inmos®

IMS B016 VMEbus master card

Incorporating the IMS F008A support software.

SGS-THOMSON

Contents

1 Introduction

1.1 How to use this manual

The manual is organised into roughly three sections:

- A general overview of the IMS B016 hardware functions and the access to those functions via the IMS F008A procedure library. This section also covers jumper configuration and installation details. Refer to chapters 2 and 3.
- A detailed description of the IMS F008A support library and example applications. This section also covers software installation, configuration and loading of the sample applications. Refer to chapter 4.
- An in–depth discussion of IMS B016 hardware functionality and the detailed procedures for setting up and using all the board level features. Refer to chapter 5.

The appendices contain memory map and connector information and also an overview of the monitor command language used in the example code.

1.2 Background

The IMS B016 is a VMEbus board incorporating a 32–bit transputer processor, local RAM, peripherals, and interface circuitry to allow efficient communication between the transputer and other VMEbus boards. See figures 1.1 and 1.2

Sustained transfer rates in excess of 12Mbytes/s are achievable, given a fast VMEbus system. The IMS T801 on–board is capable of 12.5 MIPS and has its own private fast SRAM for speed–critical code and data. Some memory on the board can be accessed by both the transputer and other VMEbus Masters. This is called the *Dual–Access Memory*. The transputer, an IMS T801, can perform Master accesses to other VMEbus slaves. It can also interrupt other VMEbus Interrupt Handlers and itself handle VMEbus interrupts.

The IMS B016 can be used as the gateway hardware in a SUN workstation hosted transputer development system. A board support software product, the IMS S514C is available to support this use of the card. S514C users need not read this manual in detail as the User Guide supplied with the software contains specific installation and configuration details for the IMS B016.

Users wishing to proceed directly to install the IMS B016 in their equipment should read section 3.1.

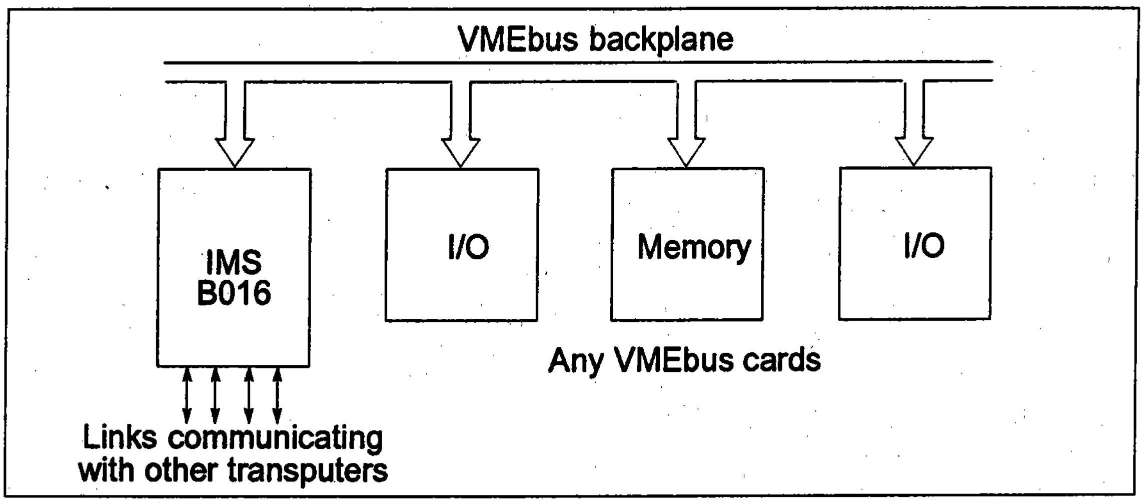


Figure 1.1 The IMS B016 used to create a VMEbus I/O subsystem

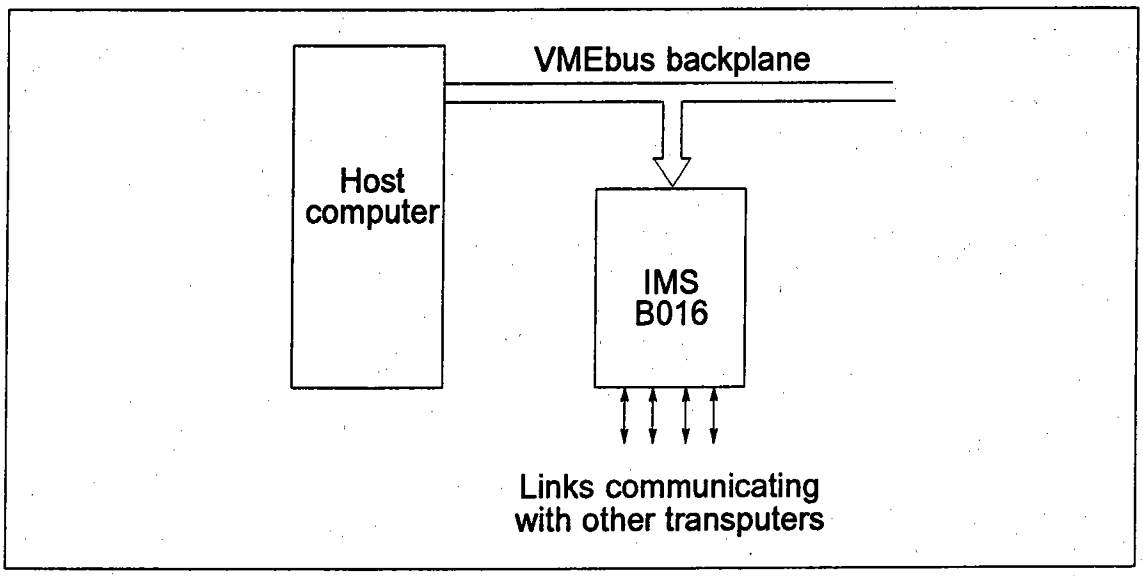


Figure 1.2 The IMS B016 used as an interface from a host computer

2 A tour of the IMS B016 hardware

INMOS provides a set of software libraries, written in the C language and compatible with transputer toolset development systems. These libraries (IMS F008A) provide the easiest way to program the card for the majority of users. This section explains the card's features and operation from the perspective of a code which makes calls to IMS F008A libraries. Users who would like to perform functions which are not catered for in the IMS F008A libraries should read section 5.1 and refer to the IMS F008A source code.

