The PATT Chip Test Setup

Maurice Garcia-Sciveres   Dipanjan Ray

February 11, 1998

Contents

Part I
System Overview

1 Introduction

The system described here was designed for automated testing of svx2 and svx3 die in wafer form. It can also be used for operating svx2 and svx3 chips that have been mounted on boards or hybrids, or even as a generic stimulus system for testing other devices with digital I/O. The main design goals were as follows:

1. A single system flexible enough to be used with svx2 and svx3 chips in their various incarnations, and with the ability to easily change the way the chips are controlled as our understanding of their operation evolves.

2. The system is put together using components already in hand, allowing for addition or substitution of new (unknown) components as needs may arise.

3. Troubleshooting is fast and simple.

The svx2 and svx3 chips have the common feature of lacking analog outputs. Therefore, they can be fully exercised with a pattern generator plus logic analyzer, or with a pattern generator plus a digital memory for statistical tests. However, pattern generator modules are not only expensive, but also slow and tedious to program (even when downloading patterns from a computer, as the computer-to-module interface tends to be slow). But for wafer probing it is necessary to switch quickly between many different chip control patterns. A computer with an output register can in fact function as a "pattern generator" more suited for wafer probing operations than a pattern generator unit. The problem with such a homemade pattern generator is that successive pattern lines will not come out with the precise timing or speed expected of a ready-made unit. In the present system, this deficiency is remedied with a simple, custom built hardware interface added downstream of the output register (described in detail later). Similarly, data acquisition is implemented using an input register to the master computer preceded by a simple, custom built interface to match the speed and timing of the svx2 and svx3 chips.

Having no need for complex standalone modules for chip operation or data acquisition, the design goals have been met by by building as much as possible of the system out of software. All intelligence is centralized in one master computer, all other hardware serving only as electrical or mechanical interface between the computer and the svx2 or svx3 chips. The software, however, is not in the form of one do-it-all executable, but instead consists of a set of object libraries, with no executable at all. The user must write a specific master program for each new task, that will basically consist of a sequence of calls to an appropriate subset of the system routines. This permits easy tailoring of the system for a wide range of possible applications, and allows the software to be structured in self-contained, independent libraries addressing the completely different elements of wafer probing. These elements are:

1. Mechanical motion of wafer and probing of die.

2. Electrical control of die functions and data acquisition.

3. Data storage and analysis.

Finally, in order to make the system as robust as possible no special utilities or libraries, such as CERN software or user interface packages, have been used. All code is written in the C language, with a conscious effort to avoid constructions that may lead to different results on different compilers (the user master program need not be written in C. All object modules can be called from a FORTRAN program). Device specific software (and perhaps firmware) is necessary for driving the various hardware modules. An effort has been made to place such routines at the lowest possible level in the software, so that new modules with new drivers can be incorporated with a minimum amount of work. This should also make the system easily portable.

2 Setup

□

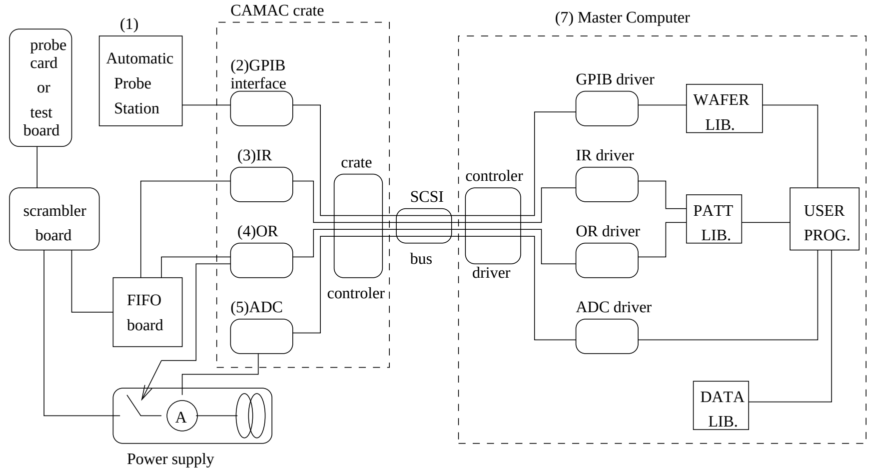

Figure 2.1: Schematic diagram of chip test setup implemented for automated wafer probing. The reference numbers in some of the components correspond to the item number of the equipment list given in the text.

Fig. 2.1 shows a diagram of the system as implemented for the first time. Hardware and software elements are shown on an equal footing, in the spirit of the design. Elements that *can* be replaced by equivalent parts with similar functionality are shown as ovals, while fixed elements are shown as rectangles. For example, the setup as depicted uses CAMAC crate mounted modules and a SCSI interface between the master computer and the CAMAC crate. But a setup could just as easily be implemented using VME modules, or a different computer to CAMAC interface, or even a PC with an I/O board instead of CAMAC or VME. In fact, all the commercial hardware is replaceable. The specific hardware components of the present setup are as follows (numbered as in the figure):

1. Automatic Probe Station: Alessi REL-2500 (Not needed to operate svx2/3 chips mounted on test boards. Used only to connect to svx2/3 chips on un-diced wafers).

2. GPIB interface: Kinetic Systems 3388 (Not needed to operate svx2/3 chips. Used only to control automatic probe station).

3. Input Register: Kinetic Systems 3430.

4. Output Register: Jorway Model 40.

5. ADC: LeCroy 8232 (Not needed to operate svx2/3 chips. Used only for automated current monitoring).

6. Crate Controller: Jorway Model 73A (connects directly to SCSI bus)

7. Master Computer: Silicon Graphics Indy.

Additionally there are two custom-built circuit boards called the PATT02 FIFO board and the Scrambler board. The FIFO board is common to all test setups and is the subject of Section 3 of this document. The scrambler board, on the other hand, is different for each type of test chip, e.g. a different scrambler is needed for SVX2 than for SVX3. A scrambler board designed for the SVX3 chip is described in Section 4.

Part II
Hardware

3 The PATT02 board

This circuit board is the element that makes a computer with an output register look like a pattern generator, and a computer with an input register look like a fast data acquisition system. In simplest terms it consists of two first-in-first-out digital memories (FIFOs) and a clock oscillator.

Binary data to control the chip under test come out of the output register at a rate typical of the hardware (approximately 0.5MHz for CAMAC) with variable timing between successive bits, due to the fact that the CAMAC and SCSI buses are shared between several devices and arbitrated. Even if the average output register frequency were near the 50MHz desired for svx2/3, these data would not do as a chip control pattern because of the arbitrary timing between successive bits. So instead of being used to directly control the chip under test, the data from the output register are stored in a FIFO, called the pattern FIFO. Once the entire pattern needed for a given chip test has been stored in the pattern FIFO, an oscillator on the PATT02 board clocks the data out of the FIFO with constant timing and at the desired frequency. The FIFO clearly must be deep enough for an entire control pattern. The components used come in depths of up to 16K, and operate at frequencies up to 50MHz. A depth of 4K has proved adequate for most svx2/3 operations so far.

At the same time as control data move from the pattern FIFO to the chip under test, a second memory, called the DAQ FIFO, is used to store any data coming back from the chip. The DAQ FIFO is strobed by a dedicated line in the pattern FIFO, so it runs synchronously with any chip control pattern. Once the entire pattern has run, any data in the DAQ FIFO can be transferred to the input register for storage in the master computer.

Note that the way the PATT02 board operates means that this system is not capable of continuous chip control and data acquisition, as would be required for a detector being used in a physics experiment. Rather, the system is meant for limited “bursts” of control/daq operation, as is adequate for chip testing.

