Towards TinyOS for 8051

TEP:	121
Group:	TinyOS 8051 Working Group
Type:	Informational
Status:	Draft
TinyOS-Version:	1.x
Author:	Anders Egeskov Petersen, Sidsel Jensen, Martin Leopold
Draft-Created:	15-Dec-2005
Draft-Version:	1
Draft-Modified:	27-Mar-2006
Draft-Discuss:	TinyOS 8051 Working Group List <Tinyos-8051wg at mail.millennium.berkeley.edu>

Abstract

This TEP covers our effort of porting TinyOS to the nRF24E1 platform. We ported the basic modules of TinyOS: Timer, UART, ADC and LEDS.

1. Project Outline

The original 8 bit 8051 chip is a member of the mcs51 family and was developed in 1980 by Intel. It is still to this date one of the most widely used microcontrollers. Porting TinyOS to the 8051 System on chip architecture makes perfect sense - the mcs51 family has been thoroughly tested, it is relatively cheap and it has a reasonable small footprint - which makes it ideal for embedded solutions and sensor networks.

For this work, we use a Nordic Semiconductor VLSI nRF24E1 evaluation-board [NSC]. The board contains an Intel 8051 compatible MCU with 4KB program memory, a 16 MHz clock, 3 different Timers (one being 8052 compatible), a 2.4 GHz wireless RF transceiver and 9 input 10 bit ADC, SPI and a RS232 Serial interface. The nRF24E1 board was chosen because the radio component matches the radio on the specially designed DIKU/DTU HogthrobV0 [HOG] boards used for research purposes at DIKU [PEH].

We ported a subset of TinyOS for the 8051 platform consisting of the Timer, UART, ADC and LED modules. We did not port the radio module and the underlying SPI-bus code.

This works attacks the two most immediate problems when porting TinyOS to 8051: the toolchain and the hardware abstraction components. The first problem when porting TinyOS to 8051-based platforms concerns the toolchain. The Gcc compiler does not support 8051. This is a major issue as the code generated by the NesC preprocessor is tailored for gcc. The second problem concerns the hardware abstraction components that must be specialized to the 8051 idiosyncracies. In a perfect world, such a specialization should not require to modify any interface. Unfortunately, we needed to modify some of the interfaces to accomodate the 8051 features.

This work was done under the supervision of Martin Leopold at University of Copenhagen.

2. Project Approach

The approach to the porting project has been pragmatic. The focus has been on producing working code, so testing and debugging have been key elements of our work. The process has been to implement new functionality in small iterative steps and do testing simultaneously.

To bootstrap the development without a JTAG module or alike, we built a small LED expansion board attachable to the port logic. The LEDs was an easy way to get instant low level test output. We also built a small stimulator based on a potentiometer (variable resistor) to get valid input from the ADC pins.

The following TinyOS application programs have been written and tested:

- Empty - test of port logic and tool chain
- mcsatomic - test of atomic and interrupts
- mcsBlink - test of LEDs
- mcsBlinkTimer - test of Timers using LEDs
- mcsSerialTest - test of UART code, simple input/output one char
- mcsSerialTest2 - test of multiple byte output
- mcsTimerSerialTest - test of UART controlled by Timer interrupts
- mcsADC - test of ADC code with Timer and LEDs

3. Development Environment and Tool Chain

The following subsections describe the different development tools, their selection and interconnection.

3.1 Selection of Development Tools/Compilers

A large number of 8051 compilers exist primarily for the DOS and Windows platforms. We have focused on two popular and regularly updated compilers: KEIL and the Small Device C Compiler (SDCC).

SDCC is an open source project hosted on the Sourceforge website, whereas the KEIL C51 compiler is a commercial compiler and Integrated Development Environment (IDE). The debugger for SDCC (SDCDB) is still fairly experimental. The KEIL suite runs on the Windows platform, and has a good interactive debugger and simulator. KEIL and SDCC accepts roughly the same syntactical dialect, which eases the work of moving between the two compilers.

