AMBuSh, the Atlas pixel module burn-in shell

Johannes Mülmenstädt

March 29, 2004

$Revision: 1.5 $

Abstract

AMBuSh, the ATLAS module burn-in shell, is part of the ATLAS module burn-in system. It controls the burning-in of pixel modules by executing a script of tests to be performed on modules. In this document we describe the components of AMBuSh, define its syntax and explain how it interacts with the rest of the module burn-in system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ATLAS pixel module burn-in system 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.

The burn-in 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. 1.1 shows how the consumer and producer interact; see also Ch. 7.

This document describes the producer, called **A**TLAS Pixel **M**odule **Bu**rn-in **Sh**ell (AMBuSh). In Ch. 2 we describe how to get, build and configured the software. In Ch. 3 we describe the structure of AMBuSh: the various instruction buffers for scheduled and conditional statements, the execution queue and data handling system. In Ch. 4 we describe the C-like language in which AMBuSh scripts are written. In Ch. 5 we delve into the gory details of the code. Only slightly scarred, we emerge in Ch. 6 to describe how to make AMBuSh aware of hardware. In Ch. 7 we talk about the interaction between AMBuSh and the data consumer. Finally, in Ch. 8, we list the functions available in AMBuSh scripts.

1.1 How to get this document

This document is the official documentation for AMBuSh. Since AMBuSh may change between releases, you should make sure that the release of the documentation matches the release of your AMBuSh software. The software release is printed when you invoke the AMBuSh shell. The release of this copy of the documentation is `$Id:$`.

If your documentation is outdated, you can find current documentation in the `doc/` directory of the AMBuSh distribution. You can obtain the AMBuSh distribution from `http://pixdata.lbl.gov`.