3.1 Board Layout

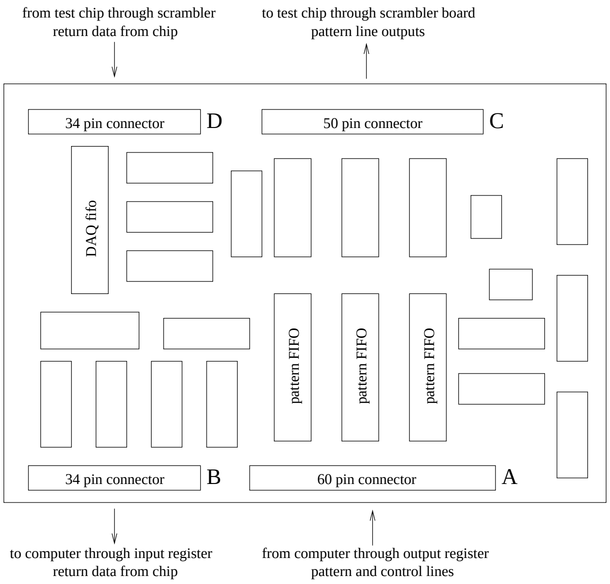

Figure 3.1: Rough layout of PATT02 board with FIFOs and connectors labeled.

Figure 3.1 shows a rough map of the PATT02 board with key components marked. Logic patterns and control signals are sent from the computer to the board via 60-pin connector A in the lower right corner of the board. Meanwhile, data in the DAQ FIFO can be read into the computer via the adjacent 34-pin connector B.

The pattern FIFO consists of three separate integrated circuits in parallel, making up a 23-bit wide FIFO memory. The output of the pattern FIFO is available via a 50-pin connector C at the top of the board, while the DAQ FIFO receives its inputs from the 34-pin connector D at the upper left of the board.

Because the FIFO IC is such an integral component of the board, additional information about its functionality and timing is given in section 3.8.

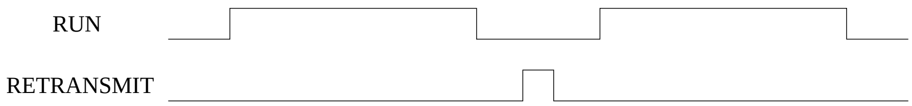

Figure 3.2: Timing for RETRANSMIT signal (with respect to RUN).

3.2 Control Lines

(pin numbers refer to 60 pin connector A leading to the pattern FIFO inputs.)

1. CONTROL (pin 1): HIGH for normal operation; pulse low to reset pattern and DAQ FIFOs. The FIFOs MUST be reset after the board is turned on and before its first use.

2. RUN (pin 3): normally LOW. To run a pattern stored in the pattern FIFOs, the RUN signal must be raised high for at least a time equal to the length of the pattern times the oscillator period.

3. WRITE (pin 5): normally LOW, this signal is used to strobe a pattern into the pattern FIFOs. Each bit of the pattern is strobed in by sending a positive pulse to WRITE.

4. RETRANSMIT (pin 7): normally LOW. After running a stored pattern, pulse RETRANSMIT high for at least 12 ns in order to place patterns in a position to be run again. Timing for this signal is shown in figure 3.2.

5. BIT 9 SELECT (pin 9): the FIFOs used on the PATT02 board are 9 bits wide. The bit 9 select signal allows one to choose whether the signal on pin 29 or pin 30 of the 34-pin connector returning from the scrambler board is sent to the DAQ FIFO's 9th data input. (B9S high selects pin 29).

6. STORE (pin 55): normally HIGH, this pattern line is lowered to tell the DAQ FIFO to store the data word at its inputs during the execution of a pattern. Usage of STORE is discussed in section 3.6.

3.3 Power Requirements

The PATT02 board runs on a single +5 V power supply, which is connected between the large vias labelled '+5 V' and 'GND'. To properly power up the board, turn on the +5 V power supply. The control lines should be set at the following levels:

control line	voltage level
CONTROL	HIGH
RETRANSMIT	LOW
RUN	LOW
WRITE	LOW

Then pulse CONTROL low for at least 12 ns, to reset the FIFOs. The first set of data retrieved from the DAQ FIFO after powerup should be discarded, as it may contain an extra data word.

3.4 Downloading a Pattern to the Board

To strobe each bit of a given pattern into the pattern FIFOs, present the data at the FIFO inputs with WRITE high; then, maintaining the data, lower WRITE. (See figures 3.3 and 3.4.) A setup time of 9 ns must be observed with respect to the falling edge of WRITE; because the pattern is strobed into the FIFOs at CAMAC speed (roughly 1 MHz), this is not really a concern.

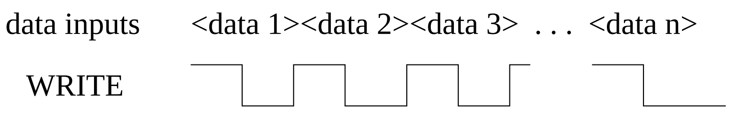

Figure 3.3: Pattern download timing diagram

3.5 Operation of the Pattern FIFO

Running a pattern causes it to appear, bit by bit, at the 50-pin output connector C, where a scrambler board may be added to provide the necessary

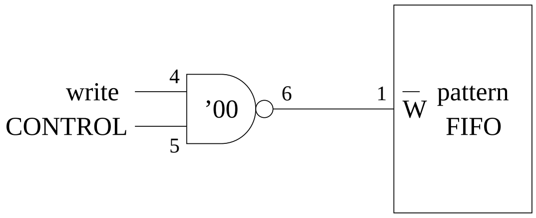

Figure 3.4: Pattern download circuitry.

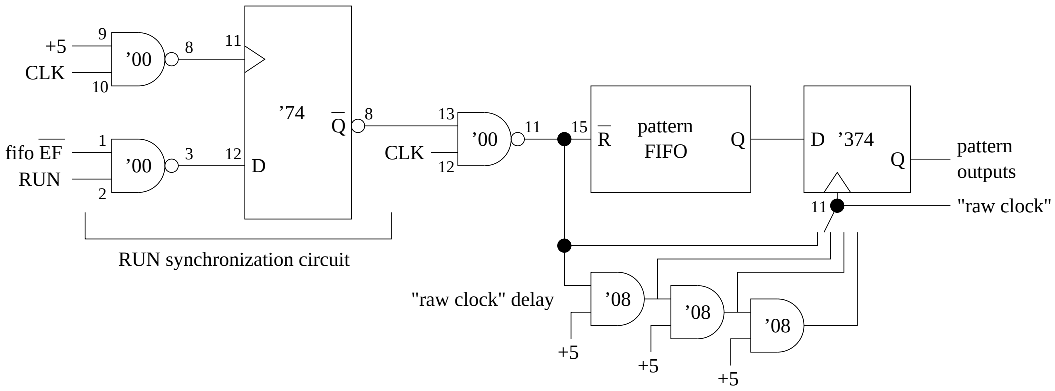

Figure 3.5: Circuitry for running pattern.

interfacing with a device under test. The speed at which the pattern is produced depends directly upon the frequency of the oscillator on the PATT02 board; one pattern bit appears per period of oscillation.

To run a pattern stored in the pattern FIFOs, simply raise RUN high for at least the amount of time required for the entire pattern to be output (i.e. the length of the pattern times the oscillator period).

The circuitry controlling the running of a pattern is shown in figure 3.5. Some features of this circuit deserve further comment:

- The RUN synchronization circuit ensures that when RUN is first raised to run a pattern, the Q-bar output of the '74 goes high while CLK is low. Thus, no glitches appear at the start of a pattern.

- Each pattern bit is first clocked out of the pattern FIFO into a '374, and then out of the '374 to the output connector C. The presence of the '374 is required because the FIFO does not maintain valid data at its outputs after a rising edge of the $\overline{\mathrm{R}}$ signal.

- The "raw clock" delay circuit allows one to delay the '374 clocking signal with respect to the FIFO read signal. This is needed in order to give the '374 enough setup time at high oscillator frequency (50 MHz).
- Note that although the oscillator chip (CLK) runs all the time, "raw clock" contains a clock signal only while a pattern is running.
- At high oscillator speeds (50 MHz), an extra pattern word is sometimes clocked out at the very end of a pattern. We have found that the FIFO's empty flag signal sometimes does not get to the critical NAND gate quickly enough to prevent an extra "raw clock" pulse. We have not observed any resultant problems in our usage of the board.