During our work with SDCC and SDCDB we encountered numerous problems and bugs. SDCC 2.4.0 suddenly returned 'fatal compiler errors' with no apparent reason. After an update of SDCC from version 2.4.0 to 2.5.0 (most recent release) the error disappeared, the code compiled, but it still did not work correctly on the board. We also discovered a serious problem regarding sign bits in SDCC 2.5.0. SDCC made a type error when reading a 32 bit signed value. Apparently SDCC did not interpret the sign bit correctly, so a very small negative number was interpreted as a very large positive number as if the value was unsigned. KEIL however interprets the value correctly. The bug was submitted by us and fixed by the SDCC development team in just two days, but the timer module still does not work using SDCC.

Our attempts using SDCC's debugger/simulator (SDCDB) was equally troublesome. SDCDB simply stopped at address 0, and running or stepping through the code returned us to the UNIX prompt with no error message. Without SDCDB, we had no debug possibility and we were forced to rethink the tool chain. We decided to substitute SDCC with the KEIL development kit. This gave us a working debug environment - with minimal change to the already produced code.

3.1.1 Our Recommendation

In our experience the SDCC compiler and associated tools are not yet mature enough to support our development. We recommend pursuing other alternatives such as KEIL or other compiler suites.

We continue to mention SDCC in the remaining text, because we encourage the use of open source software and cross-platform development. We hope SDCC will prove an reliable alternative in the future.

3.2 Tool Chain Overview

The following figure and sections are an overview of the current tool chain. The tool chain is based on TinyOS 1.x, NesC 1.1.3, avr-gcc 3.4.3, PERL v. 5.8.6 and SDCC 2.5.4 or KEIL C51 version 7.20.

Each step in the tool chain will be explained in the section below.

```
             Mangle-
TinyOS       script
-----> app.c -----> app_mangled.c --------> app.hex ------> nRF24E1
 NesC         PERL                SDCC/KEIL         nRFPROG
```

3.3 Description of the Tool Chain

The compilation of the TinyOS test program outputs two files, a 'main.exe' file and an 'app.c' file. The 'app.c' file contains all the needed code to run the TinyOS application. However the C code produced by NesC cannot be compiled for the 8051 platform directly.

One solution could be to alter the syntax NesC produces for this specific platform, by modifying the source for NesC. However as a first step we chose not to make changes to NesC, but instead changed the content of the NesC output file 'app.c'. We inserted an extra step in the tool chain in the form of a mangle script. The mangle script works as the rope, tying the output from NesC to the input of SDCC or KEIL.

After running the mangling script on the 'app.c' file we obtain an 'app_mangled.c' file which can be compiled by either SDCC or KEIL. This produces a hex file that is transferred to the chip by the nRFPROG software.

3.4 Description of the Mangling Script

The mangling script is written in PERL, a commonly used general purpose scripting language with powerful pattern matching capabilities and extensive handling of regular expressions. The mangle script handles all currently known problems, and it can easily be expanded to handle additional alterations.

To run the mangle script use the following syntax:

"./sdccMangleAppC.pl -KEIL -file build/mcs51/app.c > build/mcs51/app_mangled.c"

or

"./sdccMangleAppC.pl -SDCC -file build/mcs51/app.c > build/mcs51/app_mangled.c"

The 'sdccMangleAppC.pl' script handles a number of needed alterations:

- it alters the SFR and SBIT declarations for SDCC and KEIL respectively
- it convert 64 bit data types to 32 bit
- it alters the reserved SDCC keyword data to _data
- it removes inlining directives
- it removes preprocessor line numbering
- it alters $ in identifiers to underscore
- it alters GCC interrupt declaration to SDCC syntax

Each of these alterations will be explained in the sections below.

3.4.1 SFR and SBIT Declarations

In order to make TinyOS accept the 8051 special function registers (SFR) and special bit variables (SBIT), we have included them into the TinyOS 8051 platform folder as a 8051.h file.

SFRs are located on an address dividable by 8, whereas an SBIT addresses a specific bit within these SFR.

In order to make TinyOS accept the SFRs we have type defined them in the NesC code as:

```
typedef int sfr; sfr P0 __attribute((x80));
```

which is altered to

```
//typedef int sfr; sfr at 0x80 P0;
```

for the SDCC compiler in the mangle script and

```
sfr P0 = 0x80;
```

for the KEIL compiler and similar for the SBIT declarations.

