The Zaurus Software Development Guide

Robert Christy

August 29, 2003

Contents

1 Overview

The Bat computing platform is comprised of two layers: the upper layer, a *Sharp*® *Zaurus* and the lower layer, an *ATMEL*® *ATmega128*. There are also two controller schemes being used by the *MVWT* group for the Bat. The first of these controllers is specific to *RoboFlag*. It should provide the control and interface necessary to run the Bat as a vehicle in a *RoboFlag* game. This controller is fairly small and effectively controls for velocity using the vision system and the onboard gyro[1] However, the *RoboFlag* controller is not sufficient for most control research applications and, moreover, it is unlikely that the *Zaurus* itself is sufficiently powerful for these applications. For this reason, there is planned[2] a force controller that will be used in conjunction with an

[1]Gyro support for the *RoboFlag* controller is not yet supported because of the ongoing troubles with the *Zaurus*'s ability to receive serial data. See appendix A for more on the state of sensor data support.

[2]This controller has not yet been implemented *at all*

offboard control platform. Under this scheme, sophisticated control laws (with trajectory generation, etc.) may be implemented on a computer separate from the Bat and the output of these controllers (fan forces, specifically) will be transmitted over the network to the vehicle. Once there, the force controller will use these values as reference to control the output forces of the individual fans by closing a loop around the onboard accelerometer data.

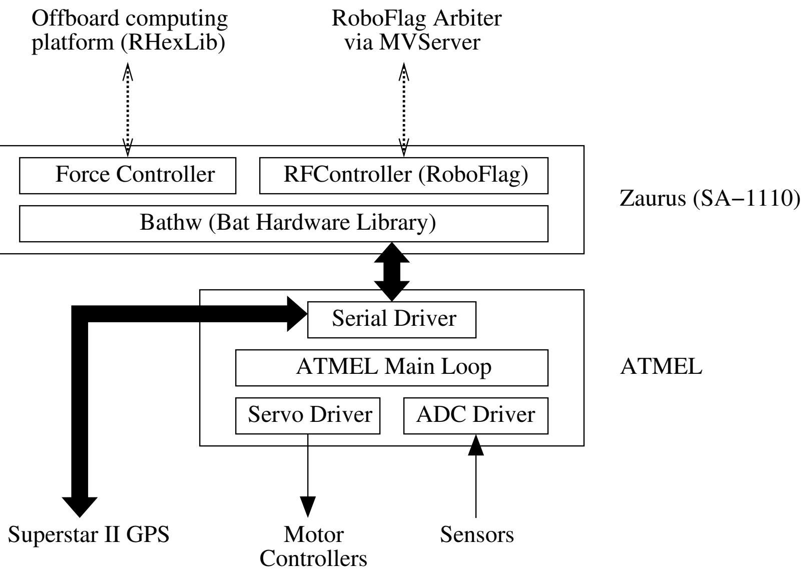

Figure 1: This diagram lays out the general structure of the Bat software. The items at the top are offboard, that is, not located on the Bat. The items at the bottom are components located on the PCB. The layers in between represent the *Zaurus* and the *ATMEL* and are labeled as such. The heavy solid, lines represent RS232 serial connections, the dotted lines represent 802.11b wireless ethernet connections, and the thin, solid lines represent PCB trace connections.

The relationship among the control schemes and computation layers is depicted in figure 1. The remainder of this document should serve to explain the operation of and, where relevant, interfaces to the various components in the "*Zaurus*" block of this figure. If developing additional sensor support or looking for information about the "*ATMEL*" block in the figure, be sure to also consult Hans Scholze's documentation for the *ATMEL* microcontroller code.

2 Writing Software for the *Zaurus*

The *Zaurus* runs the *Lineo Embedix Linux*^{TM} operating system. Therefore, developing software for the *Zaurus* is very similar to developing software for any other Linux-based platform. Since the *Zaurus* uses an *Intel*® *StrongARM*

SA-1110 processor (rather than an *Intel*® x86 processor), a cross compiler is necessary to compile programs for the *Zaurus* on a different (x86 architecture) computer. The cross compiler recommended for writing *Zaurus* programs is `arm-linux-gcc`. It can be obtained from the *Zaurus* website: `http://www.zaurus.com-/dev/tools/downloads/tools/` as an RPM. In addition to the cross compiler itself, three other packages are necessary. Namely the binary utilities, glibc standard C library, and the standard C header files are required to compile programs for the *Zaurus*. For more instructions on how to install the cross compiler, see the *RoboFlag for MVWT Users' Guide* section on setting up the Bat controller.

