1985

**Termux: a terminal multiplexor for the Cambridge Ring local area network - hardware**

Michael J. Milway  
*University of Wollongong, mjm@uow.edu.au*

Follow this and additional works at: [https://ro.uow.edu.au/compsciwp](https://ro.uow.edu.au/compsciwp)

**Recommended Citation**  

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au
The University of Wollongong

Department of Computing Science

Termux
A Terminal Multiplexor for the Cambridge Ring Local Area Network
- Hardware

Michael J. Milway

Abstract
The Terminal Multiplexor will enable several (up to ten) terminals to be connected to the Cambridge Ring Local Area Network via one Ring Node. This document describes the hardware for the first version of the Terminal Multiplexor. Some low level software and diagnostics are also described.
1. Introduction

The Department of Computing Science has installed a Cambridge Ring Local Area Network to link the department's computers and peripherals. The Ring offers more flexibility and is easier to expand than is the case when devices are connected by individual cables.

The Terminal Multiplexor project was originally a Master's project being conducted by Meng Fong under the supervision of Phillip McKerrow. However, he left early in its development and the project was handed to the department's support staff.

The Termux is designed to attach a number of terminals, printers or other peripherals to the ring via one Ring Node. Virtual links may be made around the ring between the peripheral and another device. i.e. a terminal may be connected to a host unix machine. Peripherals do not have to be permanently connected to one machine. Indeed it is desirable that any terminal on the ring may be randomly connected to either of the department's unix machines.

This document assumes that the reader is familiar with the Cambridge Ring, especially with the Ring Node and with the various VLSI chips (MC6809, MC6821, MC6828, MC6840, MC6844 and SY6551) used in the design of the Termux.

2. Design Considerations

The Termux is designed around the Motorola MC6800 microprocessor family using the MC6809 as its CPU. This family was chosen because there was already considerable design experience within the department with this family. The MC6809 was chosen because it was considered that this processor was sufficiently powerful to do the required job without the additional complexities of using a sixteen bit processor.

The MC6809 is an upgrade of the MC6800 microprocessor. It has a much better instruction set, extra indexed addressing modes and extra registers making it suitable for use with high level languages such as C.

It was decided to make the basic topology of the Termux similar to that of the Apple II computer. A mother board contains the CPU and much of the associated circuitry and a ten slot backplane is available for plug in memory and interface cards. Figure 1. shows a block diagram of the Termux.

The original specifications required that the Termux handle between eight and sixteen terminals. It was deemed that polling this many ACIAs (Asynchronous Communications Interface Adapters) every time an interrupt was serviced would take too much time. A Priority Interrupt Controller (PIC) is used to generate seven levels of interrupts. By spreading all the ACIAs between the PIC's interrupt levels polling is greatly reduced.

The Termux has DMA (Direct Memory Access) capability on its ring node interface. Once a link has been established through the ring to another device and the appropriate hand shaking taken place, data may be transferred under DMA control, thus speeding up the transfer.

The MC6809 is a memory mapped I/O (Input/Output) device. i.e. I/O shares the same address space as memory. I/O devices are addressed in the first 2K bytes of memory, from 0000H (Hexadecimal) to 07FFH. The rest of memory is taken up by either RAM (Random access Read/Write Memory) or by EPROM (Erasable Programmable Read Only Memory) up to a maximum of 62K bytes of memory. To assist with I/O decoding a signal, *IOSEL, has been provided on the backplane. This signal is asserted whenever the first 2k bytes of memory is addressed.

Note that within this document active low signals are preceded with an '*'. Some active low signals pertaining to the node interface are followed by ':N' (Logica's convention). For instance *RESET and WT:N are active low signals.
3. Hardware Description

The main PCB (Printed Circuit Board) has been divided into five basic areas or modules. Each module has a distinct function. The division was originally made to assist in the design of the PCB. It was much easier to consider one module at a time than to consider everything at once. However, by taking one module at a time it is also much easier to describe the Termux, and hence
to understand it. The modules on the main PCB are given below.

(1) CPU
(2) Data Buffers and On Board I/O decode
(3) DMA Interface
(4) Programmable Timer
(5) Node Interface

All memory and serial interfaces reside on cards plugged into the expansion bus.

The main PCB modules, memory and serial cards are described below.

3.1 CPU Module

Sheet 1. shows the circuit diagram for the CPU module. The microprocessor, UD1, is a Motorola MC6809 running at a clock speed of 1 MHz (set by X1). Inverters UB2 and associated components form the RESET circuitry. The Termux will be reset on power up or when switch SW1 is pressed.

The address bus is buffered by chips UB3 and UA2. Note that the Direct Memory Access Controller's (DMAC) address pins are connected directly to the CPU. This means that the address buffers may be permanently enabled and serve to buffer the address outputs of both the CPU and DMAC during processor or DMA cycles.

The I/O select line, *IOSEL, is produced by UA3. *IOSEL will be active whenever address bits A11 through A15 are low. This signal is used to decode the I/O address space within the first 2K bytes of memory.

The control bus, BA, BS, E, Q, R/*W and *RESET are buffered by UC7. The bus enable signal, *BE, is produced by ANDing BA and BS together. When this signal is negated the processor has released the bus and DMA may take place. When this signal is asserted, the processor has the bus and is performing either an instruction cycle or interrupt acknowledge. Note when the CPU is executing a Sync instruction it is waiting for an interrupt and has its Address, Data and control buses tri-stated. The inputs to the buffers will be floating and hence their outputs cannot be guaranteed. This may cause false reads or writes. Consequently this instruction should not be used. (use the cwai instruction instead.)

The DMAC requests the bus by asserting the *DMA/*BREQ pin on the CPU. When this pin is asserted the CPU releases the bus on the next cycle. It does not wait until the current instruction is finished. The DMAC may then use the bus for DMA operations. The CPU will claim the bus back once every fourteen cycles in order to do self refresh.

The programmable timer interrupts the CPU via the *FIRQ input.

The *IRQ interrupt has been expanded by a MC6828 PIC, UC2, to give eight vectored interrupts of which seven have been used. These interrupts are labelled *IN0 (lowest priority) through *IN7 (highest priority). *IN7 has not been used. When an interrupt occurs on one of the *IN interrupt lines the PIC asserts the CPU *IRQ input. The CPU finishes its current instruction and responds (if *IRQ interrupts are enabled) by executing an interrupt acknowledge. That is, BA and BS are set to zero and one respectively and a vector is fetched from addresses FFF8H and FFF9H. The PIC detects the vector fetch via UC1 and modifies address lines A1 through A4 depending on which *IN input has been asserted. Thus, although the CPU thinks it is fetching a vector from FFF8H and FFF9H, the vector will in fact be fetched from another location depending on which *IN interrupt is being acknowledged. If none of the *IN inputs have been asserted (default interrupt) then the PIC does not modify the address lines. If two or more *IN interrupts are asserted at the same time, the PIC will acknowledge the highest priority one.

The PIC was originally designed for use with the MC6800 processor and uses the memory space directly below FFF8H for the *IN vectors. This space is used by the MC6809 for the
*FIRQ, SWI2 and SWI3 interrupt vectors. The *IN vector space has therefore been shifted to a slightly lower areas of memory. When the CPU acknowledges an *IRQ interrupt the vector is detected by UC4 and UC5, and A5 is inverted. The combination of BA (0), BS(1), A1, A2, A3 (all zero) and A4 (1) is unique to an *IRQ acknowledge. Since A5 is always one at this time it will always be changed to zero, thus shifting the *IRQ and *IN vector table down by 20H bytes.

In addition to modifying the interrupt acknowledge vectors the PIC can also be programmed with a priority level so than only *IN interrupts with a priority equal to or greater than the mask value will be enabled. The mask is set by writing to the PIC with A1 through A4 set to the mask value. Since the PIC is not connected to the data bus the data is "don't care". If, for instance the mask is set to five (write to address FFEAH) only *IN5, *IN6 and *IN7 interrupt will be enabled. If the mask is greater than seven then no *IN interrupts will be enabled. The PIC mask has no effect on the default *IRQ. *IN interrupts have priority over the default *IRQ. If the PIC mask is set to eight then no *IN interrupt will be acknowledged, but the default *IRQ will be. The interrupt vectors and addresses for setting the PIC mask are given below.

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Vector</th>
<th>PIC Mask Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>*RESET</td>
<td>FFFE/FFFF</td>
<td></td>
</tr>
<tr>
<td>*NMI</td>
<td>FFFC/FFFD</td>
<td></td>
</tr>
<tr>
<td>SW1</td>
<td>FFFA/FFFB</td>
<td></td>
</tr>
<tr>
<td>*FIRQ</td>
<td>FFF6/FFF7</td>
<td></td>
</tr>
<tr>
<td>SW12</td>
<td>FFF4/FFF5</td>
<td></td>
</tr>
<tr>
<td>SW13</td>
<td>FFF2/FFF3</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>FFF0/FFFF</td>
<td></td>
</tr>
<tr>
<td>*IN1</td>
<td>FFFD/FFD7</td>
<td>FFF0</td>
</tr>
<tr>
<td>*IN2</td>
<td>FFDC/FFD3</td>
<td>FFEE</td>
</tr>
<tr>
<td>*IN3</td>
<td>FFDO/FFD1</td>
<td>FFE8</td>
</tr>
<tr>
<td>*IN4</td>
<td>FFCE/FFCF</td>
<td>FFE6</td>
</tr>
<tr>
<td>*IN5</td>
<td>FFCC/FFCD</td>
<td>FFE4</td>
</tr>
<tr>
<td>*IN6</td>
<td>FFCA/FFCB</td>
<td>FFE2</td>
</tr>
<tr>
<td>*IN7</td>
<td>FFCA/FFCF</td>
<td>FFE0</td>
</tr>
</tbody>
</table>

Note that the normal position for the *IRQ vector is empty and has been moved to locations FFD8/FFD9H.

The *MRDY, *NMI and *HALT inputs to the CPU have not been used and are held high by R3.

3.2 Data Buffers and On Board I/O Decode Module

Sheet 2 shows the circuit diagram for the Data Buffers and On Board I/O Decode module. The data bus from the processor is connected to the backplane by UE5. The direction of data flow through the buffer is controlled by R/*W. The buffer is enabled by *BE. When this signal is active the CPU can transfer data to and from the data bus. The buffer is disabled during DMA cycles.

The node interface, programmable timer and DMAC are all located on the main PCB. Their data buses are connected to the backplane (and hence to the CPU) by UE4. This buffer is enabled whenever any of the on board I/O devices are addressed (addresses 0600H through 07FFH). The direction of data flow through this buffer is determined by the R/*W line and whether DMA is in progress. If a CPU read is in progress the data will flow from an I/O device to the CPU. However, if a DMA read is in progress then data will be read from memory and written to the node interface. Individual I/O devices are selected by UD7 and UC8.

A switch register has been connected to the Termux at address 0600H. When this location is read the CPU will read the value of eight switches. This is useful for determining what software to
run after power up or reset or for setting baud rates etc. Sheet 2B shows the circuit diagram for the switch register.

