers, all components usually found in standard off-the-shelf LCD controllers. Outputs of the CPLD are fed to source and sink drivers which actually drive LEDs with

current. The LED matrix is organized in 16 rows and 32 columns, which yields up to 512 LEDs to be used with the control logic. The row decoder accepts BCD coded number supplied by CPU placed onto lines BCDROWS0 to BCDROWS3. Internally implemented 4-to-16 decoder decodes BCD number and asserts related row. The simplified block diagram of LED matrix logic is provided on Figure 5.

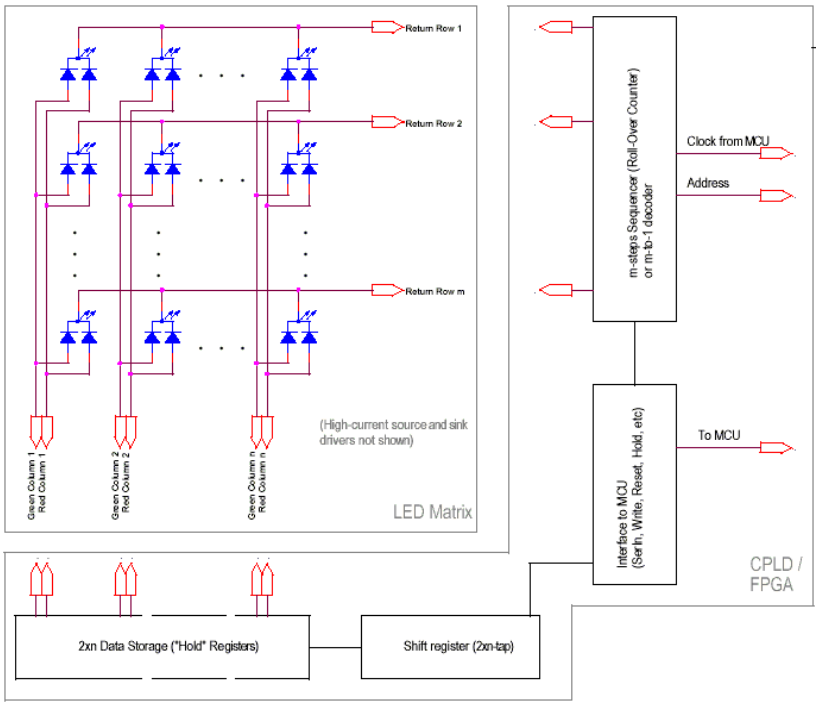

Figure 5: Simplified CPLD logic block diagram

The data storage "hold" register block is consisted of storage registers which stores bits (LED turn on/off bits) for currently displayed row and source drivers which actually drives LEDs with at least 10 mA current. That yields minimum of 640 mA required current to drive all 32 bi-color LEDs at once, providing that each LED consumes 20mA (10mA per red or green light) in normal operation. Each output line of TDA62783 source drivers has current limiting resistor to limit current to 10 mA.

The row line is input to sink driver, i.e. when the row line is asserted the corresponding sink driver (ULN2803) is enabled to sink the current from associated row. Each row can sink up to 64 LEDs (32 bi-colors) which yields up to 640 mA per row, assuming that each LED consumes 10mA in a normal operation. For that reason driver's outputs are doubled to ensure proper operation in a safe range (ULN2803 normally can sink up to 500mA per line)

The shift register doesn't convert parallel data into serial data, as one could expect judging from the name. Rather, the "shift register" decodes binary coded input SLOT0 and SLOT1 signals, serves as 2-to-4 decoder, and address one of four 8-bit hold registers to receive (latch) the incoming 8-bit data. By doing so, the shift-register virtually "shifts" (or switches) data bus internally inside CPLD to enable hold registers to "hook on" data bus and accept the date currently being delivered.

In such a scheme, it takes four data cycles (the execution times worth about 32 CPU assembly instructions) to completly update one LED matrix row (32 bi-color), which makes possible to ramp up the refreshing freq-uency to close to 1 kHz, without having LED map to flicker or data loss.