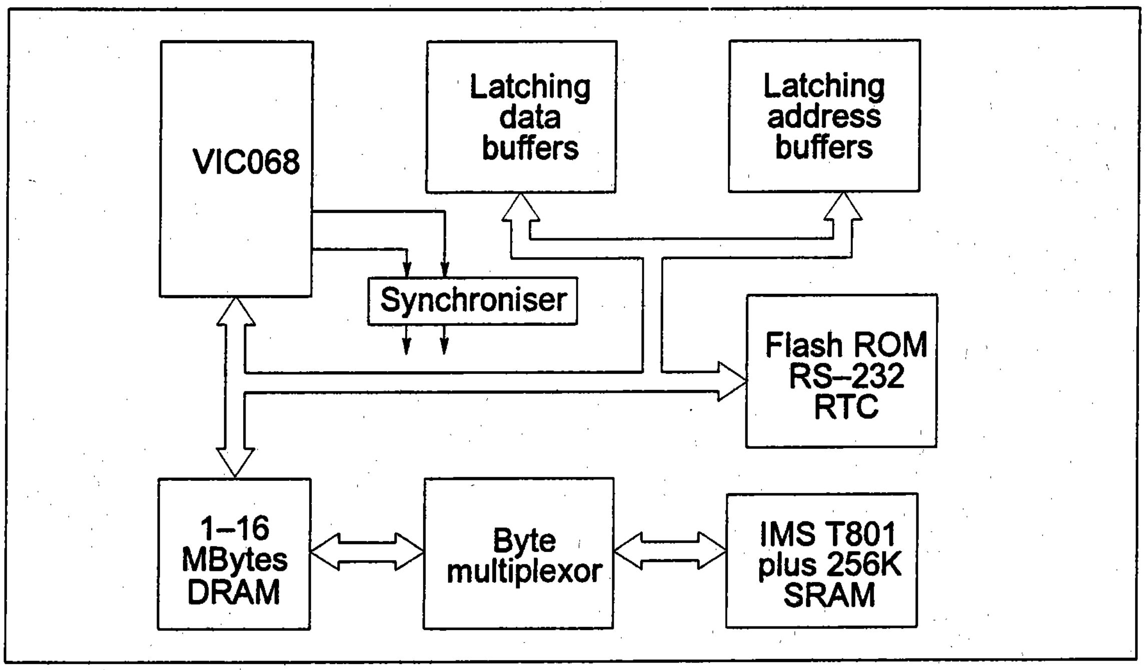


Figure 2.1 IMS B016 Block Diagram

2.1 IMS T801 and private SRAM

The card's processor, an IMS T801 floating point processor, has four INMOS serial links. These links allow connections to be made to other transputer devices on other boards via the P2 connector. 256Kbytes of fast memory (the *private SRAM*) is directly connected to the IMS T801. This cycles in 80ns and is only accessible by the transputer. Typically this memory would be used to store transputer code and data which does not need to be accessed from the VMEbus. The SRAM is about twice as fast as the next fastest available memory, as seen from the transputer, so its use is recommended whenever possible. The SRAM occupies the first 256Kbytes of the transputer's address space, minus the IMS T801's 4Kbyte internal RAM which overlays it (`#80001000`–`#8003FFFF`).

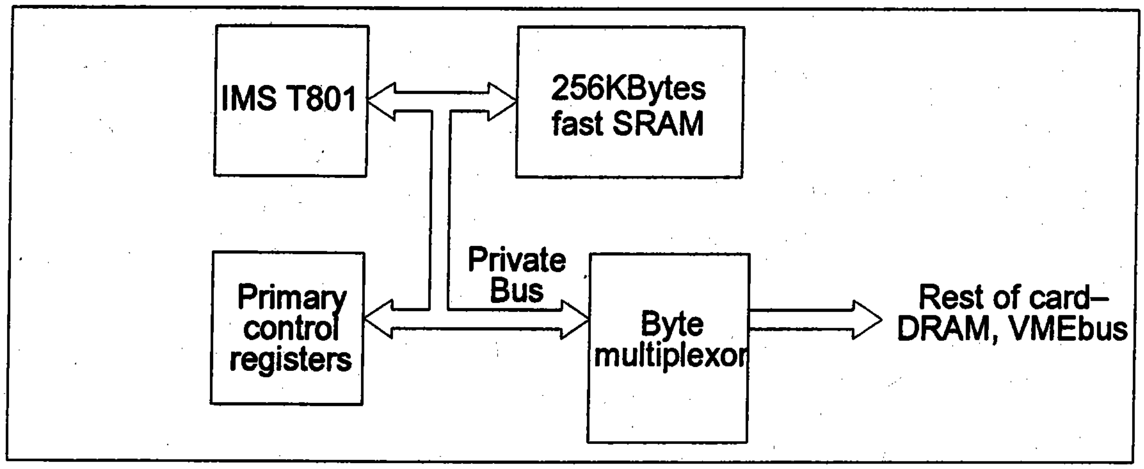


Figure 2.2 IMS T801 and private SRAM

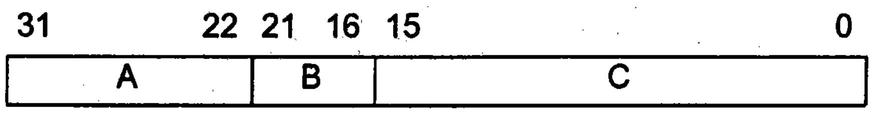
Note that most transputer compilers have storage allocation strategies which attempt to put frequently used data at a low memory address. This strategy will work well with the private SRAM as it is located just above the transputer's internal memory. However, in seeking ultimate performance, the user may wish to perform his own analysis of where code and data are placed. Refer to memory maps in Appendix A.

