Need for a programable and configurable wireless communication platform to run distributed wireless sensor networks. Small scale, low datarate, low power, medium quantity (tens to hundred), able to run and program a Bluetooth conform Host Controller Interface (HCI) protocol and support small applications.

The Bluetooth standard [1] is a system available in samples today to enable wireless local networking in the 2.4 GHz ISM band. Tested modules and software stacks are available for PC platforms. Bluetooth is designed to partition the protocol stack onto different hardware, according to the target platform. The modules available from Ericsson [2] offer an interface to the HCI protocol layer. Higher layers have to be implemented on a host platform. There is no access to the program code of the module.

1 Target Features

- Programmable Bluetooth platform
- Direct access to a Bluetooth module via daughterboard
- Configurable logic?
- LED’s for control and debugging
- Temperature or other sensor
- 3 volt design
- Battery powered

2 System Architecture

A general architecture contains a bluetooth module, a microcontroller or microprocessor, memory, and interfaces. Figure 1 and 2 show examples of current System-On-a-Chip components.
2.1 Atmel ATmega103L

The ATmega103L is a low-power CMOS 8-bit microcontroller based on the AVR RISC architecture. By executing powerful instructions in a single clock cycle, it achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed. The following features are provided on chip: 128K bytes of in-system programmable Flash, 4K bytes EEPROM, 4K bytes SRAM, 32 general purpose I/O lines, 8 input lines, 8 output lines, 32 general purpose working registers, real time counter (RTC), 4 flexible timer/counters with compare modes and PWM, UART, programmable watchdog timer with internal oscillator, an SPI serial port and three software selectable power saving modes. [3]

2.2 Triscend E5 Configurable SOC

The Triscend E5 family is based on a high speed 8032 core. It features up to 64kbytes on-chip System RAM and up tp 3200 Configurable System Logic (CSL) cells. [4]

2.3 Ericsson Bluetooth ROK 101 007
Figure 2: Triscend E5 Configurable SOC

Figure 3: Ericsson Bluetooth Module ROK 101 007 Pinout
3 Bluetooth Communication

3.1 Bluetooth HCI Interface

The Bluetooth HCI supports different transport layers: USB, RS-232 and UART for on-board serial communication.

Although the HCI directly instructs the Baseband to do these inquiry/paging instructions, there exist devices, such as cordless phones, which are envisaged to function without the use of the HCI layer. It provides a typical standard command interface to the baseband controller and link manager, and access to hardware status and control registers. Knowing this it would be up to the developer, and application/profile setup, to decide whether to implement it. The Cordless Telephony Profile doesn’t use many of the lower features (such as hold, sniff) so it makes sense for it to not implement the HCI interface, and so reducing complexity & cost.

3.2 RS-232 and UART Transport Layer

The serial transport layer supports four kinds of HCI packets:

- HCI Command Packets
- HCI ACL Data Packets
- HCI SCO Data Packets
- HCI Event Packets

The Bluetooth HCI interface contains commands for:

- Link Control
- Link Policy
- Host Controller and Baseband
3.3 Link Quality Assessment

4 Axis OpenBT Stack

This section gives hints and examples on how to use the Axis OpenBT Stack. This implementation of the Bluetooth protocol stack can be obtained from http://developer.axis.com or http://sourceforge.net/projects/axisbt/ [5]. It is currently undecided how to partition the stack and applications into the sourcecode/filestructure.

Quite a view of the remarks in this section are derived from the mailing list of the Axis OpenBT Stack.

4.1 Structure of the Axis OpenBT Stack

Short description of new stack integration in kernel

1. PPP over serial (standard way):

   user app (pppd)
   \ /------------------
   |                  |
   \                  |
   tty | line discipline (PPP)
   \             |
   tty driver | (serial driver)
   \ /--------
   | UART

2. PPP over bluetooth:

   user app (pppd)
   \ /------------------
   |                  |
   \                  |
   ttyBT0 | ldisc-> PPP ldisc
   \             |
   tty driver | (bluetooth driver)
   |<----------
ttySx is a standard serial port tty.

ttyBTx has a separate major nbr and has been registered to use bt driver as tty driver.

ttySx is opened initially from a bluetooth control-application which sets the bt line discipline and keeps the tty open.

When ttyBT is opened from pppd it is set to use ppp discipline which in turn uses the driver registered in its tty i.e the bt driver.

At the stack top it acts like a serial driver for ppp

At the bottom the stack acts like any line discipline used from the standard serial tty.