3.3 DMA Interface Module

Sheet 3 shows the circuit diagram for the DMA module. DMA within the Termux is controlled by a MC6844 DMAC (Direct Memory Access Controller). This chip can simultaneously handle four channels. Two channels, channels zero and one, are used to control DMA access to the node. Channel zero controls data input from the node and channel one controls data output to the node. It is possible to be simultaneously receiving and transmitting data under DMA control. When programming the DMAC it is essential that channel zero be set up for a DMA write (i.e. read from node, write to memory) and channel one be set up for DMA read. Channels three and four are unused and may, with some modification to the PCB, be used to control a plug in peripheral card. The DMAC should be programmed for four channel operation even if channels two and three are not being used.

The DMAC is addressed at locations 0740H through 077FH. During CPU cycles pin two is a chip select input and pin thirty three is an interrupt request output. This output is qualified by *BE and is connected directly to the CPU's *IRQ input pin, i.e. it is the default *IRQ.

During DMA cycles pin two is an output which determines (with pin thirty five) which channel is active. Pin thirty three is asserted during the last DMA cycle to tell the peripheral that the DMAC has finished transferring data.

When the DMAC receives a transfer request on one of its four TxRQ pins it asserts *DRQ1 or *DRQ2 which forces UF2 pin 3 low. This is qualified by the Q clock and clocks UF3 asserting *DMA/*BREQ on the CPU. When the transfer cycle is over the DMAC will negate both *DRQ lines thus negating *DMA/*BREQ. The CPU responds to the DMA request by finishing its current cycle and entering the Halt or Bus Grant Acknowledge state. *BE is negated thus asserting DGRNT. The next clock cycle will be a dead cycle. The DMAC will then run at least one DMA cycle. When the DMAC has finished either one cycle for cycle steal mode or several cycles for Halt steal mode it releases the bus and another dead cycle occurs. The CPU then regains the bus. To avoid random memory accesses during dead cycles UF3, UF2 and UF4 combine to form a *DMA VMA02 signal which is only active during the positive half of the E clock for DMA or CPU cycles.

When the DMAC has control of the bus it asserts TxAKA and TxAKB to enable the required channel. The DMA cycle is strobed by *TxSTB. The DMAC's address bus acts as an output to determine the memory location being accessed and the R/*W line is set to determine the direction of the data transfer. If R/*W is high then data will be read from memory and written to the peripheral and visa versa.

If channels two and three are going to be used to control a peripheral on a plug in card then TxAK2, TxAK3 and *DEND must be connected to the backplane. The traces connecting TxRQ2 and TxRQ3 to ground should be cut and these two inputs connected via inverters to the backplane. The inputs to the inverters should have pull up resistors so that if the peripheral card is unplugged TxRQ2 and TxRQ3 will be forced low and thus will not cause any spurious DMA requests.

3.4 TIMER Module

Sheet 4 shows the circuit diagram for the timer module. It consists of a MC6840 programmable timer and some support logic. The MC6840 contains three independant sixteen bit timers. Timers two and three have their clock input pins connected to a 1KHz clock that has been derived from the processor E clock. Their gate inputs are permanently asserted. They may both be used as free running timers for software timeouts, etc. They may use either their external 1KHz clock input or the internal E clock as their time base. Timer one has been configured to count node retries. When the node is retrying mini-packets it holds TDN:N negated (high) and pulses TCLK:N for each retry. The gate input is connected to TDN:N via an inverter so that the timer is active while
the node is transmitting a mini-packet. TCLK:N is qualified by TDN:N before being used as the
clock input to timer three. The outputs of all three timers are left unconnected.

The interrupt output of the timer chip is connected to the *FIRQ interrupt input of the CPU.

3.5 Node Interface Module

Sheets 5A and 5B show the circuit diagram for the Node Interface. Sheet 5A shows the
DMA control logic and interrupt logic. Sheet 5B shows the CPU interface. The ring node is
connected to the node interface by a 50 way ribbon cable.

The CPU sees the Node Interface as sixteen address locations starting at address 07C0H.
The Node registers are as follows.

<table>
<thead>
<tr>
<th>Address</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>07C0</td>
<td>Source Select Register</td>
</tr>
<tr>
<td>07C1</td>
<td>Receive Source (Read)/Transmit Destination (Write)</td>
</tr>
<tr>
<td>07C2</td>
<td>Node Address (Read only)</td>
</tr>
<tr>
<td>07C3</td>
<td>Transmit Status Register (Read only)</td>
</tr>
<tr>
<td>07C4</td>
<td>Receive Node Status (Read)/Node Control (Write)</td>
</tr>
<tr>
<td>07C5</td>
<td>Receive Byte (Read)/Transmit Byte (Write)</td>
</tr>
<tr>
<td>07C6</td>
<td>Receive Byte and Go (Read)/Transmit Byte and Go (Write)</td>
</tr>
<tr>
<td>07C7</td>
<td>Receive Go (Read)/Transmit Go (Write)</td>
</tr>
<tr>
<td>07C8</td>
<td>Extended Mode Status (Read)/Extended Mode Control (Write)</td>
</tr>
<tr>
<td>07C9</td>
<td>Reserved</td>
</tr>
<tr>
<td>07CA</td>
<td>Reserved</td>
</tr>
<tr>
<td>07CB-F</td>
<td>For User/Interface Unit</td>
</tr>
</tbody>
</table>

Most registers are Read or Write only. That is, reading from an address will access one
register (such as a status register) while writing to the same address will access a different register
(such as a control register).

When the node is accessed under CPU control TxSTB:P is inactive. The node chip select is
active for the cycle. The node address buffer, UF13, transfers the lower four bits of the address to
the node address lines. The node data buffer, UF12, transfers data to or from the ring node. If a
read cycle is in process RDN:N will be asserted while the E Clock is high for that cycle.

If a write cycle is in operation then UE14 pin 8 will be asserted while the E clock is high for
that cycle. This is used as the input to UF15. When the Q clock goes low, 250nS later, the write
strobe, WT:N, is asserted. The write strobe is cleared by the return of ECHO:N from the Node.
This delay is included because the Node latches data on the leading edge of the write strobe. During
a processor cycle the data will be stable at this time, but during a DMA cycle the data must be read
out of memory and propagate through three data buffers before the write strobe may be asserted.
The length of the write strobe delay means that memory must have an access time of less than
200nS. As a check that the Node responds in time UF15 checks that ECHO:N returns before the E
clock is negated. If ECHO:N does not return in time then an active low pulse is applied to the CA1
input of the MC6821, UE9 (Sheet 5A). This may be used to generate an interrupt.

The MC6821 (UE9 on-sheet 5A) is used to control interrupts and DMA. RDN:N and TDN:N are
connected to the CB2 and CA2 inputs of UE9 respectively. These inputs can be used to
generate receive and transmit interrupts. The CA1 input is connected to ECHO:N as discussed
above. CB1 is used as a DMA abort interrupt. This will be described below. The interrupts are
cleared by reading the associated data registers of the MC6821. The interrupt should be cleared before data is read from or written to the node or the next interrupt may be missed. The interrupt outputs from the MC6821 are connected to interrupt level *INS.

Node DMA is controlled by two pairs of flip flops, UD11 and UD12. UD11 controls the enabling and disabling of DMA transfers as a whole and UD12 requests individual transfers.

DMA input cycles (read from node, write to memory) are controlled by channel zero of the DMAC. It is essential that this channel be programmed for this direction of transfer. Input cycles (within the node) are disabled by pulsing pin 13 of UE11 low. This may be done by resetting the Termux, DEND0 from the DMAC or by pulsing the MC6821's PB1 output high. Input cycles are enabled by pulsing the MC6821's PB0 output high. The DMAC must also be programmed for the DMA operation. When input DMA is enabled pin three of UD14 is high. When a mini-packet is received RDN:N is asserted setting pin 12 of UD12 high. This is clocked through by the Q clock and asserts TxRQ0 on the DMAC. The DMAC responds by requesting the bus and then executing a DMA cycle for channel zero. TxAK0 is asserted, thus clearing the request, TxSTB:P will be asserted, initiating a node access via pin 10 of UE13, R/*W will be set low, indicating a write to memory (however since TxSTB is asserted data will be read from the node) and the address bus will point to the location in memory being written to. The first data byte of the minipacket will then be read from the node and placed in memory. Since there are two bytes per minipacket RDN:N will stay active and the second byte will be transferred. RDN:N will then go inactive and no more DMA cycles will occur until the next mini-packet is received. Note that the type bits for mini-packets being received under DMA control must be 00. If the type bits are anything else then DMA will be blocked by pin 5 of UD14 and the CB1 input to the MC6821 will be forced low. This input may be used as a DMA abort interrupt. When the last byte is being transferred the DMAC asserts DEND0 thus disabling further DMA requests from the node. During DMA cycles UF13 puts the address 06H onto the node address lines. i.e. DMA always reads from or writes to the Receive/Transmit Byte and Go register. When DMA is enabled the CB2 interrupt should be disabled otherwise the CPU will attempt to react to the same receive data as the DMAC.

Output DMA (read from memory, write to node) is controlled in the same way as input DMA. Channel one is used to control the transfer. Since the data must be read from memory before it can be written to the node a delay has been included in WT:N to allow the data bus to stabilise. Auto retries must be enabled when doing output DMA.

Input and output DMA may occur at the same time.
3.6 Backplane

Ten slots are provided on the backplane for plug in cards. Cards may be plugged into any slot. The address, data, control, interrupt, DMA and power busses are included on the backplane. This allows a wide range of peripheral and memory cards to be plugged into the backplane. The signals on the backplane are given below.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>Ground</td>
<td>25</td>
<td>+5V</td>
</tr>
<tr>
<td>27</td>
<td>*IN0</td>
<td>24</td>
<td>TxAK3</td>
</tr>
<tr>
<td>28</td>
<td>*IN1</td>
<td>23</td>
<td>TxAK2</td>
</tr>
<tr>
<td>27</td>
<td>*IN2</td>
<td>22</td>
<td>TxRQ3</td>
</tr>
<tr>
<td>30</td>
<td>*IN3</td>
<td>21</td>
<td>TxRQ2</td>
</tr>
<tr>
<td>31</td>
<td>*IN4</td>
<td>20</td>
<td>*IOSEL</td>
</tr>
<tr>
<td>32</td>
<td>*IN4</td>
<td>19</td>
<td>*DMAVMA02</td>
</tr>
<tr>
<td>33</td>
<td>*IN6</td>
<td>18</td>
<td>R/*W</td>
</tr>
<tr>
<td>34</td>
<td>*DEND</td>
<td>17</td>
<td>A15</td>
</tr>
<tr>
<td>35</td>
<td>BA</td>
<td>16</td>
<td>A14</td>
</tr>
<tr>
<td>36</td>
<td>BS</td>
<td>15</td>
<td>A13</td>
</tr>
<tr>
<td>37</td>
<td>*RESET</td>
<td>14</td>
<td>A12</td>
</tr>
<tr>
<td>38</td>
<td>Q</td>
<td>13</td>
<td>A11</td>
</tr>
<tr>
<td>39</td>
<td>*DMA/*BREQ</td>
<td>12</td>
<td>A10</td>
</tr>
<tr>
<td>40</td>
<td>E</td>
<td>11</td>
<td>A9</td>
</tr>
<tr>
<td>41</td>
<td>D0</td>
<td>10</td>
<td>A8</td>
</tr>
<tr>
<td>42</td>
<td>D1</td>
<td>9</td>
<td>A7</td>
</tr>
<tr>
<td>43</td>
<td>D2</td>
<td>8</td>
<td>A6</td>
</tr>
<tr>
<td>44</td>
<td>D3</td>
<td>7</td>
<td>A5</td>
</tr>
<tr>
<td>45</td>
<td>D4</td>
<td>6</td>
<td>A4</td>
</tr>
<tr>
<td>46</td>
<td>D5</td>
<td>5</td>
<td>A3</td>
</tr>
<tr>
<td>47</td>
<td>D6</td>
<td>4</td>
<td>A2</td>
</tr>
<tr>
<td>48</td>
<td>D7</td>
<td>3</td>
<td>A1</td>
</tr>
<tr>
<td>49</td>
<td>Ground</td>
<td>2</td>
<td>A0</td>
</tr>
<tr>
<td>50</td>
<td>+12V</td>
<td>1</td>
<td>-12V</td>
</tr>
</tbody>
</table>