Hardware: The LED matrix

To make 512 LEDs easily "countable" by firmware (dividable by 4 or 8), I split 512 LEDs into 8 clusters of 64 LEDs each. Each cluster is further-on addressed by slots (see 001-MTRX.SCH), total of 8 slots. Each slot contains 8x8 sub-matrix, i.e. 8 rows and 8 columns of LEDs. Columns are fed to one of aforementioned 8-bit "hold" registers, while rows are fed to 4-to-16 decoder, effectively making 2 "über-rows" each sharing the same SLOT0,SLOT1 signals used to decode, here-named, "über -columns" of slots. The reason for this is relatively simple: easier maintaining the code, easier naming convention, easier debugging, easier placement and easier PCB layout.

Logical organization differs from physical organization. LED are mounted on the

board to reassemble LA map of freeways. During layout I did a lot of swapping components in order to help PCB auto-router to achieve 100% routability. LEDs get swapped, thus initial logical organization (naming) didn't help me here, so I used a lookup-table placed in the firmware to match logical position on the map with the physical position on the PCB. Latter during initial design, I have added a dot matrix LCD but I didn't want to change too much an existing schematic, or rather the CPU port's organization, so I implemented a "hand off" register inside CPLD to connect the LCD's interface circuit. The 20-characters-by-2-rows LCD utilize well known Hitachi's dot matrix LCD controller which "knows how to talk" in 4 bits, thus saving some I/O pins, and some of these signals are bi-directional. I have used some of unused CPLD pins to implement bi-directional 4-bit LCD interface.

All aforementioned functions are done in an old-school "all-schematic-no-VHDL" collection of schematics, put into sub-directory \CPLD_Schematic (VHDL source would fit on one letter-size page with 8 pt font, it would be a challenge to explain and wouldn't serve large community of CCI readers).

Hardware: The rest of circuitry

The rest of circuitry is relatively simple. A keyboard is of simple scanning type with added fast-switching diodes to provide an interrupt (see 001-INOUT.SCH). In such a scheme, firmware doesn't need to scan the keyboard "once-in-a-while" using timer interrupt, only when actual keypres occurs. Therefore, upon keypress, a low logical level on KBD0..3 lines are fed to pulled-up KDNINT* line, which in turns generates an keyboard interrupt, making CPU to scan lines and figure out which key has been pressed. Key scanning is a standard one: bring row line to logical zero and see which column line is brought to logical zero as well, then map the position it into a keypress code. This interrupt scheme is bit trickier to implement on software side, but it helps to avoid multiple timer interrupts, which may cause the map to flicker.

Power supply and management circuitry is also meant to be very simple having in mind that TDDY is suppose to be powered from vehicle's battery. The vehicle's DC 12V is supplied thru DF10S, a 2A recfifier used to avoid misconnection of ground and +12V then stepped down to 5V via oldie-goldie 7805. Not ultra-efficient, but sufficient to be used in vehicles. To avoid wasting of power while vehicle idles on the parking lot, the power saving scheme is being implemented. When CPU goes asleep (after 5 minutes of idle time) it disconnects remaining power-hungry components (CPLD, bipolar drivers, LCD's backlight and pager) via associated power-MOSFETs.

Hardware: The PCB

The layout of the PCB was a real challenge. Initially I planed 4 layers, did layout on 8 layers, but the manufacturing price was jaw-dropping so I modified LEDs padstacks. Modified LED padstacks caused some complains from PCB people and ping-ponging of emails and a quite a deal of delay. But I ended up with 6 layers (2 power planes, 4 signal layers) and saved money. The very small 1.6 x 0.8 mm LED padstack I modified to include a drill on cathode pad, allowing cathodes to be connected using thru-hole.

The assembly was yet another challenge. The top side of the PCB needed to be populated only with LEDs, due to small spacing between LEDs and protection cover plastic on the top of the proposed enclosure (see Figure 1). All other components are placed on the bottom side of the PCB. Due to very small physical size it took me days to stitch together and solder down 512 LEDs. The signal traces are 3 mils wide, inner layer traces are 6 mils wide, power pins on active components are connected, other than via power planes, with 12 or 16 mils traces. Power plane is split into thre separated sub-plane to accommodate the power scheme. Figure 6 shows the silk-screen of resulting PCB, shown scaled down to 50%.

Those folks familiar with L.A. area will easily recognize characteristic layout of major LA freeways.

At the lower left corner are visible part labels for LEDs associated with the rosetta to indicate which direction is currently being shown on the map (N-E or S-W).