3.6 Operation of the DAQ FIFO

It is sometimes necessary to store data returning from the test chip during a pattern (for example, during the read-out mode of an SVX2/3 chip). This is accomplished using the "store" pattern line corresponding to pin 55 on the 60-pin connector A. Note that pin 55 does NOT control STORE in real time. Rather, the store signal must be programmed into whatever pattern one wishes to run, and the actual control of the DAQ FIFO occurs only at the time the pattern runs.

"store" is normally kept high. For each bit of the pattern where "store" is low, one byte is stored in the DAQ FIFO. Figure 3.6 shows the timing of the DAQ FIFO's write enable signal with respect to the raw clock and the '374 output associated with the "store" pattern line, for a "store" pattern of LHHLLHH. Each bit of the pattern emerges on a rising edge of the raw clock; note that a low cycle of STORE causes the DAQ FIFO to store the byte present during the **next** pattern cycle, not the pattern cycle in which STORE is actually low. Holding store low for more than one pattern cycle causes repetitive write pulses to be sent to the DAQ FIFO, with one byte being stored per pattern cycle.

For data to be stored properly in the DAQ FIFO, one must allow enough setup time before each rising edge of the DAQ FIFO $\overline{\mathrm{W}}$ signal. However, different devices under test will typically have different characteristic propagation delays from the time they are told to put data on the output bus and the time at which they actually do so. Adding to this are cable lengths and receiver propagation delays. To deal with the range of possible data delays, a DDU-39F digital delay unit allows fine tuning of the DAQ FIFO $\overline{\mathrm{W}}$ signal to provide the right setup time for any choice of test device, cable length, etc.

3.6.1 storing data: bit 9 select

The PATT02 board is designed to receive data from the chip in the form of 8-bit words, plus a ninth bit which can come from one of two user-selectable sources (typically, in tests of SVX2/3, this feature was used to store either Data Valid or Bottom Neighbor, as required by the test being run.) The bit 9 select control line (pin 9 on the 60-pin connector) is used to select which signal is stored.

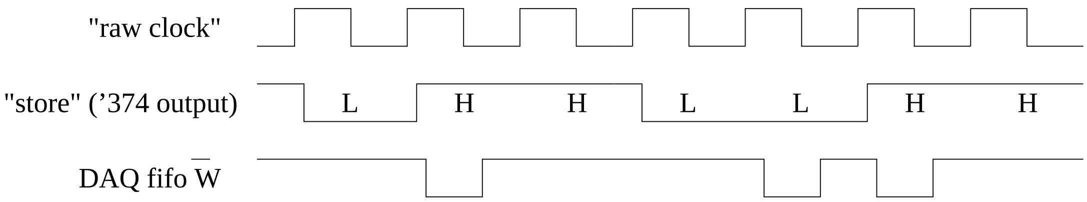

Figure 3.6: Timing diagram for STORE pattern line. The exact timing of the write pulses going to the DAQ FIFO can be fine-tuned by adjusting the DDU-39F delay unit.

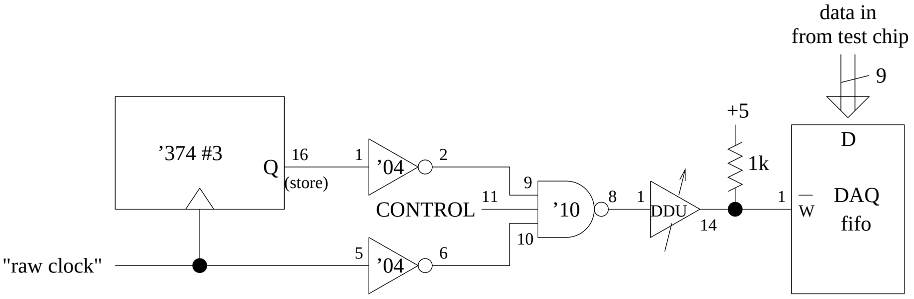

Figure 3.7: STORE pattern line circuitry.

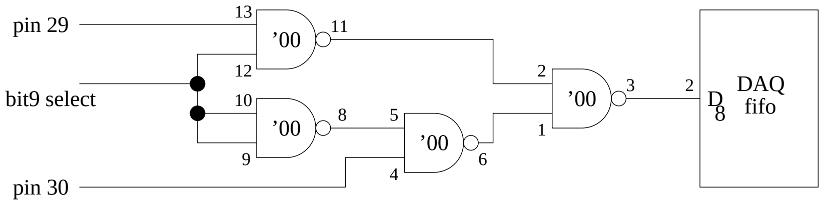

Figure 3.8: Bit 9 Select circuitry. Pins 29 and 30 refer to 34-pin input connector D.

3.7 Reading out the DAQ FIFO

After a typical test pattern has been run, data generated by the device under test may be sitting in the DAQ FIFO. These data need to be uploaded to the master computer for analysis. This is accomplished through connector B in figure 3.1.

The inputs and outputs on the 34-pin connector B are all differential. The data is strobed out of the DAQ FIFO using pins 3 and 4 (the "data out strobe".) This strobe is kept low initially. To read out one data word, the strobe is raised high, whereupon the next byte to be read appears at the outputs. The strobe is then lowered again. This procedure is repeated for each data word.

In order to tell when the DAQ fifo has been emptied of data, a 'FIFO empty' flag is provided as one of the outputs (pins 5 and 6). The empty flag signal is high while the DAQ FIFO still contains data, and goes low as the last data word in the FIFO is being read out.

3.8 A note on the IDT7204 FIFO

The IDT7204 is a first-in-first-out memory chip with asynchronous read/write, capable of storing 4096 9-bit words. Parts are available with a wide range of acces times, but the fastest version (12 ns access time, part number IDT7204L12TP) is required for 50 MHz operation. The timing information in the rest of this section applies to this 12ns access time part. A pin-compatible and functional equivalent IC that is slightly faster is made by Cypress (part number CY7C433-10PC). Deeper versions are also available from Integrated Device Technology, but tend to be available only with longer access times (IDT7205L15TP is 8192 bytes deep, with 15ns access time.)

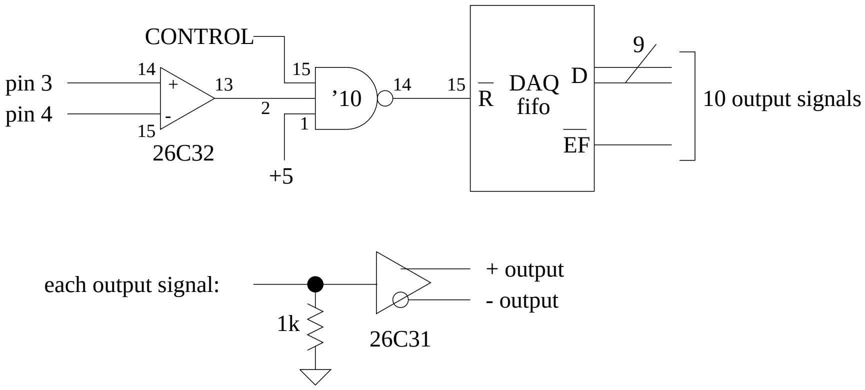

Figure 3.9: Circuit for readout of DAQ FIFO. Pins 3 and 4 refer to 34-pin connector B.

important pins: pin 1 (write enable), pin 15 (read enable), 21 (empty flag), 22 (master reset), 23 (retransmit).

writing data to the IDT7204: a setup time of 9 ns must be observed with respect to the rising edge of write enable to strobe data into the FIFO.

reading data from the IDT7204: data is guaranteed to appear 12 ns after the falling edge of read enable (in practice, data appears roughly 7 ns after the falling edge). Data is no longer guaranteed to be valid 5 ns after the rising edge of read enable. (in practice, the data persists for a longer period.)

empty flag: the empty flag signal goes low 12 ns after the falling edge of read enable corresponding to the last data word in the FIFO being read out.

resetting the IDT7204: a negative pulse of 12 ns resets the FIFO. This MUST be done after powerup and before first use.

retransmit: a negative pulse of 12 ns causes the FIFO to rewind to the beginning of its memory without affecting the stored data, so that the next read operation will cause the first word in memory to be output.

