Technical Report No. 410

Analog Test Board: Design and Operation

R. A. Montante

bobmon@cs.indiana.edu

Computer Science Department

Indiana University

Bloomington, Indiana

August 31, 1994

Contents

1 Introduction **4**

2 Physical Design **5**

2.1 Board Infrastructure 5
2.2 Inputs 6
2.2.1 Current-Drain Signals 7
2.2.2 Constant-Voltage Signals 7
2.3 Outputs 8
2.4 Timing 9
2.4.1 Triggering inputs and outputs 9
2.4.2 Single-pulse generation 9
2.5 Interface 9
2.5.1 Parallel Interface Board 10
2.5.2 Address Decode Board 13
2.5.3 Address Mapping 14
2.6 Test Circuit 14
2.7 Digital Test Circuits 15

3 ATB Control **16**

3.1 Interface Operations 16
3.1.1 Sending Data 17
3.1.2 Retrieving Values 20
3.2 ATB Device Operations 21
3.2.1 Load a Value into a DAC 21
3.2.2 Convert Test-Input Values 22
3.2.3 Trigger Test-Output Conversions 23
3.2.4 Read ADC Output Values 24
3.2.5 Initialize the ATB 25
3.3 Test Inputs and Outputs 25
3.3.1 Input Currents and Voltages 25
3.3.2 Test Output Readings 27
3.4 Testing and Data Collection 28
3.4.1 General Testing 28
3.4.2 A Sample Test of KLLA 29

4 Acknowledgements **29**

References **30**

List of Figures **31**

A ATB Control Software **45**
A.1 Library Routines 45
A.2 `ftntest` Testing Program 59
A.3 Utility Functions 88
A.4 Makefile 104

1 Introduction

The Analog Test Board, or ATB, is a device designed to exercise and test current-mode analog circuits by providing up to 38 independently adjustable inputs and capturing up to six separate outputs. It is controlled through a parallel port such as the printer port found on most "PC clone" personal computers. The inputs are generated by Digital-Analog Converter chips (DACs) which receive 12-bit digital values from the controller; the outputs are converted to 12-bit digital values by Analog-Digital Converters (ADCs) whose results are polled by the controller. Test inputs to the analog circuit can be produced asynchronously or synchronously, after setting them up one at a time; output values can also be captured synchronously or asynchronously, then sent to the controller one by one. The inputs are current drains, and the test-circuit outputs are assumed to be also.

The ATB was originally designed and built by Charles Daffinger, who used it to test his LL9 and LL10 analog logic circuits. The original design was controlled through an RS-232 serial interface connected to a dedicated port on a VAX minicomputer; the 1200bps communication rate through the interface limited testing speed. The author re-engineered the interface to use a parallel port compatible with the printer port on an 80386-based desktop computer, providing communication speeds on the order of 10 kilobytes per second along with the convenience and flexibility of a dedicated personal computer for board control. In the process some hardware bugs were detected and repaired. This new configuration is used by the author to test the LL9 and KLLA analog logic circuits and others, including a digital implementation of the LL9/KLLA logic. (The LL9/KLLA logic is described in [MBD90] and [Mil91]. A KLLA test is described in [MM93].)

This technical report is organized in a bottom-up manner. Section 2 report describes the physical design of the ATB, with an eye toward maintaining it in operating condition. Section 3 describes the algorithms and programming used to control the ATB; an optional appendix[1] lists the software control-routine library and a general-purpose program presently used in

[1]The appendix is 64 pages of C-language source listing; it is available on request .

Revised version, Copyright ©1994, R A Montante and Indiana University.

testing operations on the board. The organization makes sense for maintaining the ATB (or building a new one); if a "user-level" programming guide is desired it may be helpful to skim the report backwards first for a more top-down view of the ATB's operation.

2 Physical Design

The ATB consists of a two-foot-by-three-foot platform with components mounted on its surface. The platform is a wooden frame covered by a grounded aluminum sheet; the sheet provides an electrical ground plane and partial shielding from electromagnetic interference. On top of the aluminum sheet are a number of functional units, wired on circuit prototyping boards which are attached to the sheet, and two small "perf board" circuit boards which hold the parallel-interface circuitry. Figure 3 is a schematic diagram of the ATB. The functional blocks in Figure 3 are cross-referenced to circuit diagrams in other figures by Figure 2.

2.1 Board Infrastructure

Power. The various circuits on the ATB require 5-volt power and ground to drive TTL-level chips, and +15 volts and -15 volts to power analog-digital converters and support circuitry. These voltages are supplied by an external voltage-regulated power supply, and distributed over the board. They can be tapped into from many points on the functional-unit prototyping boards. The power-supply ground is connected to the aluminum ground-plane, which is available everywhere on the board. In the circuit schematics, the +15V supply is identified as "+Red", referring to the color of the supply wire. "-Black" wires distribute -15V; +5V for the TTL logic is routed through white wires, and Ground connections use green wires.

Controlled Voltages. Figure 5 shows two op-amp based circuits which provide reference voltages for the test output-measuring circuits. The 5V reference is used in converting output current drains to voltage levels. The 10V reference is supplied to the ADC chips. One prototyping board holds both circuits, as well as the Clock circuit and part of the Sync circuit.

A 10V reference signal is also supplied to the DAC chips. One copy of the circuit in Figure 7 generates $V\text{ref}_0$ for the 18 DACs numbered **0x00** through **0x11**. Another copy generates $V\text{ref}_1$ for DACs **0x12** through **0x23**, **0x44** and **0x45**. The circuits are mounted on spare areas of the prototyping boards that hold devices **0x45** and **0x09**.

Clock. A Clock circuit is shown in Figure 6. A 10MHz clock signal is divided down to 78.125KHz, then supplied at 10V to the ADC chips and at 5V to the Sync circuit which generates *Convert.L*[2] signals for the ADC chips.

Signal Distribution. The various devices on the ATB receive data from the controller (via the Parallel Interface Board) over a 12-bit input data bus composed of red wires, referred to as ID11 through ID0. The output devices return data to the Parallel Interface Board over a 12-bit output data bus composed of blue wires, referred to as OD11 through OD0. Device addresses are decoded into individual enable signals (identified as *Write.L* in Figures 11, 9, 8, and 10) which are sent over distinct wires from the Address Decoder Board to each device.

2.2 Inputs

The ATB generates 36 current-drain signals as inputs to an analog circuit to be tested, plus two settable voltage signals intended as test-reference voltages. Each signal is produced by latching a 12-bit value S_{in} from the common input data bus into an Analog Devices AD1208 DAC, and issuing a *Write.L* signal to that DAC. The DAC also receives a reference voltage $V\text{ref}$ (supplied by one of the circuits in Figure 7), and emits an output current that drives a TL082 op amp[3] to produce a stable voltage given by $-\frac{S_{\text{in}}}{4095} \cdot V\text{ref}$. The DAC and op amp are shown in both Figures 9 and 8.

The DACs are double-buffered, and can be controlled either to convert S_{in} immediately or to wait until a separate *Xfer.L* signal is received. When

[2]The notation *signal.L* refers to a controlling signal which is logically true when its electrical value is 0V. This notation is described in [PW87]; IC schematics often write such signals in the form $\overline{signal}$.

[3]Both 47pF and 100pF feedback capacitors are used with the op amps; the exact value is not critical.

a DAC is addressed, the Address Decode Board converts the address into a *Write.L* signal which enables the appropriate DAC to latch a new S_{in}. The constant-voltage DACs (see below) are hardwired to convert S_{in} immediately. The current-drain DACs share a common *Xfer.L* signal; when they receive it (from device **0x40**, see §2.4.1) they all convert their new values simultaneously. This makes it possible to load distinct S_{in} values to each of the current-drain DACs, and then present them all to the test circuit at the same time. Alternatively, *Xfer.L* can be reissued after each DAC (or subset of DACs) is loaded, to convert new values as they are sent.

2.2.1 Current-Drain Signals

As seen in Figure 3, there are two banks of 18 signal-DAC functional units that drain selectable amounts of current. The units have hexadecimal addresses **0x00** through **0x11** and **0x12** through **0x23**. Each unit consists of the circuit depicted in Figure 9, built on (half of) a prototyping board. The DAC/op amp output voltage, described above, is inverted through a $100K\Omega$ resistor to a positive voltage by a second op amp.[3] The positive voltage controls a 2N5486 JFET transistor which acts as a current drain. It will draw between $0\mu A$ and $100\mu A$ of current; the value is linearly determined by S_{in}. The JFET is connected to a test input by a shielded coaxial cable which provides noise immunity for rapidly changing test currents. The coax cables for all 36 current-drain input devices have matched impedances.

As stated above, these units latch a new value from the input data bus when they are addressed. They continue to present their previous current-drain signals until they receive the triggering *Xfer.L* signal.

2.2.2 Constant-Voltage Signals

Two signal-DAC functional units are configured to supply constant reference voltages to a circuit under test. Figure 8 shows the units, at addresses **0x44** and **0x45**. Device **0x45** is basically one of the current-drain units without the JFET: it inverts the DAC/op amp output through a second op amp[3] to produce a positive reference voltage, selectable between 0V and 10V based on S_{in}. As originally designed, device **0x44** supplied the negative DAC/op amp output voltage directly to a circuit under test as a negative reference voltage, selectable between 0V and -10V. Presently this output is

also routed through a second op amp, to provide another independent positive reference voltage. In effect both polarities of voltage are available from each unit, requiring only that the appropriate op amp's output be connected to the circuit under test. Coaxial cables identical to the current-drain cables are used. When addressed, these units latch a new value from the input data bus and immediately begin supplying the converted voltage.

2.3 Outputs

Six output functional units are provided, each consisting of the circuit shown in Figure 10 assembled on a prototyping board which is mounted to the aluminum plate. A shared signal causes conversion of up to six test outputs simultaneously; the results may then be retrieved by addressing each unit as desired. The units are addressed from **0xf0** through **0xf5**.

The test circuit's output (assumed to be a current drain of between $0\mu A$ and $100\mu A$) is connected through a shielded coaxial cable, identical to the input cables, to a pair of TL082 op amps which convert the signal to a voltage between 0V and 5V (dependent on the 5V reference circuit shown in Figure 5). This V_{test} voltage is supplied to an ADC1210 ADC chip.

A conversion is initiated by supplying a *Convert.L* pulse, which is gated by the ADC's $\overline{CC}$ output. (It is possible to supply the *Convert.L* pulse to any combination of the six ADCs simultaneously — see §2.4.1). If $\overline{CC}$ is True, the *Convert.L* pulse is latched to the ADC's $\overline{SC}$ input. When the ADC receives $\overline{CC}$, it begins converting the V_{test} to a 12-bit digital value V_{out} by comparing it to the reference provided by the 10V circuit of Figure 5. During the conversion the ADC holds its $\overline{CC}$ signal False. The conversion takes $100\mu sec$ (approximately 12 cycles of the 10V clock from Figure 6), after which the $\overline{CC}$ signal is set True. The 12-bit S_{out} is made available through $47K\Omega$ matching resistors to enabled 74HC245 transceivers.

S_{out} is read from any one of the units when the corresponding address is selected. It enables the appropriate 74HC245 transceivers which copy S_{out} from the ADC onto the output data bus.

An $\overline{SC}$ signal received in the midst of an ongoing conversion would corrupt the conversion; hence the use of $\overline{CC}$ to block *Convert.L* during the conversion time. If a constant *Convert.L* signal were supplied it would generate a train of successive conversions, which is not desirable for the ATB; however, the single-pulser circuitry shown in Figure 12 constrains *Convert.L* to be a single

pulse.

2.4 Timing

The input DACs can be controlled to present their test signals simultaneously even though they have to be loaded individually; the output ADCs can be triggered to convert one or more test outputs at the same time; and must be timed to avoid spurious conversions that would corrupt previously acquired readings.

2.4.1 Triggering inputs and outputs

Figure 11 shows four 74LS273 octal D-flipflops, which serve as control devices at addresses **0x40** through **0x43**. Each of the octal flipflops receives the low-order eight bits of the input data bus, and presents the bits as outputs when the chip is addressed.

When addressed, device **0x40** presents Bit 0 as the *Xfer.L* signal to all of the input DACs; the remaining bits are unconnected.

Device **0x42** presents the low-order six bits as *Sync* signals for each of the six ADCs. The value on the input data bus (1 through 63) determines what combination of ADCs are triggered to perform conversions. The Sync circuit described below converts each of these *Sync* signals to single-clock *Convert.L* pulses, as required by the ADCs.

Devices **0x41** and **0x43** are unused.

2.4.2 Single-pulse generation

Figure 12 shows a single-pulser circuit which is replicated six times. Each copy receives one of the *Sync* signals from device **0x42** and produces a one-clock-cycle *Convert.L* pulse at +10V, which signals the appropriate ADC to begin a conversion. Thus a conversion may be delayed as much as one clock cycle after device **0x42** is addressed (plus signal propagation delays).

2.5 Interface

The interfacing logic between the ATB and the controller is implemented on two "perf" boards mounted at two corners of the platform. They are

connected by a gray ribbon cable which carries address lines A0 through A7 and A18. Data values and device addresses are buffered, sent and received on the Parallel Interface Board. Addresses are decoded and distributed by the Address Decode Board. Input data values are distributed over the input data bus; output data values are collected from the output data bus.

2.5.1 Parallel Interface Board

The Parallel Interface Board communicates with the controller computer and distributes values to and from the functional units of the ATB. It makes use of the "Centronics printer" parallel port design of the early IBM PC desktop computers. This port design supplies eight Data bits which are *not* treated as bidirectional.[4] Four Control bits (three of which are inverted) are also received from the controller, and five Status bits (one inverted) can be returned to the controller. Through these bits, the interface must receive 12-bit data values to be supplied to the ATB units, and return 12-bit values received from the ADCs. It must also receive 8-bit addresses which are used to activate devices on the ATB, plus Control signals that govern operations. In a typical controller architecture the Data bits, Control bits, and Status bits are manipulated and referred to as the "Data Register", the "Control Register", and the "Status Register" respectively (see §3.1). Figure 15 maps these bits to the pins of a DB-25 parallel-port connector plug. Some of the Control and Status bits are inverted by the standard controller hardware; these inversions are compensated for in the controlling software.
Note — "Data" and "Control" in the context of the parallel port are distinct from "input data values" and "controlling signals" as used on the ATB itself. The ATB's "input data values" are transmitted to the ATB through the Data Register of the parallel port; the controlling signals are either from input data bus values or from addresses. The Control Register values described below only control the interfacing logic on the Parallel Interface Board.

[4] Many recent implementations of the PC parallel port design support bidirectional data bits, but IBM's original specification short-sightedly hardwired "write-only" functionality into the design.

Controller to ATB (input). Figure 13 is a schematic of the Parallel Interface Board. The eight Data bits are received and amplified by a 74LS244[5] octal line receiver which routes them to three 74LS374 enabled octal flipflops. The 74LS374s serve as buffers for the low-order eight bits of the ATB's input data bus, the high-order four bits of the input data bus, and eight device address bits. (The address bits are placed on the gray ribbon cable as lines A7 through A0.) The 74LS244 is permanently enabled, while a separately received Control value governs when (and whether) one of the 74LS374s actually latches in the Data bits.

Another 74LS244 receives the four Control bits; bit C_3 is routed to the Address Decode Board as line A18 of the gray address cable, while bits $C_2..C_0$ are passed to a 74LS139 enabled decoder. Bits C_2 and C_1 are decoded to form the enable signals for the 74LS374 flipflops, and C_0 enables the 74LS139 decoder. This allows the correct 74LS374-enable signal to be set up without experiencing glitches. Figure 1 lists the effect of the Control bits.

Getting a value from the parallel port. The general protocol for loading an 8-bit value into the ATB is:

1. Place the value on the Data bits of the parallel port (controller writes the value to its Data Register).

2. Select the desired receiving lines by writing a disabling value to the Control Register — for example, if the value is intended as the low-order bits of an ATB input, the appropriate four Control bits are {0101}.

3. Enable the 74LS139, and hence the desired 74LS374 latch, by writing the enabling form of the Control value to the Control Register — *viz.*, {0100}.

4. Disable the 74LS139 again by rewriting the Control value from Step 2 — *viz.*, {0101}.

[5]Some of the chips used, particularly transceivers, are actually members of the 74HC chip family rather than 74LS. Either family is acceptable on the ATB.

5. Deselect all latches by writing {0001} to the Control Register.

After Step 3 is completed, the value will be presented and held on the appropriate receiving lines of the ATB — high- or low-order input data bus, or address bus — until a new value is latched into that 74LS374.

If the loaded value is in fact a device address, then another Control value will be needed to make the Address Decode Board actually decode the address and issue the corresponding device *Write.L* signal. This Control value is bit C_3, which is routed directly to the Address Decode Board over line A18 of the gray address cable. After the desired address has been loaded onto the address cable (using the above protocol), two more steps pulse the *Write.L* signal:

6. Activate the Address Decode Board by writing {1001} to the Control Register.

7. Repeat Step 5.

Typically a device is activated by loading its address to the ATB and then immediately toggling bit C_3. In this case step 6 can be performed immediately after step 4, and the "Deselect all" value {0001} need only be sent once.

ATB to Controller (output). The Parallel Interface Board also sends output data values back to the controller through the parallel port. Since the original, basic form of the parallel port doesn't provide bidirectional Data bits, the output values are transmitted via the Status Register, four bits at a time. The output data bus lines are connected to two 74LS244 transceivers in three groups of four bits (along with four dummy bits). The 4-bit sections of the 74LS244s are individually enabled, and connected in parallel to the Status bits of the parallel port. As indicated in Figure 1, the output 74LS244s are selected and enabled by the same Control values that select the input 74LS374s, so every time a value is written to the ATB a value is also sent back to the controller's Status Register. It is up to the controller to ignore these values unless it wants them.

Sending a value over the Status lines. A complete 12-bit value must be sent four bits at a time. The protocol for returning a 4-bit value to the controller is somewhat simpler than the loading protocol. First the (12-bit) value must be written onto the output data bus by addressing one of the ADC output units. Four-bit portions of the value are then transmitted as follows:

1. Select the desired bits to receive by writing the appropriate Control Value to the Control Register — for example, to get the low-order four bits, write {0011}.

2. Read the Status Register (within the controller) and mask off the meaningless bits.

3. Repeat steps 1 and 2 for the remaining nybbles.

Note — Each output 74LS244 section can be enabled by four distinct Control values, three of which will also do something else. For example, {0011}, {0010}, {1011}, and {1010} will all send bits OD3..OD0; but setting $C_0 = 0$ will also enable the input 74LS139 decoder, and setting $C_3 = 1$ will issue a *Write.L* to some device. Neither of these additional effects is likely to be desirable. Only the values {0011}, {0101}, and {0111} should be used ({0001} is innocuous, but unnecessary).

2.5.2 Address Decode Board

The Address Decode Board, shown in Figure 14, is basically a large, enabled demultiplexer. It receives eight address bits as lines A0 through A7 and one Control bit as line A18, all from the Parallel Interface Board. Three 74LS244 octal transceivers (one unused) distribute the bits to ten 74LS138 demultiplexers that issue the *Write.L* signals, and one 74LS138 that selects the *Write.L* 74LS138s. Some addresses require an enabling Control value in order to generate a *Write.L* signal, while others generate *Write.L* immediately.

The low-order bits A2..A0 are routed to the *Write.L* 74LS138s, which are labeled in Figure 14 with their corresponding device addresses. Bits A5..A3 are decoded by the Select 74LS138, whose outputs enable the *Write.L* 74LS138s for the DACs at addresses **0x00**..**0x23**. The Select 74LS138 is itself

enabled by bit A18, in conjunction with the address bits $\overline{A7}$ and $\overline{A6}$. Thus the DACs at addresses **0x00**..**0x23** will receive a *Write.L* signal only when Control bit C_3 goes high *and* the high-order two address bits are $\{0\,0\}$.

The Trigger latches and reference DACs at addresses **0x40**..**0x45** get *Write.L* from a 74LS138 that is enabled by bits A18, $\overline{A7}$, and A6. These devices thus receive *Write.L* when Control bit C_3 goes high and the high-order address bits are $\{0\,1\}$.