3. Glue Layers for stack

"APPLICATION" (here: ttyBT)

TOP | bt_recieve_top() | bt_write_top() |
--- | --- | --- |
    | V | V |

THE STACK

BOTTOM | bt_receive_lower_stack() | bt_write_lower_driver() |
--- | --- | --- |
    | V | V |

"PHYSICAL DRIVER" (here: serial driver)

Directory Structure of Release bluetooth_20010108.tgz

apps/
  btd/ Demo Application in Kernel Mode
  sdp_server/ Bluetooth SDP Database in an XML File
  userstack/ Demo Application in User Mode (Different Makefile)

hroot/
include/ Header Files
libs/expat/ Bluetooth Stack
src/
  bluetooth.c Linux kernel integration code for bluetooth stack
  btdebug.c Miscellaneous debug functions
btmem.c  Memory management in Bluetooth stack
bt_proc.c Implementation of /proc files in bluetooth driver
hci.c Implementation of Bluetooth Host Controller Interface (UART transport layer)
hci_vendor.c All vendor specific HCI commands and events are located in this file. The hci_XX commands commands are called during startup and process_vendor_XX when vendor specific HCI events are received.
l2cap.c Implementation of Bluetooth Logical Link Control and Adaption Protocol (L2CAP)
l2cap_con.c Implementation of Bluetooth Logical Link Control and Adaption Protocol (L2CAP), Connection Manager
l2cap_sec.c Implementation of Bluetooth Logical Link Control and Adaption Protocol (L2CAP), security part
rfcomm.c Implementation of Bluetooth RFCOMM with TS 07.10, Serial Port Emulation
rfcomm_sec.c Implementation of Bluetooth RFCOMM with TS 07.10, Serial Port Emulation, security part
sdp.c Implementation of Bluetooth Service Discovery Protocol
sec_client.c Implementation of interface to Bluetooth Security
tcs.c Implementation of Bluetooth Telephony Control Protocol Specification, TCS Binary
test.c Test layer used to verify stack
unplug_test.c Test cases for the UnPlugFest 3

bt.o created in src/ is the Bluetooth protocol stack kernel module. It enables other user space programs to connect through it to talk to other devices.

Think of bt.o like ppp.o. ppp.o provides the PPP logic to enable you to do TCP/IP over a physical type connection (modem, serial, and bluetooth...)

---

**How to Generate a New App/Directory**

- Copy apps/userstack/Makefile and unplug_test.* to a new directory
- Check Makefile for included files, make links
- Replace links for files if necessary

**4.2 Getting Started on Applications**

A minimal app to setup bluetooth so that another app can open /dev/ttyBT* and use it like a char driver. It doesn’t setup scan inquiry or initiate a connection, but it’s a head start.

```c
/*
 * Note: 1) this is for a kernel mode stack -- you need to insmod bt.o before executing this
 * 2) if your h/w assumes a different initial baud rate then change it below
 */
#include <sys/time.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <fcntl.h>
```
#include <time.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <termios.h>
#include <stdlib.h>
#include <string.h>
#include <syslog.h>
#include <errno.h>
#include <stdarg.h>
#include <signal.h>
#include <getopt.h>
#include <readline/readline.h>
#include <readline/history.h>
#include "bluetooth/include/linux/bluetooth/btcommon.h"

int main(int argc, char **argv)
{
    int bt_ldisc = N_BT, ser_fd, bt_fd;
    struct termios t;

    /*
    * Setup the serial port to the initial baud rate of the bluetooth
    * device (vendor-specific).
    */
    if ((ser_fd = open("/dev/ttyS0", O_RDWR)) < 0) {
        perror("open /dev/ttyS0");
        exit(1);
    }

    if (tcgetattr(ser_fd, &t) < 0) {
        perror("tcgetattr");
        exit(1);
    }

    cfmakeraw(&t);
    t.c_cflag &= ~CBAUD;
    t.c_cflag |= B57600 | CS8 | CLOCAL; /* initial baud rate */
    t.c_oflag = 0;
    t.c_lflag = 0;
    t.c_cflag &= ~CRTSCTS; /* flow control off -- depends on vendor! */

    if (tcsetattr(ser_fd, TCSANOW, &t) < 0) {
        perror("tcsetattr");
        exit(1);
    }

    if (ioctl(ser_fd, TIOCSETD, &bt_ldisc) < 0) {
        perror("TIOCSETD");
        exit(1);
    }