2.1.1 Dual–access DRAM

A large memory, using fast DRAM devices, is accessible from both the IMS T801 and (with appropriate programming) from the VMEbus as slave memory. This kind of dual-access is sometimes called "dual-port memory" but since this is not strictly true (true dual-port memory is very expensive and about 100 times less dense than the devices used on the IMS B016) the term "dual-access" will be used here.

Two variants of the IMS B016 are available. The IMS B016–1 has four megabytes of dual-access DRAM. The IMS B016–4 has sixteen megabytes of dual-access DRAM. Otherwise the variants are identical. Without any special configuring, the dual-access DRAM appears as the first four or sixteen megabytes of the transputer's address space, minus the internal RAM and private SRAM which overlay it (`#80040000–#803FFFFF` or `#80040000–#80FFFFFF`) and cycles in 160ns (four CPU cycles).

When accessed from the VMEbus as slave memory, the dual-access DRAM can occupy any VMEbus address in any address space (except A16). Programmable decode circuitry on the card allows the card's memory to appear in one or more 64Kbyte blocks of VMEbus address space. This means that one can arrange for less than the full amount of card memory to be accessible on the VMEbus. This feature is useful both when implementing a system which uses the A24 address space (which is only 16 megabytes). It also permits decoding at multiple different locations which may be an advantage in a multi-master application. The correspondence between VMEbus addresses and IMS T801 addresses can appear slightly confusing. For a 4 Megabyte board, a VMEbus address can be represented as follows:



Bitfields “A” and “B” together form a 16–bit value which is used in a table look–up to determine if the card should respond to the VMEbus transfer. Bitfields “B” and “C” together form the 22–bit index into the card's DRAM.

To determine what address is being accessed in dual-access DRAM, write the VMEbus address in the form shown and verify that bitfields “A” and “B” do in fact match the values programmed by `prog_slave_access()` . Then take bitfields “B” and “C” and replace “A” with `1000000000` to form the corresponding IMS T801 address.

Note that some “lost” memory which the IMS T801 can not see because it is overlaid by the private SRAM will be accessible from the VMEbus. The first location in dual-access DRAM which is available to the IMS T801 is at offset `#040000` so applications sharing data between other VMEbus masters and the card's IMS T801 should not use locations below this.

When the card is first switched on or reset, VMEbus slave accesses to the dual-access DRAM are disabled. The VMEbus address decoded and the type of address modifier recognised must first be programmed by the IMST801. VMEbus slave access would then be enabled. All these configuration functions are performed by writing control registers on the card from a program running on the IMS T801.

Using the IMS F008A libraries, VMEbus slave configuration of the dual-access DRAM is achieved using the routine `prog_slave_access()`.

```
prog_slave_access(decode_address, decode_window,
address_type, writepost)
```

`decode_address` is the base of the VMEbus address space to be decoded. `decode_window` is the number of bytes of address space to decode (routine rounds this up to the nearest 64 Kbytes block). `address_type` signifies whether the card should decode A32 or A24 cycles — value is either 32 or 24. Writepost controls whether the card buffers one outstanding write from the VMEbus — value 1 to enable writeposting. After configuring the routing enables VMEbus slave accesses.

```
#include "vme.h"
#include "basic.h"

int main()
{
  basic_board_init();
  prog_slave_access((void*)0x10000000,0x00400000,32,0);
}
```

This complete program first initialises the card and then sets up for decoding the dual-access DRAM as a four megabyte region from VMEbus address `#10000000` in the A32 address space. Writeposting is not enabled. So after running this program, the IMS B016 card will function like a four Mbyte D32 VMEbus memory card in the A32 address space, decoded at address #10000000. The card will respond to both "user" and "supervisor" address modifier codes.

2.1.2 MAP–RAM

The MAP-RAM is a look–up table, connected directly to the top 12 bits of the IMS T801's address bus. For every IMS T801 memory cycle, the MAP-RAM produces a set of control information which is used to determine how the various parts of the board's circuitry behave. For instance the MAP-RAM determines what kind of VMEbus cycle will be performed when the transputer attempts to perform a VMEbus master cycle.

Since there is one MAP-RAM location per megabyte of IMS T801 address space, different functions may not be programmed for two addresses within the same megabyte page.

The writing of MAP-RAM locations and the generation of the control bit-patterns which need to be written has been automated by routines in the IMS F008A support library. For most applications it should not be necessary to write MAP-RAM contents directly.

Users wishing to perform low-level MAP-RAM programming are directed to section 5.1.

2.1.3 Byte multiplexor

There are two main databuses on the board. One is private to the IMS T801 and its SRAM (and some control registers). The other connects everything else including the dual-access DRAM and the VMEbus. Between these two 32-bit busses is a transceiver which has the ability to re-order bytes (see figure 2.2). This feature allows data to be moved at high-speed between little-endian processors (the IMS T801) and big-endian processors and peripherals (most VMEbus cards). The byte multiplexor can perform any byte reordering operation, the particular operation is programmable on a megabyte-by-megabyte basis using the MAP RAM. Up to eight reordering functions can be active at any time.