The ADC transceivers' *Write.L* 74LS138 is enabled by A7 and $\overline{A3}$, so that it issues *Write.L* for addresses **0x80** through **0x87** as soon as the address appears, regardless of C_3. In fact, all addresses whose high-order bit is $\{1\}$ are accepted by this 74LS138; the ADC output units could be addressed as any of **0x80**..**0x85**, **0x90**..**0x95**, ..., **0xf0**..**0xf5** interchangeably. The use of addresses **0xf0**..**0xf5** is arbitrary. Another, unused 74LS138 is similarly enabled by A7 and A3 to issue *Write.L* for addresses **0x88** through **0x8f** and higher-numbered aliases.[6]

2.5.3 Address Mapping

An anomaly in wiring the connection between the between the Parallel Interface Board and the Address Decode Board resulted in swapping the address bits pair by pair: A0 and A1, A2 and A3, A4 and A5, A6 and A7. (Likewise, Control bit C_3 ended up on A18 instead of the intended A19.) Rather than rewire this once it was discovered, the anomaly has been corrected by mapping logical addresses to physical addresses in the software. This is described in §3.1.1.

2.6 Test Circuit

An unoccupied area is available in the center of the ATB. Typically the circuit to be tested is a DIP-mounted "chip" which is held on a prototyping board; the ATB's input and output functional units, which surround the area, are connected to the test circuit board through shielded coaxial cables. Ground connections are made directly to the aluminum plate which forms the floor

[6]The Address Decode Board's design supports the use of some additional addresses; however, additional addresses higher than **0x80** will require a redesign of the board, at which time the aliasing should be removed.

of the unoccupied area. Figure 3 depicts two separate test circuits, "KLLA" and "null", positioned within the area.

Figure 4 details a typical test circuit and its electrical connections to the ATB. The Cmir_1 analog current-mode circuit is part of the author's KLLA test chip ([MM93]). The circuit receives inputs from pins 11 and 12 and delivers its output to pin 9 of the KLLA chip. It also uses the chip-wide reference voltage and ground (supplied to pins 30 and 10), and the chip's pads all require a comparable voltage and ground (supplied to pins 5 and 35) for proper operation. This circuit will compute the Łukasiewicz implication function, a ramp-like function. An example of this circuit's output is plotted in Figure 17.

2.7 Digital Test Circuits

A digital circuit can be easily tested on the ATB, if no more than 12 bits of input are required; six sets of 12 simultaneously captured output bits can also be collected. The procedure is to simply jumper from the input data bus to the test circuit's inputs, bypassing the DAC units entirely. (However, the point of connection of a DAC unit to the input data bus is a convenient place to put the jumpers.) The DAC *Write.L* and *Xfer.L* signals are unnecessary. A lower-level interface control is needed to put the desired input bits onto the input data bus.

Alternatively, all of the DAC units can be used in the same way as devices **0x44** and **0x45**, by tapping the voltage outputs of the op amps as single-bit inputs. This approach provides 38 bits of input which must be set up individually; however, positive- and negative-voltage inputs are available, and intermediate voltage levels are available for exploring low-voltage devices and non-saturated behaviors.

Output data bits can be similarly jumpered to the inputs of an ADC unit's 74HC245 transceivers, and the appropriate ADC addressed to read the output bits back to the controller. The ADC *Convert.L* signal is not required, since the ADC chip isn't involved. Up to 12 bits can be read simultaneously, and up to six different sets of 12 can be captured.

3 ATB Control

The ATB exercises a test circuit by providing input signals and recording output signals under the control of a computer equipped with a "Centronics" style parallel port. This section describes the control algorithms and software as implemented on an MS-DOS desktop computer based on an 80386 CPU with an 80387 floating-point coprocessor.[7] The machine in use has a parallel port with Data, Status, and Control Registers at I/O addresses 0x0378, 0x379, and 0x37a ($0378, $0379, $037A in Intel notation). Values are sent to the ATB by writing the Data and Control Registers, and received from the ATB by reading the Status Register. The software described here is written in Turbo C version 2.0, an ANSI-standard implementation of the C language with extensions suited to the Intel 8086 architecture. In particular, Turbo C offers two port-I/O functions:[8]

`void outportb(char Byte, char Address)`
— writes an 8-bit byte to an I/O register

`char inportb(char Address)`
— returns an 8-bit value from an I/O register

The controlling algorithms are described in bottom-up order, from the lowest-level hardware control operations up to overall testing procedures. Algorithms are expressed as functions in ANSI-C notation. Appendix A.1 lists the actual library subroutines used by the author to control the ATB, and Appendix A.2 shows a testing program that uses the library.[9]

3.1 Interface Operations

The fundamental operations that the controlling computer can perform are to place values on the ATB's input data bus, generate device-enabling *Write.L*

[7] The math coprocessor is not required, but speeds up some of the data manipulation substantially.

[8] Other languages may include comparable instructions. The underlying assembly-language instructions are `OUT` and `IN`.

[9] The original code has been debugged with much effort and is known to work; but with hindsight it seems rather like "a twisty maze of little passages, all different". The examples used in the text should be rather clearer, and might be preferable in actual use.

signals, and read values from the ATB's output data bus. These operations involve the Parallel Interface Board's buffering hardware, which is operated by the computer via the Control Register bits. Typically the Data Register is located at I/O Port address 0x0378; other common addresses are 0x0278 and 0x03bc. The Status Register's address is [Data register]+1; the Control Register's address is [Data Register]+2. As indicated in §2.5.1 and Figure 15, three of the Control Register bits are inverted relative to the electrical signals sent out. An "exclusive-OR" mask in the code below does the bitwise inversion of program values that are written to the Control Register.

3.1.1 Sending Data

Two kinds of data are sent to the ATB: 12-bit values for the input data bus, and 8-bit addresses. 12-bit values are transmitted as an 8-bit lower chunk and a 4-bit upper chunk, sent in either order.s[10] A data-value chunk is sent by placing it on the Data lines of the parallel port and instructing a latch on the Parallel Interface Board to accept the chunk, using the Control Register values listed in Figure 1. A complete 12-bit value takes two steps. A C function that puts a 12-bit value on the input data bus is:

```
/* I/O Register addresses: */
#define D_Reg        0x0378    /* Data-Register       */
#define C_Reg        0x037a    /* Control-Register    */
#define CTRL_X_MASK  0x0b      /* fixes inverted bits */
#define CTRL_null    0x01      /* Nothing enabled     */

void send_12bits(unsigned char Upper, unsigned char Lower)
{
    /* ID11..ID8 onto data bus */
    outportb(D_Reg, Upper);
    /* Select high-chunk latch */
    outportb(C_Reg, 0x03 ^ CTRL_X_MASK);
    /* Select latch, enable '139 */
    outportb(C_Reg, 0x02 ^ CTRL_X_MASK);
```

[10] Partial values can also be sent, changing only some of the bits on the input data bus. Doing this would require care to keep track of what value is actually present, since there is no way to read the input data bus.

```
    /* Drop enable bit again */
    outportb(C_Reg, 0x03 ^ CTRL_X_MASK);

    /* ID07..ID0 onto data bus */
    outportb(D_Reg, Lower);
    /* Select low-chunk latch */
    outportb(C_Reg, 0x05 ^ CTRL_X_MASK);
    /* Select latch, enable '139 */
    outportb(C_Reg, 0x04 ^ CTRL_X_MASK);
    /* Drop enable bit again */
    outportb(C_Reg, 0x05 ^ CTRL_X_MASK);

    /* Turn everything off */
    outportb(C_Reg, CTRL_null ^ CTRL_X_MASK);
}
```

An address is sent similarly, substituting `0x07` and `0x06` (the Address-latch "address") for `0x03`/`0x02` or `0x05`/`0x04`. The ATB device addresses **0x00..0x23** and **0x40..0x45** also require an address-enable command via bit 3 of the Control Register. As mentioned in §2.5.1, this can be sent immediately after the address is latched in, like so:

```
void send_DAC_address(unsigned char Addr)
{
    /* A7..A0 onto data bus */
    outportb(D_Reg, Addr);
    /* Select address latch */
    outportb(C_Reg, 0x07 ^ CTRL_X_MASK);
    /* Select latch, enable '139 */
    outportb(C_Reg, 0x06 ^ CTRL_X_MASK);
    /* Drop enable bit again */
    outportb(C_Reg, 0x07 ^ CTRL_X_MASK);

    /* enable DAC addresses with 0000_1001, i.e. bit C_3  */
    outportb(C_Reg, 0x09 ^ CTRL_X_MASK);

    /* Turn everything off */
    outportb(C_Reg, CTRL_null ^ CTRL_X_MASK);
```

```
}
```

The code in Appendix A.1 is slightly more layered — it provides a lowest-level `latch_in()` function that latches a byte to the Parallel Interface Board, and `set_data()` and `set_addr()` functions that make use of it to send 12-bit input data values and 8-bit addresses.

Generating addresses. As mentioned in §2.5.3 the logical device addresses must be mapped to the physical addresses that the ATB interface logic sees, to compensate for a wiring error: address bits get swapped in a pairwise fashion. A function that fixes this is:

```
unsigned char physical_address(unsigned char Logical_Address)
{
    int i;
    unsigned char Physical_Address, b1, b0;
    Physical_Address = 0;
    for (i = 0; i <= 7; ++i) {
        /*
        | Pick off the 2 highest-order bits, and shift the
        | address left in preparation for the next loop.
        */
        b0 = 0x80 & Logical_Address;
        b1 = 0x40 & Logical_Address;
        Logical_Address <<= 2;

        /*
        | Shift the bits down to low order, swapping them in
        | the process, and "OR" them onto the mapped address.
        */
        Physical_Address = (Physical_Address << 2)
                         |  (b0 >> 7)  |  (b1 >> 5);
    }
    return Physical_Address;
}
```

However, the library code in Appendix A.1 uses a faster and simpler solution — a logical-to-physical-address lookup table implemented as the array

`Addr_Map[256]`.[11] This is invoked in the low-level function `set_addr()`, so the address mapping is invisible to user-level testing programs and even to the rest of the library routines.

3.1.2 Retrieving Values

The output data bus of the ATB provides 12-bit values, which must be read through the Status Register in 4-bit nybbles. (A fourth nybble is available, and provides zeros for bits OD15..OD12.) The nybbles may be read in any order, and must be shifted and concatenated to form a complete (16-bit) value. As with the Control Register, one of the Status Register bits is inverted in the hardware, so an "exclusive-OR" mask is used to adjust this. The following function reads the ATB and returns a 12-bit value as a 16-bit integer:

```
#define S_Reg          0x0379  /* Status-Reg. I/O address */
#define STATUS_X_MASK  0x80    /* fixes inverted bit 7 */

unsigned int receive_12bits(void)
{
    unsigned int Twelve, Four;

    /*
    | Latch bits OD3..OD0 from the output data
    | bus into the Status Register, read them,
    | & copy them into the variable 'Twelve'
    */
    outportb(C_Reg, (0x03 ^ CTRL_X_MASK));
    Four = inportb(S_Reg) ^ STATUS_X_MASK;
    Twelve = (Four & 0x00f0) >> 4;

    /*
    | Latch and read bits OD7..OD4,
    | & copy them into 'Twelve'
    */
```

[11] Depending on the programming language used, this can also be a shorter solution.

```
    outportb(C_Reg, (0x05 ^ CTRL_X_MASK));
    Four = inportb(S_Reg) ^ STATUS_X_MASK;
    Twelve = Twelve | (Four & 0x00f0);

    /*
    | Latch and read bits OD11..OD8,
    | & copy them into 'Twelve'
    */
    outportb(C_Reg, (0x07 ^ CTRL_X_MASK));
    Four = inportb(S_Reg) ^ STATUS_X_MASK;
    Twelve = Twelve | (Four & 0x00f0) << 4;

    return Twelve;
}
```

This function appears (in somewhat baroque form) as `ADC_bus()` in Appendix A.1.

3.2 ATB Device Operations

The basic operations that the ATB performs are:

- Load a value into a DAC input device.
- Convert input values to test-input current drains.
- Trigger test-output conversions.
- Read ADC output values.
- Initialize the ATB.

These actions are commanded by sending the appropriate data and addresses to the ATB using the algorithms defined in §3.1.

3.2.1 Load a Value into a DAC

This operation is straightforward: put the desired value onto the input data bus, then address the desired DAC to take the value. Values must be in the 12-bit range from `0x00..0xfff` (0..4095), and the valid DAC addresses are

0x00..**0x23** and **0x40**, **0x42**..**0x45**. (Two of these, **0x40** and **0x42**, aren't actually DACs, but they operate in the same way.)

```
void load_DAC(unsigned char DAC_addr, unsigned int Value)
{
    unsigned char Upper, Lower;
    Lower = (unsigned char)(Value & 0x00ff);
    Upper = (unsigned char)((Value & 0xff00) >> 8);
    send_12bits(Upper, Lower);
    send_DAC_address(DAC_addr);
}
```

3.2.2 Convert Test-Input Values

The test-current DACs at addresses **0x00**..**0x23** can operate in "double-buffered" or "flow-through" mode. In the normal double-buffer mode the DACs simultaneously convert the values they've been loaded with, only when they receive the shared *Xfer.L* signal from device **0x40**. Once they start converting, *Xfer.L* can be turned back off and they will maintain the conversion; meanwhile, a new value can be loaded in preparation for another *Xfer.L* conversion signal. Device **0x40** issues *Xfer.L* when it is loaded with the value `0x01`, and turns off *Xfer.L* when loaded with `0x00`:

```
void Xfer_on(void)
{
    load_DAC(0x40, 0x01);
}

void Xfer_off(void)
{
    load_DAC(0x40, 0x00);
}
```

In the alternative flow-through conversion mode, each DAC begins converting a new value as soon as the value is loaded, independently of the other DACs. Flow-through mode can be turned on by calling `Xfer_on()` initially and leaving it that way; flow-through mode will be turned off again whenever `Xfer_off()` is called.

3.2.3 Trigger Test-Output Conversions

The ADC output units convert the test-output currents they see to 12-bit values, when they get a *Convert.L* signal. Each ADC receives its own *Convert.L* from device **0x42**, which generates the signals from the low-order six bits of the ATB input data bus. Writing **0x42** with `0x011`, for example, generates *Convert.L* for ADC 5 (at address **0xf5**) and ADC 0 (at address **0xf0**). The *Convert.L* signals are triggered by a transition from "bit-Off" to "bit-On", so the appropriate bit in device **0x42** must start at 0 and change to 1. The following function assures this by forcing all bits to 0 initially. For each ADC to be triggered, the corresponding function argument should be set to 1. Untriggered ADCs should have their arguments set to 0.

```
void trigger_ADC(int adc0, int adc1, int adc2,
 int adc3, int adc4, int adc5)
{
    unsigned char adcs;

    /*
    | OR together all the desired ADC arguments:
    */
    adcs = ( ((adc5 != 0) << 5)
            |  ((adc4 != 0) << 4)
            |    ((adc3 != 0) << 3)
            |      ((adc2 != 0) << 2)
            |        ((adc1 != 0) << 1)
            |           (adc0 != 0)
           );

    /*
    | Turn off all Convert.L signals first, then
    | Trigger the desired ones:
    */
    load_DAC(0x42, 0x00);
    load_DAC(0x42, adcs);
}
```

The library function `ADC_sample()` in Appendix A.1 just accepts the 8-bit `adcs` value, and leaves it to the calling function to perform the multiple `OR` if more than one ADC is to be triggered simultaneously. This avoids the overhead for single-ADC operation.

3.2.4 Read ADC Output Values

Reading an ADC's converted output is a two-step operation: first command it to put its value on the ATB output data bus by addressing it, then read the output data bus through the Status Register. Unlike the DAC addresses, the ADC output unit addresses **0xf0**..0xf5 do not require the C_3 enable signal; on the other hand, the ADC's output is present on the output data bus only as long as it is being addressed, so addressing and ADC and reading its value must be a more-or-less atomic operation.

```
unsigned int read_ADC(unsigned char Addr)
{
    unsigned int ADC_val;
    /*
    | Send the ADC's address like other addresses,
    | but without the unneeded enabling Control bit:
    */
    outportb(D_Reg, Addr);
    outportb(C_Reg, 0x07 ^ CTRL_X_MASK);
    outportb(C_Reg, 0x06 ^ CTRL_X_MASK);
    outportb(C_Reg, 0x07 ^ CTRL_X_MASK);
    outportb(C_Reg, CTRL_null ^ CTRL_X_MASK);

    ADC_val = receive_12bits();

    /*
    | Disable the ADC output again.  (Not really necessary.)
    */
    outportb(D_Reg, 0x3f);
    outportb(C_Reg, 0x07 ^ CTRL_X_MASK);
    outportb(C_Reg, 0x06 ^ CTRL_X_MASK);
    outportb(C_Reg, 0x07 ^ CTRL_X_MASK);
    outportb(C_Reg, CTRL_null ^ CTRL_X_MASK);
```

```
    return ADC_val;
}
```

There is no exactly comparable library function. Instead `ADC_sample()` combines the ADC-triggering and -reading steps; the action of disabling the ADC output by erasing its address occurs implicitly when the ADC triggers are reset (by writing `0x00` to device **0x42**) at the end.

3.2.5 Initialize the ATB

This operation just brings the ATB back to a known, inactive state by writing null control values to the ATB devices and the Parallel Interface Board.

```
void reset_ATB(void)
{
    /* Disable all interface latches and DAC addresses. */
    outportb(C_Reg, 0x01 ^ CTRL_X_MASK);

    /* Turn off DAC conversions. */
    Xfer_off();

    /* Ready ADCs for new conversions. */
    load_DAC(0x42, 0x00);

    /* Disable interface latches and DAC addresses again. */
    outportb(C_Reg, 0x01 ^ CTRL_X_MASK);
}
```

3.3 Test Inputs and Outputs

3.3.1 Input Currents and Voltages

The ATB provides two kinds of test input: current drains in the range $0\mu A$ to $100\mu A$ from devices **0x00** through **0x23**, and voltages in the ranges $0V$ to $+10V$ and $0V$ to $-10V$ from devices **0x44** and **0x45**. All devices are set by a 12-bit data value, so the current drains and voltages are available in linear steps of 1/4095 from minimum to maximum. A testing program needs to do something like the following for the test-current DACs:

```
#define Max_Current 100.0

void set_current(unsigned char Addr, float Idrain)
{
    unsigned int Int_Current;
    if ( (Idrain < 0.0) || (Max_Current < Idrain) ) {
        /* Cope with Current-Out-of-Range Error */
    }
    if ( Addr > 0x23 ) {
        /* Cope with Invalid_DAC Error */
    }

    /*
    | Convert user units to an ATB value and send the value:
    */
    Int_Current = (unsigned int)
                  ( (float)0xfff * Idrain/Max_Current );
    load_DAC(DACnum, Int_Current);
}
```

After this is done for all desired DACs, the new values should be converted with an *Xfer.L* pulse:

```
void convert_new_values(void)
{
    Xfer_on();
    Xfer_off();
}
```

The test-voltage DACs at addresses **0x44** and **0x45** are hardwired for flow-through operation, so the *Xfer.L* pulse is unneeded. This function sends out a new reference voltage immediately:

```
#define Max_Volts 10.0
void set_volts(unsigned char Addr, float Volts)
{
    unsigned int Int_Volts;
    if ( (Volts < 0.0) || (Max_Volts < Volts) ) {
```

```
        /* Cope with Voltage-Out-of-Range Error */
    }
    if ( (Addr != 0x44) && (Addr != 0x45) ) {
        /* Cope with Invalid-DAC Error */
    }

    /*
    | Convert user units to an ATB value and send the value:
    */
    Int_Volts = (unsigned int)
                ( (float)0xfff * Volts/Max_Volts );
    load_DAC(Addr, Int_Volts);
}
```

Whether the resultant test voltage is positive or negative is determined by the way the voltage-input unit is connected to the test circuit (see §2.2.2 and Figure 8).

3.3.2 Test Output Readings

The `read_ADC()` function described above returns test outputs as 12-bit values representing a fraction of the range $0\mu A$ to $100\mu A$. A single ADC can be triggered and read as follows:

```
float sample_test_current(ADC_addr)
{
    unsigned char adc;
    float fraction;
    if ( (ADC_addr < 0xf0) || (0xf5 < ADC_addr) ) {
        /* Cope with Invalid-ADC Error */
    }

    /*
    | Determine the ADC-trigger bit, turn off all
    | the Convert.L signals, and trigger the
    | desired ADC to convert:
    */
    adc = 1 << (0x07 & ADC_addr);
```

```
    load_DAC(0x42, 0x00);
    load_DAC(0x42, adc);

    /*
    | Read the ADC output and put it in user units:
    */
    fraction = (float)read_ADC(ADC_addr) / (float)0xfff;
    return ( Max_Current * fraction );
}
```

For repeatability and precision studies the 12-bit values may be more important than the equivalent currents; so the library current-input and output routines use 12-bit values exclusively. If engineering units are important the calling program must do its own conversions. Test-input voltage routines are available that work in voltage units for convenience.

