AMBuSh, the Atlas pixel module burn-in shystem

Johannes Mülmenstädt

September 24, 2003
Document revision: `$Id: doc.tex,v 1.11 2003/09/24 22:01:01 jmuelmen Exp $`

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Atlas pixel module burn-in system (AMBuSh) tests the longevity of Atlas pixel modules by subjecting them to a sequence of tests. The system consists of hardware that supplies power to an array of modules and controls them; and of software that allows the system operator to specify and modify the sequence of tests. Two considerations were principal in the design of the system. The first is to allow the concurrent testing of a large number of modules; the second is to provide a flexible mechanism for altering the test sequence without interrupting a run.

1.1 How to get this document

This document is the official documentation for the AMBuSh system. Since AMBuSh may change between releases, you should make sure that the release of the documentation matches the release of your AMBuSh software or hardware. The software release is printed when you invoke the AMBuSh shell. The hardware release is printed on the BurP board. The release tag of this copy of the documentation is `$Name:  $`.

If your documentation is outdated, you can find current documentation in the `doc/` directory of the AMBuSh distribution. You can obtain the AMBuSh distribution by CVS from blah blah blah or http from blah blah blah.

Chapter 2

Hardware

The only new piece of hardware required for AMBuSh is the BurP (Burn-in Power) board. The BurP board fans out control from a TPCC to several modules (four in the current design), multiplexes data returned by the modules and provides power and high voltage to the modules. It also provides slow controls — setting analog and digital supply voltages for each module and reading back currents and temperatures — through a USB interface. Refer to Fig. 2.2 for the physical layout of the board.

The following sections deal with using the hardware to burn in modules and hacking up the hardware if you want it to behave differently.

2.1 Using the hardware

Using the hardware is straightforward. Fig. 2.1 shows how the parts go together. The BurP board wants to be connected to a TPCC through one of the four module connectors on the TPCC. It also requires a high-voltage power supply for each module; one common bulk power supply to feed the regulators; and one 3.3 V power supply that drives the control circuitry on the board. Finally, the USB port of the board needs to be connected to a computer running the AMBuSh software. Fig. 2.2 shows the location of the connectors on the BurP board.

2.2 Hacking the hardware

The BurP board consists of four components. A schematic for each is included.

2.2.1 Regulators

Linear regulators provide VDDD and VDDA for each module connected to the BurP board. The regulators are current-limited to 1.2 A. Their output voltage ramps upon being enabled. The voltage level can be selected by the user: 80%, 90%, 100% and 110% of the nominal voltage. Fig. 2.3 shows the schematic of the regulator.

2.2.2 Clock and control fan-out

The TPCC sends control and 40 MHz clock signals over LVDS lines. These signals are converted to TTL on the BurP board and fanned out to a separate set of LVDS drivers for every module.