Circuit Side  Component Side
Viewed from above

The address bus is connected to the outputs of UB3 and UA2. The data bus is connected to pins two through nine of UE5. The control bus is connected to the output of UC7. The interrupt bus is connected to the PIC, UC2. *IN5 is also connected to UE9. The DMA bus, *DEND, TxAK2, TxAK3, TxRQ2 and TxRQ3 have not been connected. If required these signals should be connected as detailed in section 3.3. Power supplies of +5V, +12V and -12V are also connected to the backplane.

3.7 Memory Card

All memory on the Termux is included on plug in cards. Sheet 6 shows the circuit diagram for the 16K byte memory card. This card may include up to 16K bytes of memory of either RAM, EPROM or EEPROM in any combination on 2K byte boundaries. The memory chips used are HM6116-15 for RAM, 2716 for EPROM and X2816A for EEPROM. The RAM chip has an access speed of 150nS, thus satisfying the requirements for DMA.

The memory chips are selected by the 74LS138 one of eight decoder. The data buffer is enabled whenever one of the memory chips is selected.

The memory card may be addressed at any 16K byte boundary by setting two wire wrap straps as shown below.
Straps | Address  
---|---  
E - F, B - C | 0800 - 3FFF  
E - F, A - C | 4000 - 7FFF  
D - F, B - C | 8000 - BFFF  
D - F, A - C | C000 - FFFF  

*IOSEL is included in the decoding so that no memory cards are selected when I/O space is addressed (0000-07FF).

The centre pin of the three wire wrap pins below each memory chip should be strapped to the pin above it if an EPROM is used or strapped to the pin below it if a RAM chip is used. If the chip is an EEPROM then the strap should be set as for RAM if its contents are to be modified, or as for EPROM if its contents are to remain unchanged.

3.8 ACIA Card

Sheet 7 shows the circuit diagram for the ACIA card. The card can control four RS232 lines at any standard baud rate from 50 to 19.2K baud. The card can be addressed on any 100H byte boundary within the range 0000 to 05FFH by setting three wire wrap straps as shown below:

Straps | Address  
---|---  
K - J, H - G, E - F | 0000  
K - J, H - G, E - D | 0100  
K - J, H - I, E - F | 0200  
K - J, H - I, E - D | 0300  
K - L, H - G, E - F | 0400  
K - L, H - G, E - D | 0500  
Other straps should not be used as the map onto the on board I/O space.

The SY6551 ACIAs are addressed on 40H byte boundaries from the base address of the card. For instance an ACIA card addressed at 0100H will have ACIAs addressed at 0100H, 0140H, 0180H and 01C0H.

The ACIA clock is driven by an oscillator constructed from a 1.8432MHz crystal, a 74LS04 inverter, a 510 OHM resistor and a 150pF capacitor. The clock drives the XTAL1 input of the ACIAs and is used by them to generate the required baud rate. The TxD, RxD, *DCD and *DTR signals from the ACIAs are connected to a 25 way ribbon cable connector as shown. The *DCD input is used to detect the presence of a terminal or other peripheral. The RxD and TxD lines are used to receive or transmit data into or out of the card. The *DTR output may be used to tell a host computer that the Termux is attached by a RS232 line. The *DTR output from ACIA0 may be strapped so that it is permanently asserted by setting strap B - C, otherwise it is driven by the ACIA by setting strap A - B. This is done so that the *DTR line will remain asserted after the Termux has been reset. ACIA0 of the ACIA card addressed at 0100H is used by the Termux monitor as a link to a host (unix) machine. It is desirable that *DTR remain asserted so that the user does not have log back onto the host machine every time the Termux is reset. Pin one of the 26 way connector may be strapped to ground for use as a shield line by setting strap O - P.

The 26 way ribbon cable connector is connected to a 25 pin male D type connector on the back of the Termux. (pin 26 is not used.) From here the RS232 lines may be split off for individual terminals. Sheet 8 shows a typical configuration.

All four ACIAs on the card may be connected to any of the *IN interrupt lines by means of a wire wrap strap. All ACIAs on a card will be connected to the same interrupt level.

The SY6551 has an unfortunate interrupt mechanism in that all pending interrupts are cleared when the status register is read. For instance if receive interrupts are enabled and the transmitter is being polled then the act of polling the status register for Transmitter Ready may accidently clear a
receive interrupt before it can be handled, thus losing data. This problem may be solved by
disabling interrupts before the status register is polled. If the Recieve Data flag is found to be set the
interrupt handler should be called to handle the pending receive interrupt. If the Transmitter Ready
flag is also set then a character may be transmitted. Finally interrupts should be re-enabled.

A better solution to this problem is to enable both transmit and receive interrupts. After an
interrupt has occurred the handler should read both the status register and the command register for
the ACIA. Since these registers are adjacent both may be read as a single sixteen bit value. To
determine whether a receive interrupt has occurred this value should be checked to see if the
Interrupt and Receive Data flags in the status register are set and that receive interrupts are turned on
in the command register. This may be done by ANDing the status value with a mask to leave only
those bits required and then comparing it with the appropriate bit pattern. If the two are equal then a
receive interrupt has occurred. Transmit interrupts may be tested for in a similar manner. The status
register should only be read once during the execution of the handler. Any interrupt that occurs
between the status register being read and the end of the handler will remain pending until interrupts
are again enabled after a 'return from interrupt' instruction is executed. The sample interrupt
handler below should clear up any problems:

```
handler  ldd  ACIA+1  get status and command regs
          bge  exit       interrupt flag not set
          pshs d         save value
          andd #x'8803   check for IRQ flag set,
          cmpd #x'8801   *RDRF flag set and receive interrupts enabled
          bne  txmt      not a valid receive interrupt
          *
          * Put code to read and store received character here
          *
            txmt  puls d      restore status value
             andd #x'900C
             cmpd #x'9004    * TDRE flag set and transmit interrupts enabled
             bne  exit       not a valid transmit interrupt
             *
            * Put code to transmit character here
            *
          exit  rti         end of handler
```

3.9 Probe Card

Sheet 9 shows the circuit diagram for the Termux Probe Card. This card is a debugging aid
which facilitates the connection of the Termux to a Logic State Analyser. The address, data and
control busses may be connected to the state analyser by means of the pins along the top edge of the
card. To observe bus cycles the state analyser's clock should be triggered by the negative going
edge of the E clock.

The Probe Card also contains three eight bit registers that occupy consecutive addresses from
the base address of the card. The card's base address is set by wire wrap straps as follows:

<table>
<thead>
<tr>
<th>Straps</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>G - I, D - F, A - C</td>
<td>0000</td>
</tr>
<tr>
<td>G - I, D - F, A - B</td>
<td>0100</td>
</tr>
<tr>
<td>G - I, D - E, A - C</td>
<td>0200</td>
</tr>
<tr>
<td>G - I, D - F, A - B</td>
<td>0300</td>
</tr>
<tr>
<td>G - H, D - F, A - C</td>
<td>0400</td>
</tr>
<tr>
<td>G - H, D - F, A - B</td>
<td>0500</td>
</tr>
</tbody>
</table>

Other straps should not be used as they map onto the on board I/O space.
The Probe Card is generally addressed at 0000H.

These registers may be considered as three separate eight bit registers, an eight bit and a sixteen bit register or as a twenty four bit register. When an eight bit value is written to a register or a sixteen bit value written to a pair of registers during two consecutive bus cycles (i.e. STD instruction) the probe clock will pulse low once. The rising edge of the pulse corresponds to the opcode fetch for the next instruction. If the state analyser is connected so that it reads the address lines and the appropriate probe registers, and is clocked by the rising edge of the probe clock, then every time data is written to a probe register the data will be logged along with the address of the next instruction to be executed. If software is written so that a probe register is written to at the beginning of each major routine (or any other point of interest) then the state analyser will log the order of execution of the routines. The data written to the routines may be a unique value to announce each routine or may contain some status information.

3.10 Synchronous Interface

A Termux is being used as a protocol converter between the department's unix machines and the Computer Centre's Univaac. This has been achieved by adding a synchronous interface card (with attached display card) and including new software.

The synchronous interface card can be addressed on any 100H byte boundary within the range 0000H to 05FFH by setting three wire wrap straps as shown below:

<table>
<thead>
<tr>
<th>Straps</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>H - I, E - F, B - C</td>
<td>0000</td>
</tr>
<tr>
<td>H - I, E - F, B - A</td>
<td>0100</td>
</tr>
<tr>
<td>H - I, E - G, B - C</td>
<td>0200</td>
</tr>
<tr>
<td>H - I, E - G, B - A</td>
<td>0300</td>
</tr>
<tr>
<td>H - G, E - F, B - C</td>
<td>0400</td>
</tr>
<tr>
<td>H - G, E - F, B - A</td>
<td>0500</td>
</tr>
</tbody>
</table>

Other straps should not be used as they map onto the on board I/O space.

The card may be connected to to any of the *IN interrupt lines by means of a wire wrap strap. The card is nominally strapped to address 0500H and to interrupt line *IN5.

The synchronous interface card uses a Motorola MC6852 Synchronous Serial Data Adapter to interface to a Univac (Unisys) synchronous DCT line. This line connects to the Computer Centre via a pair of Racal synchronous modems. The modems supply both the receive and transmit clocks. (The transmitting modem supplies the transmit clock to the transmitting hardware (6852) and to the receiving modem. The receiving modem supplies what is now the receive clock to the receiving hardware. This ensures that there are no timing errors due to drift if the clocks were independantly generated at each end or by phase shift in a long line.

When the line starts up the 6852 asserts Request To Send (RTS) and the modem asserts Clear To Send (CTS) in response. Data Set Ready (DSR) is permanently asserted by the modem. Data Carrier Detect (DCD) is asserted by the modem when the synchronous card is polled.