3.4 Testing and Data Collection

3.4.1 General Testing

The C-language program `ftntest`, listed in Appendix A.2, is a general-purpose testing program which uses the `atbpara` library. This program presents a menu of low- and middle-level commands for the ATB; the menu screen is shown in Figure 16. `ftntest` was originally written as a board-debugging tool, but creeping featurism has evolved it into a program that can exercise and collect data from test circuits as well.

`ftntest` uses the following functions or macros from the `atbpara` library:

`adc_bus()`

`adc_sample()`

`adc_strobe()`

`adjust_Vdd()`

`adjust_Vss()`

`set_addr()`

`set_data()`

`write_device()`

It also uses some functions that are unrelated to the ATB (and reflect idiosyncrasies of the author). Appendix A.3 shows these additional functions. To round things out, a "makefile" for the `ftntest.exe` program and `atbpara.lib` library is given in Appendix A.4. This makefile uses features of the Borland "make v3.5" program, which is slightly more flexible than the "make" which accompanies Turbo C v2.0.

3.4.2 A Sample Test of KLLA

A simple test of the KLLA analog chip illustrates the use of `ftntest`. The chip's I/O pads are powered by device **0x45**; the Cmir_1 subcircuit only requires two inputs, which have been connected to devices **0x08** and **0x09a**. As shown in Figure 16, the program's output is captured to an output file by the '>' function (which prompts for a file name, here "demo.out"). Subsequent actions are recorded in the file, and data collections are written out, in a format compatible with the `gnuplot` plotting program. The output file is shown in Figure 18, and records the following actions.

The 'V' function sets device **0x45** to 10V to provide I/O power (the -Vss value is unimportant for this test). The 'w' functon is used to set the DAC-**0x08** input to three current levels (high, low, medium); at each level the 'l' function is used to sweep the DAC-**0x09** input over a range of values and record the output from ADC **0x04**.

Figure 17 shows the three output curves, plotted by supplying the output file to `gnuplot`.

4 Acknowledgements

Charles Daffinger conceived, designed, and built the Analog Test Board for his testing purposes. William Hunt traced the wiring and circuits shown in the circuit diagrams, and instructed the author in the parallel interface design. Professor J.W. Mills provided encouragement and motivation to us all, not to mention funding for the materials and time that went into the board. William Hunt and "TJ" Jones were very helpful in obtaining first an 8088-based IBM PC and later an 80386-based personal computer to operate it.

References

[MBD90] Jonathan W. Mills, Gordon Beavers, and Charles A. Daffinger. łukasiewicz logic arrays. Technical Report 296, Indiana University, Bloomington, IN, March 1990.

[Mil91] Jonathan W. Mills. Area-efficient implication circuits for very dense łukasiewicz logic arrays. Technical Report 339, Indiana University, Bloomington, IN, November 1991.

[MM93] R. A. Montante and J. W. Mills. Probabilistic error correction in arbitrarily large łukasiewicz logic arrays. Technical Report 377, Indiana University, Bloomington, IN, April 1993.

[PW87] Franklin . Prosser and David Winkel. *the Art of Digital Design.* P T R Prentice Hall, Englewood Cliffs, New Jersey, 2nd edition, 1987.

List of Figures

1 Meanings of Control bits 32
2 ATB schematic/Circuit Diagram Cross References 32
3 Analog Test Board Schematic 33
4 Connecting an analog circuit to the ATB 34
5 ADC reference voltages 35
6 ADC Clock circuit 35
7 DAC reference voltages 36
8 Test-circuit Reference Voltages 36
9 Test-circuit Inputs 37
10 Test-circuit Outputs 37
11 Triggering devices 38
12 Conversion Synchronization 38
13 Parallel Interface Board 39
14 Address Decode Board 40
15 Parallel-port pinouts 41
16 `ftntest` Command Menu 42
17 Plotted KLLA response 43
(caption for Figure 18) 43
18 `ftntest`'s Output File 44

Control bits				Input Meaning:		Output Meaning:
C_3	C_2	C_1	C_0	Latch Select	Latch Enable?	Transceiver Select
x	0	0	1	none	Disable	(OD15..OD12)
x	0	1	1	ID11..ID8	Disable	OD3..OD0
x	1	0	1	ID7..ID0	Disable	OD7..OD4
x	1	1	1	A7..A0	Disable	OD11..OD8
x	0	0	0	none	(Enable)	(OD15..OD12)
x	0	1	0	ID11..ID8	Enable	OD3..OD0
x	1	0	0	ID7..ID0	Enable	OD7..OD4
x	1	1	0	A7..A0	Enable	OD11..OD8
0	x	x	x	Disable *Write.L*		-
1	x	x	x	Enable *Write.L*		-

Figure 1: Meanings of Control bits

'x' means "don't care". The C_3 bit has no effect on the latches or transceivers, but should be held at '0' when sending or receiving Data values to avoid writing to a device inadvertently. Bits $C_2..C_0$ similarly have no effect on the *Write.L* signals. OD15..OD12 are hardwired to 0.

Block label in Figure 3:	Detailed in...
KLLA, null	Figure 4
0x00, **0x01**, ..., **0x23**	Figure 9
0xf0 .. **0xf5**	Figure 10
0x40 .. **0x43**	Figure 11
0x44, **0x45**	Figure 8
Vref0, Vref1	Figure 7
ADCref	Figure 5
Clock	Figure 6
Sync	Figure 12
Parallel Interface	Figure 13
Address Decode	Figure 14
Power, ID11..ID0, OD11..OD0	none

Figure 2: ATB schematic/Circuit Diagram Cross References

The ATB schematic in Figure 3 shows functional units which are detailed in the other figures as listed here.

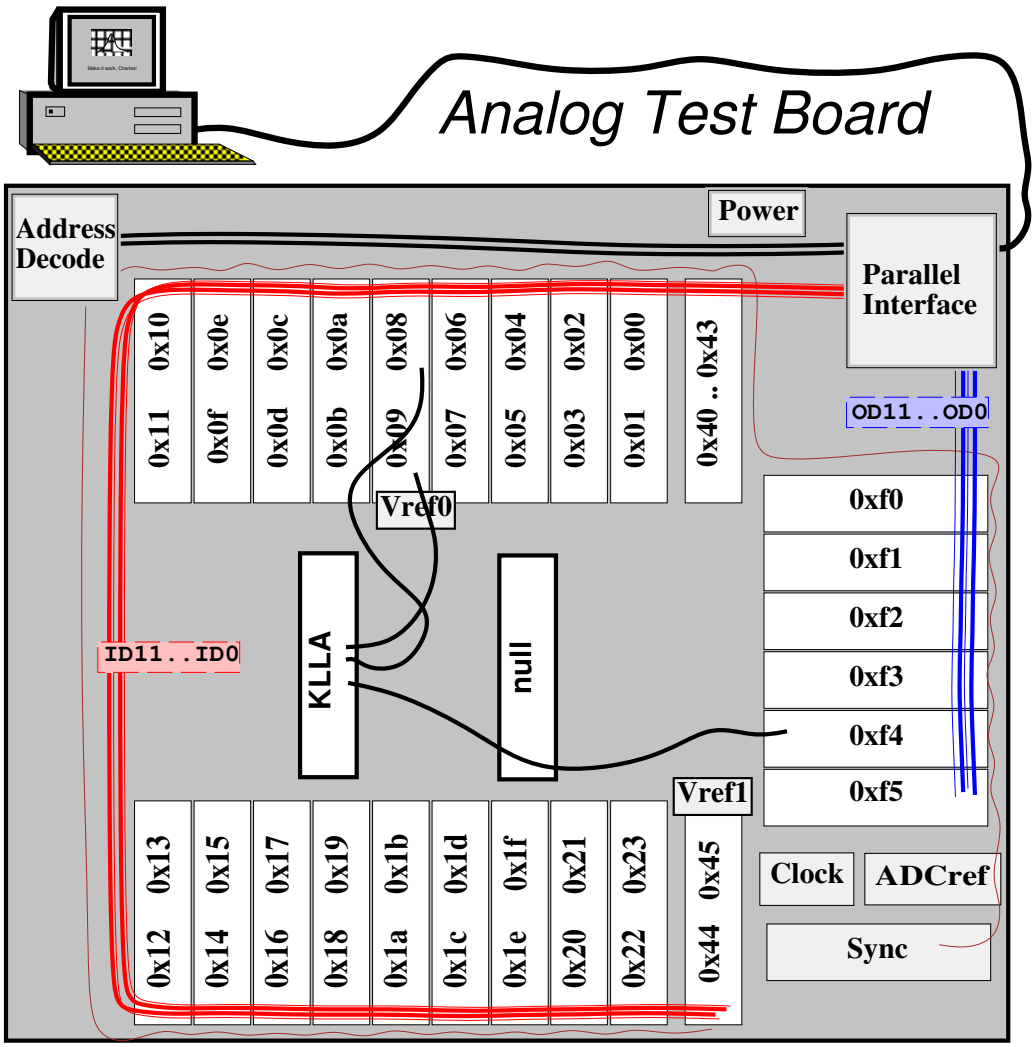

Figure 3: Analog Test Board Schematic
Major functional units on the ATB. Controllable devices are labeled with their addresses in hexadecimal. "Power" identifies the external power-supply connector. "KLLA" and "null" represent typical test circuits, with some connections to KLLA shown. Functional blocks are detailed in following figures; see Figure 2 for cross-references.

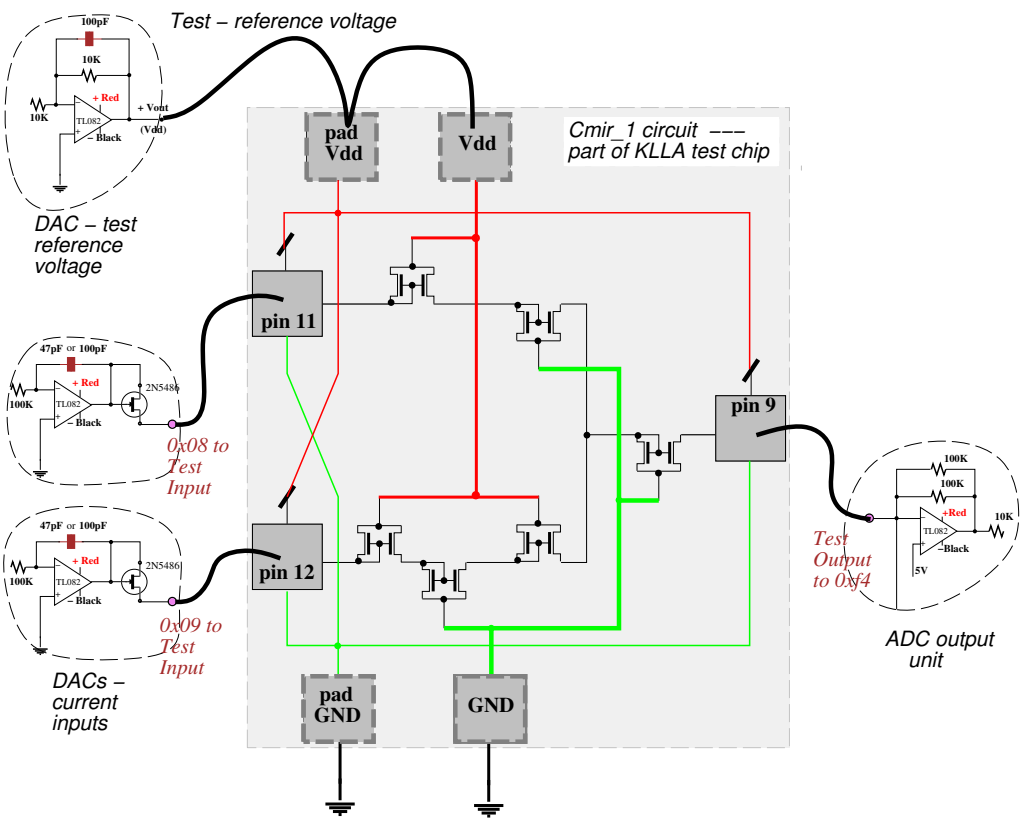

Figure 4: Connecting an analog circuit to the ATB
Portions of the test-reference-voltage circuit (Figure 8), test-current circuits (Figure 9), and test-output circuit (Figure 10) are shown connected to a typical current-mode test circuit. The circuit is typically a DIP chip mounted on a prototyping board; the connections are made through shielded coaxial cables to pins on the chip.

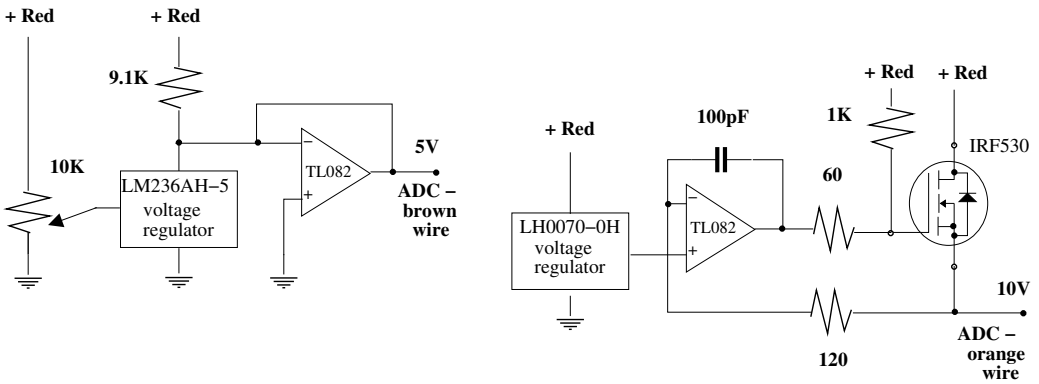

Figure 5: ADC reference voltages
These two circuits provide stable reference voltages for the Analog-Digital Conversion circuits. The voltages are routed through brown and orange wires. "+Red" is the external +15V supply.

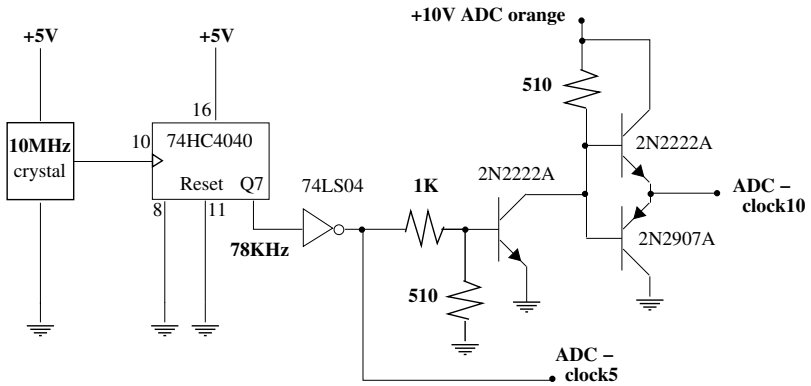

Figure 6: ADC Clock circuit
This crystal-controlled clock circuit provides a 78KHz clock at 5V for the output transceiver chips, and at 10V for the ADC chips.

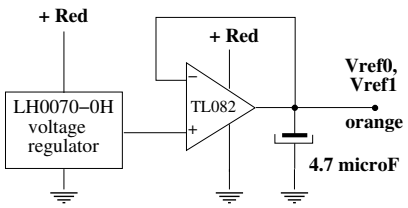

Figure 7: DAC reference voltages
Two copies of this circuit, $V\text{ref}_0$ and $V\text{ref}_1$, provide 10V references to the DAC circuits in Figures 8 and 9.

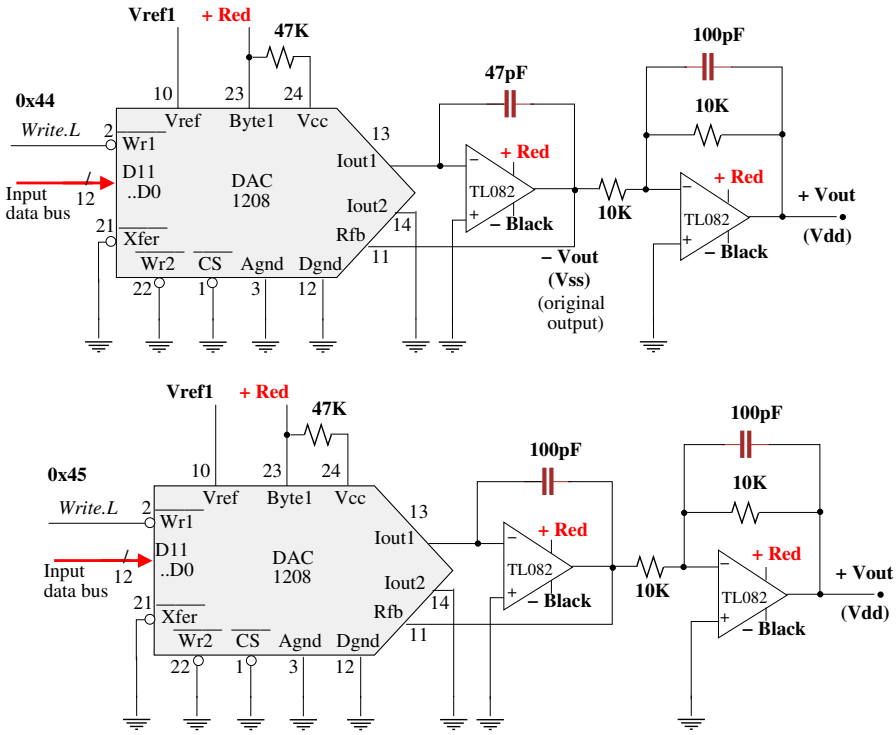

Figure 8: Test-circuit Reference Voltages
These two circuits supply a selectable reference voltage to the test circuit. Either device can supply a negative voltage (Vss) from the output of the leftmost op amp, or a positive voltage (Vdd) from the output of the rightmost op amp.

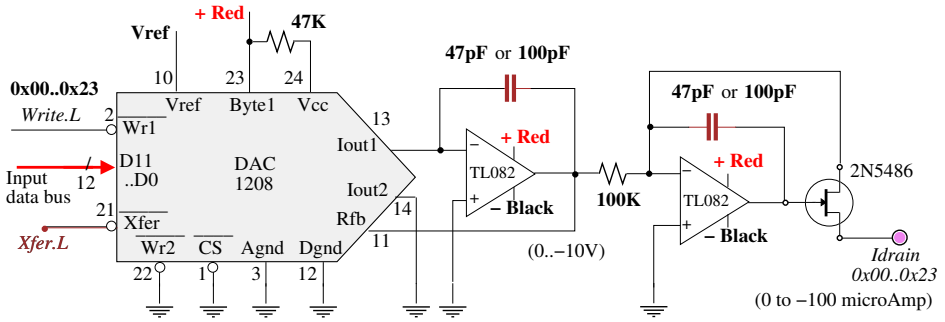

Figure 9: Test-circuit Inputs

Thirty-six copies of this circuit, with addresses from **0x00** to **0x23**, supply selectable currents to test inputs. The currents are *drains*, i.e. positive charge flows *from* the test inputs *to* the 2N5486 transistor.