3 Using the *bathw* Library

Since the *Zaurus* interfaces to the hardware through a serial connection, basic Linux device I/O will suffice to control the fans and read sensor data. However, for the sake of good programming practice, an abstraction layer exists to obviate the use of Linux I/O calls in programs that interact with the Bat hardware. This abstraction layer is manifested in the "Bat hardware library," *bathw*.

The *bathw* library is designed to provide the programmer with a comprehensive interface to the Bat hardware. In addition to basic intialization, cleanup and error handling functions, the *bathw* library contains an interface to all three fan outputs as well as an extensible interface to sensor data. While this document is not intended as a comprehensive reference (the header file is very thoroughly commented, and should be used as for that purpose), an outline of these interfaces will be provided below.

3.1 Using the fans

The programmer is provided access to the fans is provided through a pair of functions: *bathw_setfanoutputs* and *bathw_getfanoutputs*. Each of these functions take three arguments corresponding to the left, right and lift fans respectively.

The *bathw_setfanoutputs* function takes three integers, ranging between 0 and the defined constant *FAN_MAX*. This call will power each of the fans to the level specified by the given argument. If the programmer wishes to change only some subset of the fan values, he or she may opt to pass the constant *FAN_NO_CHANGE* as the argument for the fans whose levels should remain unchanged.

The *bathw_getfanoutputs* function takes pointers to three integers and stores the last values output to the fans at the location referenced by the pointers. Note that this function does *not* actually read values from the hardware. Rather it simply recalls cached values of previous *bathw_setfanoutputs* commands. As a result, if no previous calls to *bathw_setfanoutputs* have been made, then the values *FAN_UNDEF* will be returned in place of actual fan outputs. For this reason, it is good practice to always set zero fan outputs immediately after initialization.

That way, *bathw_getfanoutputs* will always return meaningful results. No values are stored for NULL arguments.

3.2 Getting sensor data

The *bathw* library contains a fairly robust interface to the sensors onboard the Bat. While at this time, the sensor interface is mostly implemented[3], it is largely untested. This is because of an ongoing dilemma with the serial communication onboard the *Zaurus*. For more information on this problem, please consult appendix A. This section, therefore, lays out the existing infrastructure and the intended functionality of the completed interface.

3.2.1 Enabling sensors

Before a sensor may be used, it must be enabled by calling the appropriate *bathw_enable* function. The existing sensors are enabled by the functions:

bathw_enable_gyro Enables the gyro

bathw_enable_accel Enables the accelerometers

bathw_enable_heading Enables the heading sensors (magnometers)

Each enable function typically takes a filename and one or more pointers to "buffers." as arguments. The filename refers to a file containing calibration data for that particular sensor. Each enable function must know how to read and interpret the data found in that file. The other arguments are pointers to variables accessable locally by the caller. When the sensors are updated (this will be covered in the next section), the updated sensor data will be stored in these variables.

3.2.2 Updating sensors

After the programmer has enabled all of the sensors he or she intends to use, the sensors must be periodically updated. This is done by calling *bathw_update_sensors*. After a call to *bathw_update_sensors*, the most recent sensor values will be stored in the buffers of each of the enabled sensors. See figure 3.2.2 for an example.

There is something very important to note about the *bathw_update_sensors* function. It performs a blocking read. This means that once it is called, the function will not return until sensor data is available. This behavior is undesirable for many applications, specifically those that perform several tasks asynchonously. Dynamic controllers and programs with networking both fall under this category. Three ways of working around this problem are presented here. The last and most elegant of these solutions is not currently supported, however, it is nevertheless described in the hopes that support may be added in the future.