3.9 PATT02 component list

component	quantity
34-pin dual row, 100-mil center header	2
50-pin dual row, 100-mil center header	1
60-pin dual row, 100-mil center header	1
28-pin THINDIP socket	4
20-pin DIP socket	4
16-pin DIP socket	6
14-pin DIP socket	8
8-pin DIP socket	2
1k resistor	14
6.8 μF tantalum capacitor	3
0.1 μF bypass capacitor	21
IDT7204L12TP FIFO	4
74AC00 2-input NAND	2
74AC04 hex inverter	2
7408 2-input AND (any family)	1
74AC10 3-input NAND	1
74LS31 delay element	1
74AC74 D-type flip-flop	1
74AC374 octal D-type flip-flop	3
DS26C31 line driver	3
DS26C32 line receiver	1
DDU-39F digital delay unit	1
oscillator chip (half size, $\leq$ 50 MHz)	1

3.10 PATT02 board pinouts

(even numbered pins are ground unless otherwise noted)

Table 3.1: 60-pin input connector A

pin number	function
1	CONTROL
3	RUN
5	WRITE
7	RETRANSMIT
9	BIT 9 SELECT
11	patt. line 0 input
13	patt. line 1 input
...	...
$2n+11$	patt. line n input
...	...
51	patt. line 20
53	patt. line 21
55	patt. line 22 (STORE)
57,59	(unused)

Table 3.2: 34-pin output connector B. All signals are differential. If pin $2n-1$ contains signal, then pin $2n$ contains $\overline{\text{signal}}$.

pin number	function
1	(unused)
3	strobe input
5	DAQ FIFO $\overline{EF}$
7	bit 9 output
9	bit 8 output
11	bit 7 output
13	bit 6 output
15	bit 5 output
17	bit 4 output
19	bit 3 output
21	bit 2 output
23	bit 1 output
25-33	(unused)

Table 3.3: 50-pin output connector C. If pin 1 is pulled high then pattern outputs 0 through 7 will tristate.

pin number	function
1	tristate in
3	raw clock out
5	patt. line 0 out
7	patt. line 1 out
...	...
$2n+5$	patt. line n out
...	...
55	patt. line 20 out
57	patt. line 21 out
59	+5V output

Table 3.4: 34-pin input connector D

pin number	function
1-9	(unused)
11	bit 1 input
13	bit 2 input
15	bit 3 input
17	bit 4 input
19	bit 5 input
21	bit 6 input
23	bit 7 input
25	bit 8 input
27	(unused)
29	bit 9 input A
30	bit 9 input B
31	SCR. POWER output
33	+5V output

4 SVX3 scrambler

The SVX3 scrambler provides the necessary interface between the FIFO board and the SVX3 chip. The scrambler board provides drivers and receivers for the bus lines and generates the differential clock levels needed by the chip.

The scrambler board receives a pattern from the FIFO board through the 50-pin connector C and passes it on to the chip via a 64-pin connector. Data from the chip returns to the DAQ FIFO of the FIFO board via the 34-pin connector D. Power for the SVX3 chip (not for the scrambler) comes through a 16-pin connector. The scrambler board itself is powered by the PATT02 board through pin 49 of the 50-pin connector and through pins 1 and 3 of the 34-pin connector. Pinouts for these connectors can be found in section 4.3.

Many of the pattern lines from the PATT02 board are passed to the chip without any processing. These are pattern lines 3 to 11; their functions are listed in table 4.1.

Table 4.1: Pattern lines passed directly through scrambler to SVX3.

Channel number	Function
3	fe_mode
4	be_mode
5	TN
6	pa_reset
7	prd1
8	prd2
9	l1a
10	cal_sr
11	chmod

4.1 Front and Back End Clocks

The scrambler board provides two different methods of generating the differential clocks needed by the SVX3 chip from the single-ended CMOS output

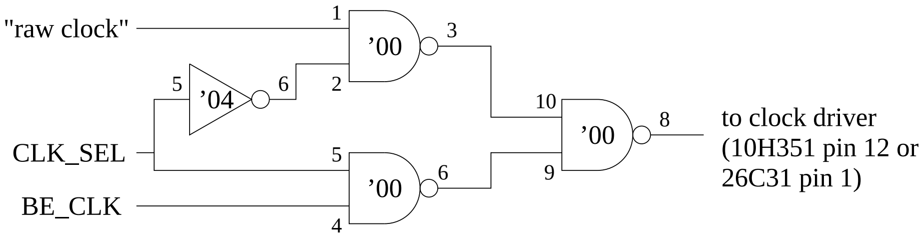

Figure 4.1: Back end clock select circuitry.

signals provided by the PATT02 board. One can use either a MC10H351 IC or a DS26C31 IC. These chips plug into sockets on the extreme right side of the board. The DS26C31 does not work at frequencies above 25 MHz, so the other chip is the one normally used; however, the DS26C31 provides a larger voltage swing (from 0.3 to 3.1 V, while the 10H351 swings between 3.2 and 4.1 V), and so is useful in certain situations. At any time one OR the other chip should be loaded, but never both.

One other feature of the scrambler board is its ability to output a maximally fast clock signal to the back end clock. This is useful, for example, during digitize mode, to make the Gray code counter run at its specification frequency. To send a fast back end clock, simply hold the CLK_SEL pattern line low. This causes the scrambler board to use the "raw clock" signal from the PATT02 board to generate the back end clock signal, rather than the BE_CLK pattern line. The "raw clock" signal completes a full clock cycle (i.e. high, then low) for each bit of the pattern being run; thus, a back end clock signal generated in this manner is twice as fast as the fastest possible clock that could be generated using the BE_CLK pattern line. (See figure 4.1.)

4.2 The Bus Lines

On SVX3, bus0 through bus3 are bidirectional; they need to be driven except during readout mode. A DS26C31 chip on the scrambler board drives these lines, and on-board logic looks at the mode lines, causing the driver to tri-state its outputs during readout mode (i.e. when fe_mode is high and be_mode is low—see figure 4.2)

The receivers used for the bus lines (and for data valid) are AT&T 41MF

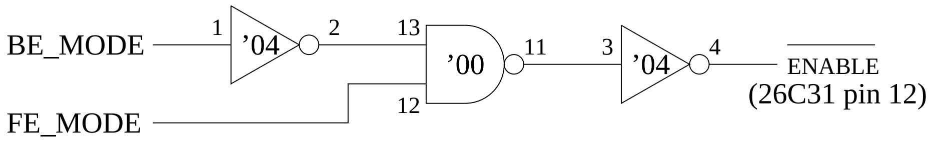

Figure 4.2: Bus0-3 driver tristate circuitry. (Note that pin 4 of the 26C31, ENABLE, is grounded, so $\overline{\text{ENABLE}}$, pin 12, solely controls whether the driver tristates.)

ICs (these being fast enough to work up to 50 MHz, unlike the 26C32.) Two 20-pin sockets in the center of the scrambler board allow for the placement of termination resistors. A common mode voltage is provided along one whole row of each socket, to provide a bias voltage for the bus lines if necessary. Two termination schemes we have used are differential termination (resistor between busN and busN*), which makes no use of the bias voltage, and termination of each bus line to the common mode voltage (resistor from busN to common-mode and from busN* to common-mode.)