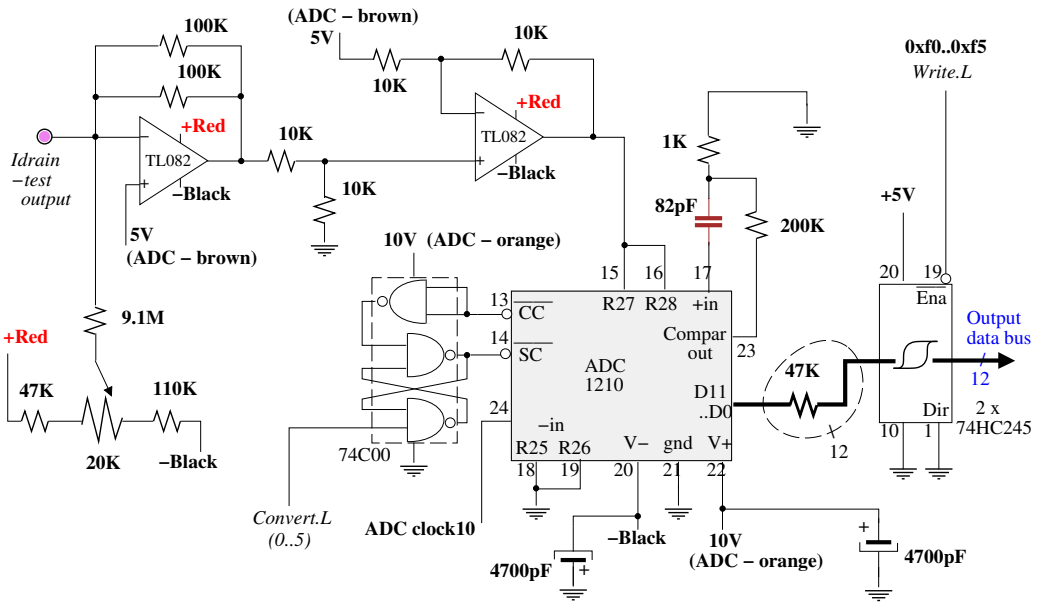

Figure 10: Test-circuit Outputs

Six copies of this circuit, addressed from **0xf0** to **0xf5**, digitize test outputs and supply 12-bit values to the output data bus. Test outputs are assumed to be current *drains*, i.e. positive charge flows *from* the leftmost op amp *to* the test circuit.

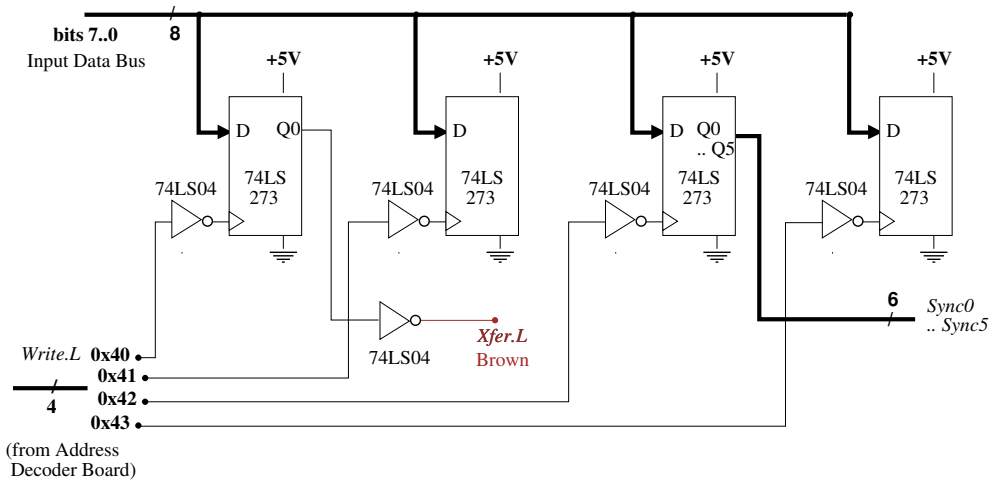

Figure 11: Triggering devices

These "devices" generate triggering signals for the DACs and ADCs when addressed. Device **0x40** uses bit 0 of the input data bus to determine whether the DACs actually receive an *Xfer.L* signal. Device **0x42** uses bits 0..5 to determine which ADCs get a *Sync* (and hence *Convert.L*) signal. (Devices **0x41** and **0x43** are unused.)

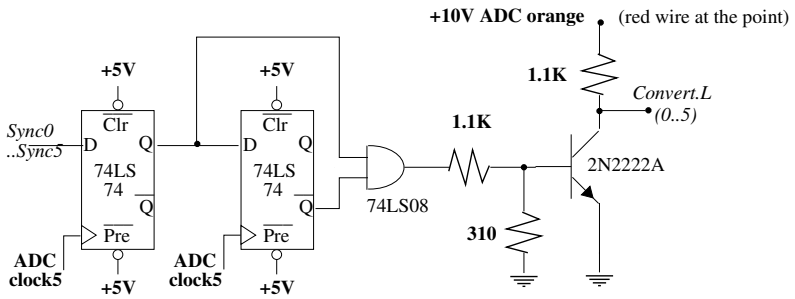

Figure 12: Conversion Synchronization

Six copies of this circuit convert each of the ADC *Sync* signals, from device **0x42**, to single-clock-cycle *Convert.L* pulses that trigger the appropriate ADC to begin a conversion.

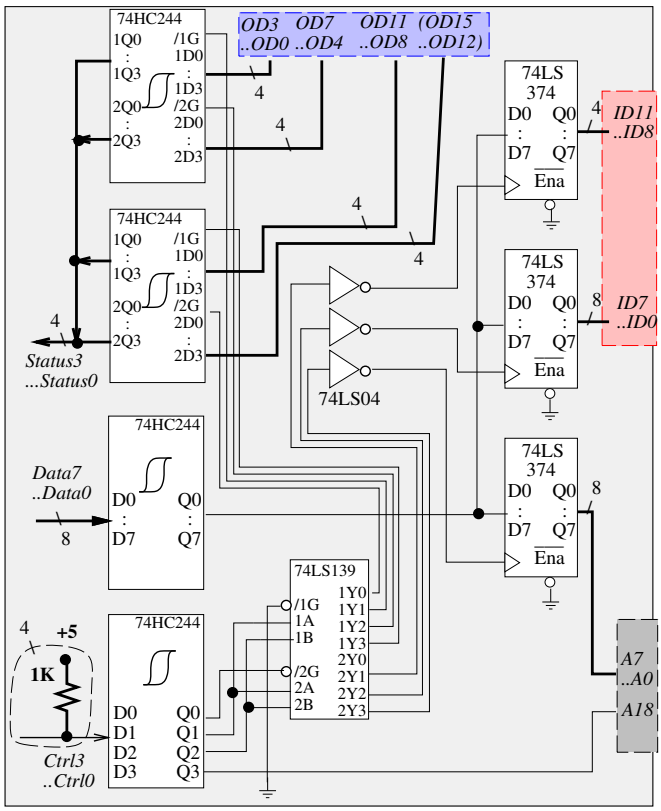

Figure 13: Parallel Interface Board

This circuit board buffers and distributes values to and from the various functional units on the ATB, in conjunction with the Address Decode Board.

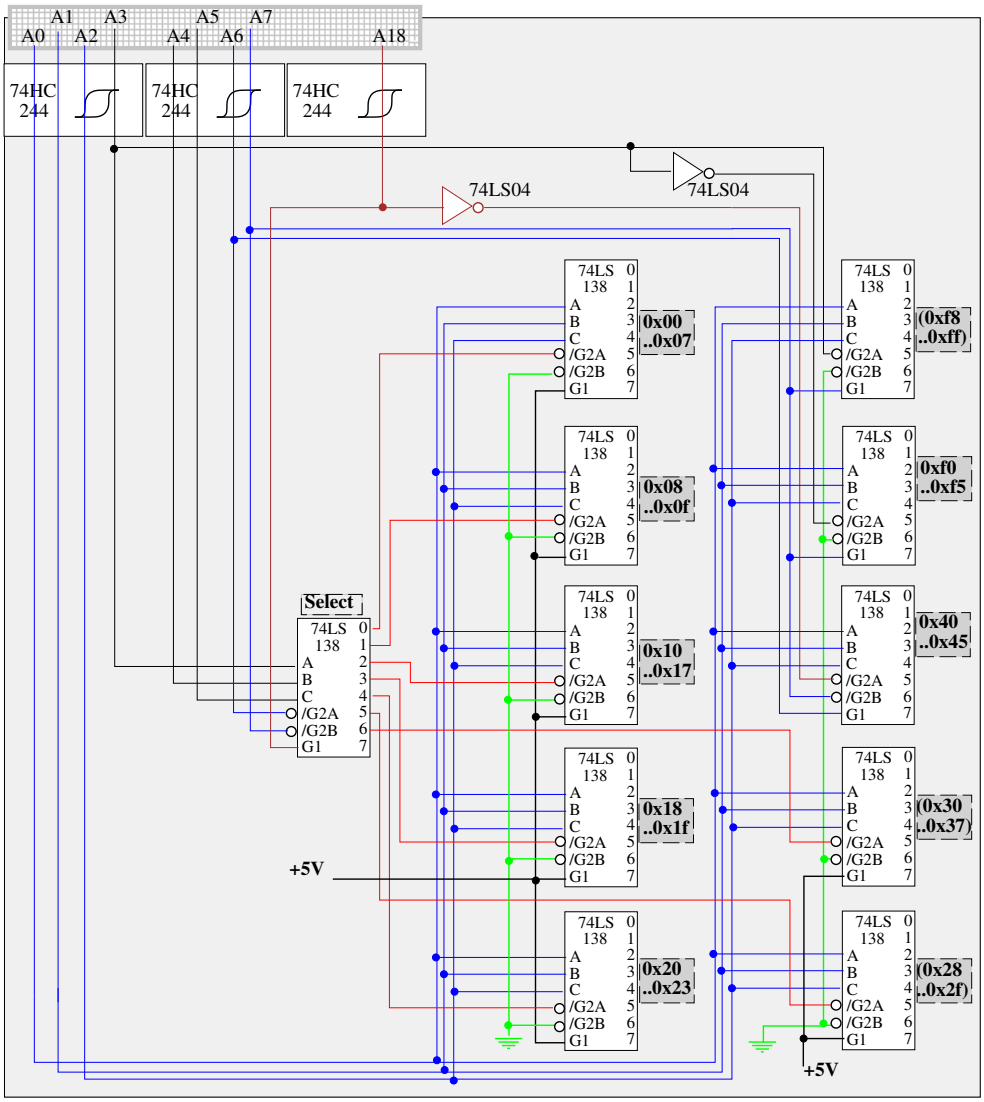

Figure 14: Address Decode Board

This circuit board decodes address lines A7..A0 to activate devices on the ATB. The 74LS138 chips issue *Write.L* signals for the device addresses they are labeled with.

Interface Signal	I/O Register Bit	DB-25 Connector Pin		I/O Register Bit	Interface Signal
$/C_0$	!Ctrl 0	1	14	!Ctrl 1	$/C_1$
D_0	Data 0	2	15	Status 3	-
D_1	Data 1	3	16	Ctrl 2	C_2
D_2	Data 2	4	17	!Ctrl 3	$/C_3$
D_3	Data 3	5	18	-	(Ground)
D_4	Data 4	6	19	-	(Ground)
D_5	Data 5	7	20	-	(Ground)
D_6	Data 6	8	21	-	(Ground)
D_7	Data 7	9	22	-	(Ground)
S_2	Status 6	10	23	-	(Ground)
$/S_3$	!Status 7	11	24	-	(Ground)
S_1	Status 5	12	25	-	(Ground)
S_0	Status 4	13			

Figure 15: Parallel-port pinouts

I/O Register bits are electrically connected to pins of a DB-25 female connector, with corresponding ATB interface signals. Register bits/connector pins are mapped to the ATB Parallel Interface signals as shown.

'-' No connection.

'!' Register bit is inverted relative to its corresponding pin (*e.g.*, [Ctrl bit 1 = 0] implies [pin 14 = 5V]).

'/' Signal is negated (due to the register/pin inversion).

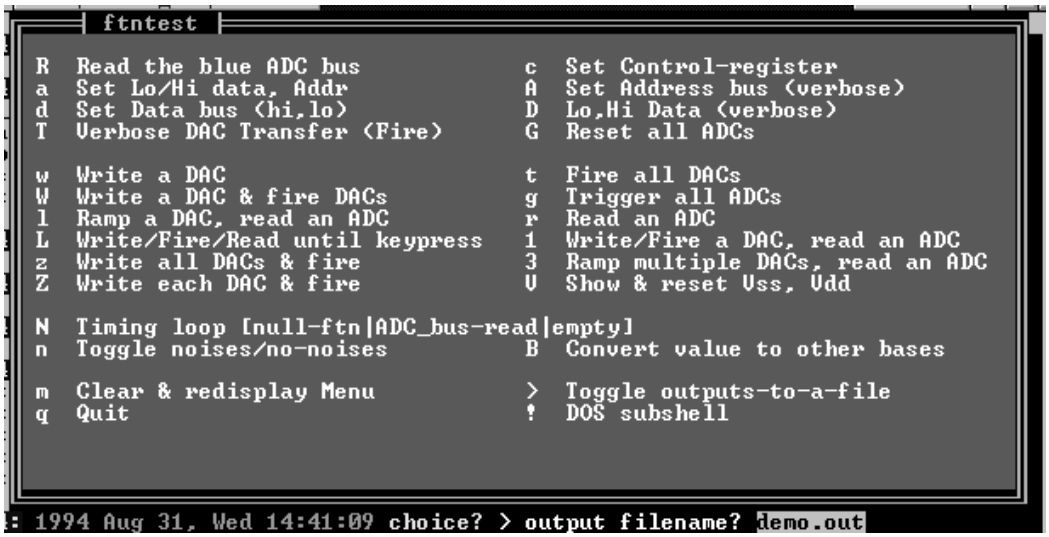

Figure 16: `ftntest` Command Menu

The `ftntest` screen shows available commands. Commands with output overwrite the menu, which can be redrawn when needed. Output can also be copied to a file: here the user has entered the '>' command, which prompts for a filename, and the user has given "demo.out".

The output file is shown in Figure 18, after the command sequence

{'V', 'w', 'l', 'w', 'l', 'w', 'l', 'q'}

is used to exercise the Cmir_1 circuit. The 'V', 'w', and 'l' commands prompt for values.

Figure 17 graphs the output file.

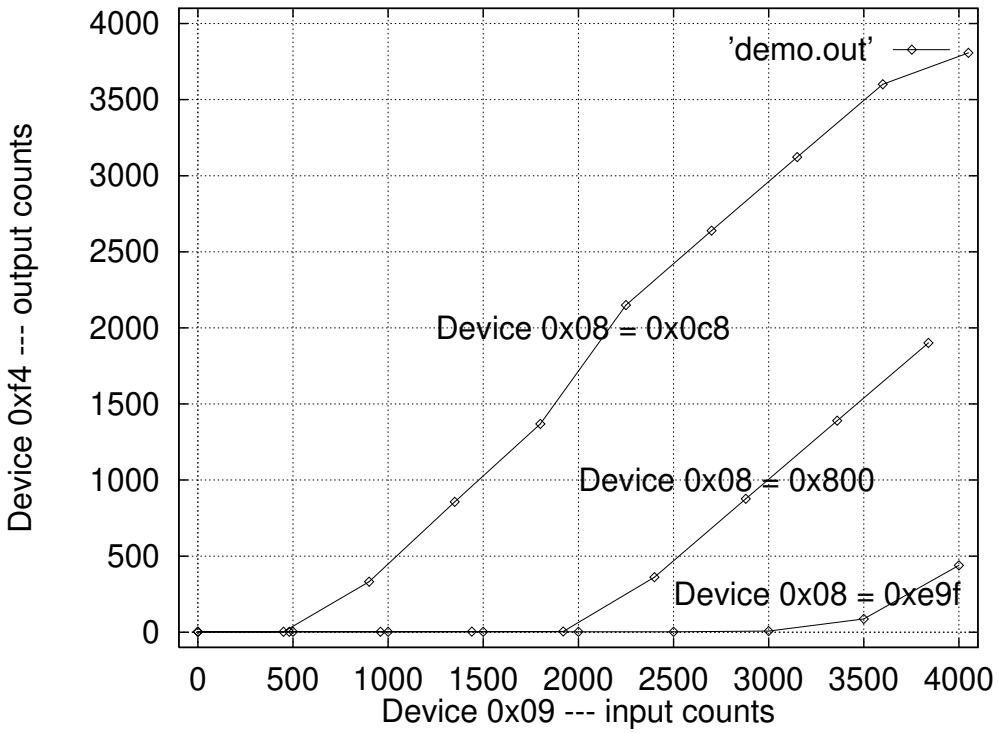

Figure 17: Plotted KLLA response
Test data from Figure 18 are plotted here, using the widely-available `gnuplot` data-plotting program.

Figure 18, next page: "demo.out" is a flat text file. The user ran the 'V' command to set the test-reference voltages (both are connected for positive voltage, but `ftntest` doesn't know that). Then 'w' to set device **0x08**, followed by 'l' to sweep device **0x09** and read device **0xf4**, were run three times to make a set of response curves — three distinct sweep intervals have been chosen. Optional hexadecimal and binary results are also available, in a format suitable for the `gnuplot` plotting program.

```
# Now:  Vss=-10.000, Vdd=10.000.
# DAC $08 gets $e9f  (1110_1001_1111)
# signal: DAC $9;  response: ADC $4
   0     3  #  $  0  $  3  ##  0000_0000_0000  0000_0000_0011
 500     3  #  $1f4  $  3  ##  0001_1111_0100  0000_0000_0011
1000     3  #  $3e8  $  3  ##  0011_1110_1000  0000_0000_0011
1500     3  #  $5dc  $  3  ##  0101_1101_1100  0000_0000_0011
2000     3  #  $7d0  $  3  ##  0111_1101_0000  0000_0000_0011
2500     3  #  $9c4  $  3  ##  1001_1100_0100  0000_0000_0011
3000     7  #  $bb8  $  7  ##  1011_1011_1000  0000_0000_0111
3500    87  #  $dac  $ 57  ##  1101_1010_1100  0000_0101_0111
4000   439  #  $fa0  $1b7  ##  1111_1010_0000  0001_1011_0111

# DAC $08 gets $0c8  (0000_1100_1000)
# signal: DAC $9;  response: ADC $4
   0     3  #  $  0  $  3  ##  0000_0000_0000  0000_0000_0011
 450     3  #  $1c2  $  3  ##  0001_1100_0010  0000_0000_0011
 900   332  #  $384  $14c  ##  0011_1000_0100  0001_0100_1100
1350   856  #  $546  $358  ##  0101_0100_0110  0011_0101_1000
1800  1369  #  $708  $559  ##  0111_0000_1000  0101_0101_1001
2250  2150  #  $8ca  $866  ##  1000_1100_1010  1000_0110_0110
2700  2639  #  $a8c  $a4f  ##  1010_1000_1100  1010_0100_1111
3150  3122  #  $c4e  $c32  ##  1100_0100_1110  1100_0011_0010
3600  3601  #  $e10  $e11  ##  1110_0001_0000  1110_0001_0001
4050  3807  #  $fd2  $edf  ##  1111_1101_0010  1110_1101_1111

# DAC $08 gets $800  (1000_0000_0000)
# signal: DAC $9;  response: ADC $4
   0     3  #  $  0  $  3  ##  0000_0000_0000  0000_0000_0011
 480     3  #  $1e0  $  3  ##  0001_1110_0000  0000_0000_0011
 960     3  #  $3c0  $  3  ##  0011_1100_0000  0000_0000_0011
1440     4  #  $5a0  $  4  ##  0101_1010_0000  0000_0000_0100
1920     5  #  $780  $  5  ##  0111_1000_0000  0000_0000_0101
2400   361  #  $960  $169  ##  1001_0110_0000  0001_0110_1001
2880   877  #  $b40  $36d  ##  1011_0100_0000  0011_0110_1101
3360  1390  #  $d20  $56e  ##  1101_0010_0000  0101_0110_1110
3840  1901  #  $f00  $76d  ##  1111_0000_0000  0111_0110_1101


# 1994 Aug 31, Wed 14:23:23
```

Figure 18: `ftntest`'s Output File

A ATB Control Software

This appendix contains listings of the author's software for testing current-mode analog circuits on the ATB. §A.1 shows the low-level routines that manipulate the functional units of the ATB. §A.2 is `ftntest`, a (largish) program that offers a menu of functions for testing portions of the ATB hardware, and for basic testing of circuits via the ATB. §A.3 contains miscellaneous routines that `ftntest` uses; §A.4 is a makefile for `ftntest`.