    /*
    * Setup the bluetooth stack.
    */
if ((bt_fd = open("/dev/ttyBT0", O_RDWR)) < 0) {
    perror("open /dev/ttyBT0");
    exit(1);
}

if (ioctl(bt_fd, BTINITSTACK) < 0) {
    perror("BTINITSTACK");
}

/*
 * Initialize the h/w (assumed digidongle)
 */
//ioctl(bt_fd, BTSETDIGIBAUDRATE, 115200); /* final baud rate */

/*
 * Set the serial port baud rate to match the new h/w rate
 */
if (tcgetattr(ser_fd, &t) < 0) {
    perror("tcgetattr");
    exit(1);
}

cfmakeraw(&t);
t.c_cflag &= ~CBAUD;
t.c_cflag |= B115200 | CS8 | CLOCAL; /* initial baud rate */
t.c_cflag = 0;
t.c_lflag = 0;
t.c_cflag &= ~CRTSCTS; /* flow control off */

if (tcsetattr(ser_fd, TCSANOW, &t) < 0) {
    perror("tcsetattr");
    exit(1);
}

/*
 * Now, the app can do whatever it wants over the bluetooth
 * tty... you can fork/exec it here or just hang around here and
 * let the app use a different /dev/ttBT*.
 */

4.3 Messages

BCSP link establishment message:

```
0xc0 0x40 0x41 0x00 0x7e 0xda 0xdc 0xed 0xed 0xa9 0x7a 0xc0
```

The big clue is the 0xc0 at either end - BCSP uses SLIP at its lowest level, and 0xc0 is the packet framing character.

Also, the ASCII text in the middle, "dad ceded," is the first of BCSP's link establishment messages. The Casira module is trying to make first contact with its host.
4.4 PPP Operation

The proper way of using the ppp? Here's my guess:

Server

btd -u /dev/ttyUB0 -s 57600 -e 0 -m -R
ppp (gives waiting for line 0 or something)

Client

btd -u /dev/ttyUB0 -s 57600 -e 0 -m -r client -R
rf_conn <bt_address> 1 0

4.5 USB Operation

Make sure the USB device is connected, the USB bluetooth driver is loaded, the USB
device is recognized by the USB Bluetooth driver (kernel messages will say this.) and
then point the bluetooth stack at the USB device node (that was created as per the
/usr/src/linux/Documentation/usb/bluetooth.txt documentation file from kernel 2.2.18
and up.)