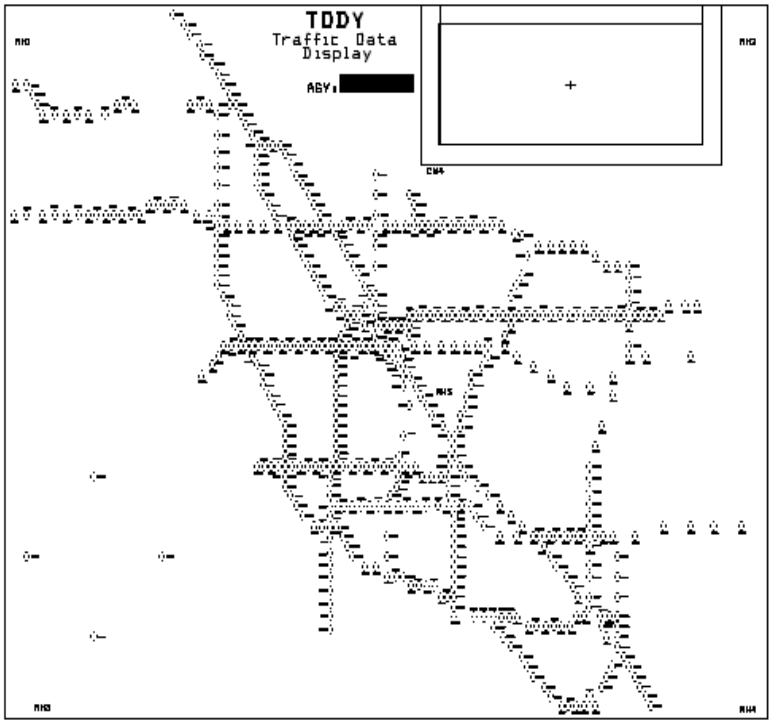

Figure 6: The PCB's silk screen, scale 1:2

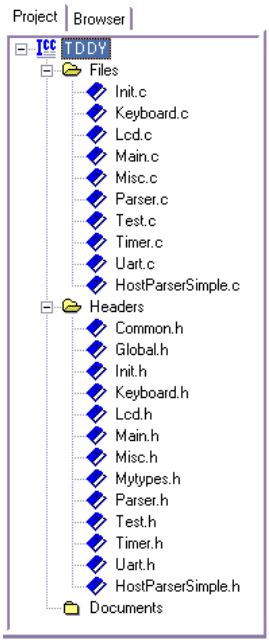

Figure 7: The project tree

Software: The Firmware

The entire firmware is done using Imagecraft's ICCAVR v6.23b C compiler, with minimum or next-to-none usage of assembly language. In few occasions I was using assembly just for very low functions usually in a form of some frequently used macros. I was using ICCAVR Standard Edition, means no code optimization used. Figure 7 shows the firmware project tree of within the Imagecraft's IDE.

The central part of the program is a super-loop function, see Main.c file. Since I had several sub-versions of the firmware, I extensively used compile-time compiler directives, as it may be seen throughout the source code, which I have provided in a separate file, FWCODE.ZIP.

Upon reset and startup initialization, on-chip peripherals are being initialized in ATmega128Init() function. The firmware then perform test to see if pager responds. The test is simple waiting for CR/LF sequence from the pager. If it times out, the main function idles while showing the error message on the LCD screen. After that timeout, the power has to be recycled.

When initialization went OK, the main function enters the main super-loop where it first checks for user input on keyboard. If flag KBHIT is set, the kbscan() function returns a character, which is subsequently being interpreted. Upon user's keypress, the interrupt INT0 is being generated, see Keyboard.c file. The KBHIT flag is set within keyboard's ISR, portd_int0_ISR(), later being read/interpreted within main loop. In this version, firmware can perform a few actions, but the most important action is checking which direction (N-W or S-E) user wants to see on the map. The function Change_Direction() switches different set of static variables which holds turn-on/off state for LEDs. Four volatile declared static variables hold LEDs states, see Global.h file. These variables are organized as static array, for example Red_NW[4][16], i.e. 512 bits wide, meant to hold states for 512 LEDs. Upon received and interpreted message, parser calls lookup-table to map bit position within array to actual physical position of LED on the map. As I mentioned earlier, logical position within the array doesn't necessarily matches the physical position of LED (for example LED named LD101 is not automatically the 101st bit in the array) so look-up is being performed then Update_LEDs() function copy (or rather to say send) entire array to CPLD. One array is defined for each distinct color, red and green. Identical bit positions

within each array are logically ANDed to define the color being displayed. If yellow color is needed to be seen, related function simply copy to CPLD an identical bit position from both red and green array. Logically, if previously lit LED need to be turned off, bit position in both red and green array is cleared, which yields no color, subseq-uently turning off the corresponding LED. Normally, if red color need to be lit, corresponding bit position in "red" array is set while bit position in "green" array is cleared. The same goes for green LEDs. With such organization four colors are defined using two arrays:

Bit pos. in array		Color
Red	Green	
0	0	Black, Turn off
0	1	Green
1	0	Red
1	1	Yellow

Two pairs off such arrays are defined for each direction: N-W (North-West) and S-E (South-East), totaling 256 bytes of RAM being used for holding LED states. Pressing key "DIR" on membrane keyboard toggles the direction arrays and regarding LEDs being lit. Four bits within an array are dedicated to rosette LEDs, therefore rest of 508 LEDs are used to compose the L.A. freeways map.

After parsing, two functions Update _LEDs() and Turn_LED() updates the storage arrays (see Parser.c file), while main transfer of data is provided within Timer1 ISR. Timer1 ISR is the low level function which talk drectly to the CPLD. Here, CPLD bits are manipulated to provide a sort of XON/XOFF handshaking. Simplified algorithm for data looks like this:

```
- enable latching into row latches
- write a new value into row latch
- enable latching into column latches
- for loop slot = 0 to 3
    -   enable green columns
    -   place slot address onto PORTC
    -   place actual data onto PORTA
    -   write to green 'hold' register
    -   enable red columns
    -   place actual data onto PORTA
    -   write to red 'hold' register
- write new value into column latche
- update row counter (next row will
  be updated upon next ISR entry)
```

Software: The Protocol

This basic concept of this interface protocol is independent of the lower level protocol that passes the commands and responses between the pager and the CPU. However, this implicitly assumes an ASCII protocol when it specifies the lengths of the commands, responses, and component fields. Since I used one-way pager this protocol doesn't include status reporting except basic error catching in form of error propagation from lower to higher level functions. Current state of the art of pager infrastructure and pager protocols allows sending only printable characters while prohibiting any control character further restricting the length of the message to about 200 characters including "From" and "Subject" fields. For that reason I defined very simplistic protocol for messages being aired over pager network, with commands for changing states (turn on/off) of LEDs.

Commands sent over pager network are defined similar to escape sequences for printers, as follows:

Description	Command
Turn On Red LED	^R
Turn On Green LED	^G
Turn On Yellow LED	^Y
Turn Off LED	^O
Toggle N-E direction	^N
Toggle S-E direction	^S
End of Message	^E

The protocol has the following structure:

```
<From: [text]><LF>
<Subject: [text]><LF>
<CMD><LED#>
[<delimiter><CMD><LED#>]
<CMD End of message>
```

The "From" and "Subject" are standard field of email message sent over pager network, while body of the email message contains commands described in the above table. These fields are separated with the "Line Feed" character. The LED number is positional number of bit within an array described earlier. The delimiter is any

printable character used to separate individual commands within string, usually a semicolon. Command ^E is used to terminate the command string. The command string is prepared, assembled and sent over email using host software. The length of email is limited and character count of entire email message being sent cannot exceed 200 characters, otherwise an ISP, the pager provider, will crop it. For example, the message and command string may have the following content:

```
From: test@usa.com<LF>Subject: more tests<LF>^N,^R101,^R203,^R417,^S,^0012,^G234,^G501,^O499,^Y212,^E
```

which reads as follows:

- "From" field (skip)
- "Subject" field (skip)
- toggle N-E direction
- turn on red LED #101
- turn on red LED #203
- turn on red LED #417
- toggle S-E direction
- turn off LED #12
- turn on green LED #234
- turn on green LED #501
- turn off LED #499
- turn on yellow LED #212
- end of message

For development purposes, I have also defined a helper protocol with commands sent by "Host" (a VB application, basically a terminal) like "RESET" and "TURN". Only few of them are being used mainly to put CPU in the well defined state during initial phase of development (see HostParser-Simple.c file). The output of such functions (if any) is redirected to UART1 port, which also serves as debug/test port. Status messages from functions as well as traffic messages are also redirected to this port.

Software: The Server Side

Email messages containing command strings are prepared on host PC system running custom sever program and sent to pager provider (or ISP) that in turns make sure the message will be delivered over the pager network. The screenshot of the prototype server program called GOTO is shown on Figure 8.