A.1 Library Routines

These three files, `low-para.h`, `atbpara.h`, and `atbpara.c`, provide basic routines to manipulate the parallel interface from a C-language program.

```
/*-------------------------------------------------------------------*\
| low-para.h                                                          |
|                                                                     |
| Low-level constants and macros that define the parallel port and its  |
| control.  Here we take care of inverted bits, weird bit/nybble      |
| positions, and such arcana....                                      |
|                                                                     |
| Foo.                                                                |
|                                                                     |
| 92-11-10 Split out from atbpara.h, because only the atbpara.c file  |
|       really needs this stuff.                                      |
| 93/01/20 ... and ftntest.c ...                                      |
| 93-03-28 Rename "S_Reg_Nybble" to "S_Reg_Upper4" so I'll remember   |
|       which nybble has the data  :(                                 |
\*-------------------------------------------------------------------*/
#if !defined(_LOW_PARA_H)
#define _LOW_PARA_H
#include <dos.h>

#define D_Reg    0x0378         /*---------------------------------*/
#define S_Reg    0x0379         /* Parallel-port register addresses */
#define C_Reg    0x037a         /*---------------------------------*/

/*-------------------------------------------------------*\
| Bit 3 of the Status register is inverted.  This fixes it. |
```

```
\*-------------------------------------------------------*/
#define STATUS_X_MASK   0x80    /* 1000_0000 */

/*-------------------------------------------------------*\
| Extract a data nybble from (the high end of) the Status |
| register.  The 0x00f0 masking really is necessary ---   |
| it's cost me a week and a half to determine that :<     |
\*-------------------------------------------------------*/
#define S_Reg_Upper4    (word)(0x00f0 & (STATUS_X_MASK ^ inportb(S_Reg)))

/*-------------------------------------------------------------*\
| Control-Port signals - ref. Lab Notebook #2, pp.13,14, etc.   |
|                                                               |
| Control bits (on the ATB interface board):                    |
|   7..4 - (no connection)                                      |
|      3 - DAC-addr enable ( ATB address line a20)              |
|      2 -  } Interface-latch selector                          |
|      1 -  }                                                   |
|      0 - '139 latch enable.L (interface addr buffer??)        |
|                                                               |
| The parallel-port Control Register inverts bits 3, 1, & 0     |
| (eat flaming death IBM), so these constants incorporate the   |
| necessary bit inversions to produce desired values at the     |
| interface board.                                              |
\*-------------------------------------------------------------*/
#define CTRL_X_MASK             0x0b    /* 0000_1011 */
#define LATCH_ENABLE_MASK       0x01

/*-------------------------------------------------------------*\
| Select one of the ATB-device-addressing latches (or nothing)  |
| to accept the byte from the Data register.  The latch isn't   |
| enabled until bit 0 goes TRUE (low at the '139 chip).         |
|                                                               |
| Usage:                                                        |
|     Send out the appropriate latch-select value.              |
|     Send it again, OR'd w/ LATCH_ENABLE_MASK, to actually     |
|       latch in the D_Reg value.                               |
|     Send it again w/o LATCH_ENABLE_MASK to disable the latch. |
|                                                               |
```

```
| This sequence avoids glitches when changing latch selects.    |
|                                                               |
| The desired value at the ATB interface board is XOR'd to      |
| convert it to what the C_Reg needs to see.                    |
\*-------------------------------------------------------------*/
#define ctrl_NULL       (0x01 ^ CTRL_X_MASK)    /* 0001 -> 1010 */
#define D_HI_LATCH      (0x03 ^ CTRL_X_MASK)    /* 0011 -> 1000 */
#define D_LO_LATCH      (0x05 ^ CTRL_X_MASK)    /* 0101 -> 1110 */
#define A_LATCH         (0x07 ^ CTRL_X_MASK)    /* 0111 -> 1100 */

/*-------------------------------------------------------------*\
| Select one of the "Blue Bus" latches.  These values           |
| incidentally select one of the device-addressing latches as   |
| well, but they are held DISabled by bit 0.  One of these      |
| latches is always unavoidably selected, and they have been    |
| hardwired to be ENabled; so there is always something on the  |
| Status-register lines.  We simply ignore the S_Reg until      |
| after we've deliberately latched a meaningful value.          |
|                                                               |
| As above, the desired value at the ATB interface board is     |
| XOR'd to convert it to what the C_Reg needs to see.           |
\*-------------------------------------------------------------*/
#define n3_READ         (0x01 ^ CTRL_X_MASK)    /* 0001 -> 1010 */
#define n2_READ         (0x07 ^ CTRL_X_MASK)    /* 0111 -> 1100 */
#define n1_READ         (0x05 ^ CTRL_X_MASK)    /* 0101 -> 1110 */
#define n0_READ         (0x03 ^ CTRL_X_MASK)    /* 0011 -> 1000 */

/*-------------------------------------------------------------*\
| Enable the outputs on the address-decoder board, so the       |
| addressed DAC will buffer a value from the Red bus.           |
\*-------------------------------------------------------------*/
#define DAC_ENABLE      (0x09 ^ CTRL_X_MASK)    /* 1001 -> 0010 */

/*-------------------------------------------------------------*\
| Produce the '139-enabling version of a ctrl-reg value.        |
|                                                               |
| The un-enabled ctrl-reg value sets up the '139 multiplexer    |
| to clock one of the '374 FF-registers on the interface board; |
| this enabling version causes the '139 to actually emit        |
```

```
| whichever "clock" pulse is set up.  The enabling protocol     |
| blocks any switching transients in the '139 from triggering   |
| one of the '374s inappropriately.                             |
\*--------------------------------------------------------------*/
#define enabled(Ctrl)           ((Ctrl) | LATCH_ENABLE_MASK)

#endif  /* _LOW_PARA_H */
```

```
/*--------------------------------------------------------------------*\
|  ATBPARA.H                                                           |
|                                                                      |
| This header file defines data types, constants, and function         |
| prototypes (or macros) for use by a program doing parallel I/O to    |
| the Analog Test Board (ATB).                                         |
|                                                                      |
| 93-05-25 ADC_delay gone again; I understand what I'm doing better... |
|       logic-analyzer timings provoke changes in atbpara.c            |
| 93-04-23 "lo(WORD)" and "hi(WORD)" moved from the newly-created      |
|          "local.h" to here.                                          |
| 93-03-20 ADC delay in MICROsecs instead of millisecs                 |
| 93-03-17 read_data() replaced by ADC_bus().                          |
| 93-03-14 Vdd, Vss can now be set using 12-bit values.  Floating-point |
|          inputs use adjust_Vdd(), adjust_Vss() instead.              |
| 93-02-07 Debugged the comments (!) as part of the distribs package.  |
| 93-02-02 Trying to figure out the post-Nov. changes... (ugh).        |
|          Cleaned up comments some.                                   |
| 92-11-10 Debug the internal changes.  Move some parallel-port stuff  |
|          to low-para.h, "simplify" some more stuff.                  |
| 92-11-03 Internal changes for efficiency - some constants mutate too |
| 92-05-24                                                             |
\*--------------------------------------------------------------------*/
#if !defined(_ATBPARA_H)
#define _ATBPARA_H

typedef unsigned char byte;
typedef unsigned int  word;
```

```
#define lo(Byte_)  (Byte_ & 0x00ff)
#define hi(Byte_)  (Byte_ >> 8)

#define DAC_XFER    0x40                /*------------------*/
#define ADC_CONV    0x42                /* Device addresses */
#define Vss_DAC     0x44                /*------------------*/
#define Vdd_DAC     0x45
#define addr_NULL   0x3f   /*  An address that shouldn't enable anything  */

#define ADC0    0                       /*-------------*/
#define ADC1    1                       /* ADC indexes */
#define ADC2    2                       /*-------------*/
#define ADC3    3
#define ADC4    4
#define ADC5    5
#define no_ADC  0xffff

/*-----------------------------------------------*\
| Conversions between User units & device counts |
\*-----------------------------------------------*/
#define COUNTS_PER_VOLT (float)(MAX_COUNT/10.0)
#define MAX_COUNT       0xfff
#define NUM_COUNTS      (MAX_COUNT+1)

/*---------------------------------------*\
|  Bring the ATB to a known initial state.  |
\*---------------------------------------*/
void reset_ATB(void);

/*-------------------------------------------------------------------------*\
|  DAC conversions --- DAC_XFER device distributes a conversion signal  |
|  to all DACs.  Not used directly.                                     |
|                                                                       |
|  write_DAC() causes the selected DAC to latch the (12-bit) value      |
|  into its input buffer.                                               |
|                                                                       |
|  trigger_DACs() causes all the DACs to transfer their input buffers   |
|  to their conversion registers, thus generating new analog outputs.   |
```

```
|                                                                      |
|  DAC_flow_through() causes all the DACs to pass their buffer values  |
|  straight through to their conversion registers.                     |
\*--------------------------------------------------------------------*/
#define DAC_convert_new()       write_device(DAC_XFER, 0x01)
#define hold_DAC_conversion()   write_device(DAC_XFER, 0x00)

#define write_DAC(addr, data)   write_device(addr, data)
#define trigger_DACs()          (DAC_convert_new(), hold_DAC_conversion())
#define DAC_flow_through()      DAC_convert_new()

/*-------------------------------------------*\
|  Reset all ADCs so they're ready to accept a  |
|  new conversion signal.  Used by reset_ATB(). |
\*-------------------------------------------*/
#define reset_ADCs()    write_device( ADC_CONV, 0 )

/*--------------------------------*\
|  Convert and read a single ADC.  |
\*--------------------------------*/
word ADC_sample(byte adc_num);

/*-----------------------------------------------------*\
|  Trigger ADC conversions, enable ADC output lines,    |
|  access the Blue output bus; reset ADC conversions.   |
|  ADC_sample() integrates these into one function.     |
|                                                       |
| 93-03-17 read_data() replaced by ADC_bus().           |
\*-----------------------------------------------------*/
#define trigger_ADC(adc)        adc_strobe( ADC_trgr(adc) )
#define trigger_all_ADCs()      adc_strobe(all_ADC_triggers)
#define latch_ADC_out(adc)      set_addr( ADC_addr(adc) )

word ADC_bus(void);             /* Read whatever's on the Blue bus. */

/*-----------------------------------------------------------*\
| High-level functions/macros that accept values in user units  |
| (eg, -10..10 volts) & control the ATB in byte-oriented terms. |
\*-----------------------------------------------------------*/
```

```
#define word_form(v)    (word)( (v) * COUNTS_PER_VOLT )
#define volt_form(c)    (float)( (c) / COUNTS_PER_VOLT )

#define set_Vdd_DAC(val)        write_DAC(Vdd_DAC, (val))
#define set_Vss_DAC(val)        write_DAC(Vss_DAC, (val))

void adjust_Vdd(float val);
void adjust_Vss(float val);

/*************************************************************************\
*  Low-level functions, constants, macros.  The high-level functions      *
*  above provide the interfaces to this stuff....                         *
*  Generally, User programs needn't refer to these directly.              *
\*************************************************************************/

void latch_in(byte val, byte ctrl);     /* Load byte to interface register */
void set_addr(byte addr);               /* load Addr interface register */
void set_data(word data);               /* load Data interface registers */

void write_device(byte addr, word data);        /* yer basic control ftn */

/*------------------------*\
| ADC addresses & triggers |
\*------------------------*/
#define ADC_addr(adc)    ( 0xf0 | adc )
#define ADC_trgr(adc)    ( 0x01 << adc )
#define all_ADC_triggers ( ADC_trgr(0) | ADC_trgr(1) | ADC_trgr(2) | \
                           ADC_trgr(3) | ADC_trgr(4) | ADC_trgr(5) )
void adc_strobe(byte adcbits);

#endif  /* _ATBPARA_H */
```

```
/*--------------------------------------------------------------------*\
|  ATBPARA.C                                                          |
|                                                                     |
| Subroutines that perform I/O to the Analog Test Board (ATB) via     |
| the parallel port.                                                  |
|                                                                     |
| 93-05-25: adc_strobe() doesn't need ADC_delay; it is slow enough to |
|       let the single-pulser flipflops reset anyway.  ADC_sample()  |
|       gets changes based on logic-analyzer timing information.      |
| 93-04-29:(  latch_in() becomes a subroutine again, after evidence   |
|       that the inline code experiences more time jitter.  I wish I  |
|       understood why.  Make it externally visible while we're at it.|
| 93-03-20: Make use of u_time() microsecond timer.                   |
| 93-03-19: Change read_data() to ADC_bus() for aesthetic reasons.    |
| 93-03-14: Floating-point Vdd & Vss ftns renamed to "adjust_V.."     |
|       while "set_V.._DAC" become macros with integer inputs.        |
|       Fluke 97 indicates a 78.1KHz (12.8us) clock.                  |
|                                                                     |
| 92-11-16: After profiling:  latch_in() becomes a macro for more speed |
| 92-11-10: After a flakey initial attempt, clean up/speed up the     |
|       functions.  Move the parallel-port #defines into low-para.h   |
| 92-06-21: ADC-trigger patch. bobmon                                 |
| 92-06-18: touchup. bobmon                                           |
\*--------------------------------------------------------------------*/
#include <stdio.h>      /* For error messages in the Vdd/Vss routines. */
#include "low-para.h"
#include "atbpara.h"
#include "timing.h"

#undef DEBUG

#define C_reg_OFF()     outportb(C_Reg, ctrl_NULL)

void reset_ATB(void)
{
    C_reg_OFF();                /* Disable all interface latches.       */
    hold_DAC_conversion();      /* enable double-buffering.             */
    reset_ADCs();               /* ready ADCs for new conversion.       */
    C_reg_OFF();                /* Disable all latches (again).         */
```

```
    u_time_calibrate(1000000L); /* prepare for microsecond delays.      */
}

/*-----------------------------------------------------------------------*\
|  Map one of 256 logical addresses to the physically-wired equivalent. |
|  (sigh... software hack for hardware quirk:  each pair of bits        |
|  (0&1, 2&3; 4&5, 6&7) is swapped.)                                    |
|  Each nybble translates as:                                           |
|       0000 : 0000     0100 : 1000     1000 : 0100     1100 : 1100     |
|       0001 : 0010     0101 : 1010     1001 : 0110     1101 : 1110     |
|       0010 : 0001     0110 : 1001     1010 : 0101     1110 : 1101     |
|       0011 : 0011     0111 : 1011     1011 : 0111     1111 : 1111     |
|  92-11-03 Okay, we make one big table, for speed.  The moot code just |
|       below shows the original mapping method.                        |
\*-----------------------------------------------------------------------*/
#if 0
static const byte Addr_Map[16] = {
    0x0, 0x2,  0x1, 0x3,    0x8, 0xa,  0x9, 0xb,
    0x4, 0x6,  0x5, 0x7,    0xc, 0xe,  0xd, 0xf
};
#define address(addr)  \
    ( (Addr_Map[ (addr & 0xf0) >> 4 ] << 4)  |  Addr_Map[ addr & 0x0f ] )
#else   /*-------------------------------------------------------*/

static const byte near Addr_Map[] = {
    0x00, 0x02, 0x01, 0x03,  0x08, 0x0a, 0x09, 0x0b,
      0x04, 0x06, 0x05, 0x07,  0x0c, 0x0e, 0x0d, 0x0f,
    0x20, 0x22, 0x21, 0x23,  0x28, 0x2a, 0x29, 0x2b,
      0x24, 0x26, 0x25, 0x27,  0x2c, 0x2e, 0x2d, 0x2f,
    0x10, 0x12, 0x11, 0x13,  0x18, 0x1a, 0x19, 0x1b,
      0x14, 0x16, 0x15, 0x17,  0x1c, 0x1e, 0x1d, 0x1f,
    0x30, 0x32, 0x31, 0x33,  0x38, 0x3a, 0x39, 0x3b,
      0x34, 0x36, 0x35, 0x37,  0x3c, 0x3e, 0x3d, 0x3f,

    0x80, 0x82, 0x81, 0x83,  0x88, 0x8a, 0x89, 0x8b,
      0x84, 0x86, 0x85, 0x87,  0x8c, 0x8e, 0x8d, 0x8f,
    0xa0, 0xa2, 0xa1, 0xa3,  0xa8, 0xaa, 0xa9, 0xab,
      0xa4, 0xa6, 0xa5, 0xa7,  0xac, 0xae, 0xad, 0xaf,
    0x90, 0x92, 0x91, 0x93,  0x98, 0x9a, 0x99, 0x9b,
```

```
      0x94, 0x96, 0x95, 0x97,  0x9c, 0x9e, 0x9d, 0x9f,
    0xb0, 0xb2, 0xb1, 0xb3,  0xb8, 0xba, 0xb9, 0xbb,
      0xb4, 0xb6, 0xb5, 0xb7,  0xbc, 0xbe, 0xbd, 0xbf,

    0x40, 0x42, 0x41, 0x43,  0x48, 0x4a, 0x49, 0x4b,
      0x44, 0x46, 0x45, 0x47,  0x4c, 0x4e, 0x4d, 0x4f,
    0x60, 0x62, 0x61, 0x63,  0x68, 0x6a, 0x69, 0x6b,
      0x64, 0x66, 0x65, 0x67,  0x6c, 0x6e, 0x6d, 0x6f,
    0x50, 0x52, 0x51, 0x53,  0x58, 0x5a, 0x59, 0x5b,
      0x54, 0x56, 0x55, 0x57,  0x5c, 0x5e, 0x5d, 0x5f,
    0x70, 0x72, 0x71, 0x73,  0x78, 0x7a, 0x79, 0x7b,
      0x74, 0x76, 0x75, 0x77,  0x7c, 0x7e, 0x7d, 0x7f,

    0xc0, 0xc2, 0xc1, 0xc3,  0xc8, 0xca, 0xc9, 0xcb,
      0xc4, 0xc6, 0xc5, 0xc7,  0xcc, 0xce, 0xcd, 0xcf,
    0xe0, 0xe2, 0xe1, 0xe3,  0xe8, 0xea, 0xe9, 0xeb,
      0xe4, 0xe6, 0xe5, 0xe7,  0xec, 0xee, 0xed, 0xef,
    0xd0, 0xd2, 0xd1, 0xd3,  0xd8, 0xda, 0xd9, 0xdb,
      0xd4, 0xd6, 0xd5, 0xd7,  0xdc, 0xde, 0xdd, 0xdf,
    0xf0, 0xf2, 0xf1, 0xf3,  0xf8, 0xfa, 0xf9, 0xfb,
      0xf4, 0xf6, 0xf5, 0xf7,  0xfc, 0xfe, 0xfd, 0xff,
};
#define address(addr)  Addr_Map[addr]

#endif  /*-------------------------------------------------------*/

/*-------------------------------------------------------------*\
| Put a byte on the parallel-port data lines, then strobe the   |
| appropriate interface-board latch to accept it.               |
| This needs to be a function to try to keep timing clean...    |
| The final shutdown (ctrl_NULL) seems to help keep things      |
| cleaner on the interface board.                               |
|                                                               |
| We'll also make it visible "just in case".                    |
\*-------------------------------------------------------------*/
void latch_in(byte val, byte ctrl)
{
    outportb(D_Reg, val);               /* Value onto data bus  */
    outportb(C_Reg, ctrl);              /* Select desired latch */
```

```
    outportb(C_Reg, enabled(ctrl));     /* Enable the latch     */
    outportb(C_Reg, ctrl);              /* Drop the enable bit  */
}

/*-------------------------------------------------------------*\
|  Latch in an address.                                        |
\*-------------------------------------------------------------*/
#define s_set_addr(a)   latch_in(address(a), A_LATCH)

void set_addr(byte addr) { s_set_addr(addr); }

/*-------------------------------------------------------------*\
| Latch in both bytes of a data-bus value.  The macro is       |
| optimized after inspecting tcc's assembly output.            |
\*-------------------------------------------------------------*/
#define s_set_data(d)   ( latch_in( (d), D_LO_LATCH ), \
                          latch_in( ((d) >> 8), D_HI_LATCH ) )

void set_data(word data) { s_set_data(data); }

/*-------------------------------------------------------------*\
|  Latch a 12-bit datum, and an 8-bit address, to the board;   |
|  then strobe the address-enable so the addressed device      |
|  accepts the data value.  Finish by setting a null address.  |
\*-------------------------------------------------------------*/
void write_device(byte addr, word data)
{
    s_set_data(data);           /* Load the Red bus.                    */
    s_set_addr(addr);           /* Choose a device...                   */
    outportb(C_Reg, DAC_ENABLE); /* All dev's enabled, only 1 addr'd.   */
    s_set_addr(addr_NULL);      /* Un-choose the device.                */
    C_reg_OFF();                /* Disable all interface latches.       */
}

/*-------------------------------------------------------------*\
| Trigger an ADC to convert, give the conversion some time,    |
| then  enable its output buffer and read the value one nybble |
| at a time.  The Nat'l Semi. ADC1210 needs 100us from the     |
| Start-Conversion trigger to produce a valid 12-bit output.   |
```

```
|                                                                  |
| 93-05-30: Do a correct ADC-reset at the end.  This function      |
|       CANNOT support reading more than one ADC after a common    |
|       triggering event.  To do that, all the ADCs should be      |
|       triggered, each ADC latched and bussed out, then           |
|       they should all be reset.                                  |
| 93-05-25: 200usec -> ~40usec error margin for conversion.        |
| 93-03-20: u_time() 4X margin of error --- delay(1) gives 10X     |
\*-----------------------------------------------------------------*/
word ADC_sample(byte adc_num)
{
    word tmp;

    trigger_ADC( adc_num );     /* Start the conversion and...  */
    u_time(200);                /*   give it time to complete.  */

    latch_ADC_out( adc_num );   /* Put result on the Blue...    */
    tmp = ADC_bus();            /*   bus, and read it out.      */

    reset_ADCs();               /* Prepare for another reading. */
    return tmp;
}

/*-----------------------------------------------------------------*\
| Reset & start conversion on ADC(s).                              |
| "trigger_ADC()" is a macro that converts from an ADC number      |
| to the "adcbits" byte needed here.                               |
|                                                                  |
| 93-05-25: No delay needed between reset & pulse --- ftn-call     |
|       overhead gives the single-pulsers plenty of time,          |
|       given the 78.1KHz/12.8us ADCclock  (meas. by Fluke 97)     |
\*-----------------------------------------------------------------*/
void adc_strobe(byte adcbits)
{
    write_device(ADC_CONV, 0);          /* Reset the ADC single-pulsers */
    write_device(ADC_CONV, adcbits);    /* Pulse all desired /SC lines  */
}

/*-----------------------------------------------------------------*\
```

```
| Read the ADC output via Status register, nybble-by-nybble.    |
| Data nybbles come in on bits 7..3 of the Status register, so  |
| they are shifted into position as they arrive.                |
|                                                               |
| 93-04-23: Delays out; they aren't the problem?                |
| 93-03-20: Add delays to increase the chance of reading the    |
|       parallel port correctly.  (A shot in the dark, this.)   |
| 93-03-17: read_data() becomes ADC_bus(), w/ change in         |
|       argument-passing style.                                 |
| 93-03-16: Read each nybble separately, OR them together.      |
| 92-11-10: I went too far... Stuff now buried in low-para.h    |
| 92-11-03: Speed things up as much as possible by coding       |
|       only what's needed (elegance & regularity lose to       |
|       efficiency).                                            |
| Original: This routine reads all four nybbles, even though    |
|       the high nybble (n3) is hardwired to 0 on the ATB.      |
\*--------------------------------------------------------------*/
#if 0
#   define Wait_usec    utime(1)
#else
#   define Wait_usec    {}
#endif

word ADC_bus(void)
{
    word hi, mid, lo;

    outportb(C_Reg, n2_READ);           /* bits 11..8 */
    Wait_usec;                          /* wait 1 usec (yeah right) */
    hi = S_Reg_Upper4;

    outportb(C_Reg, n1_READ);           /* bits  7..4 */
    Wait_usec;                          /* wait 1 usec (yeah right) */
    mid = S_Reg_Upper4;

    outportb(C_Reg, n0_READ);           /* bits  3..0 */
    Wait_usec;                          /* wait 1 usec (yeah right) */
    lo = S_Reg_Upper4;
```

```
    return ( (hi<<4) | mid | (lo>>4) );
}

/*-------------------------------------------------------------*\
|  Set the constant-V 'power-supply' DACs to a desired voltage. |
\*-------------------------------------------------------------*/
void adjust_Vdd(float val)
{
    if ( (val < 0.0) || (10.0 < val) )
        fprintf(stderr,"Invalid Vdd:  0 < %f < 10 fails.\n", val);
    else {
#ifdef DEBUG
        printf("Vdd: word_form(%f) = $%x ( %s )\n",
                 val, word_form(val), dec_bin(word_form(val),16));
#endif
        write_DAC( Vdd_DAC, word_form(val) );
    }
}

void adjust_Vss(float val)
{
    if ( (val < -10.0) || (0.0 < val) )
        fprintf(stderr,"Invalid Vss:  -10 < %f < 0 fails.\n", val);
    else {
        val = -val;
#ifdef DEBUG
        printf("(-): word_form(%f) = $%x ( %s )\n",
                 val, word_form(val), dec_bin(word_form(val),16));
#endif
        write_DAC( Vss_DAC, word_form(val) );
    }
}

/*-----------------------------  END  -------------------------------*/
```