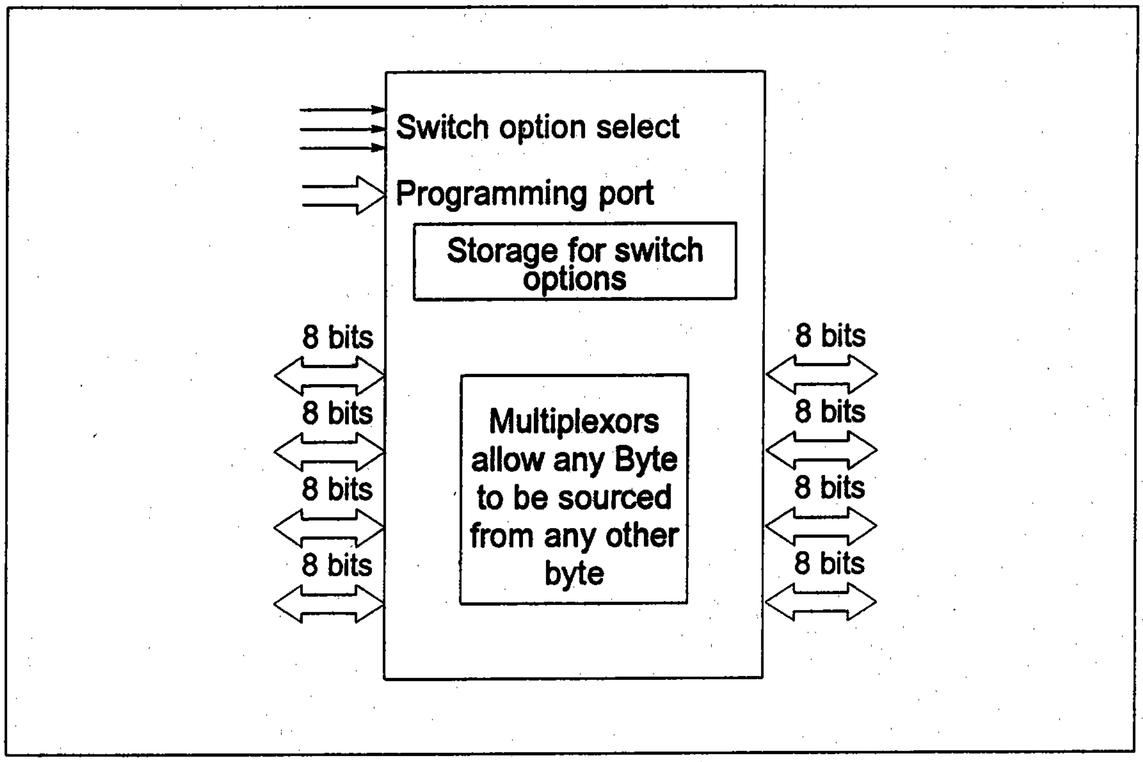

Switch option select
Programming port
Storage for switch options
8 bits
8 bits
8 bits
8 bits
Multiplexors allow any Byte to be sourced from any other byte
8 bits
8 bits
8 bits
8 bits

Figure 2.3 Byte Multiplexor

In order that the board behaves in a sane fashion at system start–up, the operation of the Byte Multiplexor is disabled on reset and must be explictly enabled via a control register. When disabled, the Byte Multiplexor functions as a simple 32-bit transceiver.

The eight re-ordering functions can be re-programmed at any time by writing the relevant registers. Note however that chaos is likely if the Byte Multiplexor is instructed to re-order memory where the currently executing program (or its data) resides. Note also that transputer byte-writes do not function correctly for regions of memory where byte multipexing is in operation. Control over the byte multiplexor is provided via various low-level IMS F008A routines. `program_byte_mux()` sets up a swap function, `turnon/off_byte_mux()` enable and disable the multiplexor. `basic_board_init()` clears the swap function code for all memory and programs swap function zero to “transparent”. Thus memory areas which have not been explicitly configured for byte-multiplexing will retain a default transparent swap function when the byte-mux is enabled for a region of memory.

Since the majority of byte-multiplexing applications are associated with master VMEbus access, the routine `prog_master_access()` allows swap functions to be specified along with other VMEbus access parameters.

2.1.4 VMEbus master operation

The IMS T801 can perform VMEbus master cycles simply by making memory accesses to certain regions of its address space. However, regions of address space must be programmed in advance for the appropriate VMEbus options.

The VMEbus supports various address spaces, including three different address sizes, 16–bit which is termed "A16", 24–bit (A24) and 32–bit (A32). Generally, simple I/O cards use the A16 space, complex I/O cards use A24, while CPU and memory cards use A32. The VMEbus also supports various data size options, including 32–bit (D32), 16–bit (D16) and 8–bit (D08). D32 tends to be used for CPU to memory transactions and for high-speed I/O, D16 for medium speed I/O and older cards and D08 for slow I/O such as serial ports.

The IMS B016 can generate many combinations of different address and data transfer VMEbus master cycles. The particular kind of transfer to be performed is determined by a look–up table, indexed on the IMS T801 address bus called the "MAP-RAM". Since the MAP-RAM has a separate entry for each and every megabyte of IMS T801 address space, the VMEbus master characteristics may be programmed on megabyte blocks. The IMS F008A routine `prog_master_access()` provides the mechanism for setting up such regions.

The VMEbus uses a byte-addressed scheme which is big-endian. That is, byte 0 corresponds to the upper byte of a 32-bit (D32) word, byte 1 to the upper middle byte and so on. Cards which support D32 transfers must also support D16 and D08 transfers to the same locations. However, all VMEbus memory spaces are conceptually 32-bit words, each composed of four bytes (called BYTE 0, BYTE 1, BYTE 2 and BYTE 3). Because the IMS T801 only performs 32-bit external memory cycles, it presents a 30–bit address to the rest of the card. This is sufficient to identify an individual VMEbus D32 longword, but lacks sufficient bits to address the words and bytes within D16 and D08 transfers. The IMS B016 works around this lack of address bits 0 and 1 by storing these bits within the MAP-RAM entry.

This means that one region of IMS T801 memory can be programmed to generate, for example a D16(BYTE3–2) transfer, while another can generate a D16(BYTE1–0) transfer. Working within the A16 or A24 address spaces, the upper 16 and 8 address bits respectively from the IMS T801 are not relevant to the VMEbus. These can be used to create multiple different regions in IMS T801 memory space which in fact generate the same A24 (or A16) VMEbus address. The effect is that all combinations of different D16 and D08 BYTEs can be addressed by the card.

In the A32 address space, this feature is not available since there are no redundant address bits. This means that practically it is not possible to use any data transfer size other than D32. Usually this would not be a limitation since the smaller data sizes are associated with I/O cards which are usually placed in the A16 and A24 address spaces.