4.6 Operation with HCI Status Parameters

The following patch allows to read the HCI Status Parameters. Upon opening a connection on the l2cap
one can inquire these parameters using the connection handle provided by the code below.

```c
--- bluetooth/src/hci.c Thu Nov 16 01:40:04 2000
+++ bluetooth-new/src/hci.c Wed Nov 22 15:52:42 2000
@@ -1403,9 +1403,53 @@
    default:
       D_REC(FNC"Not recognised !\n");
       break;
-   }
case RESET_FAILED_CONTACT_COUNTER:
    if (r_val[0]) {
        D_ERR("RESET_FAILED_CONTACT_COUNTER: %s\n",
            get_err_msg(r_val[0]));
        break;
    }
    printk("Reset failed contact counter for handle %d\n",
        CHAR2INT12(r_val[2], r_val[1]));
    break;

case GET_LINK_QUALITY:
    if (r_val[0]) {
        D_ERR("GET_LINK_QUALITY: %s\n",
            get_err_msg(r_val[0]));
        break;
    }
    printk("Link quality for handle %d is %d\n",
        CHAR2INT12(r_val[2], r_val[1]), r_val[3]);
    break;

case READ_RSSI:
    if (r_val[0]) {
        D_ERR("READ_RSSI: %s\n",
            get_err_msg(r_val[0]));
        break;
    }
    printk("RSSI for handle %d is %d\n",
        CHAR2INT12(r_val[2], r_val[1]), (char)r_val[3]);
    break;

default:
    D_REC(FNC"Not recognised !\n");
    break;

case HCI_TC: /* Test Commands */
    switch (ocf) {
        s32 c = 0;
        s32 zero = 0;
        printk(FNC" for bd address %02x:%02x:%02x:%02x:%02x:%02x:
            bd[5], bd[4], bd[3], bd[2], bd[1], bd[0]);
    }

/* Status Parameters commands */
+s32 hci_read_rssi(u32 hdl) {  
  c_pkt.type = CMD_PKT;
  c_pkt.ocf = READ_RSSI;
  c_pkt.ogf = HCI_SP;
  c_pkt.data[0] = hdl & 0xff;
  c_pkt.data[1] = (hdl >> 8) & 0xff;
  c_pkt.len = 2;
+ return send_cmd((u8*) &c_pkt ,c_pkt.len + CMD_HDR_LEN + HCI_HDR_LEN); +
+
+s32 hci_get_link_quality(u32 hdl) {
+ c_pkt.type = CMD_PKT;
+ c_pkt.ocf = GET_LINK_QUALITY;
+ c_pkt.ogf = HCI_SP;
+ c_pkt.data[0] = hdl & 0xff;
+ c_pkt.data[1] = (hdl >> 8) & 0xff;
+ c_pkt.len = 2;
+
+ return send_cmd((u8*) &c_pkt ,c_pkt.len + CMD_HDR_LEN + HCI_HDR_LEN);
+
+s32 hci_read_failed_contact_counter(u32 hdl) {
+ c_pkt.type = CMD_PKT;
+ c_pkt.ocf = READ_FAILED_CONTACT_COUNTER;
+ c_pkt.ogf = HCI_SP;
+ c_pkt.data[0] = hdl & 0xff;
+ c_pkt.data[1] = (hdl >> 8) & 0xff;
+ c_pkt.len = 2;
+
+ return send_cmd((u8*) &c_pkt ,c_pkt.len + CMD_HDR_LEN + HCI_HDR_LEN);
+
+s32 hci_reset_failed_contact_counter(u32 hdl) {
+ c_pkt.type = CMD_PKT;
+ c_pkt.ocf = RESET_FAILED_CONTACT_COUNTER;
+ c_pkt.ogf = HCI_SP;
+ c_pkt.data[0] = hdl & 0xff;
+ c_pkt.data[1] = (hdl >> 8) & 0xff;
+ c_pkt.len = 2;
+
+ return send_cmd((u8*) &c_pkt ,c_pkt.len + CMD_HDR_LEN + HCI_HDR_LEN);
+
/* Defines of Testing Commands functions */
+s32
--- bluetooth/include/linux/bluetooth/hci.h Thu Nov 16 01:40:04 2000
+++ bluetooth-new/include/linux/bluetooth/hci.h Wed Nov 22 15:18:46 2000
@@ -72,5 +72,10 @@
s32 hci_ericsson_tx_test(u8 *ctrl_seq);
s32 synth_on(void);
+s32 hci_read_rssi(u32 hdl);
s32 hci_get_link_quality(u32 hdl);
s32 hci_read_failed_contact_counter(u32 hdl);
s32 hci_reset_failed_contact_counter(u32 hdl);
+
#endif
/******************** END OF FILE hci.h *********************************************/
4.7 Demo and Test Application

Both `btdhcisp` and `btdhcisp_debug` allow a test connection setup. They are compiled from the Axis OpenBT Stack release 2001/01/08 including HCI Status parameters.

```bash
text: ./btdhcisp [options]

options:

- `u`, `--physdev <uart device>`
  sets which uart device that will be used by the stack
  default: ttyS2

- `b`, `--btdev <bluetooth device>`
  sets which bluetooth device that will be used from application
  default: ttyBT0

- `d`, `--local <local ip>`
  Sets local ipadress in pppd options

- `D`, `--remote <remote ip>`
  Sets remote ipaddress in pppd options

- `e`, `--modem`
  Use modem emulation (used when emulate modem in windows dialup.
  Can also be done from command line mode.
  default: on

- `m`, `--cmdmode`
  enters command line mode
  default: skip command line mode

- `n`, `--local-name`
  prefix used for the local bluetooth device name
  default: nothing

- `<r server>`, `--server`
  sets application role to server
  `<r client>`, `--client`
  sets application role to client
  default: server

- `R`, `--reset`
  reset bluetooth hardware before use
  default: no reset

- `s`, `--speed <speed>`
  sets uart speed
  9600, 19200, 38400, 57600, 115200, 230400, 460800
  default: 115200 baud

- `S`, `--unixsock` use local unix socket as phys device
  Used together with hci emulation to test stack locally (usermode stack only)
-T, -tcpsock <ipaddr:port> use tcp socket as phys device. Used together with hci emulation to test stack over any TCP/IP based network. Server listens on <:port>, client tries to connect to <ipaddr:port>. (usermode stack only)