A.2 `ftntest` Testing Program

This file contains the program `ftntest`, which provides a menu of testing options. It uses some of the ATB library routines, duplicates some of those routines for verification purposes, and also requires some utility functions which are listed in Appendix A.3 along with the matching ".h" files.

```
/*--------------------------------------------------------------------*\
| ftntest.c --- Trying to control the ATB via the parallel port...     |
|                                                                      |
|       Copyright 1994 R A Montante                                    |
|       All rights reserved.                                           |
|                                                                      |
| 94-08-18 Delete the hardcoded Itref option (i).                      |
|       Modify con_read(), because keyboard-macro programs affect the  |
|       console input --- 50 and 25 couldn't be entered, with my       |
|       personal macros, for example....                               |
|       Add some copyright notice per Tech Transfer's request.         |
|       Make text graphics compatible with ASCII listing, shorten      |
|       frame-drawing code.  Also, cosmetic changes.                   |
| 93-10-18 Modify "write_each_DAC()" so it's escapable.                |
| 93-10-08 Subshell supported via do_system() choice.                  |
| 93-10-07 Make noises switchable.  Update some functions?             |
| 93-07-09 Discard empty output files; prettier menu.                  |
| 93-07-05 Hacking for output-to-files, windows                        |
| 93-05-05 Prettier menu.                                              |
| 93-04-26 Conversion to "locals.lib" and .h file.  Mods to the        |
|       wr_loop() routine to support debugging of the Red Bus.         |
| 93-03-23  Try it again without the nybble-hesitation (connector      |
|       was fixed, so bits all appear again).  More menu hacking.      |
| 93-03-20  Lotsa minor changes... "void read_data(wordp)" replaced    |
|       by "word ADC_bus(void)" for aesthetic reasons; ADC_bus()       |
|       hesitates while reading nybbles, to ensure stable values.      |
| 93-03-15 Ramp_DAC() added, screen-clearing w/ menu window.           |
| 93-03-08 Settable I_tref.  write-read loop added earlier             |
|       (probably other changes too)                                   |
| 92-07-03 fold in the null-loop timing options (tidiness)             |
| 92-06-20 ftn-pointer table implemented                               |
| 92-05-23                                                             |
\*--------------------------------------------------------------------*/
```

```
#include <stdio.h>
#include <string.h>
#include <dos.h>
#include <stdlib.h>
#include <process.h>
#include <conio.h>

#include "low-para.h"
#include "atbpara.h"
#include "converts.h"
#include "local.h"
#include "timing.h"

#define FALSE   0
#define TRUE    !FALSE

#define EOL     "\r\n"
/*-------------------------------------*\
| Text-graphics   0xde  0xdd  0xb3      |
\*-------------------------------------*/
#define RBLOK   "\xde"
#define LBLOK   "\xdd"
#define VBAR    "\xb3"


/*---------------------------------------------------------------------*/

static float Vdd, Vss;
static int noisy = 1;
static int std_out = -1;
static FILE *outfile;
const char out_fmt_msg[] = "File outputs in multiple bases [ny]?";
const char step_msg[] = "\r<SPACE> to step; 'C' to complete";


/*---< Menu-option Function Prototypes >-------------------------------*/

static void set_busses(void);
static void set_Addr_bus(void);
```

```
static void do_bases(void);
static void do_c(void);
static void do_d(void);
static void do_D(void);
static void do_g(void);
static void wr_loop(void);
static void ramp_1_DAC(void);
static void ramp_multiple_DACs(void);
static void draw_menu(void);
static void do_null(void);
static void do_readADC(void);
static void do_readblue(void);
static void do_reset(void);
static void do_t(void);
static void do_T(void);
static void do_V(void);
static void wrt_a_DAC(void);
static void w_t_DAC(void);
static void wrt_each_DAC(void);
static void wrt_all_DACs(void);
static void do_1(void);
static void do__(void);
static void do_escape(void);
static void do_redirect(void);
static void fix_noise(void);
static void do_system(void);

/*------------------------------------------------------------------------*/

typedef void (*ftn_p)(void);

typedef struct _tagged_ftn {
    char tag;
    ftn_p f;
} tagged_ftn;

const tagged_ftn choice[] = {
    {'a', set_busses}   , {'A', set_Addr_bus}
    , {'B', do_bases}   , {'c', do_c}
```

```
    , {'d', do_d}       , {'D', do_D}
    , {'g', do_g}       , {'G', do_reset}
    , {'l', ramp_1_DAC} , {'L', wr_loop}
    , {'r', do_readADC} , {'R', do_readblue}
    , {'t', do_t}       , {'T', do_T}
    , {'V', do_V}
    , {'w', wrt_a_DAC}  , {'W', w_t_DAC}
    , {'z', wrt_all_DACs}, {'Z', wrt_each_DAC}
    , {'3', ramp_multiple_DACs}, {'1', do_1}
    , {'m', draw_menu}  , {'?', draw_menu}
    , {'n', fix_noise}  , {'N', do_null}
    , {'>', do_redirect}, {'!', do_system}
    , {'_', do__}       , {0x1b,do_escape}
};
#define n_choices  ( sizeof(choice) / sizeof(tagged_ftn) )


/*-------------------------------------------------------------------------*/

const char title[] =
 "\n"
 EOL"    ftntest:  portmanteau Analog Test Board exerciser"
 EOL"    version ["__DATE__", "__TIME__"]"
 "\n"
 EOL"    Copyright 1994 R A Montante"
 EOL"    All rights reserved."
 "\n\n\n"
 EOL"  Press 'm' for a menu of commands..."
;

const char menu[] =
 EOL
 " R  Read the blue ADC bus            c  Set Control-register" EOL
 " a  Set Lo/Hi data, Addr             A  Set Address bus (verbose)" EOL
 " d  Set Data bus (hi,lo)             D  Lo,Hi Data (verbose)" EOL
 " T  Verbose DAC Transfer (Fire)      G  Reset all ADCs" EOL
 "\n"
 " w  Write a DAC                      t  Fire all DACs" EOL
 " W  Write a DAC & fire DACs          g  Trigger all ADCs" EOL
```

```
 " l  Ramp a DAC, read an ADC          r  Read an ADC" EOL
 " L  Write/Fire/Read until keypress   1  Write/Fire a DAC, read an ADC" EOL
 " z  Write all DACs & fire            3  Ramp multiple DACs, read an ADC" EOL
 " Z  Write each DAC & fire            V  Show & reset Vss, Vdd" EOL
 "\n"
 " N  Timing loop [null-ftn"VBAR"ADC_bus-read"VBAR"empty]"EOL
 " n  Toggle noises/no-noises          B  Convert value to other bases"EOL
 "\n"
 " m  Clear & redisplay Menu           >  Toggle outputs-to-a-file"EOL
 " q  Quit                             !  DOS subshell"
;
/*....:....1....:....2....:....3....:....4....:....5....:....6....:....7....:.*\
\*   Menu-spacing ruler: digits mark decades, colons mark half-decades.       */


#define TOP     1
#define LEFT    3
#define frame_depth     23                      /* output lines, & borders */
#define frame_width     76                      /* menu width, borders */
#define BOTTOM          (TOP+frame_depth)       /* incl. "gutter" row  */
#define RIGHT           (LEFT+frame_width-1)

#define NoColor -1      /* invalid color number, used in show_prompt() */

/*---------------------------------------------------------------------*\
| Keep the sizes, colors, and current positions of each text window     |
| in a convenient data structure.                                       |
| WINDOWS:                                                              |
|       0 - frame;  1 - menu;  2 - output;   3 - time;  4 - "prompt"    |
\*---------------------------------------------------------------------*/
typedef struct winpos_ {
    int x, y;           /* cursor position      */
    int l, t, r, b;     /* window coordinates   */
    int bg, fg;         /* colors               */
} winpos;

winpos win_pos[] = {
      {1,1, LEFT,   TOP,   RIGHT,  BOTTOM,  BLACK,    LIGHTCYAN }
    , {0,0, LEFT+1, TOP+1, RIGHT-2,BOTTOM-2,CYAN,     WHITE     }
```

```
    , {0,0, LEFT+1, TOP+1, RIGHT-2,BOTTOM-2,LIGHTGRAY,BLACK    }
    , {0,0, LEFT+1, BOTTOM,LEFT+37,BOTTOM,  BLACK,    WHITE    }
    , {0,0, LEFT+38,BOTTOM,RIGHT,  BOTTOM,  BLACK,    WHITE    }
};
static int current_window = 0;

void hop_window(int new_window, int clear)
{
    winpos *wp;
    wp = &(win_pos[current_window]);
    wp->x = wherex();
    wp->y = wherey();
    current_window = new_window;
    wp = &(win_pos[current_window]);
    window(wp->l, wp->t, wp->r, wp->b);
    textbackground(wp->bg); textcolor(wp->fg);
    if (clear)
        clrscr();       /* Clear out the window? */
    else {
        gotoxy(wp->x,wp->y);
    }
}

#define Output_Window(CLEAR)    hop_window(2,(CLEAR))
#define Time_Window()           hop_window(3, TRUE)
#define Prompt_Window()         hop_window(4, TRUE)

/*-------------------------------------------------------------------------*\
| The "output-to-file" indicator isn't worth a window of its own,           |
| although the syntax is similar.                                           |
\*-------------------------------------------------------------------------*/
void File_Window(void)
{
    static int x, y;
    const  char flag[2] = { '>', YELLOW };
    static char locn[2];

    switch (std_out) {
      case 1:
```

```
        puttext(x,y, x,y, flag);
        break;

      case 0:
        puttext(x,y, x,y, locn);
        break;

      case -1:
        x = win_pos[0].l + (frame_width>>1);    /* middle of frame... */
        y = win_pos[0].b - 1;                   /* ...bottom edge     */
        gettext(x,y, x,y, locn);
        std_out = 0;
    }
}
/*-------------------------------------------------------------------*\
| The window frame... formed of text-graphics characters:             |
|    0xba  0xcd  0xc9  0xbb  0xc8  0xbc  0xb5  0xc6                   |
\*-------------------------------------------------------------------*/
#define PrgNm "\xb5 ftntest \xc6"
#define draw_frame() {                                  \
    winpos *wp;                                         \
    int i;                                              \
    wp = &(win_pos[0]);                                 \
    window(wp->l, wp->t, wp->r, wp->b);                 \
    textbackground(wp->bg); textcolor(wp->fg);          \
    gotoxy(wp->x, wp->y);                   /* Top */   \
    putch(0xc9);                                        \
    for (i = 2; i < frame_width-1; ++i)                 \
        putch(0xcd);                                    \
    putch(0xbb);                                        \
    gotoxy(wp->x+(sizeof(PrgNm)>>1)-1, wp->y);          \
    cputs(PrgNm);                                       \
    for (i = 2; i < frame_depth; ) {        /* Sides */ \
        gotoxy(wp->x, i);                               \
        putch(0xba);                                    \
        gotoxy(wp->x + frame_width-2, i++);             \
        putch(0xba); putch(' ');                        \
    }                                                   \
    gotoxy(wp->x, wp->y + frame_depth-1);  /* Bottom */ \
```

```
    putch(0xc8);                                        \
    for (i = 2; i < frame_width - 1; ++i)               \
        putch(0xcd);                                    \
    putch(0xbc); putch(' ');                            \
}

/*--------------------------------------------------------------------*/

void show_prompt(const char *prmpt, const int color)
{
    Prompt_Window();
    if (color != NoColor)
        textcolor(color);
    cputs(prmpt);
}

/*--------------------------------------------------------------------*/

static void cpause(void)
{
    cputs("\r <SPACE BAR> to continue");
    getch();
}

/*--------------------------------------------------------------------*\
| Text-graphics char cycles with each call, provides animated display.  |
\*--------------------------------------------------------------------*/
static char mark = 0x10;
char marker(void)
{
    if (!(mark & 0x01)) {
        mark ^= 0x0f;
    } else {
        mark &= 0xfe;
    }
    return mark;
}

/*--------------------------------------------------------------------*/
```

```
static void audible(unsigned freq)
{
    if (noisy)
        chirp(freq);
}

/*-------------------------------------------------------------------------*\
| Display a prompt, accept a character string for return, report the    |
| string's length.  Make sure a 0-length string is 0-terminated.        |
\*-------------------------------------------------------------------------*/
int con_read(char *prompt, char *bfr, int len)
{
    int x, y;
    char *b = bfr + 2;

    cputs(prompt); clreol();
    x = wherex(); y = wherey();
    fgets(b, len-2, stdin);
    gotoxy(x, y); cputs(EOL); clreol();

    b[ lookup('\n',b) ] = (char)0;
    b[ lookup('\r',b) ] = (char)0;
    return stringlen(b);
}

/*-------------------------------------------------------------------------*/

word get_word_num(char *prompt)
{
    char ans[81];

    sprintf(ans, "%s ", prompt);
    audible(660);
    Prompt_Window();
    con_read(ans, ans, 81);
    return (word)numeric(ans+2);
}

/*-------------------------------------------------------------------------*/
```

```
void con_log_string(char *bfr)
{
    cputs(bfr);
    if (std_out) {
        fprintf(outfile, "# %s", bfr+2);
        flushall();
    }
}
        /*-----------------------------------*/

void rslt_log(word data, word result, int verbose)
{
    if (!std_out)
        return;

    fprintf(outfile, "%4u  %4u  #  $%3x  $%3x", data,result, data,result);
    if (verbose) {
        fprintf(outfile, "  ##  %s  ", dec_bin(data,12) );
        fputs(dec_bin(result,12), outfile);
    }
    fputc('\n', outfile);
    flushall();
}
        /*-----------------------------------*/

void newline(void)
{
    putch('\n');
    if (std_out) {
        putc('\n', outfile);
        flushall();
    }
}

/*-------------------------------------------------------------------------*/

/*
| Ctrl-signal to inverting-register map, following the original logic.
```

```
*/
static const byte Ctrl_Val[16] = {
    0xb, 0xa, 0x9, 0x8, 0xf, 0xe, 0xd, 0xc,
    0x3, 0x2, 0x1, 0x0, 0x7, 0x6, 0x5, 0x4
};

void verbose_latch(byte val, byte ctrl)
{
    unsigned ctrl_enable = ctrl | LATCH_ENABLE_MASK;

    Output_Window(FALSE);
    outportb(D_Reg, val);                       /* Value onto data bus. */
    cprintf( EOL"Write %s data.", dec_bin(val, 8) );
    cpause();

    outportb(C_Reg, Ctrl_Val[ctrl] );           /* Select desired latch */
    cprintf("  ctrl: %s", dec_bin(ctrl, 4));
    cpause();

    outportb(C_Reg, Ctrl_Val[ctrl_enable] );    /* Enable the latch;    */
    cprintf("    : %s", dec_bin(ctrl_enable, 4));
    cpause();

    outportb(C_Reg, Ctrl_Val[ctrl] );           /* Drop the enable bit. */
    cprintf("    : %s\n", dec_bin(ctrl, 4));
}


/*-------------------------------------------------------------*\
| Reset & start conversion on ADC(s).                           |
| 93-05-25: No delay needed between reset & pulse --- ftn-call  |
|       overhead gives the single-pulsers plenty of time,       |
|       given the 78.1KHz/12.8us ADCclock  (meas. by Fluke 97)  |
\*-------------------------------------------------------------*/
void local_adc_strobe(byte adcbits)
{
    write_device(ADC_CONV, 0);          /* Reset the ADC single-pulsers */
    write_device(ADC_CONV, adcbits);    /* Pulse all desired /SC lines  */
}
```

```
/*-------------------------------------------------------------------*\
| Expanded local version of ADC_sample...                            |
\*-------------------------------------------------------------------*/
word verbose_ADC_sample(byte adc_num)
{
    word tmp;

    adc_strobe( ADC_trgr(adc_num) );    /* Trigger the ADC sync/cvt/SC  */
    cpause();

    latch_ADC_out( adc_num );   /* Latch the ADC onto the Blue Bus...   */
    cpause();

    tmp = ADC_bus();            /* read 3 nybbles thru Control Port;    */
    cpause();

    reset_ADCs();

    return tmp;
}


/*--------------------------*\
| Intercept ctrl_C interrupts |
\*--------------------------*/
static int ctrl_C(void)
{
    flushall();
    audible(1100);
    show_prompt("  ARGghhhh...  ", LIGHTRED);
    gotoxy(1,25);
    return 0;
}


/*****************************************************\
*                                                     *
```

```
*       utility functions above, menu options below     *
*                                                       *
\*******************************************************/


/*********************\
* ATB-control options *
\*********************/

static const byte Addr_Map[16] = {
    0x0, 0x2, 0x1, 0x3, 0x8, 0xa, 0x9, 0xb,
    0x4, 0x6, 0x5, 0x7, 0xc, 0xe, 0xd, 0xf
};

#define address(addr)  \
    ( (Addr_Map[ (addr & 0xf0) >> 4 ] << 4)  |  Addr_Map[ addr & 0x0f ] )