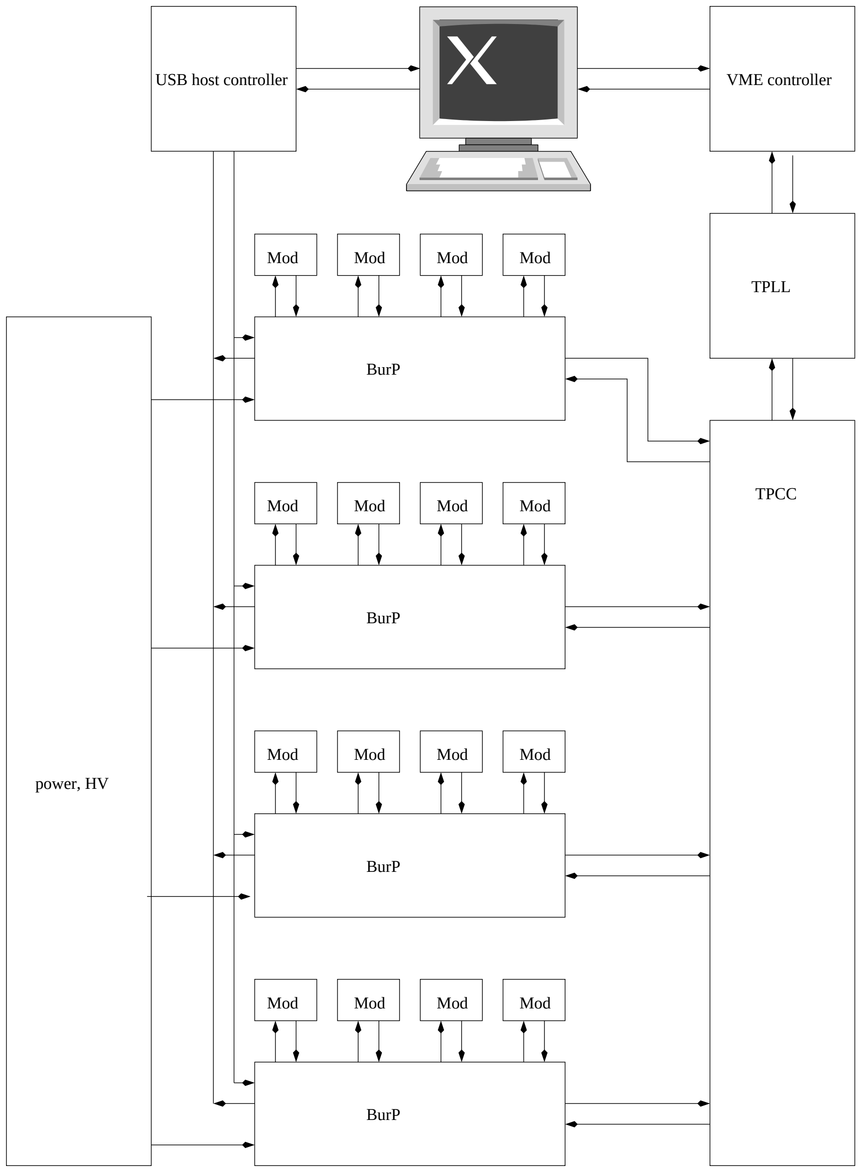

Figure 2.1: Schematic of AMBuSh hardware

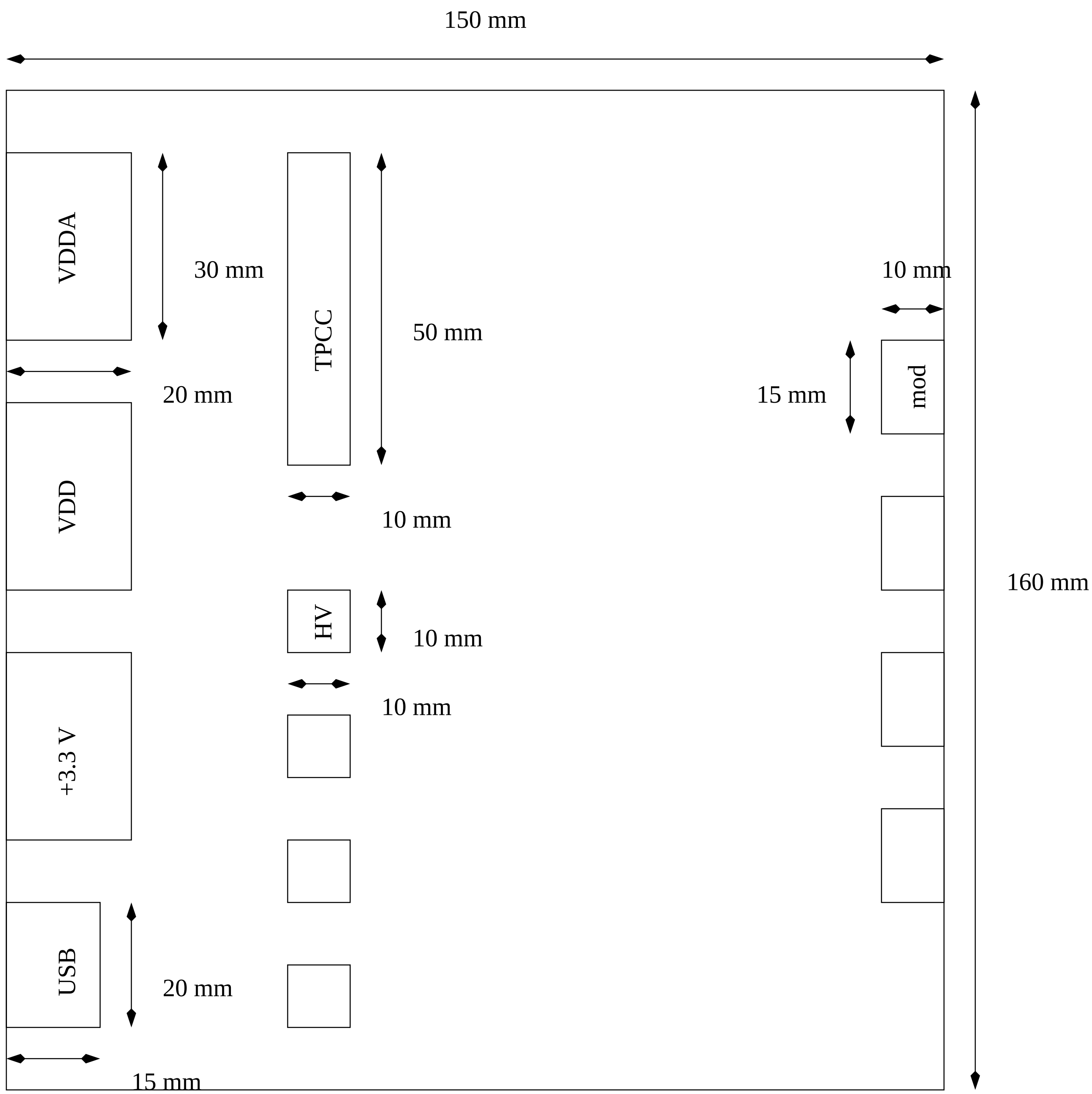

Figure 2.2: Physical layout of the BurP board

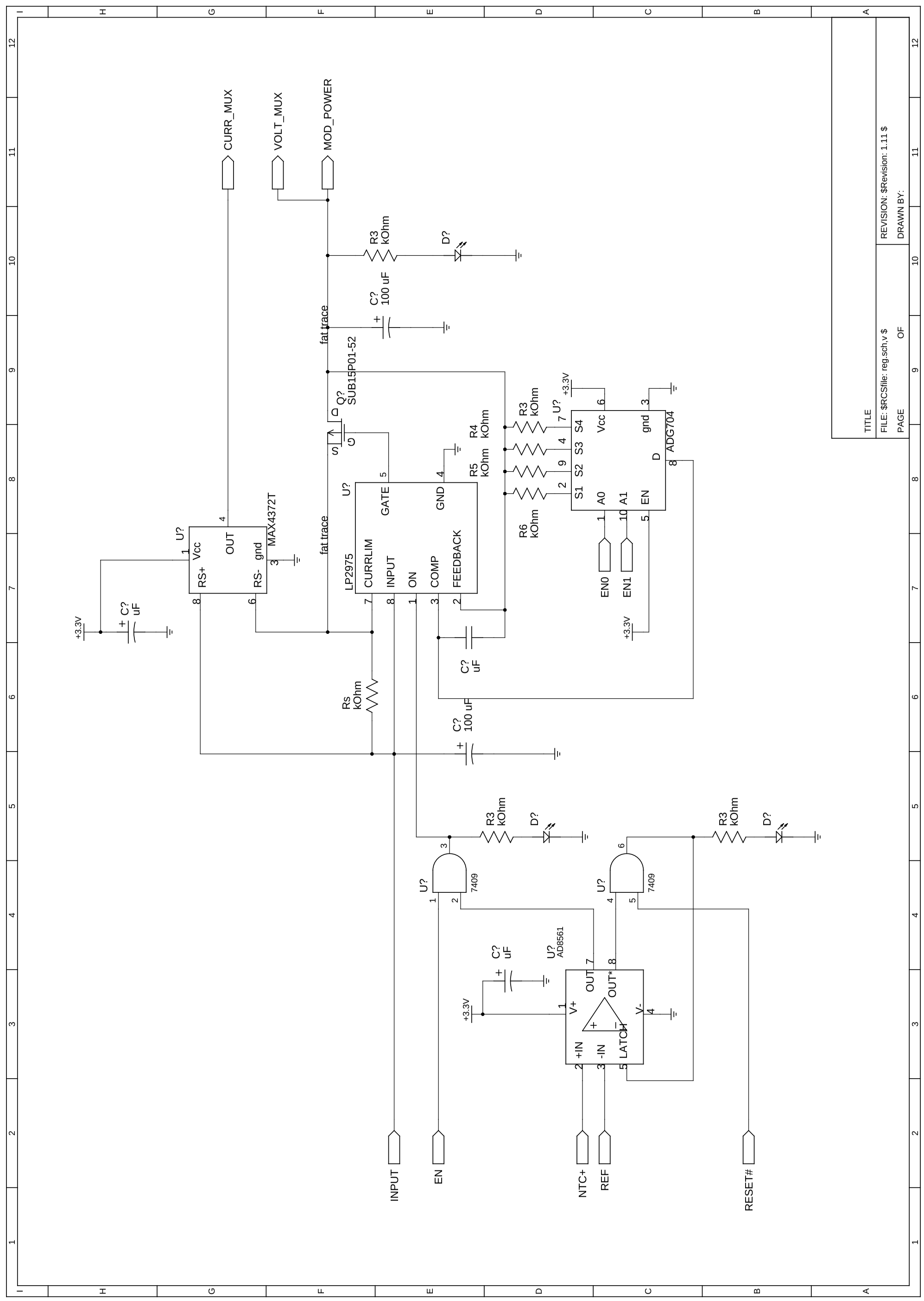

Figure 2.3: Regulator schematic

The control signals from the TPCC are fanned out to all modules connected to the BurP board, as is the 40 MHz clock. The fan-out is accomplished by converting the LVDS clock and control signals from the TPCC to CMOS and then distributing the CMOS signals to one LVDS driver for each module. Consult the top halves of Fig. 2.4 and Fig. 2.5.

2.2.3 Data multiplexing

Each module connected to the BurP board receives the control signals from the TPCC, but only one module at a time is allowed to respond. For this reason the BurP board multiplexes the two data bits returning from the modules. The USB interface on the board ensures that only one module is able to return data by setting the enable bits of the module-side LVDS receivers (bottom half of Fig. 2.4). The data bits from the selected module are passed through LVDS drivers and transmitted back to the TPCC (bottom half of Fig. 2.5).

2.2.4 Clock return

In the pixel module scheme, the board closest to the modules is supposed to return the 40 MHz clock. The LVDS driver (U?) in Fig. 2.6 handles the clock return; the jumper (J?) allows inserting a delay cable to simulate the effect of long lines between BurP and the modules.

2.2.5 Temperature interlock

2.2.6 USB controller

All slow-control interaction between the user and the BurP board is through the USB controller. Our USB controller chip (a Cypress EZ-USB AN2131) is a combination of USB line driver, RAM, souped-up 8051 microprocessor and I2C controller. One of the convenient features of the EZ-USB is that the USB host (the computer running AMBuSh) can download firmware into the 8051; this is discussed in the software chapter.

The EZ-USB has two eight-bit digital I/O ports. Of the sixteen available bits, eight are used to turn on the eight regulators; one resets regulators that have shut down due to the temperature interlock; two set the levels of all digital regulators; two set the levels of all analog signals; and two select which module can transmit data back to the TPCC.

BurP uses the I2C bus of the EZ-USB to talk to an EEPROM and to three ADCs. The EEPROM is used to store the 8051 firmware (see the software chapter). Each of the ADCs multiplexes eight channels. The 8051 selects one of the eight regulator output voltages to be digitized at the first ADC; one of the eight regulator currents to be digitized at the second ADC; and one of the four module temperatures to be digitized at the third ADC.