A 24 pin DIP socket on the synchronous interface card connects to a display card mounted on the front of the cabinet. Six LEDs monitor the state of the synchronous line. Six seven segment displays are also available to the software to display status information.

In normal operation the synchronous line is plugged into the modem connector and an asynchronous line, connecting to a unix machine running the correct software, is plugged into the unix connector. A terminal may optionally be plugged into the terminal connector to monitor traffic. Sw8 of the DIP switch on the mother board is turned on so that when the micro is reset it will automatically boot the communications software.

During debugging sw8 of the DIP switch is off so that the monitor will boot on reset. A terminal is plugged into the terminal connector and a normal unix terminal line is plugged into the down line loading connector. This latter line is used as a transparent link to unix and for down line
loading new software.

Appendix B details the connections to micro and the front panel display LEDs. Drawings of the synchronous interface card and the display card are included at the end of this document.

3.12 Power Supply

The Termux uses an Apple II type switch mode power supply, capable of supplying +5V at 6A and +/- 12V at 1A. This is quite adequate for a fully configured Termux. The power supply has a 5V adjust but no 12V adjust.

Originally the Termux was designed to use a TAPS 2U linear power supply. This supply required a fan and additional 240V wiring as shown in sheet 10. The new supply requires neither, thus simplifying construction, reducing noise and removing the 240V safety hazard.

3.13 Standard Memory Map

The table below shows the standard memory map for a fully configured Termux.

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 - 00FF</td>
<td>Reserved for Termux Probe Card</td>
</tr>
<tr>
<td>0100 - 01FF</td>
<td>ACIA Card 0</td>
</tr>
<tr>
<td>0200 - 02FF</td>
<td>ACIA Card 1</td>
</tr>
<tr>
<td>0300 - 03FF</td>
<td>ACIA Card 2</td>
</tr>
<tr>
<td>0400 - 04FF</td>
<td>ACIA Card 3</td>
</tr>
<tr>
<td>0500 - 05FF</td>
<td>Spare, Available for special purpose cards</td>
</tr>
<tr>
<td>0600 - 063F</td>
<td>Switch Register</td>
</tr>
<tr>
<td>0640 - 06FF</td>
<td>Unused (On Board I/O)</td>
</tr>
<tr>
<td>0700 - 073F</td>
<td>Programmable Timer (On Board I/O)</td>
</tr>
<tr>
<td>0740 - 077F</td>
<td>DMAC (On Board I/O)</td>
</tr>
<tr>
<td>0780 - 07BF</td>
<td>Node PIA (On Board I/O)</td>
</tr>
<tr>
<td>07C0 - 07FF</td>
<td>Node (On Board I/O)</td>
</tr>
<tr>
<td>0800 - 3FFF</td>
<td>Memory Card 0, RAM</td>
</tr>
<tr>
<td>4000 - 7FFF</td>
<td>Memory Card 1, RAM</td>
</tr>
<tr>
<td>8000 - BFFF</td>
<td>Memory Card 2, RAM</td>
</tr>
<tr>
<td>C000 - DFFF</td>
<td>Memory Card 3, RAM</td>
</tr>
<tr>
<td>E000 - E7FF</td>
<td>Memory Card 3, EPROM</td>
</tr>
<tr>
<td>E800 - EFFF</td>
<td>Memory Card 3, EEPROM</td>
</tr>
<tr>
<td>F000 - FFFF</td>
<td>Memory Card 3, Tennon 3.0 monitor</td>
</tr>
</tbody>
</table>

The first 100H locations are reserved for the Probe Card, although this card does not always have to be plugged in. Locations 0500 - 05FFH are available for additional special purpose cards. The fully configured Termux has two spare slots on the backplane. One slot is reserved for the Probe Card and the other is available for a special purpose card. If more than one special purpose card is required then the number of ACIA cards should be reduced to make room in the address space and on the backplane.

A minimum configuration includes ACIA Card 0 addressed at 0100H and Memory Card 3 addressed at C000H. This allows for the inclusion of the monitor, 8K bytes of RAM and four ACIAs, including the monitor's console (addressed at 0140H) and the monitor's link to a host machine (addressed at 0100H). The minimum configuration is expanded by adding additional Memory Cards building down from C000H and ACIA Cards building up from 0200H to a maximum of four Memory and four ACIA cards.

A Termux being used as a synchronous protocol converter requires one ACIA card, four memory cards and one synchronous interface card.

13
3.14 Packaging

The Termux is housed in a Bicc-Vero D case, part number 48-8846J. Sheet 11 details the holes required for mounting the Termux. This sheet refers to the old TAPS 2U power supply. The fan has now been eliminated. The Apple II type power supply mounts with its 240V plug and power switch in a cut out in the rear panel. The main PCB is mounted on the base of the case adhesive PCB standoffs. The power supply is mounted on the base of the case on the right hand side. Four male 25 pin D type connectors are mounted on the rear of the case. The ACIA Cards should be mounted in the slots directly behind these connectors. A hole in the bottom left hand corner of the rear panel allows a 50 way ribbon cable to be plugged into the node interface, connecting the Termux to the ring node. If needed, plastic card supports may be inserted into the main PCB to support the plug in cards. A parts list is included in Appendix A.

4. Software

This section covers some of the low level software associated with the Termux. The software includes the Termux monitor, a ring test program, a DMA test program and the ring down line loader.

4.1 Termux Monitor

The monitor used in the Termux was written by David Wilson as a third year student project. It was originally designed to run on the Motorola 6800 and has been ported to the 6809. It is not proposed to cover the monitor in detail, but rather to show how to use it and detail such things as interrupt vectors and some useful monitor subroutines.

The commands available are as follows:

- `<address>/g`: Run a program at `<address>`
- `/r`: Display all user registers
- `/<regname>`: Display and modify individual user register
- `<range>/d`: Display memory contents in both Hex and ASCII
- `<range>/m/<address>`: move/copy address contents to new location
- `<range>/f[<fill>]`: fill memory with given constant (default zero)
- `<address>/a`: inspect and alter memory one byte at a time (rubout to exit)
- `/<address>/xs`: transparent link to host (CRTL A to exit)
- `<address>/xa`: set start address of symbol table
- `/xd`: display symbol table

Numbers are hexadecimal. `<address>` is a valid address in the range 0000H to 0xFFFFH. `<range>` is a range of addresses of the form `<start address>;<count>` or `<start address>:<end address>`. If `/d`, `/m` or `/a` are used without their range prefix then they will use the current start address and count as defined when one of these commands was previously used. For instance 1000;100/d followed by `/d` will display 100H bytes from location 1000H then 100H bytes from 1100H. Executing `/d` again will display then next 100H bytes of memory from 1200H.

The source code for the monitor can be found in the directory `/usr/hardware/ring/termux/termux` on system A. When the Termux starts up it reads the switch register at location 0600H. If the most significant bit is a one (switch 8 open) then the monitor will be booted otherwise (switch 8 closed) execution will commence at location E000H.

The monitor uses the ACIA at location 0140H as its console and the ACIA at location 0100H for a transparent link to a host machine. Both ACIAs are initialised on reset or power up to 8 data bits, no parity, one stop bit. The baud rate is determined by the lower four bits of the switch register. The transparent link allows the user to log on to a host machine. Programs may be down line loaded and executed using the unix dll command.
The following table gives the switch settings for the available baud rates:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Baud Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4321</td>
<td>19200 baud</td>
</tr>
<tr>
<td>1111</td>
<td>9600 baud</td>
</tr>
<tr>
<td>1110</td>
<td>7200 baud</td>
</tr>
<tr>
<td>1101</td>
<td>4800 baud</td>
</tr>
<tr>
<td>1100</td>
<td>3600 baud</td>
</tr>
<tr>
<td>1011</td>
<td>2400 baud</td>
</tr>
<tr>
<td>1010</td>
<td>1800 baud</td>
</tr>
<tr>
<td>1001</td>
<td>1200 baud</td>
</tr>
<tr>
<td>1000</td>
<td>600 baud</td>
</tr>
<tr>
<td>0111</td>
<td>300 baud</td>
</tr>
<tr>
<td>0110</td>
<td>150 baud</td>
</tr>
<tr>
<td>0101</td>
<td>134.58 baud</td>
</tr>
<tr>
<td>0100</td>
<td>109.92 (110) baud</td>
</tr>
<tr>
<td>0011</td>
<td>75 baud</td>
</tr>
<tr>
<td>0010</td>
<td>50 baud</td>
</tr>
<tr>
<td>0001</td>
<td>1200 baud</td>
</tr>
<tr>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>

The interrupt vectors in the monitor EPROM point to three byte blocks of memory in RAM which should be loaded with a jump instruction pointing the the appropriate interrupt service routine. The service routine should be exited using the RTI instruction. The RAM locations to hold the jump instructions are as follows:

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMI</td>
<td>DFD2</td>
</tr>
<tr>
<td>SWI</td>
<td>DFD5</td>
</tr>
<tr>
<td>IRQ</td>
<td>DFD8</td>
</tr>
<tr>
<td>FIRQ</td>
<td>DFDB</td>
</tr>
<tr>
<td>SWI2</td>
<td>DFDE</td>
</tr>
<tr>
<td>SWI3</td>
<td>DFE1</td>
</tr>
<tr>
<td>Reserved</td>
<td>DFE4</td>
</tr>
<tr>
<td>IN7</td>
<td>DFE7</td>
</tr>
<tr>
<td>IN6</td>
<td>DFEA</td>
</tr>
<tr>
<td>IN5</td>
<td>DFED</td>
</tr>
<tr>
<td>IN4</td>
<td>DFF0</td>
</tr>
<tr>
<td>IN3</td>
<td>DFF3</td>
</tr>
<tr>
<td>IN2</td>
<td>DFF6</td>
</tr>
<tr>
<td>IN1</td>
<td>DFF9</td>
</tr>
<tr>
<td>IN0</td>
<td>DFFC</td>
</tr>
</tbody>
</table>
The monitor contains a number of useful subroutines which may be called by a user program, although this is not advised except for test programs as the monitor may change in the future. Some of the more useful ones for doing I/O are given here. Anyone who really wants to use the monitor subroutines instead of writing a stand alone program is referred to the source code for the monitor for more information. The following is a table of the more useful monitor routines:

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>termon</td>
<td>F811</td>
<td>Warm start of monitor</td>
</tr>
<tr>
<td>puts</td>
<td>F9CA</td>
<td>Prints null terminated string pointed to by the X register</td>
</tr>
<tr>
<td>put4h</td>
<td>FB89</td>
<td>Prints four digit hex number contained in the X register</td>
</tr>
<tr>
<td>put2h</td>
<td>FB9B</td>
<td>Prints two digit hex number contained in the A register</td>
</tr>
<tr>
<td>putc</td>
<td>F9E7</td>
<td>Prints ASCII character contained in the A register. 'n' is printed as carriage return, linefeed.</td>
</tr>
<tr>
<td>getc</td>
<td>FA6F</td>
<td>Reads ASCII character from console and puts it into the A register</td>
</tr>
</tbody>
</table>

4.2 Nodetest.c

This program tests the Termux, the node and the ring by continuously sending mini-packets to itself. It is also a good example of accessing the ring through a C program. The source for this program is included in listing 1. The header files "node.h" and "terminal.h" are included in listings 2. and 3. respectively.