static void set_Addr_bus(void)
{
    word addr;
    addr = get_word_num("device address?");
    Output_Window(FALSE);
    cprintf(EOL"%02x maps to ", addr);
    addr = address(addr);
    cprintf("%02x; addr: ", addr);
    verbose_latch(addr, A_LATCH);
    putch('\n');
}

/*-------------------------------------------------------------------------*/
static void set_busses(void)
{
    word addr, data;

    addr = get_word_num("device address?");
    data = get_word_num("(Low-1st) Data-bus value?");
    Output_Window(FALSE);
    cprintf( EOL"%s low", dec_bin(lo(data), 8) );
    cprintf( ", %s high", dec_bin(hi(data), 8) );
```

```
    cprintf( ";  Addr %s\n", dec_bin(addr, 8) );
    set_data( data );
    set_addr( addr );
}

/*-------------------------------------------------------------------------*/
static void do_c(void)
{
    word data;

    data = get_word_num("Control-register value?");
    Output_Window(FALSE);
    cprintf( EOL"%s maps to Ctrl pattern: ", dec_bin(data, 4) );
    data = Ctrl_Val[data];
    cputs(dec_bin(data, 4));
    putch('\n');
    outportb( C_Reg, data );
}

/*-------------------------------------------------------------------------*/
static void do_d(void)
{
    word data;
    data = get_word_num("Data-bus value?");
    set_data( data );
}

/*-------------------------------------------------------------------------*/
static void do_D(void)
{
    word data;
    data = get_word_num("Data-bus value to write low-1st?");
    Output_Window(FALSE);
    cputs(EOL"low:  "); verbose_latch(lo(data), D_LO_LATCH); cpause();
    cputs(EOL"high: "); verbose_latch(hi(data), D_HI_LATCH);
    putch('\n');
}

/*-------------------------------------------------------------------------*/
```

```
static void do_g(void)
{
    trigger_all_ADCs();
    show_prompt("trigger_all_ADCs() to convert.", NoColor);
}

/*-------------------------------------------------------------------------*\
| Repetitively send values to the chip, and read a result.                  |
\*-------------------------------------------------------------------------*/
static void wr_loop(void)
{
    word src, rslt, dac_addr, adc_addr;
    char bfr[81];
    int ans;

    /*---------------------------*\
    | Set up for a particular DAC |
    \*---------------------------*/
    dac_addr = get_word_num("Input DAC?");
    src = get_word_num("Data to write?");

    /*-----------------------*\
    | Identify the output ADC |
    \*-----------------------*/
    adc_addr = get_word_num("ADC [0..5]?");

    if (std_out) {
        cputs(out_fmt_msg);
        ans = ('y' == getche());
    } else
        ans = 0;

    sprintf(bfr, EOL"DAC $%x  to  ADC $%x\n", dac_addr, adc_addr);
    Output_Window(TRUE);
    con_log_string(bfr);

    do {
        set_data( ~src );               /* flip the bits on the Red bus */
        write_device(dac_addr, src);
```

```
        trigger_DACs();                 /* trigger the DACs */
        u_time(1000);                   /* wait 1 milliseconds... */
        rslt = ADC_sample(adc_addr);

        cprintf("\r%c  %s --> ", marker(), dec_bin(src,12));
        cprintf( "%s  (%4u --> %4u)", dec_bin(rslt,12),  src, rslt );
        rslt_log(src, rslt, ans);
    } while (!kbhit());
    getch();
    newline();
}


/*-------------------------------------------------------------------------*\
| Output a succession of signals to one or more DACs, and read an ADC's |
| responses.                                                              |
\*-------------------------------------------------------------------------*/
typedef struct _addr_node {
    word addr;
    struct _addr_node *next;
} addr_node;

static void ramp_DACs(addr_node *addrs)
{
    word rampout, data, result, step;
    addr_node *this;
    char bfr[256], c;
    int ans, pos;

    rampout = get_word_num("output ADC [0..5]?");
    step = get_word_num("Stepsize ( >100 suggested )?");
    if (std_out) {
        cputs(out_fmt_msg);
        ans = ('y' == getche());
    } else
        ans = 0;

    this = addrs;
    pos = sprintf(bfr, EOL"signal DAC: $%x", this->addr);
```

```
    this = this->next;
    while ( (this != 0) && (pos < 243) ) {
        pos += sprintf(bfr+pos, ", $%x", this->addr);
        this = this->next;
    }
    sprintf(bfr+pos, "; Response ADC: $%x\n", rampout);

    show_prompt(step_msg, LIGHTBLUE);
    Output_Window(TRUE);
    con_log_string(bfr);
    c = getch();

    for (data = 0; data < NUM_COUNTS; data += step) {
        this =addrs;
        do {
            write_DAC(this->addr, data);
            this = this->next;
        } while (this != 0);
        trigger_DACs();
        u_time(1000);           /* Let the test circuit stabilize.. */
        result = ADC_sample( rampout );

        rslt_log(data, result, ans);
        if ( (1 != std_out) || ('C' != c) ) {
            cprintf(EOL"input: $%3x / %s " LBLOK " output: $%3x / ",
                        data, dec_bin(data,12), result);
            cputs(dec_bin(result,12));
        } else {
            cprintf("\r%c", marker());
        }
        if ('C' != c) {
            c = getch();
        }
    }
    newline();
    Prompt_Window();
}

/*-------------------------------------------------------------------*\
```

```
| Ramp one DAC only.                                                    |
\*--------------------------------------------------------------------*/
static void ramp_1_DAC(void)
{
    addr_node one;
    one.next = 0;
    one.addr = get_word_num("Ramp DAC #?");
    ramp_DACs(&one);
}

/*--------------------------------------------------------------------*\
| Ramp multiple DACs.                                                   |
/*--------------------------------------------------------------------*/
static void ramp_multiple_DACs(void)
{
    addr_node head, *this, *last;
    int addr;

    this = &head;
    addr = get_word_num("Ramp DAC #?");
    do {
        this->addr = addr;
        addr = get_word_num("Another DAC #? [-1 to quit]");
        if (addr == -1) {
            break;
        }
        this->next = (addr_node *)malloc(sizeof(addr_node));
        this = this->next;
    } while (1);
    this->next = 0;

    ramp_DACs(&head);
    this = head.next;
    while (this != 0) {
        last = this;
        this = this->next;
        free(last);
    }
}
```

```
/*-------------------------------------------------------------------*/
static void do_readADC(void)
{
    word addr, data;

    addr = get_word_num("ADC [0..5 or 10..15]?");
    if (addr <= 5) {
        data = ADC_sample( addr );
    } else {
        addr -= 10;
        data = verbose_ADC_sample( addr );
    }
    Output_Window(FALSE);
    cprintf(EOL"ADC %d reports $%03x (%s)\n",addr,data,dec_bin(data,12));
}

/*-------------------------------------------------------------------*/
static void do_readblue(void)
{
    word blue;
    blue = ADC_bus();
    Output_Window(FALSE);
    cprintf( "Blue (ADC) bus reports %d/$%s/%%",
             blue,val_to_ASCII(blue,16,3) );
    cputs(val_to_ASCII(blue,2,12));
}

/*-------------------------------------------------------------------*/
static void do_reset(void)
{
    reset_ADCs();
    show_prompt("Reset'd all ADCs.", NoColor);
}

/*-------------------------------------------------------------------*/
static void do_t(void)
{
    show_prompt("issuing XFER/trigger to all DACs", NoColor);
```

```
    trigger_DACs();
}

/*-------------------------------------------------------------------------*/
static void do_T(void)
{
    DAC_convert_new();
    Prompt_Window();
    cprintf( "$%02x gets new-convert signal;  ", DAC_XFER );
    cpause();

    hold_DAC_conversion();
    Prompt_Window();
    cprintf( "$%02x gets hold signal.\r", DAC_XFER );
}

/*-------------------------------------------------------------------------*/
static void do_V(void)
{
    char bfr[82];
    float v;
    Output_Window(FALSE);
    audible(550);
    sprintf(bfr, "\rCurrently:  Vss=%.3f, Vdd=%.3f.\r\nNew Vss? ", Vss, Vdd);
    if ( con_read(bfr, bfr, 82) ) {
        putch('\r');
        v = atof(bfr+2);
        if (v <= 0) {
            Vss = v;
            adjust_Vss(Vss);
        } else {
            audible(1200);
            cputs("Bad Value!"EOL);
        }
    }
    audible(550);
    if ( con_read("\rNew Vdd? ", bfr, 82) ) {
        putch('\r');
        v = atof(bfr+2);
```

```
        if ( (10 >= v) && (v >= 0) ) {
            Vdd = v;
            adjust_Vdd( Vdd );
        } else {
            audible(1200);
            cputs("Bad Value!"EOL);
        }
    }
    sprintf(bfr, EOL"Now:  Vss=%.3f, Vdd=%.3f.\n", Vss, Vdd);
    con_log_string(bfr);
}

/*-------------------------------------------------------------------*/
static void wrt_a_DAC(void)
{
    word addr, data;
    char bfr[81];

    addr = get_word_num("DAC address?");
    data = get_word_num("Data to write?");
    write_device(addr, data);
    Output_Window(FALSE);
    sprintf(bfr, EOL"DAC $%02x gets $%03x  (%s)\n",
            addr, data, dec_bin(data, 12));
    con_log_string(bfr);
}

/*-------------------------------------------------------------------*/
static void w_t_DAC(void)
{
    wrt_a_DAC();
    do_t();
}

/*-------------------------------------------------------------------*/
static void wrt_each_DAC(void)
{
    word addr, data;
    char bfr[82], dacstr[32];
```

```
    Output_Window(FALSE);
    cputs(EOL"Write each DAC.  Nil entry to stop."EOL);
    for (addr = 0x00; addr <= 0x23; ++addr) {
        sprintf(dacstr, "DAC $%02x value? ", addr);
        if (!con_read(dacstr, bfr, 82)) {
            return;
        }
        data = atoi(bfr+2);
        write_DAC( addr, data );

        sprintf( bfr, " $%02x gets %u (%04x, %s)"EOL,
                addr, data, data, dec_bin(data, 12) );
        con_log_string(bfr);
    }
}

/*-------------------------------------------------------------------------*/
static void wrt_all_DACs(void)
{
    word addr, data;
    char bfr[82];

    Output_Window(FALSE);
    data = get_word_num("global DAC value?");
    for (addr = 0x00; addr <= 0x23; ++addr) {
        write_DAC( addr, data );                /* Set to uniform value */
    }
    sprintf( bfr, " DACS $00..$23 get %u (%04x, %s)"EOL,
                data, data, dec_bin(data, 12) );
    con_log_string(bfr);
}


/*-------------------------------------------------------------------------*/
static void do_1(void)
{
    wrt_a_DAC();                /* Write a specified DAC */
    do_t();                     /* (trigger the DACs)    */
```

```
    do_readADC();               /* Read a specified ADC  */
}


/*******************************\
* Program-control, misc. options *
\*******************************/

/*---------------------------------------------------------------------------*/
static void null_ftn(void) {}

static void do_null(void)
{
    char bfr[82], ans[6];
    unsigned long tym;
    unsigned long v;
    float utymf, vf;
    word blue;

    Output_Window(FALSE);
    if (!con_read("\rftn/bus/empty loop [fbe]?", ans, 6))
        return;

    con_read("How many loops ( >= 1 ) ? ", bfr, 82);
    vf = atof(bfr+2);
    v = atol(bfr+2);
    switch (ans[2]) {
      case 'f':
        cputs("Null_ftn():");
        tym = millisecs();
        do {
            null_ftn();
        } while (--v != 0);
        tym = millisecs() - tym;
        break;
      case 'b':
        cputs("Blue bus read:");
        tym = millisecs();
        do {
```

```
            blue = ADC_bus();
        } while (--v != 0);
        tym = millisecs() - tym;
        cprintf( "  Blue (ADC) bus reports %d/$%s/%%",
                blue, val_to_ASCII(blue,16,3) );
        cprintf("%s", val_to_ASCII(blue,2,12));
        break;
      case 'e':
        cputs("Empty loop:");
        tym = millisecs();
        do {
        } while (--v != 0);
        tym = millisecs() - tym;
        break;
    }
    utymf = 1000.0 * (float)tym;
    sprintf(bfr, EOL" %lu msec, %f usec/iteration\n", tym, (utymf/vf) );
    con_log_string(bfr);
}

/*-------------------------------------------------------------------------*/
static void do_bases(void)
{
    char str[40], bfr[127];
    int pos;
    float val;
    long unsigned lua;

    Output_Window(FALSE);
    con_read("\rValue? ", str, 38);
    val = cvt_val(str+2);
    lua = (long unsigned)val;

    pos = sprintf(bfr, "%s = %#g " VBAR " %lu " VBAR " %%%s",
                str+2, val, lua, val_to_ASCII(lua, 2, 0));
    pos += sprintf((bfr+pos), " " VBAR " @%s", val_to_ASCII(lua, 8, 0));
    sprintf((bfr+pos), " " VBAR " $%s\n", val_to_ASCII(lua, 16, 0));
    clreol(); con_log_string(bfr);
}
```

```
/*---------------------------------------------------------------------*/
static void do_system(void)
{
    show_prompt("DOS shell...", LIGHTGRAY);
    textcolor(BLACK);
    system("");
    Prompt_Window();
}

/*---------------------------------------------------------------------*\
| Toy commands.                                                         |
\*---------------------------------------------------------------------*/
static void do_escape(void)
{
    hop_window(1, TRUE);        /* copyright notice... */
    cputs(title);
    audible(150);
    show_prompt("no escape.", BROWN);
}

static void do__(void)
{
    show_prompt("PLEASE DON'T DO THAT.", LIGHTRED);
}

/*---------------------------------------------------------------------*\
| Toggle the echo-to-output-file status.  The global variable "std_out" |
| keeps track of this status, for reference by all routines             |
\*---------------------------------------------------------------------*/
static void do_redirect(void)
{
    static char name_buf[81];
    char *name, *m;
    fpos_t len;

    name = name_buf + 2;
    Prompt_Window();
```

```
    if (std_out) {

        audible(220); audible(220);
        len = ftell(outfile);
        if (len) {
            fprintf(outfile, "\n# %s\n", time_now());
            fclose(outfile);
            m = "Closed";
        } else {
            fclose(outfile);
            unlink(name);
            m = "Discarded";
        }
        std_out = 0;

    } else {

        audible(700);
        if ( 0 == con_read("output filename? ", name_buf, 81) ) {
            name[0] = 0;
            m = "No";
        } else {
            if ( NULL == (outfile = fopen(name, "w")) ) {
                audible(990);
                m = "Can't fopen()";
            } else {
                std_out = 1;
                m = "Echoing to";
            }
        }
    }
    cprintf(EOL"%s file %s", m, name);
    File_Window();
}

/*---------------------------------------------------------------------*/
static void fix_noise(void)
{
    noisy = !noisy;
```

```
    audible(660);
}

/*-------------------------------------------------------------------------*/

void draw_menu(void)
{
    hop_window(1, TRUE);        /* draw the menu... */
    cputs(menu);
}

/*************************************************************************/

int main(int argc, char **argv)
{
    unsigned i;
    char sec, ans;
    char *oldscreen;
    int ctrlbrk_setting = getcbrk();

    (void)argv;(void)argc;      /* suppress "never used" messages */

    setcbrk(1);
    ctrlbrk(ctrl_C);
    chirp(0);

    oldscreen = (char *)malloc(2 + 2*( (RIGHT-LEFT+1) * (BOTTOM-TOP+1) ));
    oldscreen[0] = (char)wherex();
    oldscreen[1] = (char)wherey();
    gettext(LEFT,TOP, RIGHT,BOTTOM, oldscreen+2);

    draw_frame();
    File_Window();
    hop_window(1, TRUE);        /* copyright notice... */
    cputs(title);
    Prompt_Window();

LOOP: {
        flushkeys();
```

```
        audible(440);
        sec = -1;       /* unlikely second (for a few years, anyway) */

        Time_Window();
        do {
            if (sec != (time_now())[24]) {
                gotoxy(2,1);
                textcolor(LIGHTGREEN); cputs(time_now());
                textcolor(YELLOW); cputs(" choice? ");
                textcolor(WHITE); sec = time_now()[24];
            }
        } while (!kbhit());
        ans = getche();
        Prompt_Window();

        if ( ('q'== ans) )
            goto QUIT;

        for (i = 0; i < n_choices; ++i) {
            if (ans == choice[i].tag) {
                choice[i].f();
                i += n_choices;
            }
        }
        if (n_choices == i) {
            textcolor(LIGHTRED);
            cprintf("`%c' is unknown.", ans);
            audible(1760); audible(440); audible(860); audible(1660);
        }

    } goto LOOP;

QUIT:
    if (std_out)
        do_redirect();

    audible(1000);
    window(1,1,80,25);
    puttext(LEFT,TOP, RIGHT,BOTTOM, oldscreen+2);
```

```
    gotoxy((int)oldscreen[0], (int)oldscreen[1]);
    free(oldscreen);
    setcbrk(ctrlbrk_setting);
    return 0;
}
/*------------------------------  END  ---------------------------------*/
```