[3]Several areas of code, such as the loading of sensor calibration data are missing, at the time of the writing of this document. See section 4 for more info.

```
/* Example code snippet to demonstrate enabling and updating sensors */
#include <stdio.h>
#include "bathw.h"

int main()
{
    float gyro_value;

    /* Initialization code */
    if (bathw_init(DEFAULT_COMM_DEVICE, DEFAULT_COMM_SPEED) == -1)
    {
        fprintf(stderr, "Could not initialize Bat hardware: %s\n",
                bathw_stderr());
        return 1;
    }

    if (bathw_enable_gyro("gyro_calib.data", &gyro_value) == -1)
    {
        fprintf(stderr, "Could not enable gyro sensor: %s\n",
                bathw_stderr());
        return 1;
    }

    /* Main loop goes here ... */

        bathw_update_sensors();

        /* At this point gyro_value will contain the most recent gyro sensor
         * value */

    /* ... End of main loop */

    /* Cleanup code */
    bathw_disable_gyro();
    bathw_cleanup();

    return 0;
}
```

Figure 2: This program sample program demonstrates how the programmer can enable sensors and update them.

1. Use the *bathw_get_serial_fd* to obtain the file descriptor for the serial port and use the libc function *fcntl* to set the *O_NONBLOCK* flag. This will cause all serial port reads to be non-blocking. As a result, when *bathw_update_sensors* is called when no serial data is ready, -1 will be returned with a *bathw*error code of *BATERDERR*. This option is undesirable for several reasons:
   - It violates the stated goal of abstracting the programmer from Linux I/O
   - The programmer cannot distinguish the unavailablity of new sensor data from a genuine I/O error
   - Non-blocking I/O often requires polling on the part of the programmer, which is generally bad practice.
2. Use the *bathw_get_serial_fd* to obtain the file descriptor for the serial port and use the libc function *select* to determine when data is available on the serial port. In this manner, the programmer can ensure that *bathw_update_sensors* is only called when the read will not block. While this method still violates the abstraction layer (as the previous method did), this method has the added advantage that a *select* can be performed on many file descriptors simultaneously, thus providing a simple means for performing I/O with many devices simultaneously. This is very useful for networking applications. In general, this method is the best of the supported methods.
3. Use POSIX threads to set up a separate "sensor thread" whose sole responsibility is to retrieve sensor data while the main thread is remains available to perform other tasks. While this is clearly the most elegant solution (it does not violate the abstraction layer), it is not currently supported since the standard C I/O routines are not thread-safe. This means that having a sensor thread read sensor data while the main thread sends fan commands could have unpredictable results. Also POSIX threads carry extra computational overhead which should be considered a minor detractor of this method.

3.2.3 Disabling sensors

Sensors may be disabled as well. This is useful if you want to top using a particular sensor part of the way through the execution of a program. Sensors should also be disabled at the end of the program. At the moment, there is no true need to do this, in the future, neglecting to do this may affect the performance of subsequently running programs. Therefore, it is good programming practice to do so.

4 Completing and Extending *bathw*

As has already been stated, the *bathw* library is not entirely complete. Moreover, in the future, sensors will be added to the vehicle, which will require the addition of support for those sensors. For both of these reasons, modification of the *bathw* code will be necessary. This section is to make the developer's job as painless as possible in doing either of these tasks.

4.1 Finishing existing sensor support

The only thing missing from *bathw* currently is the code to load calibration data for each of the existing sensors and the code to use that calibration data to convert the raw *ATMEL* output values to meaningful, phyical values. The code changes must be made directly to the `bathw.c` source file and the locations where changes must be made are marked with the comment "FIXME." Once the serial dilemma[4] is resolved and the format of the calibration data determined, these pieces of code should be fairly straight forward to implement. At this time, I do not forsee the need to add any more than 20 lines of code to the existing *bathw* codebase.

4.2 Adding additional sensor support

The *bathw* sensor interface was designed with the intent that it be relatively simple to add support for new sensors. There are five steps to adding support for a new sensor device to the *bathw* code[5].

1. In `bathw.h`, add a constant of the form *SENSOR_*sensorname (where *sensorname* is an abbreviated name for your sensor) to the list of existing sensor constants found in the anonymous enumeration (`enum` in C). The constant for the new sensor should be inserted immediately above the line containg the constant *SENSOR_COUNT*. Note that since the constants are actually defined through an enumueration there is no need to specify an integer, the next integer in the sequence will automatically be assigned to your sensor.