The following example program, using IMS F008A library routines, programs the card such that for VMEbus A24 address region #100000–#1FFFFFF , various regions in IMS T801 address space generate VMEbus master transfers with all the different sizes and byte selections.

```
#include "vme.h"
#include "basic.h"

#define  MEGABYTE   0x00100000

int main()
  {
  int i=0;
  unsigned int
*d32,*d16b01,*d16b23,*d08b0,*d08b1,*d08b2,*d08b3;
  unsigned int t1,t2;

    d32    =    (unsigned int *)0x00100000;
    d16b01 =    (unsigned int *)0x04100000;
    d16b23 =    (unsigned int *)0x05100000;
    d08b0  =    (unsigned int *)0x06100000;
    d08b1  =    (unsigned int *)0x07100000;
    d08b2  =    (unsigned int *)0x08100000;
    d08b3  =    (unsigned int *)0x09100000;

    basic_board_init();
    i=init_vme_master(3,RELMODE_ROR,TRUE);
    i=prog_master_access(d32,MEGABYTE,32,24,0,0);
    i=prog_master_access(d16b01,MEGABYTE,16,24,0,1);
    i=prog_master_access(d16b23,MEGABYTE,16,24,0,2);
    i=prog_master_access(d08b0,MEGABYTE,8,24,0,3);
    i=prog_master_access(d08b1,MEGABYTE,8,24,0,4);
    i=prog_master_access(d08b2,MEGABYTE,8,24,0,5);
    i=prog_master_access(d08b3,MEGABYTE,8,24,0,6);
    i=turnon_vme_master();

  }
```

Example number 3 in the IMS F008A contains a more comprehensive version of this program, including byte–multiplexor and a second IMS B016 used as the slave card.

IMS T801 byte writes

Unlike many other microprocessors, the IMS T801 can *only* perform 32–bit and 8–bit write operations and only 32–bit reads. The VMEbus supports many more data transfer options and usually these are supported by using the IMS T801 performing only 32–bit reads and writes and using the card's capability to map those accesses into whatever VMEbus cycle is required. However, should the IMST801 perform an 8–bit write operation on a memory region programmed for VMEbus D32 transfers, the card will in fact generate the appropriate D08 VMEbus cycle.

That is, D08(BYTE0) for an upper byte write, D08(BYTE1) for an upper-middle byte write and so on. This feature means that for a region programmed to be VMEbus D32, the IMS T801's view of memory–including byte-write–will be identical to on-card memory. Thus such regions of VMEbus memory can be used for program code and data storage.

VMEbus error

Should a VMEbus transfer cycle, where the IMS B016 is master, receive a bus error, circuitry on the card causes an *event* to be signaled to the IMS T801. The IMS T801 has no feature corresponding to the M68000 the bus error pin and this mechanism (where an event is raised) is the only way to inform the CPU asynchronously of a bus error. The IMS T801 memory cycle which resulted in the bus error'ed VMEbus transfer will complete but with meaningless data if it is a read. The actions taken on bus error depend upon the application program. IMS F008A provides a very simple interrupt handler for bus error which sets a global variable–this can be checked every so often. More sophisticated schemes using channels to signal the error can be implemented by modifying the F008A code.

IMS F008A provides an event handler, complete with an event service routine for bus error interrupts. User's application code can make use of these library routines to detect bus error and take appropriate action.

```
#include "vme.h"
#include "event.h"
#include "basic.h"

int main()
{
  basic_board_init();
  start_event_handler();
  install_berr_handler(standard_berr_handler);

  my_vmebus_master_setup();

  for (;;) {
    my_vmebus_master_activity();
    if (berr_happened == 1)
      exit(1);
  }
}
```

VMEbus request modes

IMS F008A routine `init_vme_master()` should be called before attempting to perform any master operations. It configures the VMEbus request level which the card will use, the request mode and whether the card will perform master writeposting.

Request modes allowed:

- **ROR** for "release on request". Meaning that the card captures the VMEbus on a master transfer and retains bus ownership until it sees another card requesting the bus.
- **RWD** for "release when done". Meaning that the card captures the VME-bus on a master transaction and then releases the bus immediately.
- **ROC** for "release on bus clear". Meaning that the card captures the VME-bus on a master transaction and then releases the bus when it sees a bus clear signal.
- **BCAP** for "bus capture and hold". Meaning that once the VMEbus has been captured by the card, it is never released.

Users who are in doubt about which bus request mode to use should probably be using ROR. Bus request level 3 should always work. Some system controllers only support this level.

VMEbus write posting

The VIC chip, and consequently the IMS B016, is capable of performing *posted* VMEbus write cycles. When a write cycle is initiated—either by the T801 as a VME-bus master or by another VMEbus master as a slave cycle to the IMS B016's dual-access DRAM, the card's circuitry can store the data to be written and immediately acknowledge the writing device. The writing device can then proceed to perform other cycles and the write cycle completes some time later. This is termed "write posting". Write posting can greatly speed up many applications. However, it has the disadvantage that if the posted cycle actually never completes, then the writing device will have assumed that the write cycle did complete. Master and slave write posting are separately enabled. It is not possible to select writeposting on an ad-dress-basis.

System controller options

Certain VMEbus backplane housekeeping functions, such as system clock gener-ation and bus arbitration, must be undertaken by the card installed in slot 1 (left-most). Any VMEbus system *must* have a system controller and it must be in logical slot 1. In workstations which have a VMEbus the system controller function is usu-ally performed by the mothercard.

The IMS B016 can perform system controller functions, when enabled by fitting jumper K3. Users who are not installing the card in slot 1 of their backplane need not read this section.

IMS F008A routine `init_slot_one()` configures both the backplane arbitration style and the bus timeout delay.

Arbitration style can be either *priority* or *round-robin* meaning that the next master granted the bus is either the one with the highest request level, or one requesting on the level given access least recently.

Bus timeout delay is the time after which the system controller signals bus error on a transaction which has not completed. A timeout delay of 32µs is normal.

2.1.5 VMEbus block transfer (BLT)

The IMS B016 does not support master BLT operation. Slave BLT transfers are handled.

2.1.6 Events and interrupts

The IMS B016 provides the capability to handle VMEbus interrupts. It also provides several local on-card interrupt sources such as the Serial Ports, Bus Error detection and Interprocessor Communication Registers. All these interrupt conditions can be programmed to cause an IMS T801 *event*[1].