4.3 SVX3 scrambler connector pinouts

(even numbered pins are connected to ground unless otherwise noted)

Table 4.2: 16-pin chip power connector

pin number	function
1	DVDD
3	unused
5	AVDD
7	AVDD2
9-15	unused

Table 4.3: 34-pin output connector to PATT02 board

pin number	function
1-9	(unused)
11	bus0
13	bus1
15	bus2
17	bus3
19	bus4
21	bus5
23	bus6
25	bus7
27	(unused)
29	BN
30	DV
31	+5 V
33	+5 V

Table 4.4: 64-pin output connector to SVX3 chip

pin number	function
1	AVDD2
3	ramp-ref[1]
5	ramp-ped[1]
7	Vcal
9-13	(unused)
15	AVDD
17-19	(unused)
21	TN
23	pa_reset
25	prd1
27	prd2
29	l1a
31	cal/sr
33	fe_mode
35	be_mode
37	ch_mode
39(40)	fe_clk(*)
41(42)	be_clk(*)
43(44)	DV(*)
45(46)	bus0(*)
47(48)	bus1(*)
49(50)	bus2(*)
51(52)	bus3(*)
53(54)	bus4(*)
55(56)	bus5(*)
57(58)	bus6(*)
59(60)	bus7(*)
61	BN
63	DVDD

[1]Only used by first version of SVX3 BE chip.

Table 4.5: 50-pin input connector from PATT02 board

pin number	function	channel number
1	tristate out	N/A
3	raw clock	N/A
5	fe_clk	0
7	clk_sel	1
9	be_clk	2
11	fe_mode	3
13	be_mode	4
15	TN	5
17	pa_reset	6
19	prd1	7
21	prd2	8
23	l1a	9
25	cal_sr	10
27	chmod	11
29	bus0	12
31	bus1	13
33	bus2	14
35	bus3	15
37	(trig)	16
39	(spare)	17
41	(spare)	18
43	(spare)	19
45	(spare)	20
47	(spare)	21
49	+5 V power	N/A

Part III
Software

5 Introduction

Functions (the C equivalent of FORTRAN subroutines) responsible for operating the automatic probe station, stimulating die on wafers, and collecting and manipulating data are divided into various packages of functions that a high-level user program can call. There is no "master program" that the user can run to probe a wafer or test a chip. Rather, the user must write individual programs for each specific application. Each program can then take the form of a UNIX command. Section *** contains detailed descriptions of currently available general purpose programs, which will also be useful as tutorial examples. Each of these programs consists basically of a succession of calls to functions in the various software packages.

5.1 The Packages

Package	Library	Include	Purpose
PATT	libpatt.a	patt.h	"Pattern generator" to stimulate die under test as well as data acquisition system to use on die under test. Requires PATT02 board hardware interface.
DATA	libio.a	io.h	Store and retrieve test data to/from disk files in binary format.
LOG	libio.a	io.h	Handle status, warning, and error message. Store in log file. Allows user program to automatically generate log files.
WAFER	libwafer.a	wafer.h	Mechanical manipulation of wafer on automatic probe station.
CAMAC MODULES	libmodules.a libcamac.a	camac_modules.h camac_sjc.h	Control of specific CAMAC mounted hardware units.

6 PATT package

6.1 Introduction

An electrical logic pattern is the language that the chips under test understands. Such a pattern can be represented by a matrix of 1's and 0's. It would therefore be quite natural to have ASCII files of 1's and 0's on a computer disk, each file representing a different chip control pattern. To exercise a chip under test in a particual way one would then input the name of the appropriate file to a control program.

file → control pattern

But generating and modifying these files would be an unpleasant task, and different files would be needed for tasks that are similar but not identical. The

PATT package provides a pattern management environment to allow for efficient creation of complex control patterns from a set of user-defined building blocks.

In the PATT package there are ASCII files containing information to control chips under test, but the intention is for each file to contain a small but elementary fragment of a pattern, analogous to a word or frequently used collection of words in language. A program can then combine the small set of fragments in many different ways to achieve a wide range of chip operations. The combination of fragments that will ultimately become the full control pattern for the chip under test is called the "output pattern", while the fragments that can be used to fill the output pattern are called "bitstream sets". It is perfectly possible for a bitstream set to be an entire control pattern, but normally several bitstream sets will be combined (within a program) to compose a control pattern.

$bs1 + bs2 + ... + bsn \rightarrow$ output pattern $\rightarrow$ control pattern

The ASCII data files of the PATT package are representations of bitstram sets. One file one bitstream set. However, once a file is read in by a program a bitstream set is created as an object in memory (with a more compact format), so that any given file never has to be read in more than once during one execution. This apparent complication has the purpose of minimizing I/O to speed up wafer probing.

The following definitions will be useful:

bitstream a stream of bits of any length. For example, 010001110.

bitstream set a set of any number of bitstreams, normally all of the same length. for example:
stream a = 110011011110
stream b = 000000000000
stream c = 000010011111
In the PATT package any number of bitstream sets can be stored in memory at any time. Each bitstream set is identified by a name. For example, the above example set could be called "example1".

.pat files A .pat file (eg. "example1.pat") is a representation of a bitstream set using ASCII characters. A detailed description is given in section 6.3.2. .Pat files can be read in, resulting in the creation of a bitstream set in the computer memory with the same name as the file (minus the extension).

.ini files An .ini file (ex. "example2.ini") is a description of a single bitstream which can be interpreted by the PATT package. The result is a bitstream set (with additional bitstreams generated automatically) with the name of the file minus the extension. See section 6.3.4 for more details.

output pattern The control pattern to be sent to the chip under test. This pattern is itself a set of bitstreams. The number of bitstreams in the output pattern is given by a constant defined in patt.h, which must correspond to the number of physical output channels of the hardware interface being used. Also the maximum allowed length of the output pattern cannot exceed the capacity of the FIFO memories on the hardware interface. When running a program, there is a single matrix of bits in the computer memory holding the output pattern. This pattern can be re-programmed whenever desired by adding bitstream sets.

6.2 Include File "patt.h"

This file declares some constants defining parameters of the PATT package, such as the maximum number of bitstream sets allowed in memory, or the size of the output pattern. It may be necessary to make changes to this file to use more memory, or a different hardware interface.

6.3 Input files to the PATT package

The PATT package gets all its bitstream data from ASCII files. There are three kinds of files, differentiated by their extension:

Extension	File type
.chn	channel definition
.pat	pattern
.ini	intialization stream

The following rules apply to the text in all these files:

1. Case does not matter, i.e. "clock" is the same as "CLOCK" or "cLoCk".
2. Blank lines are ignored.
3. An exclamation point "!" indicates everything following it on on that line is a comment.

6.3.1 Channel Definition Files

These contain an assignment of channel names to bitstream numbers of the output pattern. A given bitstream number is allowed to have many names, but any given name can only identify one number. This assignment is necessary, as pattern files can only identify channels by name, whereas the output pattern is simply a matrix, with the top row being bitstream 0, the row below bitstream 1, etc. The number assigned to a given channel name directly determines wat pin on the PATT board the associated bitstream will be sent to. (See table 3.1.)

Example file:

```
clock     1
strobe    1
signal    0
RESET     2
load      3
input-A   4
in-A      4
in1       4
```

6.3.2 Pattern Files:

A pattern file represents the contents of a bitstream as follows:

```
name:  XXXXX_X_______XX_________XXXXX____XXX____X_X_X_X_X_
```

Spaces are ignored. Everything before the colon is considered the channel name and will be matched to a name previously defined through a .chn file. After the colon the underscore character represents logic 0 while any other non-blank character represents logic 1. The bitstream cannot wrap onto the next line. For a longer bitstream than fits on a line the name must be repeated:

```
name:  XXXXX_X_______XX_________XXXXX____XXX____X_X_X_X_X_
name:    ________________X_______________X_____
```

Bitstream data for the same channel name need not be consecutive. For example, the following:

```
clock:   _1_2_3_4_5_6_7_8_9_0
signal:  HHHHHHHHH___________
in-A:    ____________________

clock:   _1_2_3_4_5_6_7_8_9_0
signal:  ___________QQ_______
in-A:    _______MMMMMMMMM____
```

is equivalent to:

```
clock:   _1_2_3_4_5_6_7_8_9_0
clock:   _1_2_3_4_5_6_7_8_9_0

signal:  HHHHHHHHH___________
signal:  ___________QQ_______

in-A:    ____________________
in-A:    _______MMMMMMMMM____
```

and also to:

```
clock:   _1_2_3_4_5_6_7_8_9_0_1_2_3_4_5_6_7_8_9_0
signal:  HHHHHHHHH______________________QQ_______
in-A:    ___________________________MMMMMMMMM____
```

6.3.3 Special Signals

There are two signals that are never sent by the PATT02 board to a chip unser test, but rather are used to control functions of the PATT02 and scrambler boards. These signals are lines numeber 20, known as STORE, and line number 1 (depending on the scrambler board used) known as CLK_SEL (for clock select). The clock select signal controls a switch in the scrambler board that allows one to send the PATT02 board oscillator output as a digitization clock to the device under test, instead of the BE_CLK bitstream, when CLK_SEL is low. The STORE signal controls what is stored in the DAQ FIFO of the PATT02 board. For every cycle that STORE is low, the byte following that cycle at the PATT02 board DAQ input will be saved. See the hardware section for more details on these two controls.