The screen is first prepared by clearing it and then writing out all the headings required for status information. Later when status information changes the screen is updated using cursor addressing to change only the required information.

The node is then reset and enabled with auto retries disabled. The node address is then read and displayed. The node address is used to program the source select register and the destination register so that the node will only receive mini-packets from itself and will send to itself. The receive data register is cleared of any spurious mini-packet and the type bits are set.

An infinite loop is now entered to transmit mini-packets around the ring. The transmit status is read and displayed. The program waits until the Transmit Done flag is set before transmitting the next mini-packet. If an error is detected then the program aborts. Once the mini-packet has been transmitted the program loops on the transmit status register, displaying its contents, until the Transmit Done flag is set. The program aborts if an error is detected. The program now waits until the Receive Data flag is set (which should actually occur before the Transmit Done flag is set). The extended mode status (type bits) is read and displayed and then the mini-packet is read and displayed. If the mini-packet data is read first the possibility exists that a new mini-packet will arrive before the extended mode status is read. The program aborts if the transmitted and received data differ. Finally the mini-packet data is changed for the next iteration and the mini-packet count is updated.

The user can abort the program at any time by pressing either carriage return or rubout. These are read by interrupt so they will be detected at the instant they are pressed. Pressing rubout aborts the program immediately. Pressing carriage return sets a flag which is tested at the end of the main loop. The program will then abort at the end of a cycle rather than in the middle of one.

Since this program does not use node interrupts or DMA the node PIA is not programmed or used. The console ACIA is initialised by the program so that it is totally independent from the monitor.
4.3 Dmacdup.s

This program tests the DMA interface to the node by sending a block of data around the ring under DMA control. It demonstrates the full duplex DMA capability of the Termux by transmitting and receiving the data under DMA control. The source for this program is included in listing 4.

The program starts by disabling interrupts within the CPU. It then loads a three byte jump instruction pointing to the default IRQ interrupt service routine and prints out a startup message. The node is then initialised as per ctestnode.c, except that transmit retries are enabled.

The DMA interface is initialised next. The DMA controller chip is programmed for rotating priority, channels zero and one enabled, interrupts enabled for channel zero (receive channel) only, four channel operation and data chaining disabled. Four channel operation is required because the DMA hardware has been set up for four channel operation even though channels two and three are not used. The data buffers are loaded next. The channel zero data buffer points to where the receive data will be stored and the channel one data buffer points to the data to be transferred. Both channels are loaded with the same count. The channel zero control register is programmed for cycle steal operation, data transferred from the peripheral to memory (receive data) and the address register increments after data transfers. The channel one control register is similarly programmed except that data is transferred from memory to the peripheral (transmit data). The transfer directions of these two channels must reflect the DMA interface hardware as discussed in section 3.3.

The node PIA is now programmed so that the DMA interface hardware can be enabled. The B side of the PIA is programmed so that bits zero to three are outputs. Data bits zero and two are set high and then low to enable both the transmit and receive sides of the DMA interface hardware. At this point, since TDN:N is set TxRQ1 will be set and a transmit DMA cycle will occur. Once the mini-packet returns RDN:N will be asserted and a receive DMA cycle will occur. TDN:N will also have been re-asserted so another mini-packet will be transmitted. This will continue until all the required bytes have been transmitted. Since each mini-packet contains two data bytes the count should be even.

Meanwhile the CPU has enabled interrupts and is waiting for an interrupt from channel zero to say that all the data has been transferred. When the last mini-packet is transferred by each channel DEND is asserted for one cycle which disables the interface hardware for that channel. When channel zero completes its transfer it interrupts the CPU. The interrupt handler clears the interrupt request from the DMAC and then compares the receive data buffer with the transmit data buffer. If these differ then an error has occurred. Finally the program exits to the monitor.

4.4 Rdll.c

This program is used to down line load software using the ring. It should prove to be much faster than the present method of loading software over baud terminal line (which might not always be available). It is still in the development stage and has not yet been fully tested. In its final form this program will automatically start up on power up or reset. It will then poll one or more nodes with load requests until it gets an answer from a device which will load it with software. The software will then be loaded and executed. By loading software each time the Termux is reset, rather than having it permanently residing in EPROM, it is easy to update the software that the Termux is running. If a Termux is shifted to another location where there is a different configuration of terminals the Termux is easily reconfigured to match.

Rdll.c has been written so that it does not have a bss (uninitialised data) segment and the only initialised data is constant character strings. All variables reside on the stack. This makes it possible to load the program into EPROM. The source for this program is included in listing 5. The source for the header file "loader.h" is included in listing 6.

The program starts by prompting for the node address of a host machine that will down line load the Termux and for a "program number". This will be made automatic in the final version. It then sends a mini-packet to the host requesting that it be loaded. If no reply is received within ten
seconds the program times out. If a reply is received it is checked to see if the host can in fact load the data. If it can then the program sends an "accept load" mini-packet back to the host.

The process of loading data now begins. The host transmits a stream of mini-packets to the Termux. Each mini-packet contains command information in its high order byte and some data in its low order byte. The possible commands are:

- initaddr: reset address pointer to zero
- address: increment address pointer by shifting eight bits left then adding data byte
- data: load data byte into memory and increment address pointer by one
- call: call function pointed to by the address pointer
- endload: terminate call or loading process
- abort: abort load because of error

If one of these commands is not received then the Termux aborts the load by sending an abort mini-packet to the host machine.

### 4.5 Developing Programs

All software for the Termux has been developed on the department's unix machines in either C or 6809 assembly language. The following table lists the tools required for program development:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc09</td>
<td>C compiler</td>
</tr>
<tr>
<td>as09</td>
<td>6809 assembler</td>
</tr>
<tr>
<td>ld09</td>
<td>6809 link editor</td>
</tr>
<tr>
<td>dl09</td>
<td>down line loader (into Termux RAM)</td>
</tr>
<tr>
<td>rel09</td>
<td>relocation editor (for ROMable code)</td>
</tr>
<tr>
<td>stoi</td>
<td>S record to Intel record converter (specific to the department's EPROM programmer)</td>
</tr>
<tr>
<td>load</td>
<td>loads code into EPROM programmer</td>
</tr>
</tbody>
</table>

C programs are compiled into assembly source using cc09. Assembly language programs are assembled using as09. The object files are then linked using ld09, which concatenates object files and sets the start address of the program to the required value. The object code generated by ld09 may be down line loaded into the TERMUX's RAM.

Consider two programs, prog1.c and prog2.s, written in C and assembler respectively. A typical compilation sequence would be:

```
cc09 -S -o prog1.s prog1.c
as09 prog1.s
mv m.out prog1.out
mv m.lst prog1.lst
as09 prog2.s
mv m.out prog2.out
mv m.lst prog2.lst
ld09 -d -r -o prog.out -k 0xYYYY prog1.out prog2.out
```

which produces two object files prog1.out and prog2.out and combines them into prog.out which would be loaded and run at hex address YYYY. The program may now be down line loaded into the Termux using the dll command. Ld09 constrains the text (program code), data (initialised data) and bss (uninitialised data) to be loaded consecutively into memory.

If it is desired to burn a program (such as the monitor) into EPROM then the text and bss segments must be totally separate since the EPROM containing the text segment will reside at a different address to the RAM containing the bss segment. If the data segment, containing initialised data, is to be used then it should contain only constant data and be included in the EPROM. If the data segment was put in RAM, so that its contents could be changed, its contents would not be correct when the program was started. (A start up routine that loads the initialised data from
EPROM into RAM could overcome this problem.)

The text, data and bss segments may be separated by running the output of ld09 through rel09 which allows the text, data and bss segments to be originated at separate locations. Rel09 produces Motorola S record format output since the a.out format files produced by as09 or ld09 cannot handle separate origins. Rel09 can also be used to initialise the reset and interrupt vectors.

The output of rel09 is passed to stoi which converts it into Intel record format which is required by the department's EPROM programmer. Stoi is also used to determine which part of the object code is to be programmed at this time. For instance, if a 6K program was to be burnt into 2K EPROMs, it would have to be done 2K at a time.

Finally the code is loaded into the EPROM programmer using load and may now be burnt into the EPROM.

As an example the following steps create a new version of the Termux monitor:

```
as09 -l termon3.l.s
mv m.lst termon3.1.lst
mv m.out termon3.1.o
rel09 -Tf800 -P0 pn0 -P1 pn1 -P2 pn2 -P3 pn3 -P4 pn4 -P5 pn5
  -P6 pn6 -P7 pn7 -PD pnD -RE_reset -NM_nmi -IR_irq -FI_firq
  -SW swi -S2 swi2 -S3 swi3 termon3.1.o > termon3.1.rel
stoi termon3.1.rel >termon3.1.i
  (clear EPROM programmer memory)
load termon3.1.i
  (now burn EPROM)
```

Since the whole monitor fits into 2K stoi does not have to split it into several parts. Rel09 sets the start of the text segment to F800H. The data (constant data) and bss segments (not used in this case) follow immediately after the text segment. All the required 6809 and 6828 interrupt vectors are programmed from labels defined in the source code.

5. Future

The future of the Termux is very difficult to predict since at the time of revision (July 1987) the Termux has only just been put into production. Any future upgrade of the Termux is likely to involve the use of a Motorola 68000 microprocessor as the CPU and to use a commercial bus and mounting system. The current Termux, while not yet being used for its intended purpose, has been found to be quite useful for other applications. Its general purpose bussed architecture makes it very flexible. One Termux is serving the role as a protocol converter between the department's Perkin Elmer unix systems and the Computer Centre's Univac.

6. Bibliography

2. Motorola Microprocessors Data Manual; Motorola Inc; 1981
3. Synertek Data Book; Synertek Inc; 1983

7. Acknowledgements

Thanks go to Meng Fong and Phillip McKerrow from whom this project was inherited, to David Wilson for his monitor and to the rest of the Department of Computing Science who have from time to time had some input to this epic.
8. Addendum (July 1987)

Since this preprint was first written a number of errors have been detected and some more work has been done. These have been revised in the body of the text.

8.1 Corrected Errors

The following errors have been corrected:

- p10 Ada card address selection table (address 0500 only).
- p11 Probe card address selection table (address 0300 only).
- p14 Interrupt vector table (second entry for IN6 deleted and IN5 - IN0 corrected).
- P15 Revised table of entry points intoTermon3.1(instead of Termon2.0)

8.2 Changes to hardware and software

Section 3.10 has been added to describe the additional hardware required for the synchronous link.

The TAPS 2U power supply has been replaced with an Apple II type supply.

Testing of production software has shown that the Termux is capable of supporting ten ports all running at 1200 baud or a fewer number of ports running at faster speeds. The abstract has been changed to reflect this.