The IMS T801 lacks any vectored interrupt system. In the IMS B016, extra circuitry is provided to allow VMEbus interrupt vectors (status/ID) to be read from a VIC status register. When this register is read, if a VMEbus interrupt is pending then the VIC requests the bus and fetches the Status/ID byte (vector). The value fetched is returned as the contents of the register to the IMS T801. In the case of a local interrupt, the vector is fetched from one of the vector registers within the VIC. Provided that the programmer has ensured that all interrupt sources have distinct vectors, an event process running on the transputer can distinguish between interrupts using the vector. IMS F008A provides an event handler library which allows interrupts, both VME and local, to be used from "C" programs. `start_event_handler()` is called, allowing handler routines to be installed via calls to routines such as `install_vme_int_handler()`

```
#include "vme.h"
#include "event.h"
#include "basic.h"

void my_int_handler_1(void)
{
  printf("Got Interrupt on irq 1\n");
}

int main()
{
  basic_board_init();
  start_event_handler();
vec=install_vme_int_handler(1,VECTOR_INT_1,my_int_handler_1
);

  printf("\n Waiting for interrupts");
  for (;;) ;
}
```

This example code programs a handler routine for VMEbus interrupt level 1. After initialising, the program loops forever, printing a message if an interrupt is received.

1.
"Event" is the transputer terminology for "interrupt". Users should consult the Transputer Databook for details of event.

IMS F008A provides a header file containing suggested vector values for the local interrupt sources.

2.1.7 VMEbus interrupter

VMEbus interrupts can be *generated* (as opposed to handled) by the card. The IMS F008A routine `send_vme_interrupt(level)` generates an interrupt on the level specified (1–7). This routine will not return until a VMEbus interrupt handler has acknowledged the interrupt. The routine `prog_vme_interrupt_vector(level,vector)` programs the vector (status/ID byte) returned by the card during such an acknowledge cycle.

```
#include "vme.h"
#include "basic.h"
#include "event.h"

int main()
{
  basic_board_init();
  prog_vme_interrupt_vector(1,VECTOR_INT_1);

}
```

A separate handler can be installed to deal with interrupt acknowledge cycles `install_vme_int_ack_handler`.

2.1.8 Control of the local bus

The local bus on the board (the bus to which the Dual-Access DRAM is connected) can be under the control of any of three entities:

1 The IMS T801.

2 The VMEbus control circuitry (VIC), when a VMEbus SLAVE cycle is in progress.

3 DRAM refresh circuitry.

Arbitration is performed at the end of each access. For the IMS T801 this means at the end of every memory cycle which requires the local bus (all memory cycles *except* those for private SRAM, primary registers and Byte-switch configuration registers). For the VIC this means the end of every VMEbus slave access to the IMS B016[2]. The arbiter reverts to an idle state after every refresh cycle. The design of the arbiter and the board is such that fair access is allowed to the Dual-Access DRAM for both the IMS T801 and VMEbus masters performing slave cycles. That is, if both the IMS T801 and a VMEbus master attempt to saturate the Dual-Access DRAM, then DRAM cycles will be given fairly to each requester.

2.
On BLT (Block Transfer) slave cycles, the bus is secured for the the duration of the whole block transfer.

2.1.9 F–ROM

The IMS T801 may boot from a link or from the on-board 256 Kbyte ROM. These ROM devices are electrically reprogrammable *Flash* devices which are reprogrammed in-situ. Reprogramming is achieved using the IMS T801, booted via a link with programming software.

For programming and erasing to function, jumper K4 must be fitted correctly for F–ROM.

The devices are soldered to the board for extra reliability. F-ROMs can be erased and reprogrammed in excess of 10000 times.

IMS F008A supports F-ROM programming and erasing from a host development system. Library routines which erase and program individual ROMs and ROM locations are also provided. These can provide a useful basis for writing custom F-ROM applications.

The IMS F008A monitor can be programmed into the F–ROMS and the card will then boot stand-alone–communicating via the serial port. The makefile used to operate the ROM version of the monitor provides an example which can be followed by users building stand-alone ROM applications.

2.1.10 Serial ports

The card provides two independent asynchronous serial ports, buffered to EIA-RS232 levels on connector P2. IMS F008A provides a library of functions which drive the serial ports, including support for Xon/Xoff flow control and buffering using interrupts.

Baud rates up to 19.2K are supported and the IMS F008A monitor program can be configured to talk to a terminal via the serial ports.

2.1.11 Resets and transputer system services

The card circuitry is reset at power-on and following a VMEbus reset and someone pushing the front panel reset button. All these actions create a *board reset*. In addition, the IMS T801 is reset at this time. However, the IMST801 can also be reset from the “ServicesUp” port on connector P2. This reset will not affect any other circuitry apart from the tranpsuter and allows the transputer to be re-booted without disturbing the rest of the card’s circuitry. Note that this means that after a *board reset*, all the configuration for VMEbus access and suchlike will be lost. However following an IMS T801 reboot this configuration will not be lost.

It is possible to stop the operation of the CPU completely by performing a memory access to an address which does not correspond to either an on-board resource like DRAM or F-ROM or to a region programmed for VMEbus master access. In this event it is necessary to perform a *board reset* in order to recover correct operation.

Note that transputers do not typically clear their error signals on reset, a program which explicitly clears the error signal must be run. Transputer programs usually have this code included automatically. This means that users should not be surprised if the "error" LED remains lit after the card is reset.

The IMS T801 is provided with a 'subsystem" services port which is electrically compatible with other INMOS cards. In addition, three extra subsystem ports which are electrically compatible are provided for users who wish to control four independent sub-networks of transputers. These three extra ports are not software compatible with any other card and are dealt with in section 5.8.

The "traditional" subsystem port is addressed as two of the registers in the primary control registers (see figure 5.2). The reset register allows assertion of the notSubReset signal. Similarly the analyse register allows assertion of the notSubAnalyse register. Both these registers use only bit 0 and are not readable. In addition, the card's circuitry (including the IMS T801) can be reset when a VMEbus master writes the appropriate data into the VIC mailbox registers. This feature allows a F-ROM booted card to be re-booted under the control of another VMEbus master card (see section 5.7).

2.1.12 The front panel

There is a push–button on the front panel (see figure 2.4). This resets all the card circuitry (including the IMS T801) and also produces a VMEbus reset if the card is configured as a slot 1 system controller.