Additional signals can be included in a pattern to run synchronous devices other than the chip under test. Typically a trigger signal is included in order to send to a scope trigger. Another example is a LASER signal to pulse a laser for charge injection into a silicon sensor.

6.3.4 Initialization files

Svx2/3 chips have many adjustable parameters which are downloaded into the chip during Initialize mode. The parameters, in binary notation, are clocked serially into a shift register within the chip, through the "TN" pad. They are then latched into a shadow register.

Initialization files allow initialization data to be presented in a format easy to read and to modify, rather than as the cryptic sequence of 1's and 0's which ends up being strobed into TN. Reading in of an initialization file automatically generates a bitstream set in memory using the following channels (all of which must have been defined by a .chn file before an .ini file can be loaded):

Channel name	Purpose
ini_clock	clock line for strobing in of data
mode0 OR fe_mode mode1 OR be_mode chmod	place chip into Initialize mode
sr_load	signal chip to latch in ini data
TN	shift register input pad

The contents of the .ini file specify the bitstream for "TN". Each line of the file specifies the contents of one or more bits. The format of each line consists of a descriptor, always ending with a colon, followed by a value. Continuation of a line is not allowed; every non-comment line must start with a descriptor. Valid descriptors are:

1. bit descriptor, eg. "5:"

2. bit range descriptor, eg. "31-55:"

3. conversion descriptor, eg. "20>26:" or "44<53:"

The numbers in every descriptor refer to a bit number in the "TN" bitstream, the first bit being number 0. When two numbers are given in a descriptor, they indicate that all bits between and including them are being described. For example, "20>26:" refers to bits 20, 21, 22, 23, 24, 25 and 26. The first number in a range or conversion descriptor must ALWAYS be less than the second. A .ini file can have no holes, that is, all bits between 0 and the highest number appearing in a descriptor must be specified in the file.

Following a descriptor must be a value for the bits specified by the descriptor. A bit descriptor can be followed by either "1" or "0". A bit range descriptor is followed by a stream of the exact number of "1" or "0" values to fill the range specified. (Spaces are always ignored).

Conversion descriptors are followed by a decimal value, which is converted by the program to a string of bits filling the positions specified by the descriptor. If the descriptor has a ">" sign, the most significant bit will take the place of the first number in the descriptor (msb first), while if the "<" sign is used then the most significant bit goes in the position of the second number (lsb first).

```
example:
        31>35: 11      will produce bits 001011 in
                       positions 31 through 35.

        31<35: 11      will produce bits 110100 in
                       positions 31 through 35.
```

Keywords The decimal value after a conversion descriptor may be followed by a keyword. Current allowed keywords are "gray" (or "GRAY" or "Gray") and a name for the field being described, for example "pipe-depth".

The Gray keyword means the bits are to be encoded in Gray instead binary.

```
example:
        6>10: 5         will produce bits 00101

        6>10: 5 Gray    will produce bits 00111
```

The field name keyword has no immediate effect. However, function addini can be called with arguments specifying a field and a new decimal value for that field, in which case the given value will override the value in the .ini file. This is useful for stepping through an initialization file parameter in a program loop.

```
example:

 78>84: 10  pipe-depth
```

will produce bits 001010 when interpreted by the call addini(filename), but will produce the bits 101001 when interpreted by the call addini_modify(filename,"pipe-depth",41).

The following is an example of a .ini file:

```
0>3:    3       ! pipe-depth
4-9:    010110  ! test pulse mask
10<15:  49      ! ID code
16:     1       ! ramp polarity
17>22:  12 Gray ! counter limit
```

The resulting TN bit sequence will be 0011/010110/100011/1/1010. (The slashes indicate where one line of the .ini file ends and the next one begins.)

6.4 Usage of the PATT Package

A program that uses the patt package must have the following minimum content:

```
#include patt.h

begin program

  initialization block

  load bitstream sets into memory

  add desired bitstream sets to output pattern

  transfer output pattern to PATT02 board

  run PATT02 board to stimulate chip under test

end program
```

Each block in the above program will consist of a series of calls to various functions of the PATT package, and very likely also functions in the other packages (discussed later). For simplicity, the main functions of the PATT package are described below under headings corresponding to the blocks of the above program. This list does not include all the functions in the package, but rather the ones of most immediate utility to the non-expert user. But even expert users should have no need for functions not listed below under most circumstances.

6.4.1 Initialization Block

The initialization block must call the three PATT package functions below, in the given order. Additionally there will be functions from other packages (such as MODULES), called in the initialization block.

patt_reset() This function has no arguments and returns none. It simply clears some internal arrays and initializes some internal variables.

set_path(char *path) The argument of this function is a character string, for example "/home/cdfsg6/d1/mgs/patterns", in order to tell the PATT package where to find .chn, .pat, and .ini input files. Note that all files must be in the same directory (naturally one is free to use logical links).

chn_load(char *name) The argument of this function is a character string, for example "scrambler". In this example the file "scrambler.chn" would be loaded by the program. Note that no extension is given in the argument of chn_load. The file loaded must be a .chn file as defined in the previous section. This file is required, otherwise the PATT package will not understand .pat and .ini files.

6.4.2 Loading and Adding Bitstream Sets

Almost excusively one will want to load a bitstream set only if it is not already in memory at the time of adding it to the output pattern. There are therefore functions that perform this conditional loading automatically, so that as far as the user is concerned the program block "load bistream sets into memory" can be ignored.

addpat(char *name) This function returns and integer value that is 0 if there are no errors. The argument is the name of a .pat file minus the extension. As far as the user is concerned the routine will add the bitstream set in the specified .pat file to the output pattern. More specifically, the .pat file will be read in only if the routine has not been called before with this argument (note that the .pat file is assumed to be in the directory given with set_path in the initialization block). Upon reading in the .pat file a bitstream set is created in memory to hold the information. this memory bitstream set is then added to the output pattern. If addpat had been called before with this argument, then no loading is performed, as the bitstream set created the first time around will still be in memory.

addini(char *name) This function returns and integer value that is 0 if there are no errors. The argument is the name of a .ini file minus the extension. As far as the user is concerned the routine will add a bitstream set corresponding to the specified .ini file to the output pattern. More specifically, the .ini file will be read in only if the routine has not been called before with this argument (as always .ini file is assumed to be in the directory given with set_path in the initialization block). Upon reading in the .ini file a bitstream set is created in memory with the information from the file in the "TN" bitstream, as well as clock and mode bitstreams generated automatically. this memory bitstream is then added to the output pattern. If addini had been called before with this argument, then no loading is performed, as the bitstream set created the first time around will still be in memory.

addini_modify(char *name, char*parameter, int value) This function performs the same operation as *addini* on the *name* argument, but then replaces the data for the given parameter name with the value passed as an argument. See the section on .ini files for more details. To change more than one parameter, use function *addini_change*.

bs_clear This function clears all bitstream sets stored in memory. The output pattern is not affected. It has no arguments and is of type *void*.

bs_delete(char *name) This function erases from memory only the bistream set given in the argument. Note that the *name* argument does not have an extension, and it makes no difference whether the bitsream set was generated from a .pat file or a .ini file. The function returns 0 if successful.