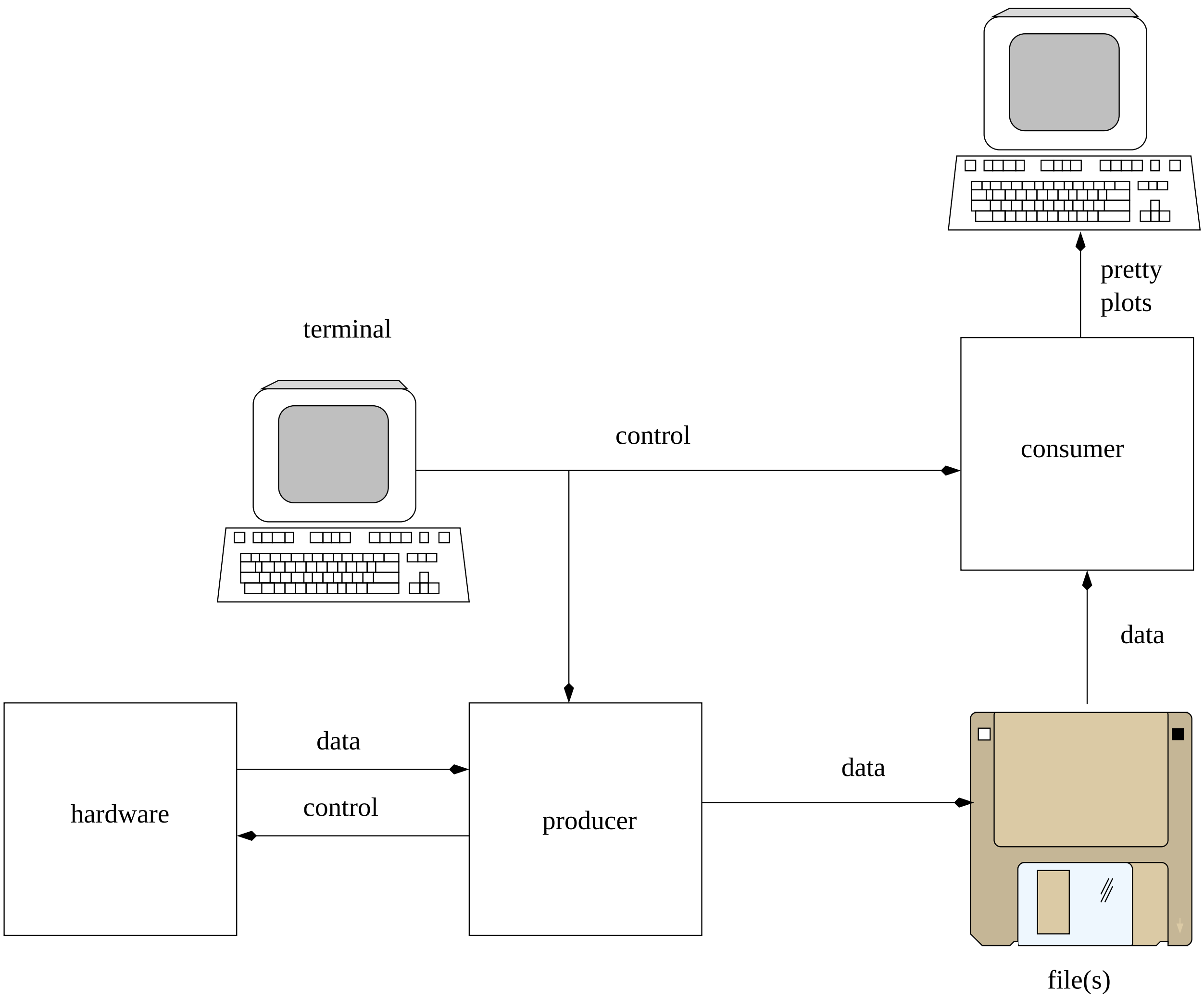

Figure 1.1: Data producer and data consumer

Chapter 2

Getting and building AMBuSh

2.1 Getting

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

2.2 Building

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.

2.3 Configuring

blah blah blah Configuration crap

Chapter 3

The structure of AMBuSh

The data producer consists of a shell, a task scheduler, a condition checker and a data management system. (See Fig. 3.1.) 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. **??** explains each producer component in detail.