The top LED (amber) lights when the IMS T801 error pin is asserted (please read section 2.1.11 about the error flag's behaviour).

The other three LED's light respectively when the IMS T801 makes a VMEbus master access; when another VMEbus master makes a slave access to the IMS B016's dual-access DRAM; and when the local bus is accessed by either the T801 or by the VMEbus. The brightness of the LED's indicates the density of cycles being performed. Note however, that the LED's are not intended to be balanced for brightness or calibrated with respect to each other.

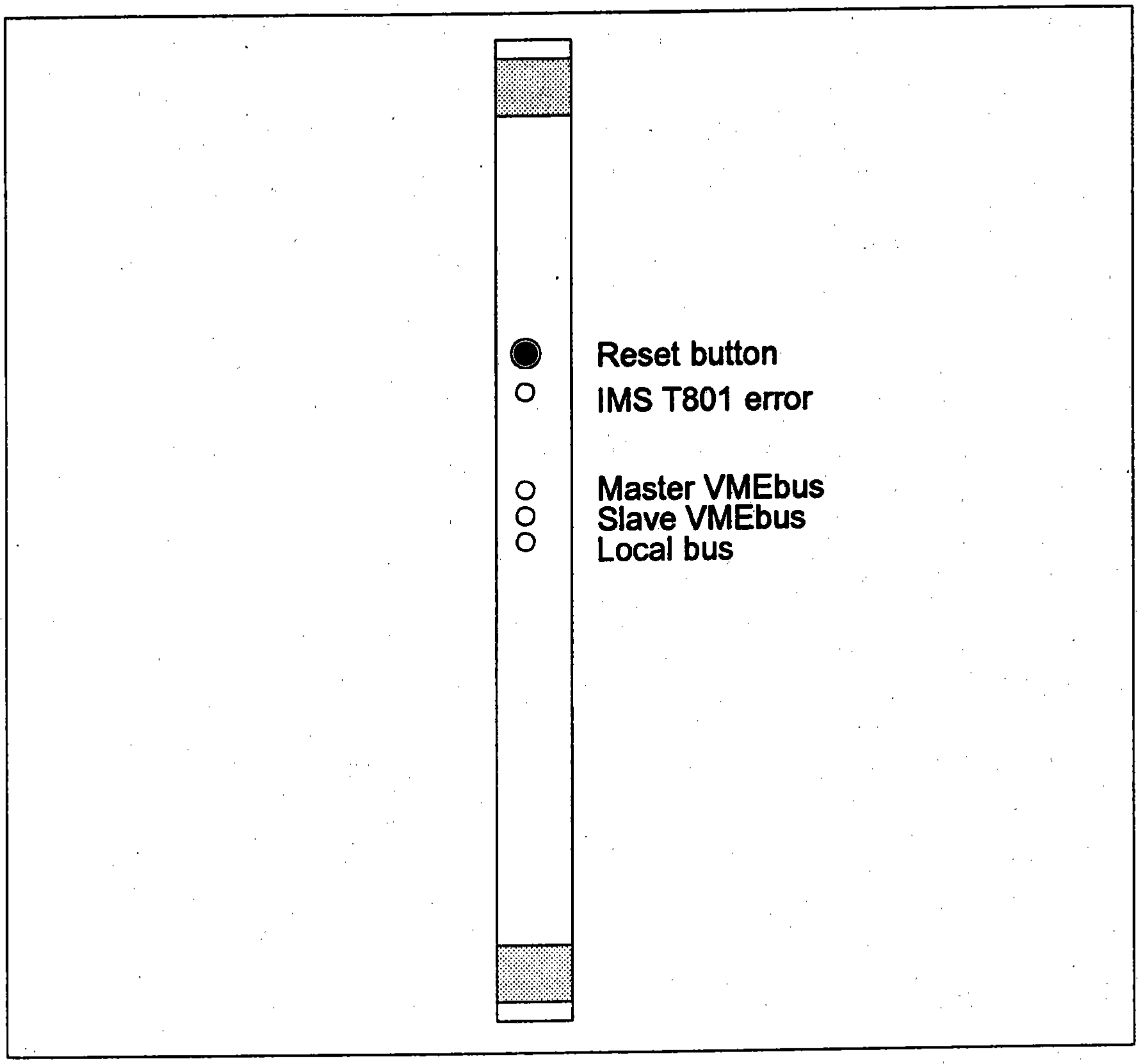


Figure 2.4 Front panel showing LEDs

3 Installing the IMS B016

3.1 Configuring prior to installation

Since the IMS B016 is based around a VLSI controller, there are far fewer configuration switches and jumpers than are usually found on VMEbus cards. However, some functions are controlled by switches and jumpers.

The VMEbus address of the VIC VMEbus slave registers (mailboxes and so on) is selected by two hex switches (see figure 3.1). In setting the VMEbus address, a unique 8-bit binary number is being selected which will be compared with the VMEbus addresses. The upper four bits of this number are set by SW2 while the lower four bits are set by SW3. A small screwdriver or trim-tool can be used to rotate the pointers on SW2, 3. The pointer must be rotated to the desired hex character Here is an example setting:

The desired addresses are `#DB00`–`#DBFF`. This means that the 8-bit number to be compared is `#DB` . Its upper four bits are `#D` , its lower four bits are `#B` . Accordingly we set SW2 to "D" and SW3 to "B".

Care should be taken not to select a VMEbus slave decode address which conflicts with addresses which may be generated by the IMS 801 performing VMEbus *master* accesses (self-accesses). Note also that users should ensure that they do not have a situation where the slave registers occupy the same address region as another A16 slave on the bus.

The IMS T801's serial links can be set to work at two different speeds. For a link on one device to talk successfully to another device, they must be set to the same speed. Current technology allows link speeds of 10 or 20Mbits/s. Jumper K2 sets the link speeds for all four of the IMS T801's links. When K2 is fitted, the links are set to 10Mbits/s, otherwise they are set to 20Mbits/s.