bs_print_index This function prints on the screen a list of names of all bitstream sets currently in memory.

bs_print_contents(int id) This function prints on the screen a matrix of 1's and 0's corresponding to the bitsteam set numbered *id*. Note that the argument is a number rather than the name of a bitstream set. The integer function bs_id(char *name) converts the name of a bitstram set into a number.

bs_pad_level(char *list) This function fills channels in the putput pattern up to the level of the longest channel currently contained. The argument is a list of channel names, or blank to indicate that all channels must be padded. The value used to fill each channel is the value of the last bit currently in that channel. For example, if the output pattern at some point contains:

```
ch0:00000000
ch1:00000000
ch2:11111111
ch3:01010101
ch4:0000000000
ch5:1111
ch6:1100
ch7:0000
ch8:
ch9:
```

then calling the function *bs_pad_level* with null argument will result in the following output pattern contents:

```
ch0:0000000000
ch1:0000000000
ch2:1111111111
ch3:0101010111
ch4:0000000000
ch5:1111111111
ch6:1100000000
ch7:0000000000
ch8:0000000000
ch9:0000000000
```

out_copy(char *nameA, char *nameB) This function takes the contents of the output pattern for the channel identified by *nameA* and uses them to overwrite the contents for channel *nameB*. Returns 0 if successful.

out_invert(char *name) This function performa a logic inversion of the output pattern contents for the channel name given. Returns 0 if successful.

6.4.3 Transferring Patterns to the PATT02 Board

A pattern cannot be sent to a chip under test directly from the computer memory. Once a program has finished composing an output pattern to exercise a chip (for example, a pattern could be made to initialize a chip, make the chip acquire data, digitze it, and read it out), the full pattern must be transferred to the PATT02 hardware interface before it can be used to stimulate the chip. This operation simply makes a copy of the pattern in the memory of the PATTO2 board. The following functions handle the output pattern as a single entity.

out_clear This function erases the contents of the output pattern.

out_print This function prints on the screen the contents of the output pattern as a matrix of 1's and 0's.

out_dump This function transfers the output pattern to the PATT02 board. The previous contents of the PATT02 board are destroyed. The output pattern in the computer memory is not changed.

out_reset_fifo This function clears all the memories of the PATT02 board.

6.4.4 Running the PATT02 Board

out_run This function causes the PATT02 board to present at it's pattern output connector the pattern data loaded in its memory at the frequency of the clock oscillator chip loaded on the board. The entire pattern will be output asynchronously. After the pattern has been output once, the board goes into a standby state, witing for the next *out_run* command.

6.4.5 running the PATT03 board

out_start

out_stop

out_daq

out_nodaq

7 Usage of the MODULES Package

The computer running the software communicates with the PATT02 board via interface modules, such as CAMAC modules or PC expansion cards. For each hardware module there is a soft module, which is a collection of functions with the name of the module in question, thus permitting the programmer to control the modules in an intuitive way. The details of CAMAC, SCSI, etc., are hidden from the user. For each module that a program will use, the user *must* tell the program where physically to find the module. These module location functions will be called in the initialization block of a program.

7.1 Initialiation Block Modules

7.1.1 outreg_address(int slot, int subaddress)

Tells the program the CAMAC slot and subaddress of the output register being used to transfer data to the PATT02 board. This must be called in order to use the PATT02 board.

7.1.2 outreg0_address(int slot, int subaddress)

Tells the program the CAMAC slot and subaddress of output register channel to be used for static logic signals, that can be individually changed by the user. These are useful for controlling power supplies, etc., but also necessary for the PATT02 board to operate.

7.1.3 out_set_controls(int control, int run, int retransmit, int B9S)

This routine tells the PATT package which bits of the output register channel being used for static signals have been connected to the control, run, retransmit and B9S inputs of the PATT02 board. It *must* be called if the PATT02 board is used.

7.1.4 inreg_address(int slot, int subaddress)

An input register can be used to read data back from the PATT02 board into the computer (see *inreg* function below). This functions locates a Kinetic Systems 2232 CAMAC input register. Different hardware modules will need to have their own software modules.

7.1.5 aps_address(int slot, int subaddress)

For wafer probing an Allessi REL-2500 semi-automatic proobe stattion is used, controlled through GPIB using a Kineic Systems 3388 CAMAC module. This function declares the CAMAC slot of the 3388 module, and assumes that the probe station is connected to it.

7.2 Other Modules

Other modules are used for wafer probing, automated calibration voltage setting, etc. Each module will have an address function that mus tbe called in the initialization block off the program.

7.3 The Output Register

The ouptput register is used mainly to send data to the PATT02 board, and for this purpose the *out_dump* function automatically controls the module. Sometimes, however, the user will want to use a logic output that ccan be controlled within a program, for example to automatically switch on power supplies. The following function is included for this purpose.

7.3.1 outreg0(int N_on, int *bits_on, int N_off, int *bits_off)

This funtion takes as inputs two arrays of integer and a value for the length of each array. The first array is to contain all output register bits (for the channel specified with *outreg0_addres*) tthat are to be set to the logicc high state, while the second array lists all bits that are to be set to logic low. To turn on bit number nine, for example, the following call would be used,

```
i = 9;
oureg0(1,&i,0,NULL);
```

7.4 The Input Register

When the PATT02 board runs a control pattern it has the capability for storing data retturning from the device under test by inclusion of an appropriate STORE signal bitstream in the pattern. Actually, since the STORE signal causes the DAQ FIFO memory to STORE a byte without prejudice towards its origin, one can simply route some patttern outputs to the PATT02 board DAQ inputs and, as a self-test, store the very data being sent out by the PATT02 board itself. But regardless of how data got into the DAQ FIFO, after a pattern has finished running it may be desireable to transfer the contents of the DAQ FIFO to the computer. An input register module can be set up to this. Note that there is no function within the PATT package that deals with data in the DAQ FIFO.

7.4.1 inreg(char *word, char *bit9)

This function controls a Kinetic Systems 3430 input register to transfer the ccontents of the DAQ FIFO into the two argument arrays. The entire contents of the FIFO are transferred whenever the routine is called, so the arrays *word* and *bit9* must be declared deep enough to hold the maximum number of bytes expected. The DAQ FIFO is 9 bits wide, and the first 8 bits are intergpreted as an 88-bit binary number that is put in *word*, while each element of *bit9* is filled with a 1 or 0 acccording to the ninth bit. The function is of type integer and returns the number of bytes actually transferred.

8 Usage of the DATA Package

The DATA pacckage allows the user to store to disk and read back the data returned by the *inreg* function. Data are stored in a binary format described later.

Each record of data is a self-contained unit, with all relevant information on origin and nature of the data in the record. For example, each test performed on a die in a wafer may store its data in a single record. Files are collections of records, but since all information about a given record is in the record itself, it does not matter for data integrity how one may choose to collect records into file (i.e. one file per wafer; 5 files per wafer; one file per test, each file with records from all wafers; it doesn't matter). Each time a die or daisy chain of die is read out the data recorder constitute one event. Any number of events can be stored in a single record. In addition, a record contains a header, which specifies identifying information about the events in the record, such as wafer and die number, time and date, etc.

The DATA package maintains a buffer in memory which has a header and an arbitrary number of events. When function *save_buffer* is called, a new record is created at the end of the selected output file and the contents of of the memory buffer are placed in that record. The memory buffer is not automatically erased, and one must therefore normally call *kill_buffer* after *save_buffer*. Save_buffer opens and closes the output file in every call, so the file will not be corrupted in case of a program crash. Also, switching between different output files is easily accomplished without having to remember to close old files. A typical program flow is as follows:

In Initialization Block and/or Beginning of Program Block

```
set_output_file("my_file");
set_header(wafer,die,test_number,clock_frequency,user_length,user);
```

In a DAQ Loop After Call to *inreg*

```
store_event(data,bit9,type,bit9_id,n_words);
```

After End of DAQ Loop

```
save_buffer();
        kill_buffer();
```

In testing a wafer, for example, the main program would set the output file name, wafer number and clock frequency of the header. The die number of the header would be set in a loop over all die, and each subroutine performing some test would set the test number, oblivious to the die or wafer numbers. Obvioulsy, set_header can be called many times before finally calling save_buffer to immortalize the memory buffer contents in the output data file. Similarly, a new event is added to the memory buffer every time store_event is called.