NOTE: The SDCC website refers to a PERL script (keil2sdcc.pl - last updated June 2003) for translating SFR and SBIT declarations from KEIL to SDCC, but it produces code with illegal syntax, so either do not use it, or alter it to produce code with the right syntax.

3.4.2 SDCC/KEIL Data Types

TinyOS and SDCC/KEIL do not support the same data types, so some alterations were needed to compile the code with SDCC and KEIL.

SDCC/KEIL supports the following data types:

- char (8 bits, 1 byte)
- short (16 bits, 2 bytes)
- int (16 bits, 2 bytes)
- long (32 bit, 4 bytes)
- float (32 bit, 4 bytes).

TinyOS supports an extra data type - 64 bit long long (u)int64_t. Since we are working with software that does not support this data type, on a very small hardware memory model, we decided to change the NesC 64 bit data types to 32 bit. This is done in the mangling script.

3.4.3 Reserved Keywords in SDCC

A number of keywords are reserved in SDCC. Half of them represent a directive to the compiler, defining which memory segment on the nRF24E1 the specific lines of code refer to. To ensure that the developer does not break code by unintentionally and unaware of their effect use them fx. as a variable name in the NesC code, they need to be replaced or altered to something else, e.g. data to _data, before compiling for SDCC. Right now the mangle script only handles the reserved keyword data. None of the other keywords except SFR, SBIT and interrupt are currently in use in the code. This might pose as a problem to future work, but the mangle script can easily be expanded to handle misuse of the other keywords.

However, if the code size increases significantly in the future, it might be nessecary to insert the keywords into the code, to support the architectures segmented memory model. Right now, everything is stored in the directly addressable memory segment, which is quite small.

The reserved keywords are:

- data / near
- xdata / far
- idata / pdata
- code
- bit
- SFR / SBIT
- interrupt

- critical

Variables declared with keyword storage class data/near will be allocated in the directly addressable portion of the internal RAM. This is the default option for the small memory model. Variables declared with storage class xdata/far will be placed in external RAM, which is default for the large memory model.

Variables declared with keyword storage class idata will be allocated in the indirectly addressable portion of the internal RAM. The first 128 byte of idata physically access the same RAM as the data memory. The original 8051 had 128 byte idata, but nowadays most devices have 256 byte idata memory. Paged xdata access (pdata) is just as straightforward.

The different memory segments that the keywords apply to, can be seen in the figure below.

nRF24E1 Internal Data Memory Structure Overview:

```
                          IRAM                 SFR
                 +---------------------+---------------------+
             FFh |                     |                     | FFh
                 |   Accessible by     |  Accessible by      |
Upper 128 bytes  |   indirect          |  direct             |
                 |   addressing only   |  addressing only    |
             80h |                     |                     | 80h
                 +---------------------+---------------------+
             7Fh |                     |
                 | Addressable by      |
Lower 128 bytes  | direct and          |
                 | indirect addressing |
             00h |                     |
                 +---------------------+
```

The prefered memory model can be defined in SDCC through CLI option --model-small or --model-large - the small memory model is default. In KEIL it can be changed through selecting it in the options pane for the target.

3.4.4 Removal of inlining

NesC assumes that GCC is being used for the final compilation. GCC supports inline functions and can be made to optimize code quite aggressively, so the code generated by NesC does not need to be very efficient. Unfortunately SDCC does not support code inlining, so the inline statements have to be removed, when compiling for SDCC.

Lines with the following format are affected:

static inline void TOSH_sleep(void); static __inline void TOSH_SET_RED_LED_PIN(void); __inline void__nesc_enable_interrupt(void);

Lines with the noinline attribute is substituted with the #pragma NO_INLINE.

3.4.5 Removal of Preprocessor Line Numbering

Also NesC produce preprocessor line number meta data, to allow the compiler to report error messages referring to the original code. We do not really need them for anything, so we filter them out to minimize the code size. It also eases the code reading significantly. If needed for debug purposes the regular expression in the mangle script which remove them can be commented out.

3.4.6 Change $ in Identifiers

The SDCC compiler is very strict when it comes to valid symbols in identifiers. NesC produce GCC-code which inserts $ as a separator character in identifiers. We mangle the $ to two underscores in order to enable SDCC/KEIL to compile.