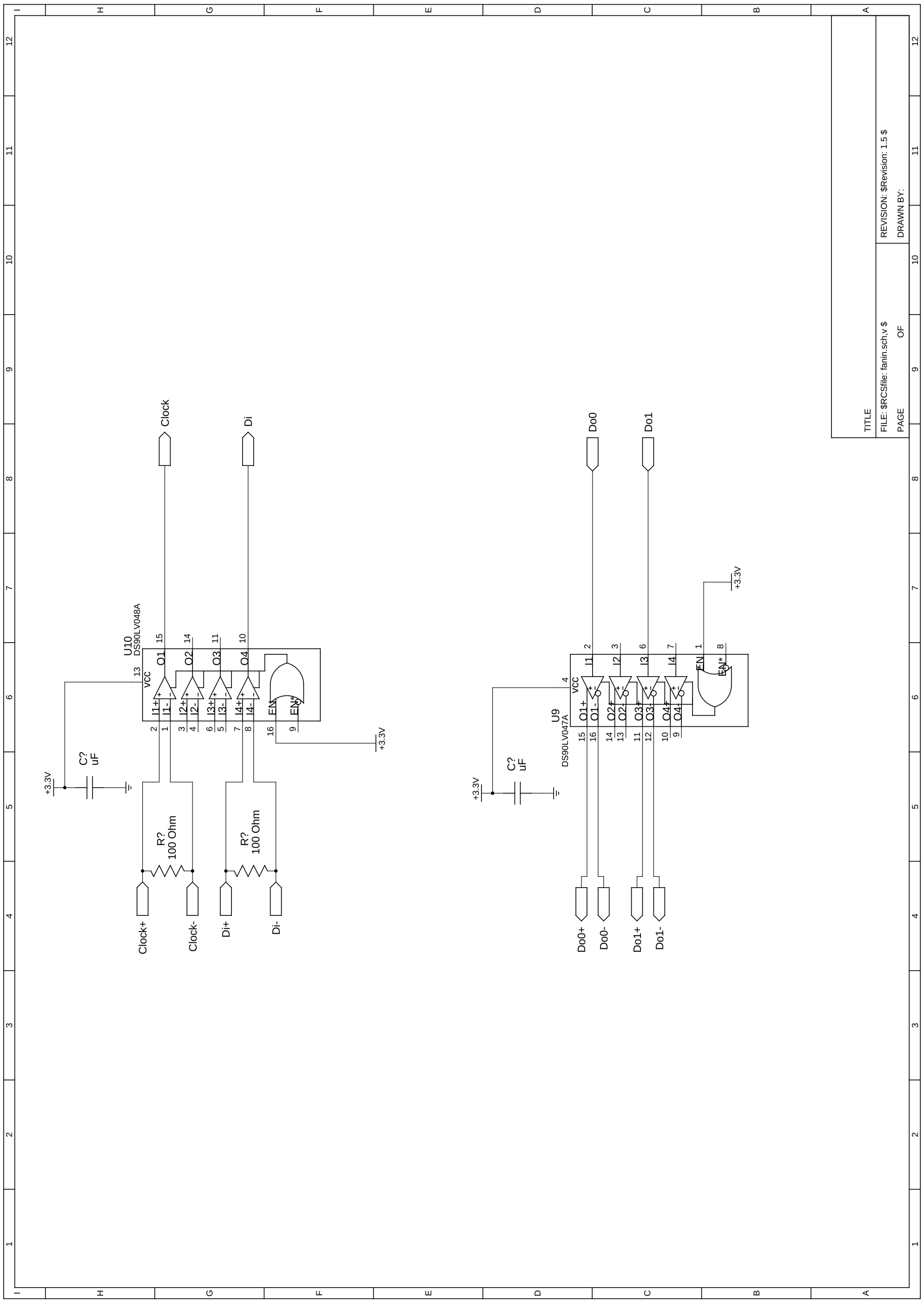

Figure 2.4: TPCC-side LVDS buffers

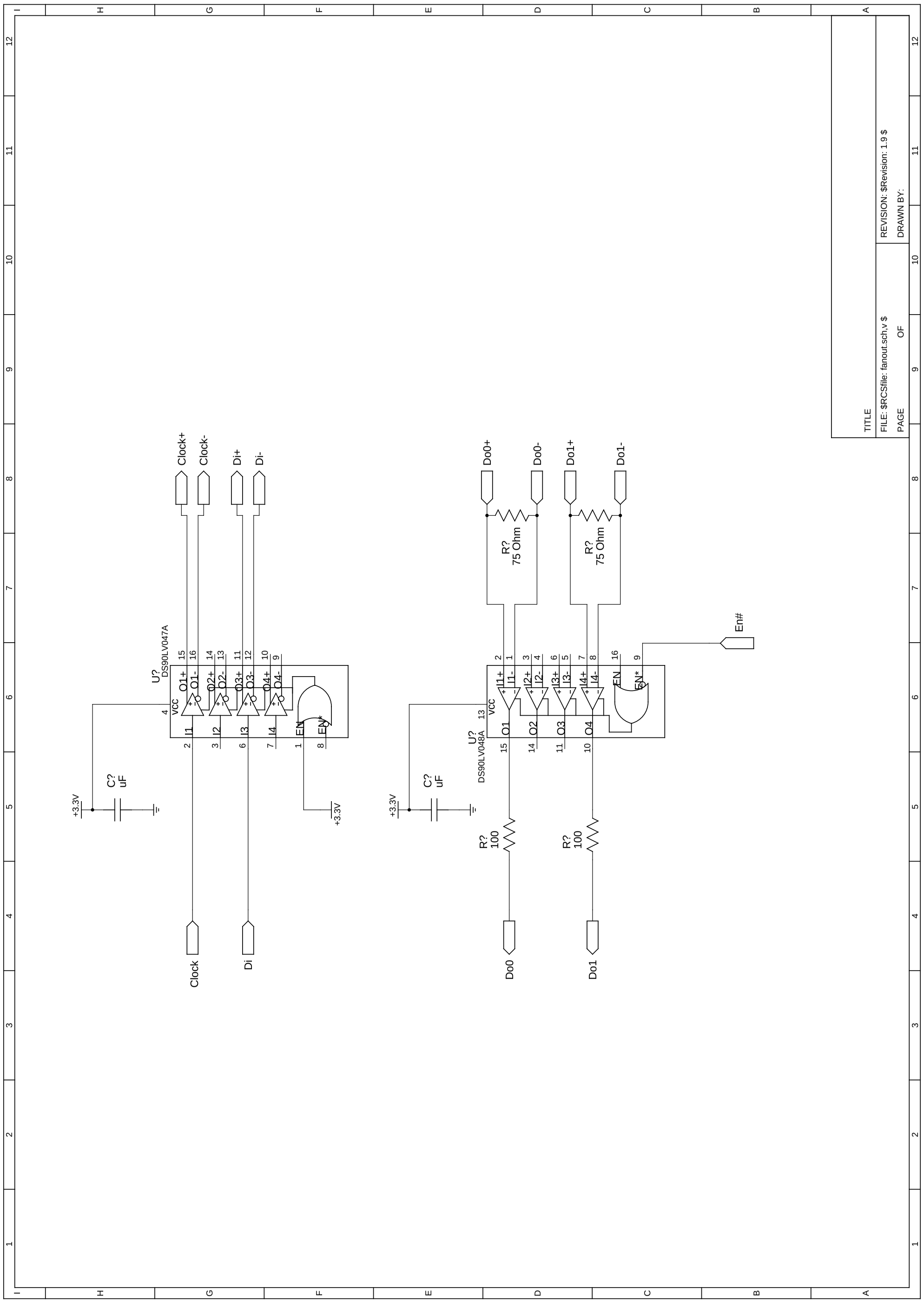

Figure 2.5: Module-side LVDS buffers

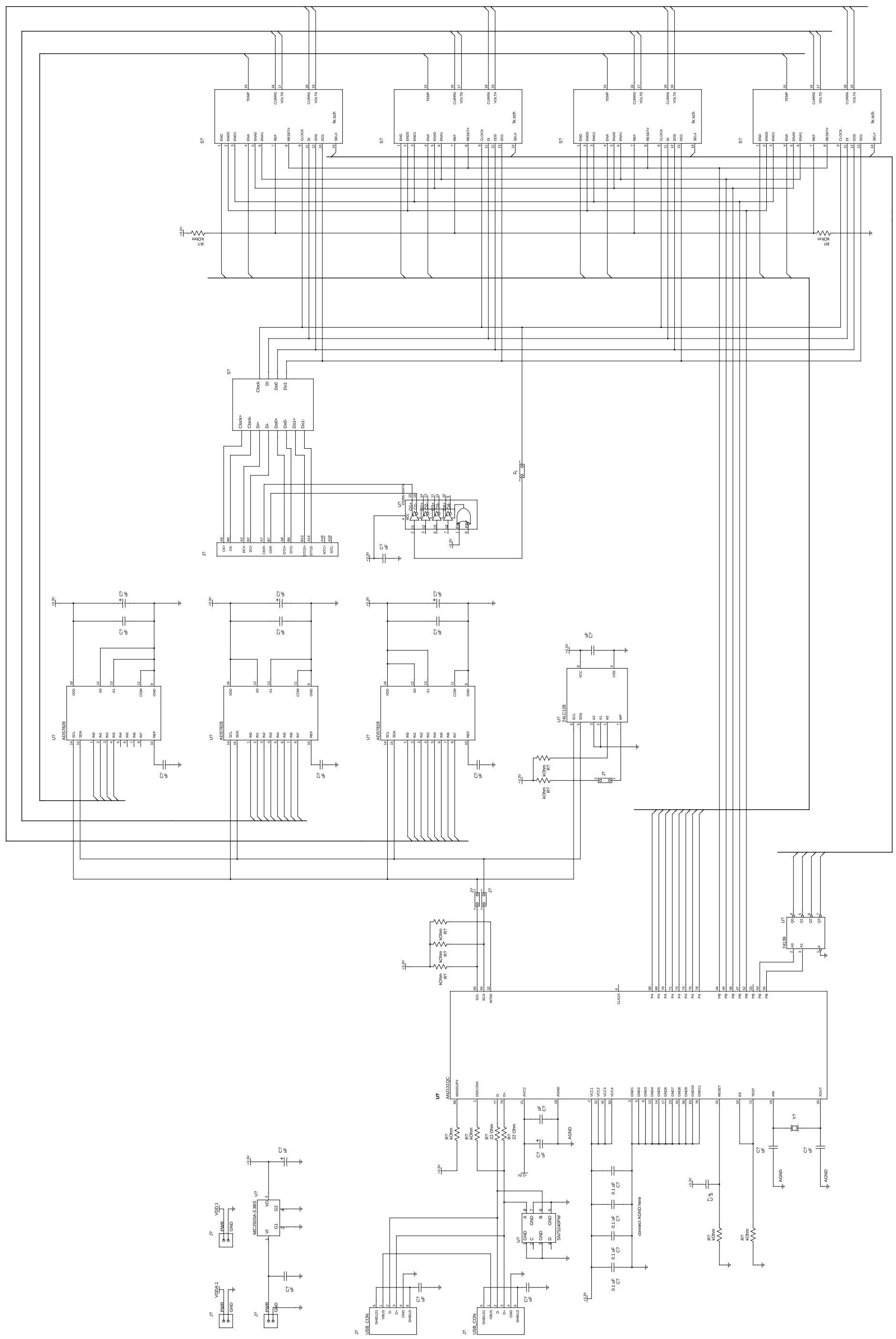

Figure 2.6: The BurP board

Chapter 3

Software

The AMBuSh software follows the producer-consumer model. One process, the *producer*, concerns itself with controlling the hardware and storing the data the hardware produces. This process is designed for maximal stability; it has to run problem-free until the scheduled tests are done. Another process, the *consumer*, looks at the data from the producer, and turns it into plots. Since this process does not control the hardware, it will not interfere with the running of the tests if it crashes. Fig. 3.1 shows how the consumer and producer interact; see also Sec. 3.3.3.

In Sec. 3.1 we describe how to get and build the software, in Sec. 3.2 we show to use the data producer and fiddle with the code, in Sec. 3.3 the consumer.

3.1 Getting and building the software

The software is available from cvs or http. Blah blah blah.

To build the source, you need GNU make and an ISO-compliant C compiler. You also need a POSIX operating system; if you are building on a windows machine, Cygwin will help you out.

Running make on the Makefile in the top level of the source tree will build the data producer binary in `main/ambush` and the data consumer `somewhere else`, along with the libraries they need.

3.2 The data producer

The data producer consists of a shell, a task scheduler, a condition checker and a data management system. (See Fig. 3.2.) The *shell* interprets user input in a C-like language that allows scheduling and canceling tests to perform on modules; issuing slow-control commands; and responding to error (or any, really) conditions. Scheduled tasks get run at their intended time by the *scheduler*. The *error checker* periodically checks for error conditions; and a *queue* takes care that concurrent interactions with the hardware don't result in resource contention. The *data management system* writes to files the data that is generated by testing and monitoring the modules. Sec. 3.2.2 explains each producer component in detail.

3.2.1 Using the software

The user interacts with AMBuSh through the AMBuSh shell. This section describes the syntax of the shell scripting language and how to use the language to run a burn-in job. We also explain how to make new hardware systems available.