The current version of the monitor in use is version 3.1. This version has the following changes made to it:

- Section "4.5 Developing Programs" has been rewritten.
- The EEPROM has been shifted into the hole at F000H left by the monitor.
- The program testnode has been replaced by nodetest which runs much faster and gives a count of mini-packets.
Listing 1. nodetest.c

```c
#include "node.h"
#include "terminal.h"
#define IN6 Oxdfe4
#define JMP Oxe7e
#define RUBOUT Oxe7f
#define CLS Oxe18
#define DLE Oxe10
#define LF Oxe0a
#define CR Oxe0d
#define w50ms 5000
char stop; /* stop flag */
char c; /* char used by interrupt handler */
int intt();
main()
{
  int i;
  int txdata; /* data to be sent around ring */
  int rxdata;
  char rubbish;
  char source;
  char dest;
  char nodeaddr;
  char rxstatus;
  char txstatus;
  char typebits;
  int hcount, lcount;
  stop = 0;
  initcia();
  txdata = 0;
  hcount = 0;
  lcount = 0;

  /* prepare screen */
  putchar(CLS); /* clear screen */
  for(i=0;i<w50ms;i++); /* delay till terminal ready */
  putline("Node address:
");
  putline("Source address:
");
  putline("Destination address:
");
  putline("Transmit status:
");
  putline("Transmit data:
");
  putline("Receive status:
");
  putline("Receive data:
");
  putline("Mini-packet count:");

  /* reset node */
  *((char *) node + nrxcr) = nnen | nrrst | ntrst;

  /* display node address */
  nodeaddr = *((char *) node + naddr);
  cursor(0,22);
  outhex8(nodeaddr);

  /* set up source selector */
  source = nodeaddr;

  /* select self only */
  *((char *) node + nssr) = source;
  cursor(1,22);
  outhex8(source);

  /* set up destination address */
```
dest = nodeaddr;
/* send to self */
*((char *) node + ntxdst) = dest;
cursor (2,22);
outhex8(dest);
/* clear rdn */
rubbish = *((char *) node + nrxg);
/* set type bits */
typebits = typeb;
*((char *) node + nxcr) = typebits;
for (; ;)
{
  /* display transmit status */
  do
  {
    txstatus = *((char *) node + ntxsr);
cursor(4,22);
    outhex8(txstatus);
  } while(! (txstatus & ntdn));
  if (txstatus & nter)
    error("Transmit error\n");
  /* send data to self */
  *((int *) (node + ntxb)) = txdata;
  /* display transmit data */
cursor(5,22);
  outhex16(txdata);
  /* make sure it got there */
  do
  {
    txstatus = *((char *) node + ntxsr);
cursor(6,22);
    outhex8(txstatus);
  } while(! (txstatus & ntdn));
  if (txstatus & nter)
    error("Transmit error");
  /* display receive status */
  do
  {
    rxstatus = *((char *) node + nrxs);
cursor(8,22);
    outhex8(rxstatus);
  } while (! (rxstatus & nrdn));
  /* ignore rejected mini-packets */
  if (rxstatus & nrrj)
    error("Receive error\n");
  /* display extended mode status */
cursor(9,22);
  outhex8*((char *) node + nxsr);
  /* Display receive data */
cursor(10,22);
  rxdata = *((int *) (node + nrxb));
  outhex16(rxdata);
  if (rxdata != txdata)
    error("data mismatch\n");
  txdata += 0x0201; /* change data for next time */
  /* display m-p count */
if(++lcount == 0)
  ++hcount;
cursor(12,22);
outhex16(hcount);
outhex16(lcount);
/* RETURN key pressed */
if (stop)
  error("Keyboard termination\n");

putline(s)
char *s;
{
    while (*s)
        putchar(*s++);
}
putchar(c)
char c;
{
    while(!((*(char *)(terminal + 1)) & 0x10)); /* wait for TDRE (bit 4) to be 1 */
    *((char *) terminal) = c;
    if (c=='\n')
        putchar('\r');
}
getsta()
{ return*((char *)(terminal + tstatus)) & rdrf); /* return non zero if data available */
}
getchar()
{
    while(!getsta()); /* wait for a character */
    return*((char *)(terminal + trxd)) & 0x7f); /* get and return 7 bit character */
}
outhex16(i)
int i;
{
    int k;
    for(k=0;k<4;k++)
    {
        putchar("0123456789ABCDEF"[(i>>12)&0x0f]);
        i<<=4;
    }
}
outhex8(i)
int i;
{
    int k;
    for(k=0;k<2;k++)
    {
        putchar("0123456789ABCDEF"[(i>>4)&0x0f]);
        i<<=4;
    }
}
error(s)
char *s;
{  
cursor(14,0);
pútline(s);
asm(" orcc  #x'10"); /* disable IRQ interrupts */
asm(" jmp  [x'ffe]"); /* reset */
}

inizcia()
{
/* set up interrupt vector */
*(char *) IN6 = JMP;
*(int *) (IN6 + 1) = (int) intt;
/* reprogram ACIA */
*(terminal + treset) = 1;
*(terminal + tcntr1) = (*char *) 0x0600 & 0x0f | 0x10;
*(terminal + tcomm) = 0x09;
/* enable PIC IN6 interrupt */
*(char *) Oxffec = 1;
*terminal; /* read any spurious character */
/* enable CPU IRQ interrupts */
asm(" andcc  #x'ef");
}

cursor(row, col)
char row;
char col;
{
putchar(DLE);
putchar(row + Ox20);
putchar(col + Ox20);
}

intrr()
{
asm(" .globl  _intt");
asm(" _intt:");
asm(" jbs  _handle");
asm(" rti");
}

handle()
{
*(terminal + tstatus); /* clear interrupt */
c = *terminal & 0x7f; /* get character */
if (c == CR)
    stop = 1; /* set flag for main loop */
else if (c == RUBOUT)
    error("Keyboard Abort");
/* else ignore character */
Listing 2. node.h

/* node registers */
#define nssr 0
#define nrxsre 1
#define ntxdst 1
#define nadd 2
#define ntxsr 3
#define nrxsre 4
#define nrxcn 4
#define nrb 5
#define ntxb 5
#define nrxb 6
#define ntxb 6
#define nrxbg 7
#define ntxbg 7
#define nrxg 7
#define ntxg 7
#define nxsr 8
#define nxcr 8

/* various control and status bits */
/* Transmit status */
#define ntdn 0x80
#define nte 0x40
#define nbce 0x20
#define ntce 0x10
#define ntig 0x08
#define ntc 0x04
#define nts 0x02
#define ntby 0x01

/* Receive status */
#define nrdn 0x80
#define nbpr 0x40
#define nrrj 0x20
#define ntrst 0x08
#define nrrst 0x04
#define nrtys 0x02
#define njen 0x01

/* Extended mode status/control */
#define nln 0xe0
#define nack 0x10
#define nron 0x08
#define n40b 0x04
#define nrtbn 0x03
#define terminal Ox0140
/* terminal registers */
#define trxd 0
#define ttxd 0
#define treset 1
#define tstatus 1
#define tcomm 2
#define tcntrl 3
Listing 4. dmacdup.s

* program to test dma transfer using dma controller.
* created 21/1/83, amended
*
* test comprises read from node; buffer address and no of bytes,
* m-p data are determined by user before execution of program.
* this program will set up a dma transfer in from the node and then
* continuously send a mini-packet to itself via the ring.
* when dma finishes, it should generate an interrupt which will
* then print out buffer read in.
*
* node naming convention
* node registers prefixed by "n"

nsr equ 0 ; source select register
rxsrc equ 1 ; receive source
txdst equ 1 ; transmit destination
addr equ 2 ; node address
txsr equ 3 ; transmit status register
rxsr equ 4 ; receive status register
rxcr equ 4 ; receive control register
rxb equ 5 ; receive byte
txb equ 5 ; transmit byt
rxbg equ 6 ; receive byte and go
txbg equ 6 ; transmit byte and go
rxg equ 7 ; receive go
txg equ 7 ; transmit go
rxequ 8 ; extended mode status register
rxer equ 8 ; extended mode control register
nable equ x'01 ; node enable
nretry equ x'02 ; retries enabled
nretry equ x'00 ; retries disabled
rxrst equ x'04 ; receiver reset
txrst equ x'08 ; transmitter reset
reset equ nenable+nrtry+nrxrst+ntxrst ;node reset mask
rstatok equ nenable+nrtry
*
* correct initialised status for retry

ter equ x'40 ; transmit error occurred
tdn equ x'80 ; tdn set
tac equ x'04 ; m-p accepted
bce equ x'20 ; broadcast enabled
tok equ tdn+tac+bce ;transmission successful mask
typebits equ x'00 ; type bits => dma block
*
* base address
node equ x'07c0
*
* pia naming convention
* registers prefixed by "pia" offsets are:
picra equ 2 ; control register a
piaeq equ 0 ; data direction register a
piapra equ 0 ; peripheral register a
picrba equ 3 ; control register b
piaeqb equ 1 ; data direction register b
piaeqb equ 1 ; peripheral register b
* base addresses
pia equ x'0780 ; node pia
*
* dmac naming convention
* registers prefixed by "dx" where x is for channel no.
0addhi equ 0 ; channel 0 address high byte
0addlo equ 1 ; " low "
0bythi equ 2 ; count byte high
0bytlo equ 3 ; " " low
1addhi equ 4 ; channel 1 address high byte
1addlo equ 5 ; " " low
1bythi equ 6 ; count byte high
1bytlo equ 7 ; " " low
0chancr equ x'10 ; channel 0 control
1chancr equ x'11 ; " 1 control
dpriocr equ x'14 ; priority control
dintcr equ x'15 ; interrupt control
ddchcr equ x'16 ; data chain control
* base address
dmac equ x'0740
*
fixprio equ x'00 ; fixed priority
rotprio equ x'80 ; rotating priority
chanel3 equ x'08
chanel2 equ x'04
chanel1 equ x'02
chanel0 equ x'01
prmask equ rotprio+chanel0+chanel1 ;priority mask
countup equ x'00 ; address increment
tscsteal equ x'04 ; mode 1; tsc steal
pcontmem equ x'00 ; peripheral cont -> memory
mempcont equ x'0 ; memory -> peripheral cont
cholmask equ countup+tscsteal+pcontmem
chilmask equ countup+tscsteal+mempcont
intmask equ chanel0 ; enable irq for input chanel only
chainoff equ 0 ; data chain disabled
fourchan equ x'08 ; enable four channel operation
dchmask equ chainoff+fourchan
*
* addresses of routines already in termon (output to terminal 1)

termon equ x'f023 ; address of monitor
puts equ x'f21a ; outputs string pointed to by X reg
put4h equ x'f81c ; outputs word in X as 4 hex chars
put2h equ x'f82e ; outputs word in A reg 2 hex chars
putc equ x'f239 ; outputs char in A reg
getc equ x'f2c6 ; gets char from terminal in A reg
*
intoff equ x'50 ; interrupts disabled at mpu
*
orcc #intoff ; maskable interrupts disabled
irqv equ x'dfd8 ; default irq vector
* put jump instruction to jump to handler
lda #x'7e
sta irqv
ldd #intrrr
std irqv+1
ldx #startmsg
jsr puts