A.3 Utility Functions

These files contain functions unrelated to ATB operation, but used by the `ftntest` program for timekeeping, numeric conversions, and other "bells and whistles".

```
/*-------------------------------------------------------------------*\
|  CONVERTS.H --- Functions to convert values back and forth.         |
| 93-11-10  cvt_val() added; allows for conversion to floating point. |
|           Calls numeric() if floating-point fails.                  |
| 93-04-22  legal_digits[] moved to here, as part of generalized      |
|           "spacers" support and use of lookup().  to_ascii() and    |
|           to_numer() also modified.  Set up machinery for maintaining |
|           allowable spacers as used by to_ascii() and to_numer().   |
| 93-03-25  Support conversion of "negative" strings to negative values |
\*-------------------------------------------------------------------*/
#if !defined(_CONVERTS_H)
#define _CONVERTS_H

#if !defined(NULL)
#   define NULL  0
#endif

#define ASCII_MAX_LEN   81    /* Longest possible string, +1 */
#define legal_digits    "0123456789abcdef"
#define legal_size      (sizeof(legal_digits) - 1)

        /* Convert a string to a floating-point value if possible;   *\
        \* if not, pass the string on to numeric().                  */
double cvt_val(char *strng);

        /* Determine the base of a # string, and convert it to an    *\
        \* integer.  '$', '@', '%' prefixes accepted.                */
long numeric(char *strng);

        /* Convert ASCII-string to integer, in the given base.       *\
        \* Tolerate underscores or commas, e.g., "a_4317", "123,456" */
long ASCII_val(char *strng, int base);

        /* Convert unsigned long integer to ASCII string with a      *\
        |* spacer char separating groups of digits:  viz., "1_0010", *|
```

```
        |* "7,324".  The first character in the "spacers" string is  *|
        |* used.  set_spacers() controls this; set a null string to  *|
        |* get no spacer.  'base' specifies the desired base of the  *|
        |* converted string.  The 'len' specifies *minimum* string   *|
        \* size.  Char string changes w/ each use.                   */
char *val_to_ASCII(unsigned long value, int base, unsigned len);

#define bin_dec(bits)           ASCII_val((bits), 2)
#define dec_bin(value, len)     val_to_ASCII(value, 2, len)
#define to_ascii(value, base)   val_to_ASCII(value, base, 0)

        /* Define acceptable spacer characters in inputs. */
char *set_spacers(char *list);

#define DEFAULT_SPACERS "_,'"
#if defined(_SPACER_C)
    char *spacers = DEFAULT_SPACERS;
    int   spacer_ct = sizeof(DEFAULT_SPACERS)-1;
#else
    extern char *spacers;
    extern int   spacer_ct;
#endif

/*-----------------------------*\
| My version of index()/strpos() |
\*-----------------------------*/
int lookup(int c, const char *list);
#define stringlen(STRING)  lookup(0,STRING)

void chirp(unsigned freq);

#define flushkeys() {while (kbhit()) getch();}
/*--------------------------------------------------------------------*/
#endif  /* _CONVERTS_H */
#if 0 /* LaTeX formatting commands here... */
```

```
#endif
/*-------------------------------------------------------------------------*\
| timing.h --- microsecond and millisecond functions.                       |
| 93-03-20                                                                  |
\*-------------------------------------------------------------------------*/
#if !defined(_TIMING_H)
#define _TIMING_H

/*---------------------------------------------*\
| Day-date string in my preferred format, and   |
| timing functions in units of milliseconds.    |
\*---------------------------------------------*/
char *time_now(void);             /*  Current time-&-date char-string   */
unsigned long millisecs(void);    /*  Number of msec since 1970-01-01   */
char *duration(unsigned long ms); /*  Millisecs-as-minutes char-string  */

extern const char *day[7];
extern const char *month[12];

/*-------------------------------------------------------------*\
| u_time():  Burn microseconds.  Accuracy is limited by the     |
| machine speed; a 25MHz '386 may be good to one or a few       |
| microseconds.  However, times under a dozen usec or so will   |
| be dominated by the function-call overhead anyway.  (Still |
| better than the delay() function's millisecond resolution).   |
| Default calibration is for my generic 25MHz no-cache 386box;  |
|generally a recalibration should be done before use.   |
|                                                               |
| u_time_calibrate() allows recalibrating for a fixed time      |
| interval; u_calibrate() gives a 5-second recalibration run.   |
\*-------------------------------------------------------------*/
void u_time(float usec);
void u_time_calibrate(long unsigned cal_usecs);
#define u_calibrate()   u_time_calibrate(5000000L)

/*-------------------------------------------------------------------------*/
#endif  /* _TIMING_H */
#if 0 /* LaTeX formatting commands here... */
```

```
#endif
/*-------------------------------------------------------------------------*\
| spacer.c --- Set up spacer characters for to_ascii() and to_numer()       |
|                                                                           |
| 93-04-22                                                                  |
\*-------------------------------------------------------------------------*/
#include <string.h>
#define _SPACER_C
#include "converts.h"

char *set_spacers(char *list)
{
    if (NULL != list)
        spacers = list;
    spacer_ct = strlen( spacers );
    return spacers;
}
#if 0 /* LaTeX formatting commands here... */
```

```
#endif
/*-------------------------------------------------------------------------*\
| TO_ASCII.C --- convert numeric values to ASCII strings.                   |
|                                                                           |
| Generate the char-string form of a supplied value, using the supplied     |
| base.  To improve readability, strings contain a supplied character       |
| (ex.: underscore) every 3 characters (bases 8, 10) or 4 characters        |
| (all other bases).  This can be suppressed by supplying a NULL            |
| character (0).                                                            |
|                                                                           |
| 'len' specifies the *minimum* length of the returned string;              |
| ASCII_MAX_LEN (in coverts.h) defines the maximum length.                  |
| NOTE:  The returned char string is changed on every call.                 |
|                                                                           |
| Example:  binary strings of the form "1_0101_0000_1001_0000"              |
|                                                                           |
| 92-11-03 - Replace dec_bin() with a call to val_to_ASCII                  |
| 93-02-02 - Jazz up the comments, discard the ifdef'd dec_bin(),           |
|            move ASCII_MAX_LEN to converts.h                               |
\*-------------------------------------------------------------------------*/
#include <stdlib.h>
#include <string.h>
#include "converts.h"

static char near S_String[ASCII_MAX_LEN];

char *val_to_ASCII(unsigned long value, int base, unsigned len)
{
    char near *s, spacer;
    int near spacing, chunk;

    if (base >= sizeof(legal_digits))   /* Error-check: requested */
        return NULL;                    /*   base out of range?   */

    spacer = spacers[0];
    if (spacer) {
        switch (base) {                 /* If a spacer character */
          case 8:                       /* is requested, how     */
          case 10:                      /* frequently should it  */
```

```
            chunk = 3;                  /* be added?            */
            break;
          default:
            chunk = 4;
        }
        spacing = chunk;
    }

    *(s = S_String + ASCII_MAX_LEN - 1) = 0;    /* NULL the end-of-string */
    do {
        if (spacer) {
            if (0 == spacing) {                 /* Time to add a */
                *(--s) = spacer;                /* spacer char.? */
                spacing = chunk;
            }
            --spacing;
        }

        *(--s) = legal_digits[(unsigned)(value%base)];  /* each digit... */
        value /= base;
        if (len)                                        /* At requested */
            --len;                                      /* length yet?  */
    } while ( (value || len) && (s > S_String) );

    return s;
}
/*------------------------------  END  -------------------------------*/
#if 0 /* LaTeX formatting commands here... */
```

```
#endif
/*-------------------------------------------------------------------------*\
|  TO_NUMER.C --- convert ASCII strings into values.                       |
|       This code supports underscores or commas (for readability) in      |
|       number strings; these separators are (the most common?) options    |
|       for visual formatting that to_ascii() does.                        |
|                                                                          |
| 93-11-10 - cvt_val() for floating- & general integer-format operands.    |
| 93-07-16 - numeric() strips leading whitespace.  ASCII_val() stops       |
|       when it hits nondigits, so these can parse a line.                 |
| 93-04-22 - Remove strtol() usage, for no very good reason.               |
|       Hack up the treatment of number-string spacers, for mucho          |
|       generality via the new lookup() function.  Also no very            |
|       good reason.  Test code removed to a separate file.                |
| 93-03-25 - Handle negative numbers in the form                           |
|               [<sign>][<base-marker>]<digits>                            |
|            where <sign> and <base-marker> are optional.                  |
| 92-11-03 - Abandon the redundant special cases.                          |
\*-------------------------------------------------------------------------*/
#include "converts.h"

double cvt_val(char *strng)
{
    double v;
    if ( 0 == (v = atof(strng)) ) /* Does atof() return 0 on failure?? */
        v = numeric(strng);
    return v;
}

/*-------------------------------------------------------------------------*\
| Determine the base of a # string, and convert it to an integer.          |
| 93-07-16 Strip leading blanks the UGLY way.  This is a good example      |
|       of tail recursion, which is optimized into a "goto", yes that's    |
| right, a "goto".  (cf. '-', which isn't tail recursive.)                 |
\*-------------------------------------------------------------------------*/
long numeric(char *strng)
{
What_Kind:
    switch (*strng) {
```

```
      case '%':                                 /* binary w/underscores */
        return ASCII_val( strng+1, 2 );

      case '@':                                 /* octal w/underscores  */
        return ASCII_val( strng+1, 8 );

      case '$':                                 /* hex w/underscores    */
        return ASCII_val( strng+1, 16 );

      case '-':                                 /*  Recurses!           */
        return ( - numeric(strng+1) );          /*   Negative value...  */

      case '0':                                 /* C syntax:            */
        if ('x' == strng[1])
            return ASCII_val( strng+2, 16 );    /* 0x-hex, underscores  */
        else
            return ASCII_val( strng+1, 8 );     /* 0-octal, underscores */

      case '\t':
      case ',':
      case ' ':                                 /* Tail Recurses!       */
        ++strng;                                /* skip a whitespace,   */
        goto What_Kind;                         /* try again...         */

      default:                                  /* assume it's decimal! */
        return ASCII_val( strng, 10 );
    }
}

/*-------------------------------------------------------------------*\
| Convert an ASCII string to a long integer, in a given base.         |
| Bases through hexadecimal are accepted, using [0..9a..f] (or [A..F]). |
| A leading '-' indicates a negative value.  Strings may contain      |
| spacing characters; other characters terminate the conversion.      |
\*-------------------------------------------------------------------*/
long ASCII_val(char *strng, int base)
{
    unsigned long near value;
    register char near c;
```

```
    register int near v;
    int near neg;

    if ('-' == *strng) {
        ++strng;
        neg = 1;
    } else {
        neg = 0;
    }
    value = 0;
    while ( 0 != (c = *(strng++)) ) {
      /*-------------------------------------------*\
      | Look for legal digits, possibly uppercase... |
      \*-------------------------------------------*/
        if ( legal_size > (v = lookup(c, legal_digits))
             ||  legal_size > (v = lookup(c|0x20, legal_digits)) )
        {
            value = value*base + v;
            continue;
        }
      /*----------------*\
      | Cosmetic spacer? |
      \*----------------*/
        if (spacer_ct > lookup(c, spacers))
            continue;

      /*-----------------------------------------------------*\
      | If we got here, we must've hit the ASCII string's end. |
      \*-----------------------------------------------------*/
        break;
    }

    return (neg  ?  -value  :  value);
}
/*------------------------------  END  --------------------------------*/
#if 0 /* LaTeX formatting commands here... */
```

```
#endif
/*-------------------------------------------------------------------------*\
| timeftns.c --- reinventing the wheel...                                   |
\*-------------------------------------------------------------------------*/
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "timing.h"

/*---------------------------------------------------*\
|  Return the number of milliseconds since 1970-01-01   |
\*---------------------------------------------------*/
#include <sys/timeb.h>

unsigned long millisecs(void)
{
    struct timeb t;
    ftime(&t);
    return ( (unsigned long)1000*t.time + (unsigned long)t.millitm );
}

/*---------------------------------------------------*\
|  convert milliseconds to minutes-etc. char. string.   |
|  Overwritten by each call...                          |
\*---------------------------------------------------*/
#define MS_PER_HR  3600000L
#define MS_PER_MIN   60000L
#define MS_PER_SEC    1000L

static char minute[40];

char *duration(unsigned long ms)
{
    unsigned long near i;

    minute[0] = 0;
    if ( 0 != (i = ms / MS_PER_HR) ) {
        sprintf(minute, "%uh", i);
        ms %= MS_PER_HR;
```

```
    }
    if ( 0 != (i = ms / MS_PER_MIN) ) {
        sprintf(minute+strlen(minute), "%um", i);
        ms %= MS_PER_MIN;
    }
    i = ms / MS_PER_SEC;
    ms %= MS_PER_SEC;
    sprintf(minute+strlen(minute), "%lu.%03lus", i, ms);
    return minute;
}

/*-----------------------------------------------------*\
|  Return current-time-and-date character string.       |
|  Overwritten by each call...                          |
\*-----------------------------------------------------*/
const char *day[7] = {
    "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat",
};
const char *month[12] = {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};

static char time_string[26];

char *time_now(void)
{
    time_t tymt;
    struct tm *tym;
    tymt = time(NULL);
    tym = localtime(&tymt);
    sprintf(time_string,
                "%04u %3s %2d, %3s %2u:%02u:%02u",
                1900+tym->tm_year, month[tym->tm_mon], tym->tm_mday,
                day[tym->tm_wday], tym->tm_hour, tym->tm_min, tym->tm_sec);
    return time_string;
}

/*-----------------------------------------------------------------*/
```

```
#ifdef TEST_MILLI
void test_milli(void)
{
        long i;

        printf("TIME==>%s<==\n", time_now());
        puts(duration(millisecs()));
        for (i=0; i<15000; i +=1271) {
                puts(duration(i));
        }
}
#endif
/*------------------------------  END  -------------------------------*/
#if 0 /* LaTeX formatting commands here... */
```

```
#endif
/*-------------------------------------------------------------------------*\
| utimer.c --- provide sub-millisecond delays via software loops.           |
| 93-03-20                                                                  |
\*-------------------------------------------------------------------------*/
#include <dos.h>
#include "timing.h"

static float near loops_per_usec = 0.562;  /* An informed starting guess... */

/*-------------------------------------------------------------*\
| Burn the requested microseconds by spinning a null loop       |
| "enough" times, as determined by the "loops_per_usec"         |
| calibration factor.                                           |
\*-------------------------------------------------------------*/
void u_time(float usec)
{
    register long unsigned near loops;
    loops = loops_per_usec * usec;
    if (!loops)
        return;
    do {} while (--loops);
}

/*-------------------------------------------------------------*\
| Run a fixed number of loops, count the number of reported     |
| centiseconds between start and finish, and convert to a       |
| floating-point loops/microsecond calibration factor.          |
\*-------------------------------------------------------------*/
void u_time_calibrate(long unsigned cal_usecs)
{
    struct time ts, tf;
    float near usecs;

    do {
        gettime(&ts);
        u_time(cal_usecs);
        gettime(&tf);
    } while (ts.ti_hour > tf.ti_hour);
```

```
    usecs = 1.0e4 * (float)(
        100L * ((3600L * ((long)tf.ti_hour - (long)ts.ti_hour)) +
                (  60L * ((long)tf.ti_min  - (long)ts.ti_min )) +
                (        ((long)tf.ti_sec  - (long)ts.ti_sec ))  )
        + (long)tf.ti_hund - (long)ts.ti_hund );
    loops_per_usec *= (float)cal_usecs / usecs;
}

/*-------------------------------------------------------------------------*/
#ifdef TEST_MICRO
#include <stdio.h>

void test_micro(void)
{
    struct time ts, tf;
    unsigned long sample;
    fprintf(stderr,"\nUTIMER starting calibration: %f\n", loops_per_usec);
    u_time_calibrate(20000000L);
    fprintf(stderr,"New loops per usec: %f\n", loops_per_usec);

    for (sample = 1.0e4; sample <= 1.0e7; sample *= 10.0) {
        gettime(&ts);
        u_time(sample);
        gettime(&tf);
        fprintf(stderr,
                "%lu microseconds ---\n"
                " start: %2u:%02u:%02u.%02u\n"
                "finish: %2u:%02u:%02u.%02u\n",
                sample,
                ts.ti_hour, ts.ti_min, ts.ti_sec, ts.ti_hund,
                tf.ti_hour, tf.ti_min, tf.ti_sec, tf.ti_hund);
    }
}
#endif
/*-----------------------------  END  ------------------------------*/
#if 0 /* LaTeX formatting commands here... */
```

```
#endif
#include <stdlib.h>
#include <dos.h>

/*-------------------------------------------------------------------------*\
| chirp --- make a controlled noise.                                        |
\*-------------------------------------------------------------------------*/
void chirp(unsigned freq)
{
    sound(0);  nosound();  sound(freq); delay(10); nosound();
}

/*-------------------------------------------------------------------------*\
| lookup                                                                    |
|       Find the position of a character in a string.                       |
|       Returns: 0-based position of character within string; or            |
|                Length of string if no match or character is '\0'; or      |
|                -1 if a null string is supplied.                           |
|                                                                           |
|       stringlen() can be implemented in terms of lookup, as:              |
|                                                                           |
|       #define stringlen(foo)  lookup( 0, foo )                           |
|                                                                           |
| 93-04-22-bob,mon.                                                         |
\*-------------------------------------------------------------------------*/
int lookup(int c, const char *list)
{
    int i, lpos;
    i = -1;

    if (NULL != list) {
        do {
            lpos = list[ ++i ];
            if (c == lpos)
                return i;
        } while (NULL != lpos);
    }
    return i;
}
```

```
/*-------------------------------------------------------------------*/
#if defined(TEST)
#include <stdio.h>
const char alphabet[] =
    "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

int main(int argc, char **argv)
{
    printf("alphabet: %s\nsizeof alphabet = %d, stringlen(alphabet) = %d\n",
        alphabet, sizeof(alphabet), stringlen(alphabet));
    while (--argc) {
        printf("arg %d: '%s' is pos'n %d\n",
                argc, argv[argc], lookup(argv[argc][0], alphabet));
    }
    printf("'%c' & null string:  pos'n %d\n",  'A', lookup('A', 0));
    return 0;
}
#endif
/*-------------------------------------------------------------------*/
```

A.4 Makefile

This makefile expects the Borland "make v3.5" program. It makes the library of ATB routines, and the `ftntest` testing program.

```
#-----------------------------------------------------------------#
# makefile for ftntest.exe, atbs.lib, and ftntest.zip archive.
#
# 94-07-20 Distribution version. -bob,mon.
#-----------------------------------------------------------------#

#---------------------------------#
# Paths to the files under consideration...
# Fix these up as needed to find the tcc, tlink, & tlib programs
# and the \include and \lib subdirectories.
#
### ATBLIB = .\libatb
ATBLIB = .
WHERE = .
CPATH = c:\tc20

#---------------------------------#
# If a 'x87 coprocessor is available, change these to
# FPLIB = fp87
# FPFLAG = f87
# Change MDL only if a different memory model is really needed for
# some sick-puppy reason.
#
FPLIB = emu
FPFLAG = f
MDL = s

#-----------------------------------------------------------------#
# In theory, nothing beyond this point needs to be customized.
#-----------------------------------------------------------------#

CC = $(CPATH)\bin\tcc
TLINK = tlink
TLIB = tlib
```

```
INC = $(CPATH)\include
LIB = $(CPATH)\lib

CCFLAGS = -1 -O -Z -d -w -r -$(FPFLAG)
CFLAGS = $(CCFLAGS) -c

#---------------------------------#

FTNOBJS = ftntest.obj utility.obj
FTNLIBS = atbs.lib
FTNFLAGS = -N -w-stk

#---------------------------------#

ATBOBJS = $(ATBLIB)\atbpara.obj
ATBADDS =  -+atbpara

#---------------------------------#

ARCER = zip -u9rD
### ARCER = pkzip -ex -uorp
SRC_ARC = _ftntest.zip

#-----------------------------------------------------------------#

no_args:
    @echo make targets:
    @echo - "make clean" ......removes .obj files
    @echo - "make veryclean" ..removes .obj, .bak, .lst, .exe, .lib files
    @echo - "make archive" ....updates $(SRC_ARC)
    @echo - "make atb" ........make a library
    @echo - "make ftntest" ....make the portmanteau exercise program
    @echo .

#---------------------------------#

clean:
    -rm -f $(WHERE)\*.obj
    -rm $(WHERE)\*.bak
```

```
veryclean: clean
    -rm -i *.exe *.lib ../bin
    -rm -f *.obj *.bak *.lst *.lnk
    -rm -f $(WHERE)\*.obj $(WHERE)\*.bak $(WHERE)\*.lst

#--------------------------------#

archive:
    -rm *.obj *.bak *.map *.lst foo*.*
    -$(ARCER) $(SRC_ARC) mak* *.c *.h *.tc *.prj *.lib *.bat *.lst lib*.*
    @echo $(SRC_ARC) is updated.
    @echo .
    -@ls -l $(SRC_ARC)

#--------------------------------#

atb: atb$(MDL).lib

atb$(MDL).lib: $(ATBOBJS)
    $(TLIB) $(WHERE)\atb$(MDL) /E $(ATBADDS), $(WHERE)\atb$(MDL).lst

atbpara.obj: atbpara.h low-para.h timing.h converts.h

#--------------------------------#

ftntest: ftntest.exe

ftntest.exe: $(FTNOBJS) $(FTNLIBS)
    @echo Using $(FPLIB) f.p. library...
    $(TLINK) @&&!
/L$(LIB) c0s.obj $(FTNOBJS)
$&.exe
$&.map
$(FTNLIBS) $(FPLIB) maths cs
!
    @type $&.map
    @ls -l $<
    @-rm -f $&.map
```

```
ftntest.obj: ftntest.c low-para.h atbpara.h converts.h timing.h local.h
    $(CC) -m$(MDL) $(CFLAGS) $(FTNFLAGS) -I$(INC) $&

#---------------------------------#

utility.obj: utility.c local.h converts.h timing.h

#-----------------------------------------------------------------#

.c.obj:
    $(CC) -m$(MDL) $(CFLAGS) -I$(INC) {$< }

#-----------------------------------------------------------------#
```