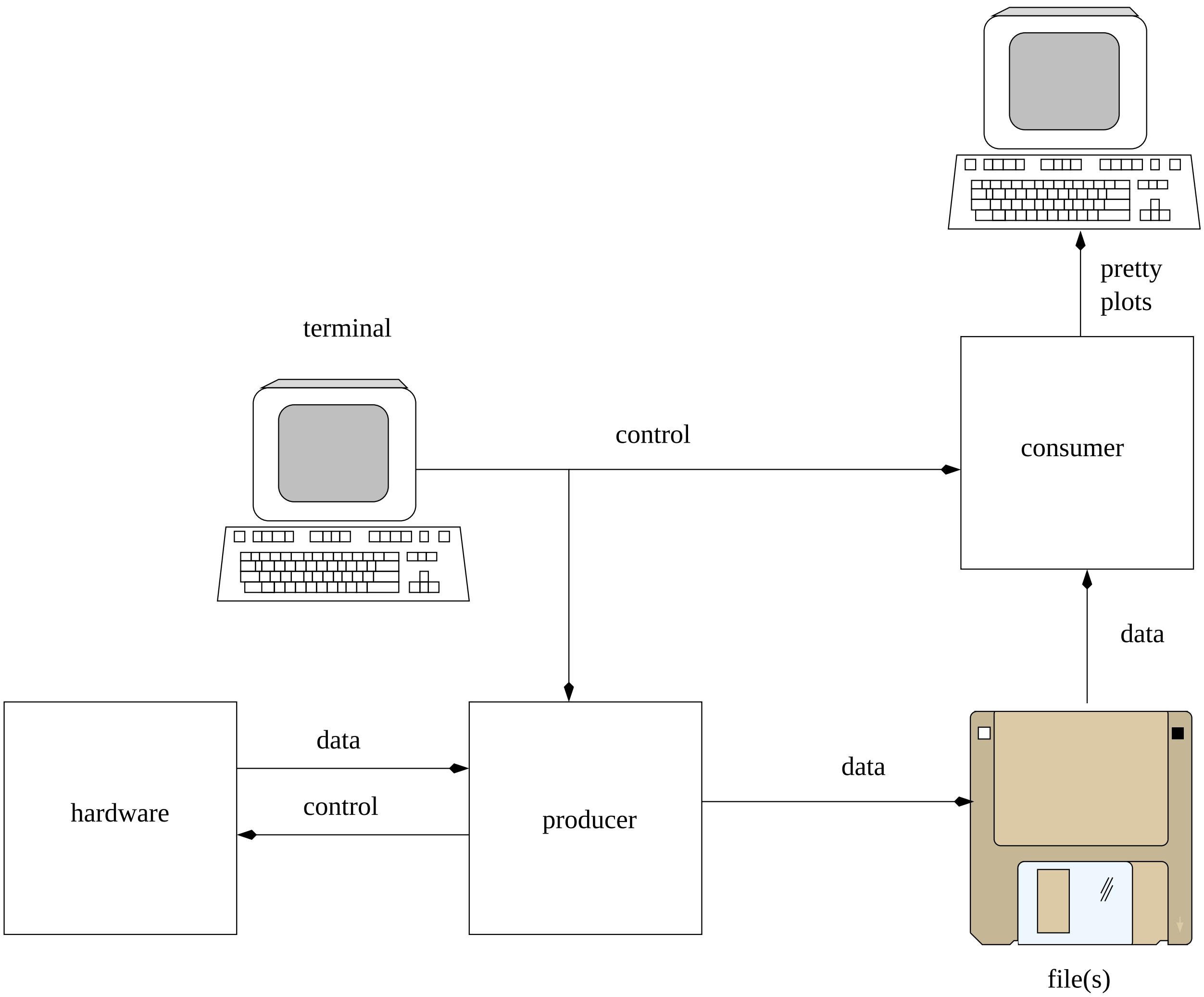

Figure 3.1: Data producer and data consumer

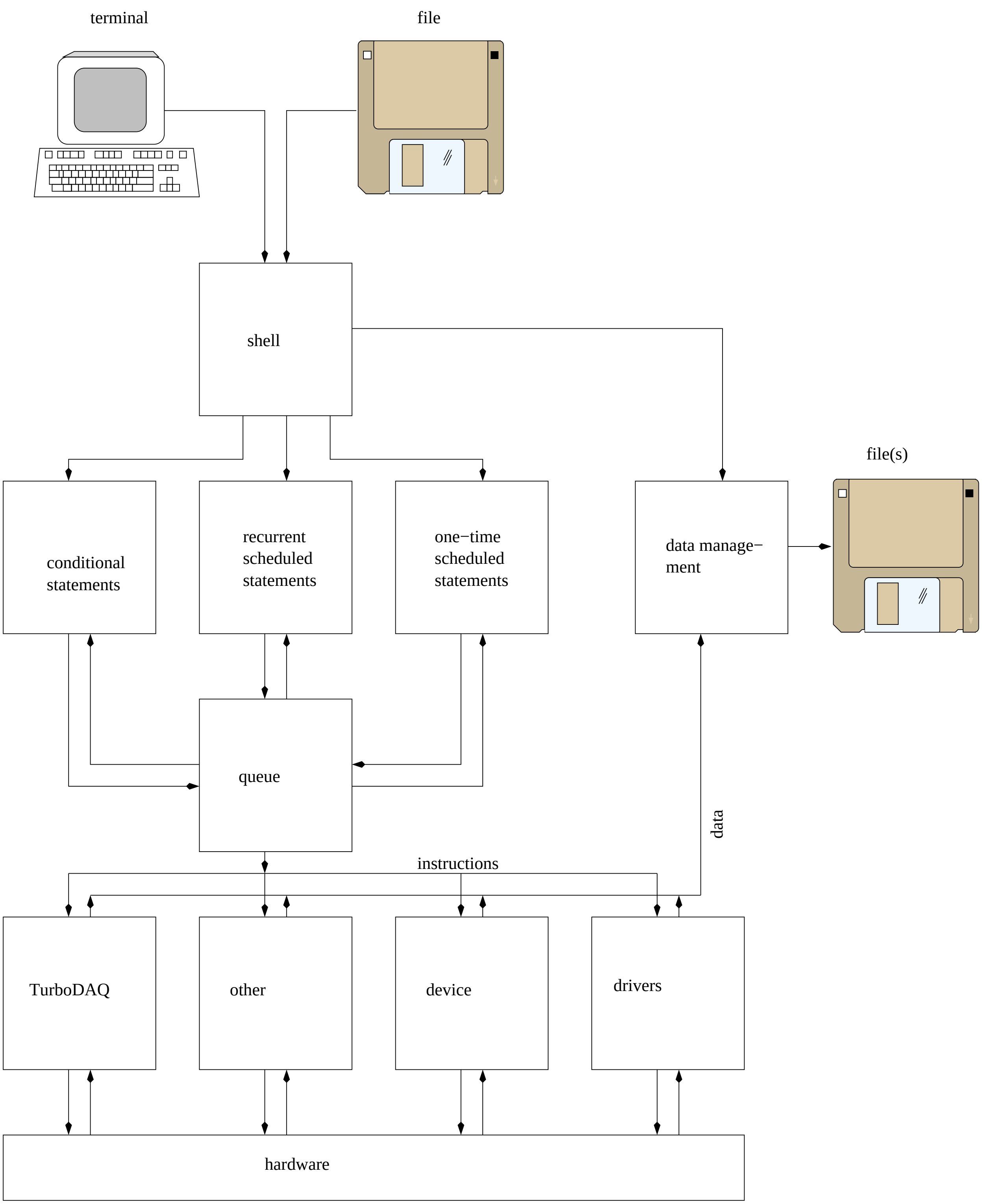

Figure 3.2: Structure of the AMBuSh data producer

3.2.1.1 Language syntax

3.2.1.1.1 Data types Valid data types are `int`, `float`, `string`, `time` and `period`. `int` and `float` are used to store integer and floating point numbers, `string` to store character strings, `time` to store time values for task scheduling and `period` to store the period of recurring tasks. Conversion between data types happens automatically when it makes sense; when it doesn't, the shell will complain. There should be a table (Table **??**) here to show what conversions take place.