"set up node interface"
lda #nodreset
sta node+nrxc ;node ready
lda node+naddr ;talk to self
sta node+nssr ;receive only from self
sta node+ntxdst ;select destination node as self
ldb #typebits
stb node+nxcxcr ; configure type bits
lda node+nrxsr ; clear rdn, kill spurious m-p

* check whether initialised properly
lda node+nrxsr ; read receive status
cmpa #rstatok
bne initerr
ldx #nokmsg
jsr puts ; debug message
bra dmacinit ; proceed

initerr
ldx #nerrmsg
jsr puts
jsr put2h
jmp exit

* initialise dmac for dma transfer
dmacinit

ldx #dmamsr
jsr puts ; output warning
lda #prmask
sta dmac+dprioccr ; initialise priority control
lda #intmask
sta dmac+dintcr ; initialise interrupt control
lda #dchmask
sta dmac+dchcr ; disable data chain

ldd inbufadd
std dmac+d0addhi ; input buffer
ldd outbufadd
std dmac+d1addhi ; output buffer
ldd bytes
std dmac+d0bythi
std dmac+d1bythi ; transfer count
lda #ch0mask
sta dmac+d0chancr ; input control reg
lda #ch1mask
sta dmac+d1chancr ; output control reg

*To enable node tdn to trigger TxRQ1 (output channel) and node rdn to
* trigger TxRQ0 (input channel) need to set up node PIA and flipflops
lda #0
sta pial+piacrb ; select data direction register B
lda #x'0f
sta pial+piaddrb ; PB0-3 output
lda #x'04
sta pial+piacrb ; select data register B
lda #x'00
sta pial+piaprb ; set clr to high on both f/fs

*  
ldx #dokmsg
jsr puts ; diagnostic

*  
loop
lda #x'05
sta pial+piaprb
lda #x'00
sta pial+piaprb ; enable DMAout and DMAin
* enable interrupts
    sta x'ffe0 ;enable all interrupts at pic
    andcc #x'ef
*
    ;wait for interrupt
    ldx #exitmsg
    jsr puts
    bra exit ;got to monitor
*
    ;DMA. interrupt to exit
*
    exit jmp termon
* message buffers

exitmsg fcc /\nreturning to monitor\n\0\n/
startmsg fcc /\nStart of DMA test\n\0\n/
dmamsg fcc /output DMA address must be in /
dmamsg fcc /x'd000,1\n/
dmamsg fcc /input DMA address must be in /
dmamsg fcc /x'd002,3\n/
dmamsg fcc /byte count must be in /
dmamsg fcc /x'd004,5\n/
nokmsg fcc /node all set up\n\0\n/
nerrmsg fcc /\node unable to be set /
nerrmsg fcc /properly; receive status: \0/
dokmsg fcc /\nma set up\n\0/
terrmsg fcc /\nUnsuccessful transmission\n\0/
*
*
* interrupt handler for default IRQ : dmac
* dmac will generate an interrupt when dma is completed.
*
*
intrr
    lda dmac+d0chancr ;clear interrupt request from dmac
    lda dmac+d1chancr
    ldx #intirqmsg
    jsr puts ;diagnostic
*
    now compare both buffers
    ldx outbufadd ;point to output buffer
    ldy inbufadd ;point to input buffer
    ldu bytes ;count of bytes

loop1
    lda ,x+
    cmpa ,y+
    bne comperror
    leau -1,u ;next please
    bne loop1
    lda #bufmatch
    jsr puts
    rti
compperror
    ldx #compmsg
    jsr puts
    tfr y,x
    leax -1,x ;have gone one past error
    jsr put4h
    rti
comppmsg fcc /Buffer compare error at \0/
bufmatch fcc /Both buffers match\n\0/
*
intrrqmsg fcc /\n* Default IRQ interrupt (/
outbufadd equ x'd000
inbufadd equ x'd002
bytes equ x'd004 ; address and count for transfer
Listing 5. rdll.c

#include "node.h"
#include "terminal.h"
#include "loader.h"
#define typeb 2
#define timeout 0xffff
#define maxtries 400
main()
{
    int addr;
    char pgm;
    char s[4];
    putline("Enter node address: ");
    getline(s, sizeof(s));
    addr = atohex(s);
    putline("Enter program number ");
    getline(s, sizeof(s));
    pgm = atohex(s);
    rdll(addr, pgm);
    writenode(nrxcr, nrrst|ntrst); /* reset and disable node */
    asm(" jmp [x'ffe] "); /* jump to reset vector */
}
rdll(nodeaddr, pgm)
char nodeaddr;
char pgm;
{
    if (initnode(nodeaddr, nodeaddr, typeb))
        /* talk only to requested node, basic block header */
        return (-1); /* node not initialised */
    if (requestload(pgm))
        return (-1); /* timed out or load refused */
    return (load());
}
initnode(source, dest, typebits)
char source;
char dest;
char typebits;
{
    writenode(nrxcr, ntrst|nrrst|nnen); /* reset node */
    writenode(nssr, source); /* set source */
    writenode(ntxdst, dest); /* set destination */
    if (readnode(nrxg)); /* ensure RDN is clear */
    writenode(nxcr, typebits & 0x03); /* set type bits */
    return (0);
}
requestload(pgm)
char pgm;
{
    int ndata;
    int count;
    /* send mp requesting load */
    ndata = ((loader + loadme)<<8) + (pgm &0xff);
    if(transmp(ndata))
        return(-1);
    count = 0;
    while(!((readnode(nrxsr) & nrdr))
    {
        count++;
        if (count == timeout)
break;
    } /* wait for reply, but timeout after 10 sec. */
if (! (readnode(nrxsr) & nrdn))
{
    putline("1: Timeout error\n");
    return(-1); /* failed on timeout */
}
ndata = readmpdata();
/* see if abort load */
if((ndata & 0xff00) == ((loader + abort) << 8))
{
    putline("2: Loader aborted\n");
    return (-1);
}
/* see if correct reply */
else if((ndata & 0xff00) != ((loader + yesican)<<8))
{
    putline("3: Bad reply to load request\n");
    abortload();
    return(-1);
}
else if(! (ndata & 0xff)) /*look for zero byte */
{
    putline("4: Cannot load\n");
    abortload();
    return(-1);
}
/* send accept load */
if(transmp((loader + accept)<<8) + 1)
    return(-1);
else
    return(0); /* Ok to load */
}
load()
{
    int caddr; /* current load address */
    int ndata; /* mini packet data */
    for (;;)
    {
        ndata = receivemp();
        switch (ndata & 0xff00) /* what type of data */
        {
            case ((loader + initaddr)<<8):
                caddr = 0; /* reset load address */
                break;
            case ((loader + address)<<8):
                caddr = (caddr<<8) + (ndata & 0xff); /* build address */
                break;
            case ((loader + data)<<8):
                *(char *) caddr ++ = ndata & 0xff; /* load a byte */
                break;
            case ((loader + call)<<8):
                putline("5: Branching to function\n");
                (* (int (*)()) caddr)();
                putline("6: Returned from function\n"); /* call what caddr points to and return integer to here */
            case ((loader + endload)<<8):
        }
putline("7: End load\n");
return (0); /* end load on endload or call */
break;
case((loader + abort) « 8):
putline("8: Loader aborted\n");
return (-1);
break;
default: /* not loader or valid function */
putline("9: Aborting load");
abortload();
return(-1);
break;
}

readnode(nodereg)
char nodereg;
{
    return( *((char *) node + nodereg));
}

writenode(nodereg, ndata)
char nodereg;
char ndata;
{
    *((char *) node + nodereg) = ndata;
}

int readmpdata()
{
    return( *(( int *) (node + nrxb)));
}

writempdata(ndata)
int ndata;
{
    *((int *) (node + ntxb)) = ndata;
}

int receivemp()
{
    while(! (readnode(nrxsr) & nrdn));
    /* wait for receive done */
    return(readmpdata());
}

transmp(ndata)
int ndata;
{
    char status;
    int retries;
    int i;
    while(! (readnode(ntxsr) & ntdn));
    /* wait for transmit done */
    for(retries = 0; retries < maxtries; retries++)
    {
        writempdata(ndata);
        while(! ((status = readnode(ntxsr)) & ntdn));
        /* wait for TDN */
        if (! (status & ntby)) /* not busy */
            break;
        for(i = 0; i < 50;i++); /* waste some time */
    }
    if (status & nter) /* got an error */
    {
        /* handle error */
    }
if (status & ntce)
    putline("10: Transmit compare error\n");
else if (status & ntig)
    putline("11: Transmit m-p ignored\n");
else if (status & ntus)
    putline("12: Transmit m-p not selected\n");
else if (status & ntby)
    putline("13: Transmit busy\n");
return (status);

else
    return (0);
}

dumpload()
{
    return (transmp((loader + abort)<<8) + 0); /* send abort packet */
}
putline(s)
char *s;
{
    while (*s)
        putchar(*s++);
}
putchar(c)
char c;
{
    while (!((*(char *) (terminal + 1)) & 0x10))
        *(char *) terminal = c;
    if (c==’\n‘)
        putchar(’\r‘);
}
getline (ptr, size) /* get a line of characters */
char *ptr; /* points to character buffer */
int size; /* size of character buffer */
{
    char *q;
    char ch;
    q = ptr;
    do
    {
        ch = getchar();
        putchar(ch);
        switch (ch)
        {
        case (’@‘):
            q = ptr; /* start again */
            putchar(’\n‘);
            break;
        case (’\b‘):
            putchar(’ ’);
            putchar(’\b‘); /* backspace over last char */
            if (q > ptr)
                q--;
            /* don’t backspace past start of line */
            break;
        default:
            *q = ch;
            /* put the character in the buffer */
            if (q < (ptr + size -1))
                putchar(ch);
        
           
    }
q++;
}
while(ch != '\n');
*q = 0; /* null terminate the string */
}

getchar()
{
char c;
while(!(*((char *) terminal + 1) & 0x08));
c = (*((char *) terminal)) & 0x7f;
if (c == '\r')
c = '\n';
return(c);
}
atohex(s) /*convert string to hex integer */
char *s;
{
int i;
char c;
i = 0;
for(;;)
{
    c = *s++;
    if(c>='0'&&c<='9')
        i = (i<<4) + c - '0';
    else if (c>='A'&&c<='F')
        i = (i<<4) + c - 'A' + 0x0a;
    else if (c>='a'&&c<='f')
        i = (i<<4) + c - 'a' + 0x0a;
    else /*not a hex char*/
        return(i);
}
}
Listing 6. loader.h

/* down line loader format */
define loader 0x10
#define loadme 0x01
#define yesican 0x02
#define abort 0x03
#define initaddr 0x04
#define accept 0x05
#define endload 0x06
#define address 0x08
#define data 0x09
#define call 0x0a
Appendix A. Termux 1.0 Component list.