The GOTO program is based on mapping software Microsoft MapPoint 2004 and its ActiveX component. The skeleton of the program is made using Visual Basic 6.0 and the email engine is standard Microsoft technology called CDONTS (Collaboration Data Objects for Windows NT® Server) accessed using MS Scripting engine.

Figure 8: GOTO, prototype server program

Forming the email using CDONTS object may look like this:

```
'send email and attachment (if any)

 sub SendEmail (strFile)

       Dim objMail
       Set objMail= server.CreateObject
           ("CDONTS.NewMail")

       with objMail
              .From = "test@usa.com"
              .To = strEmailTo
              .Subject = strSubject
              .Body = strBody
              .Attachment = strFile
              .Send
       end with
       Set objMail = nothing ' clean up

 end sub
```

IMHO, quite effective and easy to use.

Using CDONTS I avoided using complex low level functions and/or using third party components. In above VB excerpt, string strBody contains formated commands to be sent to the pager.

The actual commands are generated by internal GOTO program's functions which takes MapPoint ActiveX component geo coordinates as an input. MS MapPoint has ability to attach a "pushpin", an object "push-pined" to the location of interest, which is a loca-tion of the traffic incident or other information.

"Pushpin" object contains longitude and latitude data, which I'm converting into LED# by simple calculation that takes location of the "pushpin", physical dimension of the map and degrees per pixel (d/p) constant. The LED number is generated using look-up table to match vicinity of the "pushpin" and LED# to be lit. Each LED# data is concatenated to form a command string then wrapped into an email message and sent out using CDONTS engine.

Testing

It was possible to easily test the whole system even without using a bit expensive pager account and above mentioned server software. Any program capable of sending ASCII email messages is adequate to test the finalized hardware, for starter Microsoft Outlook Express, simply by sending ASCII email to pager account. That way, I have tested the system by sending number of emails to make al LEDs lit. Next couple of pictures (Figures 9, 10 and 11) shows results of the test. To get better contrast I took pictures in dimmed room with longer camera exposure:

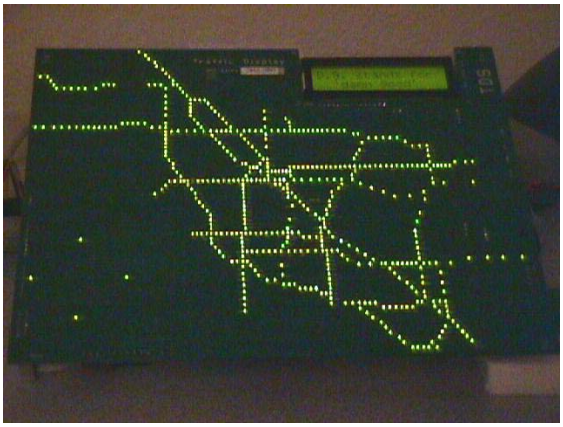

Figure 9: All GREEN test

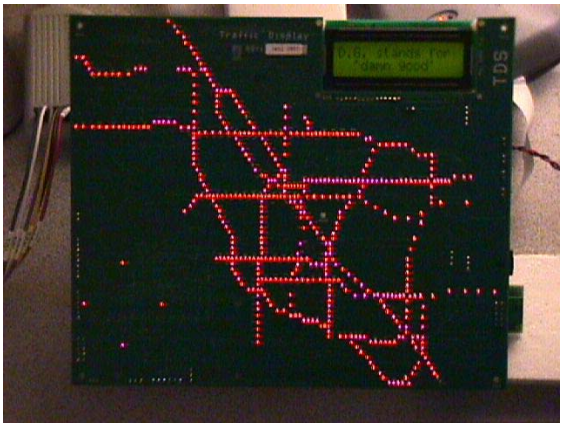

Figure 10: All RED test

Figure 11: All YELLOW test

Presented bellow are pictures (Figures 12 and 13) of the entire system, close capture. On the side of the main PCB is visible a mock-up keyboard and power supply wire. Figure 14 shows TDDY system running, with most of LEDs green color on.