3.4.7 Interrupt Vectors

The syntax for declaration of interrupt vectors are different in GCC and SDCC/KEIL. So we mangle the interrupt declaration:

From: void __attribute((interrupt)) __vector_5(void) To: void __vector_5(void) interrupt 5

Additionally KEIL does not understand that the interrupt vector is defined previous to its use. So we remove the forward declaration of the vectors in the mangle script, when compiling for KEIL.

4. TinyOS Modifications

TinyOS is based on modules with different levels of hardware abstraction. When porting TinyOS to a new platform, you change the underlying hardware dependencies in TinyOS, and you have to rebuild the modules bottom up. Hence, it has been necessary to modify a number of modules in TinyOS. The figure below shows the topological hierarchy of the TinyOS modules we have focused on. By far, most of the work has been done in the Hardware Presentation Layer (HPL), but certain changes also affected the higher abstractions, such as changes in interfaces and interrupt handling.

Modified TinyOS modules overview:

```
+-----------------------------------------------------------+
|                    TinyOS Application                     |  App
+-----------------------------------------------------------+
  \/    /\            \/    /\           \/   /\      \/  /\   -----
+----------+        +----------+      +----------+ +--------+
|  Timer   |        |  UART    |      |   ADC    | |  LEDs  |  HAL
+----------+        +----------+      +----------+ +--------+
  \/    /\            \/    /\           \/   /\               -----
+----------+  +---------------------+ +----------+
| HPLClock |  |      HPLUART        | | HPLADC   |     \/       HPL
+----------+  +---------------------+ +----------+
  \/    /\        \/         \/   /\     \/   /\               -----
+----------+  +--------+ > +--------+ +----------+ +--------+
|  Timer2  |  | Timer1 | > | Serial | | Sensors  | |  Port  |  HW
+----------+  +--------+   |  Port  | +----------+ +--------+
                           +--------+
```

The following sections describe the changes to the four groups of modules.

4.1 HPLClock and related modules

The 8051 chip has three independent timer/counter circuits: Timer0, Timer1 and Timer2, which can run in various modes. Each timer/counter consists of a 16-bit register that is accessible to software as three SFRs (TL0/TH0, TL1/TH1 and TL2/TH2). Timer0 and Timer1 can be used as 13 or 16 bit timer/counter or as 8 bit counters with auto-reload. Timer2 is only capable of running as a 16 bit timer/counter, but can remain as such even in auto-reload mode. Reload is desirable for loading the clock with a start value.

We have chosen to use Timer2 for the clock module, since it gives our design a maximum clock interval of 49.15 ms at a 16 MHz system clock. Using a different timer circuit would limit the interval to 0.192 ms, which would result in a great deal of interrupts and consume processing power for administrational overhead.

4.1.1 Timer

The Timer module (HAL) uses the HPLClock module to handle the hardware timing. These two modules communicate through the clock interface. However, the standard TinyOS clock interface is designed for an MCU with a more flexible prescaler, then the 8051 chip is equipped with. The 8051 is limited to a prescaler with a factor of 1/4 or 1/12 of the CPU clock frequency, whereas the TinyOS clock interface currently uses an 8 bit prescaler and an 8 bit timer. Because of the 8051s limited prescaler options, and the possibility to use a 16 bit timer/counter to compensate for the reduced prescaler options, we decided to widen the clock interface from 8 to 16 bit. We are using the factor 1/4 for the prescaler.

The interface change has affected the following methods: result_t setRate(uint16_t interval, char scale) void setInterval(uint16_t value) void setNextInterval(uint16_t value) uint16_t getInterval() result_t setIntervalAndScale(uint16_t interval, uint8_t scale) uint16_t readCounter() void setCounter(uint16_t n)

See: Clock.h Clock.nc HPLClock.nc TimerM.nc TimerC.nc 8051.h

4.2 HPLUART

The UART is depending on a timer to generate a baud rate for the serial port. The architecture only allows two of the three timers (Timer1 or Timer2), to act as such. Since Timer2 is already used by the clock module, this leaves only Timer1 available for the UART module.

When using Timer1 as the baud rate generator, 5 different baud rates can be obtained: 1.20 KiB/s, 2.4 KiB/s, 4.8 KiB/s, 9.6 KiB/s or 19.2 KiB/s. We chose to use a baud rate of 19.2 KiB/s with 8 data bits, no parity and one stop bit, since this speed is commonly used and the fastest speed possible using this timer.