This list consists of four sections:
(1) Main PCB board.
(2) Memory card.
(3) ACIA card.
(4) Hardware

(1) **Main PCB board**

<table>
<thead>
<tr>
<th>Qty.</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MC6809 UD1</td>
</tr>
<tr>
<td>1</td>
<td>MC6821 UE9</td>
</tr>
<tr>
<td>1</td>
<td>MC6828 UC2</td>
</tr>
<tr>
<td>1</td>
<td>MC6840 UF9</td>
</tr>
<tr>
<td>1</td>
<td>MC6844 UE1</td>
</tr>
<tr>
<td>5</td>
<td>74LS00 UC6, UE13, UF4, UF8, UF14</td>
</tr>
<tr>
<td>1</td>
<td>74LS02 UD13</td>
</tr>
<tr>
<td>1</td>
<td>74LS03 UF7</td>
</tr>
<tr>
<td>1</td>
<td>74LS04 UF6</td>
</tr>
<tr>
<td>2</td>
<td>74LS08 UF7, UF1</td>
</tr>
<tr>
<td>1</td>
<td>74LS10 UE14</td>
</tr>
<tr>
<td>1</td>
<td>74LS11 UD14</td>
</tr>
<tr>
<td>1</td>
<td>74LS14 UB2</td>
</tr>
<tr>
<td>1</td>
<td>74LS21 UC5</td>
</tr>
<tr>
<td>2</td>
<td>74LS27 UC9, UD10</td>
</tr>
<tr>
<td>1</td>
<td>74LS32 UA3</td>
</tr>
<tr>
<td>4</td>
<td>74LS74 UD11, UD12, UF3, UF15</td>
</tr>
<tr>
<td>2</td>
<td>74LS86 UC4, UF2</td>
</tr>
<tr>
<td>3</td>
<td>74LS90 UD8, UD9, UE8</td>
</tr>
<tr>
<td>1</td>
<td>74LS133 UC1</td>
</tr>
<tr>
<td>1</td>
<td>74LS138 UC8</td>
</tr>
<tr>
<td>1</td>
<td>74LS139 UF5</td>
</tr>
<tr>
<td>1</td>
<td>74LS157 UF13</td>
</tr>
<tr>
<td>2</td>
<td>74LS243 UC12, UC13</td>
</tr>
<tr>
<td>3</td>
<td>74LS244 UA2, UB3, UC7</td>
</tr>
<tr>
<td>3</td>
<td>L4LS245 UE4, UE5</td>
</tr>
<tr>
<td>1</td>
<td>RGLD7X222J 2K2 x 7 resistor pack</td>
</tr>
<tr>
<td>2</td>
<td>4K4 1/4W</td>
</tr>
<tr>
<td>17</td>
<td>10K 1/4W</td>
</tr>
<tr>
<td>1</td>
<td>1N914</td>
</tr>
<tr>
<td>2</td>
<td>24pF Ceramic</td>
</tr>
<tr>
<td>25</td>
<td>.1uF monocap</td>
</tr>
<tr>
<td>3</td>
<td>10uF Tantalum</td>
</tr>
<tr>
<td>1</td>
<td>100uF Electro</td>
</tr>
<tr>
<td>1</td>
<td>1000uF Electro</td>
</tr>
<tr>
<td>1</td>
<td>4Mhz Crystal</td>
</tr>
<tr>
<td>1</td>
<td>SPST Push switch</td>
</tr>
<tr>
<td>1</td>
<td>8 position SPST DIP switch</td>
</tr>
<tr>
<td>1</td>
<td>50 way Right angle Ribbon cable conn.</td>
</tr>
<tr>
<td>10</td>
<td>25 way, double sided, .1in pitch PCB edge conn.</td>
</tr>
<tr>
<td>1</td>
<td>24pin DIL socket</td>
</tr>
<tr>
<td>1</td>
<td>28pin DIL socket</td>
</tr>
<tr>
<td>3</td>
<td>30pin DIL socket</td>
</tr>
<tr>
<td>1</td>
<td>Termux 1.0 PCB</td>
</tr>
<tr>
<td>10</td>
<td>Richco Card support</td>
</tr>
<tr>
<td>6</td>
<td>Richco PCB support</td>
</tr>
<tr>
<td>1</td>
<td>250mm x 50 way ribbon cable</td>
</tr>
</tbody>
</table>
(2) Memory Card (For quantity of four)

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>74LS00</td>
</tr>
<tr>
<td>4</td>
<td>74LS10</td>
</tr>
<tr>
<td>4</td>
<td>74LS138</td>
</tr>
<tr>
<td>4</td>
<td>74LS245</td>
</tr>
<tr>
<td>27</td>
<td>HM6116LP-15 RAM (2K RAM)</td>
</tr>
<tr>
<td>3</td>
<td>2716 EPROM (2K EPROM)</td>
</tr>
<tr>
<td>1</td>
<td>X2816A EEPROM (2K EEPROM)</td>
</tr>
<tr>
<td>8</td>
<td>24 pin DIL sockets (ROM card only)</td>
</tr>
<tr>
<td>4</td>
<td>.01uF Greencap</td>
</tr>
<tr>
<td>4</td>
<td>10uF Tantalum</td>
</tr>
<tr>
<td>152</td>
<td>Wire Wrap pins</td>
</tr>
<tr>
<td>4</td>
<td>Termux RAM-ROM PCB</td>
</tr>
</tbody>
</table>

(3) ACIA Card (For quantity of four)

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>74LS04</td>
</tr>
<tr>
<td>4</td>
<td>74LS10</td>
</tr>
<tr>
<td>4</td>
<td>74LS138</td>
</tr>
<tr>
<td>4</td>
<td>75LS245</td>
</tr>
<tr>
<td>16</td>
<td>SY6551 ACIA</td>
</tr>
<tr>
<td>4</td>
<td>1488 Line driver</td>
</tr>
<tr>
<td>8</td>
<td>1489 Line receiver</td>
</tr>
<tr>
<td>4</td>
<td>150pF Ceramic</td>
</tr>
<tr>
<td>4</td>
<td>.01uF Greencap</td>
</tr>
<tr>
<td>8</td>
<td>10uF Tantalum</td>
</tr>
<tr>
<td>4</td>
<td>22uF Tantalum</td>
</tr>
<tr>
<td>4</td>
<td>510R 1/4W</td>
</tr>
<tr>
<td>8</td>
<td>1N4004</td>
</tr>
<tr>
<td>4</td>
<td>1.8432Mhz Crystal</td>
</tr>
<tr>
<td>4</td>
<td>28 pin DIL socket</td>
</tr>
<tr>
<td>4</td>
<td>26 Way Right angle ribbon cable conn.</td>
</tr>
<tr>
<td>4</td>
<td>26 Way ribbon cable header</td>
</tr>
<tr>
<td>4</td>
<td>25 pin male D type conn. (with pins)</td>
</tr>
<tr>
<td>4</td>
<td>100mm length 26 Way ribbon cable.</td>
</tr>
<tr>
<td>124</td>
<td>Wire Wrap pins</td>
</tr>
<tr>
<td>4</td>
<td>Termux ACIA Card PCB</td>
</tr>
</tbody>
</table>
(4) Hardware

1 Vero Case no. 48-8446J, colour soft beige, no 451A
1 Vero front panel no. 50-0769H
1 240V power cord
1 Apple II type Power supply
1 Molex 15 pin plug
1 Molex 15 pin socket
4 Molex male pin
4 Molex female pin
1 4m x 18 gauge hook up wire
4 1/2 in hex spacers (1/8 in thread)
8 1/8 in by 3/8 in pan head screw
Appendix B. Synchronous Link Connections

RS232c Connections (all 25 pin Female)

Terminal (debug) port (DCE)

<table>
<thead>
<tr>
<th>Pin</th>
<th>Direction</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>in</td>
<td>TxD</td>
</tr>
<tr>
<td>3</td>
<td>out</td>
<td>RxD</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Gnd</td>
</tr>
<tr>
<td>20</td>
<td>in</td>
<td>DTR</td>
</tr>
</tbody>
</table>

DLL (Down Lne Loading/Transparent Link) port (DTE)

<table>
<thead>
<tr>
<th>Pin</th>
<th>Direction</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>out</td>
<td>TxD</td>
</tr>
<tr>
<td>3</td>
<td>in</td>
<td>RxD</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Gnd</td>
</tr>
<tr>
<td>20</td>
<td>out</td>
<td>DTR</td>
</tr>
</tbody>
</table>

Unix (normal communications) port (DTE)

<table>
<thead>
<tr>
<th>Pin</th>
<th>Direction</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>out</td>
<td>TxD</td>
</tr>
<tr>
<td>3</td>
<td>in</td>
<td>RxD</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Gnd</td>
</tr>
<tr>
<td>20</td>
<td>out</td>
<td>DTR</td>
</tr>
</tbody>
</table>

Modem (Synchronous) port (DTE)

<table>
<thead>
<tr>
<th>Pin</th>
<th>Direction</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>out</td>
<td>TxD</td>
</tr>
<tr>
<td>3</td>
<td>in</td>
<td>RxD</td>
</tr>
<tr>
<td>4</td>
<td>out</td>
<td>RTS</td>
</tr>
<tr>
<td>5</td>
<td>in</td>
<td>CTS</td>
</tr>
<tr>
<td>6</td>
<td>in</td>
<td>DSR</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Gnd</td>
</tr>
<tr>
<td>8</td>
<td>in</td>
<td>DCD</td>
</tr>
<tr>
<td>15</td>
<td>in</td>
<td>TxCLK</td>
</tr>
<tr>
<td>17</td>
<td>in</td>
<td>RxCLK</td>
</tr>
</tbody>
</table>

LED Indicators (Top to Bottom)

- TxD
- RxD
- RTS
- CTS
- DSR
- DCD
8 x 10K Pullup Resistors

LS243 UC12

10
5
4
3
2
1

LS243 UC13

6
5
4
3
2
1

UC8 Pin 15
0600-06BFH

Termux 1.0
Sheet 2B Switch Register
Drawn M.Milway 9/85
Termux 1.0
Sheet 3 DMA
Drawn M. Milway 11/84
Copper Issue 2

TERMUX 1.0
SHEET 6.16K MEMORY CARD
DRAWN M.MILWAY 9/85
Back Panel to Peripheral Wiring

26 Way Female IDC Connector 25 Pin Male D Type Connector

ACIA Card to Back Panel Patch Cable

Part of 25 Pin Female D Type Conn. 25 Pin Male D Type Conn.

DTE is Terminal

Part of 25 Pin Female D Type Conn.

DCE is Unix link

Back Panel to Peripheral Wiring

Termux 1.0
Sheet 8 RS232 Wiring
Drawn M.Milway 10/85
24Pin DIP Header

1K x 7 DIP U22

RxD

TxD

RTS

CTS

DSR

DCD

LTS315R seven segment display U1

Total = 6 displays

74LS273 U16

1K x 8 DIP U8

74LS640 U15

74LS138 U7

Title: Termux Display Card

Drawn: M. Milway

Date: 21/7/1987

Sheet: 1