5 BTNode Status

- we have 4 fully running systems now. there is a system software providing operating-system like features.
- btnodes can communicate wirelessly using bluetooth (yes, we did it!). every btnode can inquire other bluetooth devices within communication range. btnodes can establish a communication channel to any other bluetooth device found, e.g. other btnodes, laptops equipped with a bt pcmcia card. devices can than communicate reliably (no corrupted packets, no packet duplications, packet order is preserved). both communication partners are notified when a channel is lost. unfortunately i still couldn’t get a commitment of any bluetooth supplier to ship modules in the quantity we need.
- sensors can be attached to the analog or digital inputs. there are no spi or i2c interfaces. we connected a temperature and light sensor (basically resistors on a daughterboard) to the btnode as a proof of concept.
- for programming, debugging and connecting to the outside world there is a serial port. there are also 4 LEDs, one of which is used as a heartbeat signal.
- finally, there is a system clock and a 16 bit timer on external or intenal clock.

5.1 System Software

The system software consists of low level drivers (2 serial ports, ADC, random number generator, system clock), a scheduler, and the bluetooth stack (HCI and L2CAP layers, hopefully SDP soon). applications need to be linked against the system software code and then uploaded to the device.
5.1.1 Interfaces

- the scheduler allows application to register callback functions to be called upon certain events. events currently supported are: data received on uart, uart ready to write data, timeout expired, and user events. the granularity of the timeout is roughly 250ms. if higher precision is required, the additional timer needs to be programmed on an interrupt level. the user events are triggered by application code. they are useful to trigger a function being called from within an hardware interrupt. the scheduler calls the function when it is save to do so. adding additional event types to the scheduler is simple.

- the interface provided to the serial ports is equivalent to the unix read() and write() functions.

- both bluetooth stack and scheduler are available for the atmel as well as linux. therefore applications typically run on both systems without modification.

5.1.2 Memory Consumption

- the size of the system software plus a very tiny demo application now is

<table>
<thead>
<tr>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>dec</th>
<th>hex</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>36058</td>
<td>776</td>
<td>1019</td>
<td>37853</td>
<td>93dd</td>
<td>btsched.elf</td>
</tr>
</tbody>
</table>

that is 1795 bytes of ram. the vast majority is taken by the bluetooth stack. there is still some potential to decrease ram consumption further. however, unless we redesign the bluetooth stack entirely, the stack will need at least 1k.

- yet i have no figures on the runtime stack requirements (for parameter passing and auto variables) of the system software. but i assume about 1k.

- with the current figures apps have roughly 1.2 Kbytes to themselves.

5.2 BTNode Hardware

- microcontroller atmel ATmega 103L: 4MHz, 4MIPS, 4k ram, 128 flash, 4k eeprom, 64 pin package, 3.3V: 5.5mA active, 1.6mA idle, 1µA powered down. features not useed in the btnode: external sram option needing no external components (but the memory ;) and spi (but no i2c).

- 8 x 10bit AD converter, can also be used as digital ins

- 16 x digital i/o

- 3 x interrupts, edge or level triggered

- 4 x LEDs

- 2 x UART, one hardware @ 57.6Kbps, one software @9.6 Kbps

- all components but the bluetooth module are standard and easy to acquire

5.3 Programming

- programming of the atmel is straight forward: i pretty much use my usual programming style, except for dynamic allocations.
- the bnode as well as the atmel platform (demo board from atmel) is very reliable: we had no mysterious reboots, non-deterministic behavior, etc.

- the gnu compiler and tools are reliable and complete (though built from a cvs snapshot). there is a compiler (gcc), debugger (gdb), make, ar, nm, objdump, objcopy, strings and size. the last 5 are for getting excessive information on binaries (like size of text, data, bss segments, which variable goes where, is initialized when, etc.), for disassembly and so froth. these tools offer a wealth of information i’ve never seen in a point-and-click ide. and they are scriptable ;)

- there is a libc: functions like memcpy, strcpy, strlen, index, strcmp, etc. are already implemented. there is even malloc and free, though we haven’t used that.

5.4 Planned Work

- the bluetooth stack still needs some work, particularly, i’d like to extract a well defined api for application programming and further reduce static ram and runtime stack requirements.

- there is an Ericsson Mobile Phone R520 (bluetooth built in) floating around somewhere in the institute. it should be possible to use such a phone as gateway to the internet for the bnodes. i’d like to try that (or rather, have a student doing it).

References