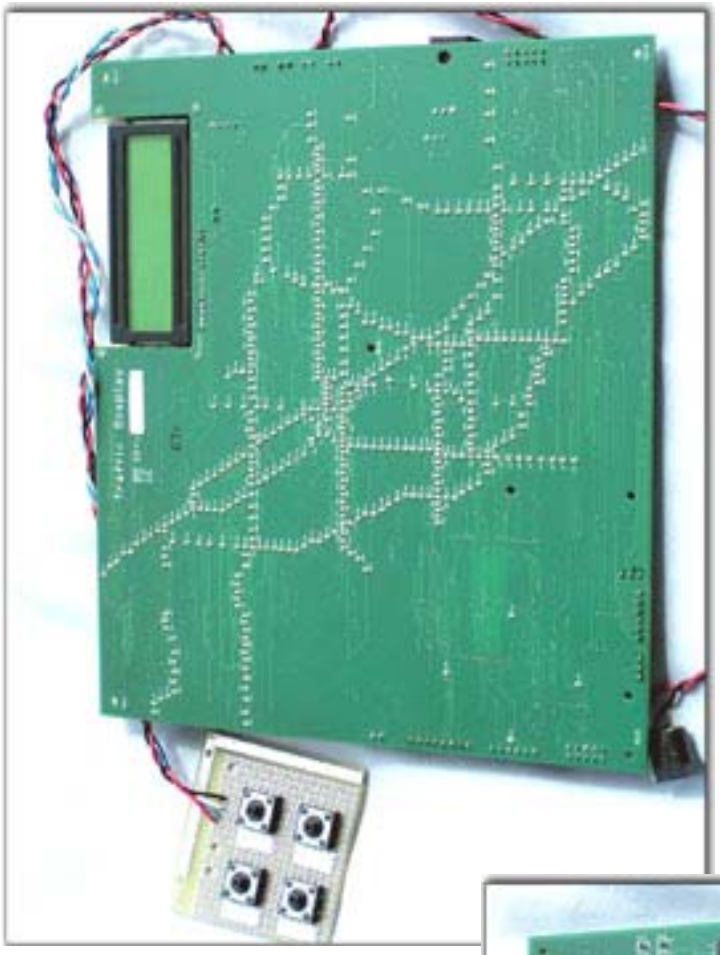

Figure 12: The TDDY board, top side. Top side contains only bi-color LEDs. Flattened with the top side plane is the top face of the dot matrix LCD. Right to the main board is visible the mock-up keyboard.

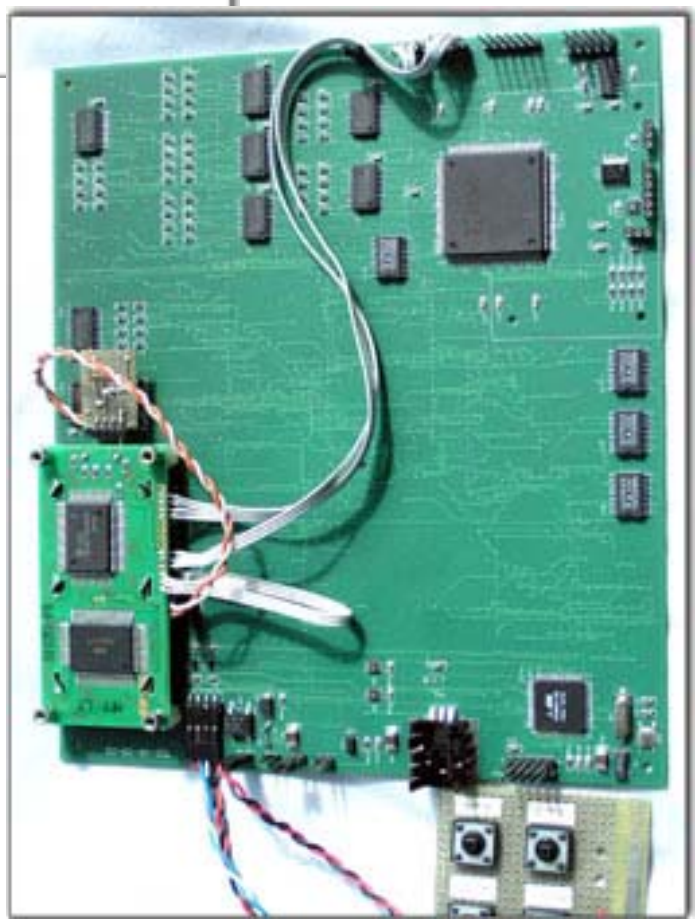

Figure 13: The TDDY board, bottom side On the left is visible Atmel Atmega128 microcontroller while on right is visible Xilinx CPLD in PQ160 package. Right to the main board is visible the mock-up keyboard. The dot matrix LCD is connected to the CPLD via flat cable.