2. Near the top of the file `bathw.c`, add a line to the array *sensors* containing information about your sensor. *sensors* is an array of type `struct sensor_info`. This structure has the following elements (note the order does matter):

   `char` ***mnemonic*** This field should be set to the character or byte that the *ATMEL* sends just before it sends data from the sensor. That is,

[4] See appendix A

[5] This does not include any steps necessary to add support for the sensor to the *ATMEL* code. There will, invariably, need to be changes to that code in order to get new sensor devices to work on the Bat. Please consult Hans Scholze's documentation for instructions on how to do that.

this byte is the header byte that is used to inform the *bathw* as to which type of sensor data is waiting on the serial port.

`char` ***enabled*** This character is a boolean value that represents whether the sensor is currently enabled. In the structure definition, this value should always be 0, since all sensors are, at startup, disabled.

int (*handler`)(uint8_t *)` This is a pointer to the handler function that *bathw_update_sensors* will call when data of the appropriate sensor type arrives. The argument to this function is the raw sensor data as read off the serial port. This function will be written at step 4.

`uint8_t` ***datasize*** This value is the length of sensor data in bytes. More specifically, it is the number of bytes that *bathw_update_sensors* should read off of the serial port and pass to *handler* when a byte equal to *mnemonic* is read.

The element should be added at the end of the array (ie. it should be the *SENSOR_*sensorname'th element in *sensors*).

3. Write an enable function. The convention used thus far for naming this function is *bathw_enable_*sensorname where *sensorname* is the abbreviated name of the sensor being added. This function must do several things:

    - Load any calibration data necessary for the sensor and or prepare any precalcuated values necessary for the sensor.
    - Send any commands to the *ATMEL*that are necessary to start the *ATMEL*sending the sensor data. At the moment, no such commands exist.
    - Set the variable *sensors*`[`*SENSOR_*sensorname`].`*enabled* equal to a non-zero value (1 will do just fine).
    - Record "buffer" pointer(s) so that the caller may access the processed sensor data.

4. Write the update function. This is the function to which *handler* in step 2 must be set. This function takes one argument, `uint8_t *`*buffer*, which is a pointer to the raw sensor data. The handler must then take this data, convert it into a meaningful value, and store that in locations referenced by the "buffer" pointer(s) recorded by the enable function.

5. Write a disable function. This function must simply reset the *enabled* flag that was set by the enable function and send any commands necessary to shut the sensor down on the *ATMEL*. At the moment, no such command exist.

A The serial problem

At the time of the writing of this document, there is a very curious problem with serial communication on the *Zaurus*. On a certain subset of the Bats, the

Zaurus can send data over the serial port but it cannot receive. That is, fan commands are sent without trouble yet peculiar behavior obscures the sensor data. The strange behavior is this: for most of the time, the serial port appears to be receiving no data. Every once in a while a read event occurs on the serial port (as indicated by the *select* libc call) but when a *read* call is attempted it blocks indefinitely. It should be pointed out that this *read* call only attempts to read a single byte, and therefore, if data is in fact available, the *read* should never block.

The behavior seems to follow the *Zaurus*'s rather than the *ATMEL*. The broken *Zaurus*'s have also been tested against a computer and have exhibited the same faults. Moreover, the *ATMEL*'s paired with broken *Zaurus*'s have been connected to working *Zaurus*'s and shown to work. This seems to indicate that the problem is localized to the *Zaurus*.

The exact same program is used on each vehicle, therefore it does not seem likely to be a high-level software issue. The status of a *Zaurus*, however, has been known to change after reinstalling all of the software (see the *MVWT II* documentation for the *Zaurus* installation procedure). Namely, a working *Zaurus* has become broken after a reinstallation. Therefore it seems unlikely that the *Zaurus* hardware is at fault. Additionally, the ailment seems to affect at least three *Zaurus*'s, which makes a hardware problem seem further unlikely.

While we have made remarkably little headway in tracking down the origins of this problem, there are still a few experiments that may be worthwhile. For example, it may be educational to try to get the *Zaurus* communicating with a computer at a baud rate slower than 115.2kbps. This should be attempted first with a standard piece of software where confidence that the serial I/O code is correct is high. And then should be tried with custom code, such as `democtlr`, where the confidence is not as high.