3.1 Shell

3.2 Scheduler

3.2.1 atbuf

3.2.2 schedbuf

3.3 Condition checker

3.4 Queue

3.5 Hardware interface

3.6 Data management system

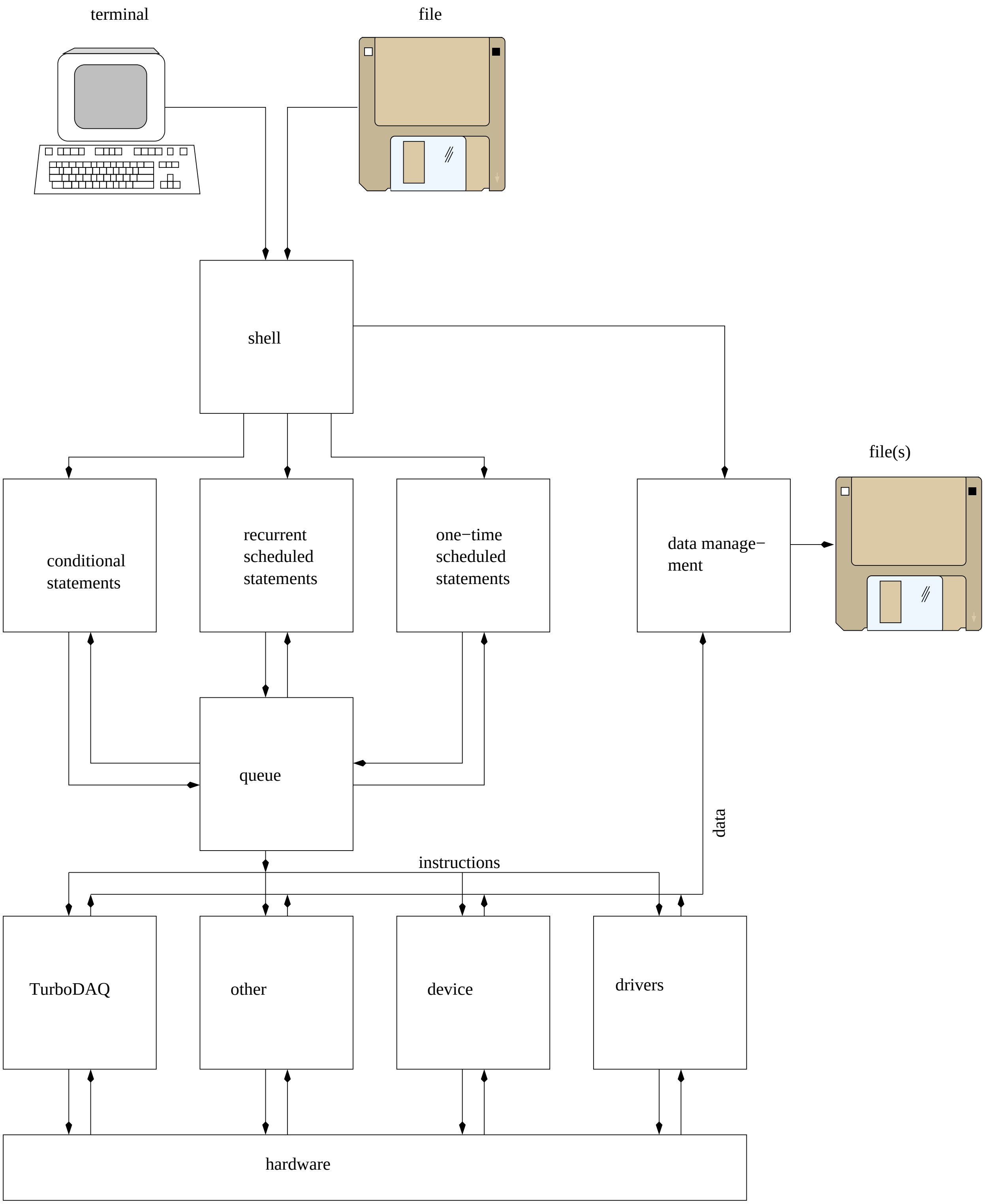

Figure 3.1: Structure of the AMBuSh data producer

Chapter 4

Syntax

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.

4.1 Language syntax

4.1.1 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.

4.1.2 Variables

Variables are declared as in C:

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

The scope of all variables is global.

blah blah blah blah.

4.1.3 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 (subsection ??) or functions you have implemented in real C and told AMBuSh about (some other subsection)).

Note that every instruction generated by the shell can is executed in parallel with all others. blah blah blah. Refer to Sec. 3.4 for discussion of the threading.

4.1.4 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:

```
{
statement_1
statement_2
...
}
```

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.

4.1.4.1 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.

4.1.4.2 Schedule statements

Schedule statements are the heart and soul of AMBuSh. They come in two variants. The first one, the `at` statement, schedules a task for execution at a specified time:

```
at (expr) statement
```

expr should evaluate to a time in the future.

The second type of schedule statement is the `every` statement, which repeatedly executes a task with a specified period:

```
every (expr) statement
```

expr should evaluate to a time period.

4.1.4.3 Init statement

4.1.4.4 Atomic statements

The AMBuSh system allows several tasks to be performed concurrently. Frequently a set of instructions needs to be executed as a unit, *i.e.* no instructions from another tasks can be executed until the entire group of instructions has been processed. Such a group of instructions is called *atomic*. To create an atomic group in a shell script, you can use an *atomic statement*:

```
atomic statement
```

All the instructions generated by the statement are guaranteed to be executed as a group. (Note that even an expression statement can generate multiple instructions.)

4.1.4.5 Sequential statements

4.1.5 Built-in functions

We need lots of built-in functions to have a working system. There should be a chapter (Ch. 8) to list them.

4.2 Running a burn-in job

Easy. Here is the sequence:

4.2.1 Preparing a script

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

4.2.2 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.

Chapter 5

AMBuSh internals

Chapter 6

Interaction with hardware

6.1 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.

6.2 Auxiliary programs

Like pontifex

Chapter 7

Interaction between the consumer and AMBuSh

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.

file names; sample data;

configuration socket

Chapter 8

AMBuSh function manual

In this chapter we list all the functions that can be called from the shell. We've tried to make the order sensible.

8.1 Builtin functions

8.2 Surf functions

8.3 TurboDAQ functions

8.4 Environmental chamber functions

8.5 Logging functions