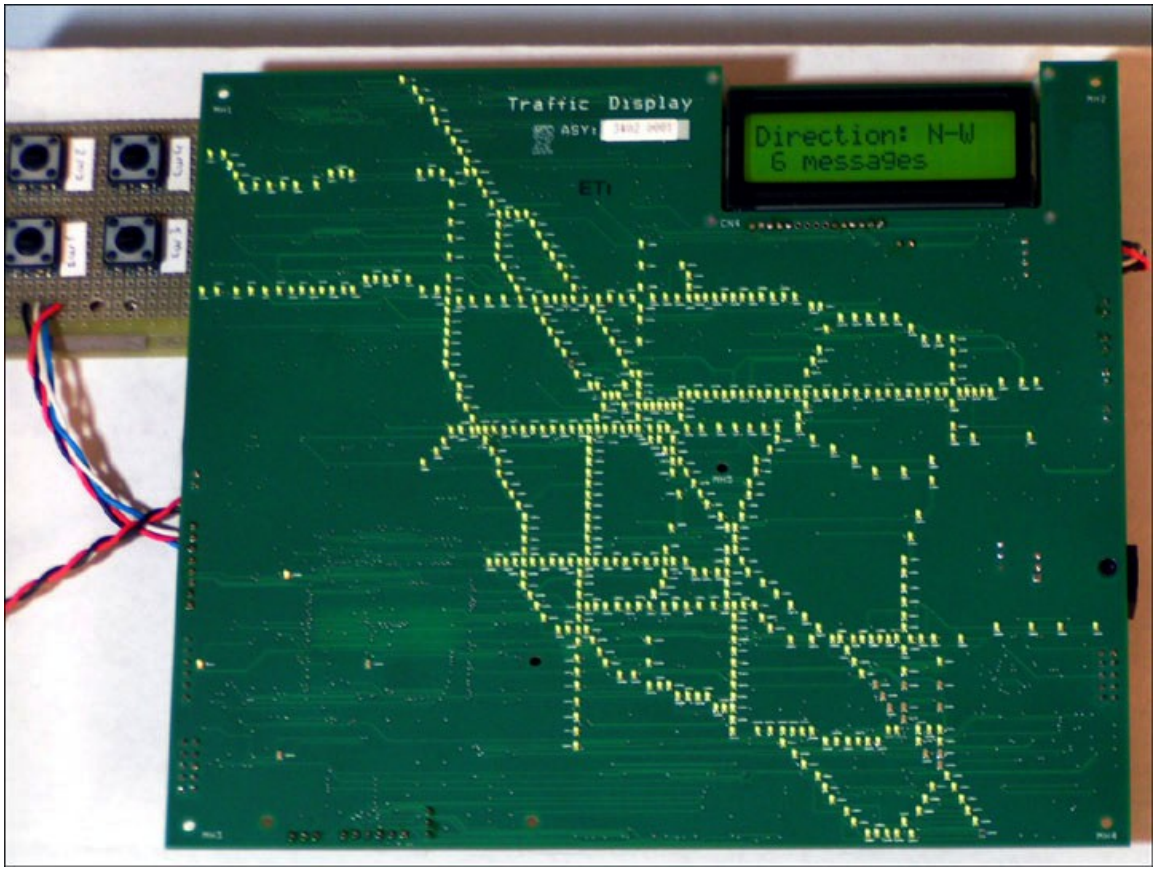

Figure 14: The finalized TDDY, contest design #A3437, in action

Tools

Tools I have used to design the TDDY contest design are:

- Atmel AVR Studio 4, version 4.7.240
- Atmel STK500 Development System
- Atmel STK501 extension
- Atmel ICD (In-Circuit Debugger)
- Cadence OrCad Lite, Release 9.2
- ImageCraft ICCAVR Standard, version 6.23b
- Microsoft CDONTS extension
- Microsoft MapPoint 2004
- Microsoft Visual Basic Enterprise version 6.0 + Service Pack 5
- Xilinx CPLD StartUp kit
- Xilinx ISE WebPACK version 4.2WP3.0
- Xilinx ModelSim version 5.1
- Xilinx Downloader cable