To access the data stored in a binary file, set_input_file is used to open the file and set the file pointer at the beginning of the file. Then, records are loaded into the memory buffer one by one. The routine next_from_file will load the record currently pointed to into memory, while file_search_forward will search sequentially through the input file for a record with particular header parameters and if found, load it. Once a record has been loaded into memory, its header data can be accessed using get_header. The number of events contained in the record is ascertained using buffer_size, and the data of each event is retrieved using get_event.

8.1 DATA Package Routines

All routines return 0 on success, unless otherwise noted. The header file *io.h* must be included to use these routines.

8.1.1 Buffer Handling Routines

set_header(int w, int d, int t, int MHz, int ulen, unsigned short *user) Changes values of header parameters in memory buffer. If a 0 is passed for any argument the previous value is unchanged. If no value is ever given other than 0 then a 0 value will be used (wafer=0 may be desired if not running on a wafer, for example).

Arguments:

w = wafer number 0-127
d = die number 0-255
t = test number 0-1024
MHz = clock frequency in MHz
ulen = length of array user 0-255

additional information needed to characterize the record can be stored in the user array part of the header. The most significant bit of each user word must be zero. If no user words need to be stored, set_header can be called with last argument NULL.

Return value is 1 if it is passed invalid parameters.

get_header(int *w, int *d, int *t, int *MHz, int *ulen, unsigned short *user, int *day, int *mon, int *year) Retrieves the header parameters of the record currently in the memory buffer. The caller must allocate a 250 element array of unsigned short's to store any user words. The number of user words will be returned in ulen, and the date the data was stored will be in mon, day, and year. The main header parameters are as in set_header.

Return value is 1 if the header parameters of the buffer have never been set.

store_event(char *data, char *bit9, char *type, int bit9id, unsigned long nwords) Appends a new event to the end of the current record in the memory buffer, dynamically expanding the size of the memory buffer if necessary. The data are stored in the format given later. nwords specifies the number of data words to write; hence, the arrays data, bit9, and type should each contain at least this number of data elements. bit9id is presumed to be constant throughout the event and so is not passed as an array; it can be 1 or 0.

Returns value 1 if it cannot allocate additional memory.

get_event(char *data, char *bit9, char *type, int *bit9id, unsigned long *nwords, int *event_num) Fetches the next event in the current buffer (starting with the first event the first time it is called). The number of data words is returned in nwords, and the data words themselves are copied into the arrays data, bit9, and type, and the variable bit9id. The event number is placed in event_num (the first event in the buffer being number 0). To get the first event of the buffer call first_event before get_event.

Returns value 1 if the buffer contains no more events.

buffer_size() Returns the number of events in the buffer. If the buffer is empty or invalid, then the function returns 0.

first_event() Causes the next call to get_event to fetch the first event (number 0) of the buffer.

kill_buffer() kill_buffer erases the memory buffer. Normally *kill_buffer* will be called after *store_event*.

8.1.2 File Output Functions

set_output_file(char *name) Sets the name of the output file (which is written to by save_buffer), but does not open it. (The routines which write to the output file open and close it automatically.)

Returns value 1 if the filename supplied is longer than 80 characters.

save_buffer() Appends a new record to the output file with the current contents of the memory buffer. The memory buffer is NOT erased after save_buffer. Therefore kill_buffer must usually be called after save_buffer.

Returns value 1 if the buffer is empty or invalid. 2 on an I/O error.

8.1.3 File Input Functions

set_input_file(char *name) Sets the name of the input file (which is read by next_from_file and file_search_forward), and attempts to open it. If there is currently an open input file, set_input_file will first close it.

Returns value 1 if the filename supplied is longer than 80 characters, 2 if it could not open the new input file. Note that the previously open input file will not be closed on an error.

next_from_file() Loads into the buffer a record from the input file starting at the current position of the input file pointer, overwriting the current contents of the buffer.

Returns value 1 if the end of the input file has been reached, 2 on an I/O or out-of-memory error.

rewind_input_file() Sets the input file pointer to the beginning of the file. This is useful, for instance, in cases where repeated uses of file_search_forward have advanced the input file pointer to the end of the file.

Returns value 1 if the input file is not currently open.

file_search_forward(int w, int d, int t, int MHz, int ulen, unsigned short *user) Searches sequentially through the input file, starting at the current file position, searching for a record with header parameters matching those passed to file_search_forward. If such a record is found, it is loaded into the buffer. If no such record is found, then the input file pointer is reset to its original position.

If any of the parameters passed to file_search_forward is 0, then any value of that parameter is considered to be a match.

Returns value 1 if no matching record was found, 2 if invalid header parameters were passed to it, 3 on an I/O error.

close_input_file() Closes the input file, if it is open.

Returns value 1 if there was no open input file.

8.2 Format of Binary Data Files

Data are stored in variable length records of 16 bit words. The meaning of each word in a record is as follows:

```
 lsb 1  2  3  4  5  6  7   0  1  2  3  4  5  6 msb
*************************************************** \
        * <-- header length ---> * <- version number >  1 * |
*************************************************** |
        * <----- record length including header ----->  0 * |
*************************************************** |
        * <-- die number ------> * <-test number lsb->  0 * |
*************************************************** |
        * <- month->  <- year -> * <-day of m.->  x  x  0 * |
*************************************************** |
        * <-- wafer number ----> * <tstmsb> <clk*5MHz>  0 * | HEADER
*************************************************** |
        * ############## 15 bit user word 1 ##########  0 * |
*************************************************** |
        * ############## 15 bit user word 2 ##########  0 * |
*************************************************** |
        * ############## 15 bit user word n ##########  0 *     /
***************************************************
        * <---- data word -----> * B9 id <typ> 0  x  x  0 * \
*************************************************** |
        * <---- data word -----> * B9 id <typ> 0  x  x  0 * | EVENT
*************************************************** |
        * <---- data word -----> * B9 id <typ> 0  x  x  0 * |  #1
*************************************************** |
        * <---- data word -----> * B9 id <typ> 0  x  x  0 * |
*************************************************** |
        * <---- data word -----> * B9 id <typ> 0  x  x  0 * /
***************************************************
        * <---- data word -----> * B9 id <typ> 1  x  x  0 * \
*************************************************** |
        * <---- data word -----> * B9 id <typ> 1  x  x  0 * |
*************************************************** | EVENT
        * <---- data word -----> * B9 id <typ> 1  x  x  0 * |
*************************************************** |  #2
```

```
        * <---- data word -----> * B9 id <typ> 1  x  x  0 * |
*************************************************** |
        * <---- data word -----> * B9 id <typ> 1  x  x  0 * |
*************************************************** |
        * <---- data word -----> * B9 id <typ> 1  x  x  0 * |
*************************************************** |
        * <---- data word -----> * B9 id <typ> 1  x  x  0 * /
***************************************************
        * <---- data word -----> * B9 id <typ> 0  x  x  0 * \
*************************************************** |  EVENT
        * <---- data word -----> * B9 id <typ> 0  x  x  0 * |
*************************************************** |   #3
        * <---- data word -----> * B9 id <typ> 0  x  x  0 * /
***************************************************
        * 0  0  0  0  0  0  0  0 * 0  0  0  0  0  0  0  1 * END OF RECORD
***************************************************

Key:
----
        version number: currently zero.
<tstmsb>= 3 most significant bits of test number. The test number
                   is 10 bits in all.
B9      = 9th data bit.
id      = identity code of 9th data bit.
typ     = type of data word (0=chip ID, 1=ch. number, 2=data in Gray
  code, 3=data in binary code).
bit following <typ> = same event flag. Separates adjacent events.
x       = spare.
```

9 The Log Package