3.2.1.1.2 Variables Variables are declared as in C:

```
type variable_name [ = initializer ];
```

The scope of all variables is global.

blah blah blah blah.

3.2.1.1.3 Expressions The shell recognizes all C arithmetic expressions valid for the data types above. It also recognizes function calls. (The shell does not provide procedural abstraction, so it is not possible to *define* functions; you can only call built-in functions (paragraph ??) or functions you have implemented in real C and told AMBuSh about (some other paragraph)).

3.2.1.1.4 Statements Expression statements, compound statements, selection statements, atomic statements and schedule statements are supported by the shell. *Expression statements* are familiar from C: they are of the form

```
expr;
```

Compound statements are equally familiar:

```
{
statement1
statement2
...
}
```

The remaining statement types are not found in C or found in modified form. They exist to add concurrency to the AMBuSh shell scripting language. We describe each statement type separately.

Selection ("condition") statements A *selection statement* has the form

```
if (expr) statement
```

This is syntactically identical to an `if` statement in C, but semantically different. Instead of immediately evaluating the predicate consequent, the shell continuously re-evaluates the predicate and evaluates the consequent once the predicate is met. This allows shell scripts to watch for conditions (tripped power supply, for example) and respond to them once they occur.

The shell also understands C-style `if` statements, where the consequent is evaluated immediately. These look thus:

```
immediate if (expr) statement
```

The C `switch` selection statement is not supported.

Atomic statements

Schedule statements

3.2.1.1.5 Built-in functions We need lots of built-in functions to have a working system. There should be an appendix to list them.

3.2.1.2 Running a burn-in job

Easy. Here is the sequence:

3.2.1.2.1 Preparing a script blah blah blah. Script starts running if the syntax is OK, and you get your command prompt back.

3.2.1.2.2 Making modifications along the way Imagine the burn-in setup has been running happily for three days. Suddenly a module fails and stops returning data. There is little sense in wasting time on running further tests on this module; the shell should provide a way to unschedule the tests scheduled for the module. The shell does indeed provide this functionality. Even while scheduled tasks are running, AMBuSh provides a fully functional shell. A list of scheduled tasks and conditions can be obtained using `list_sched()` and `list_cond()`, and you can cancel any of them by calling `cancel_sched(int idx)` or `cancel_cond(int idx)`. You can also schedule new statements or add conditions at any time from the command line, or slurp in another script.

3.2.1.3 Adding functionality

It is easy to extend the functionality AMBuSh supports. To make a new instrument or other new functionality available, you need to provide a *driver*, *i.e.* a set of functions that you want to put at the user's disposal.

Once you have written the driver, AMBuSh needs to know about the new functions available from the shell. This is best done by placing a header file in the `inc/shell` directory of the source tree and re-running make at the root of the source tree. Please note that because of the lobotomized nature of the C interpreter, everything in your header file but the function prototypes is ignored, and the function prototypes should only use `int`, `float`, `char *` and `void` type specifiers; if a function takes no arguments, you must declare the argument list `void`.

If your driver routines need exclusive access to any hardware systems, they should lock those systems using the `LOCK(`*system name*`[, `*other system name ...*`])` macro. The systems you lock remain locked for the duration of your routine. If your driver provides access to a new system that AMBuSh doesn't know about yet, you must give the system a name before you can call the locking macro. You can do this using the `PROVIDES(`*system name*`)` macro, which wants a `char *` argument. AMBuSh uses the information provided with the `LOCK` call to figure out which tasks it can run concurrently without causing resource contention.

3.2.2 Hacking the software

Fig. 3.2 shows the structure of the AMBuSh producer. Gory detailed blah blah blah.

3.3 The data consumer

The data consumer's responsibility is to turn the data acquired from the hardware into pretty plots. It is also in charge of reducing the data and entering it into a database or central storage location, and of anything else that does not require direct interaction with the burn-in hardware. This way, any problems in running the consumer leave the producer unaffected, reducing the risk that a burn-in run has to be aborted.

3.3.1 Using the software

This is completely open for discussion. Since most of the data generated in the burn-in is from TurboDAQ, the module data analysis framework developed in Bonn is a good solution for making pretty plots.

There should also be some mechanism for reducing the data from TurboDAQ — some pseudo-mechanized way of telling whether the module passed the burn-in — and storing the reduced data in a central location.

3.3.2 Hacking the software

And this should be written by the consumer author.

3.3.3 Interaction between consumer and producer

The interaction between the consumer and the producer must leave the stability of the producer unaffected. That requires that producer and consumer be separate processes. Communication between the processes should be limited; unless there is good reason, it should be restricted to files written by the producer and read by the consumer.

Currently we plan to write data into a TurboDAQ-like directory structure: one directory per module, named after the module serial number. In addition to the standard TurboDAQ subdirectories, there will be a directory containing slow-control data — called, say, `slow`. This directory will contain an ASCII file that records the history of supply voltages, currents, and module temperatures along with a time stamp.