We have also expanded the HPLUART interface to include a put2 method. This method is able to send more than one byte, by taking two pointers as arguments. These pointers refer to the first and last bytes to be sent. The HPLUART interrupt handler was also modified to take the multiple byte data into account.

See: 8051.h HPLUART.nc HPLUARTC.nc HPLUARTM.nc

4.3 HPLADC

The TinyOS standard ADC interface was developed for the AVR which includes hardware functionality for repetitive sampling at a given interval. Implementing this functionality on the 8051, which does not support this in hardware, would require use of the last timer. We chose not to implement repetitive sampling, therefore the setSampleRate method currently has no use.

See: 8051.h ADCM.nc HPLADCC.nc HPLADCM.nc

4.4 LEDS

TinyOS features three standard LEDs (Red, Green and Yellow), but the nRF24E1 evaluation board is not equipped with programmable LEDs so we used the general purpose ports (GPIO).

The standard 8051 platform features four 8 bit GPIO, however the nRF24E1 evaluation board is only equipped with two ports: Port0 and Port1, where Port0 has eight bits and Port1 has only three bits.

Intuitively the best solution would have been to place the standard three TinyOS LED bits on Port1, but unfortunately we were unable to control the most significant bit on Port1, since it is hard-wired as input and controlled by external SPI_CTRL. The Yellow LED was moved to Port0.

To visualize the status of the GPIO, including the three standard LEDs, we built a LED expansion board.

The three LEDs are currently wired to: Red -> P1.0 Green -> P1.1 Yellow -> P0.7.

See: 8051.h hardware.h mcs51hardware.h LedsC.nc

4.5 Interrupts

In TinyOS interrupts are not implemented as a single module, they are mainly facilitated in atomic blocks and in the init, start and stop methods of the various HPL modules. The init, start and stop methods only handle interrupts that are specific to the module, i.e. timer interrupt for the HPLClock module and serial interrupt for the HPLUART module. While the atomic block handle the enabling of global interrupts. This is used to avoid preempting code execution in critical blocks.

5. Conclusion

The project have reached a plateau of development in porting TinyOS to the 8051 platform, on which future development can be based. The basic modules (Timer, UART, ADC and LEDS) have been implemented making 8051 accessible for the TinyOS community. However a essential module for the field of sensor networks, the radio module, is still missing.

The result of our work will be uploaded to the TinyOS 8051 Working Group website.

6. Future Work

The work presented in this TEP is short of being a complete porting of TinyOS to the 8051 platform. Two obvious future tasks are implementing a Radio module involving the SPI interface and Power Management for duty cycling. The radio module is currently under development, in which the main hurdle is the three wire SPI interface.

This work is done for TinyOS 1.x, but looking forward, the 8051 port should target TinyOS 2.0. This might be a challenge with the timer interface being so different from TinyOS 1.x.

7. Authors

Anders Egeskov Petersen
University of Copenhagen, Dept. of Computer Science
Universitetsparken 1
DK-2100 København Ø
Denmark

Sidsel Jensen
University of Copenhagen, Dept. of Computer Science
Universitetsparken 1
DK-2100 København Ø
Denmark

email - purps@diku.dk

Martin Leoold
University of Copenhagen, Dept. of Computer Science
Universitetsparken 1
DK-2100 København Ø
Denmark

Phone +45 3532 1464

email - leopold@diku.dk

8. Citations

[NSC] Nordic Semiconductor. nRF24E1 Evalutaion board. http://www.nordicsemi.no/files/Product/development_tools/nRF24E1_EVBOARD_rev1_0.pdf

[PEH] Martin Leopold. “Power Estimation using the Hogthrob Prototype Platform” *M.Sc. Thesis, DIKU, Copenhagen University, Denmark, December 2004* .

[HOG] Kashif Virk, Jan Madsen, Andreas Vad Lorentzen, Martin Leopold, Philippe Bonnet. “Design of A Development Platform for HW/SW Codesign ofWireless Integrated Sensor Nodes” *Eighth Euromicro Symposium on Digital Systems Design*, 2005.