The IMS B016 can perform the VMEbus 'Slot 1' system controller functions. To enable this function, fit jumper K3. K3 should only be fitted if the board is installed in slot 1 of a VMEbus rack.

Jumper site K4 has three pins. The jumper can be fitted either over the center pin and the left pin, or over the center pin and the right pin. When programming the on-board F–ROMs, the programming voltage should be enabled by fitting K4 towards the INMOS logo. If the F–ROMs are not to be programmed, extra security of their contents can be achieved by fitting K4 towards the IMS T801.

Table 3.1 summarises the jumper functions.

ID	Function
K1	IMS T801 boots from link when fitted, otherwise from F–ROM
K2	IMS T801 links are 10MBits/s when fitted, 20 MBits/s otherwise.
K3	When fitted, IMS B016 performs VMEbus slot 1 functions. Do not fit unless the IMS B016 is installed in slot 1 of the VMEbus card–cage.
K4	When not fitted F–ROM programming cannot occur. It must be fitted for F–ROM programming to function.

Table 3.1 Jumper functions

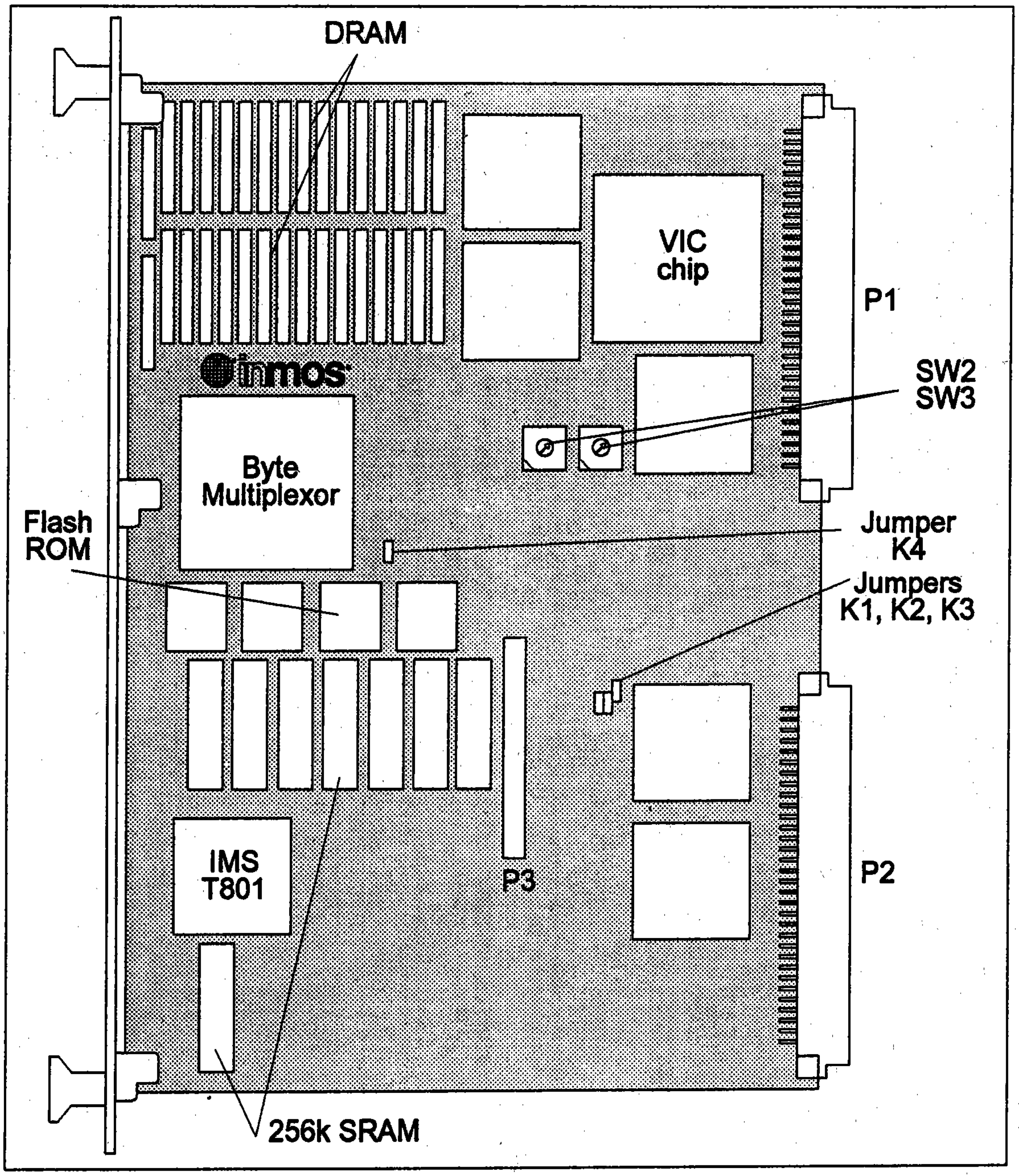


Figure 3.1 Board layout

3.2 Handling

The unpacking note in the shipping carton will give details on how to unpack the IMS B016. Standard anti-static precautions should be observed since the IMS B016 contains MOS devices which are liable to static-discharge damage.

Some VMEbus compatible card-cages, notably SUN workstations, make use of the user-defined pins on connector P2. It is extremely important that the IMS B016 is not plugged into such a card-cage because permanent destruction of the IMS B016 and/or SUN can result. This restriction only applies to some slots in some kinds of SUN (and probably other card-cages) but users should always be aware of the danger. The solution required for the SUN is to use a special holding frame which isolates the P2 user-defined pins from the backplane. Such a frame is available from INMOS, part number IMS CA12.

3.3 Installing the IMS B016 for use with IMS S514C in a workstation

If you are using the IMS B016 as the gateway hardware from a SUN workstation to your transputer development system, INMOS board support software product IMS S514C is required. The S514C User Guide contains installation information specific to this application and should be consulted in preference to the following sections.

3.4 Installing the IMS B016 in a VMEbus card–cage

Before installing any board, first make sure that the power is turned off.

Inspect the VMEbus connectors for bent pins. Next, align the corners of the board with the ends of the card-guides. The component side of the board should be to the right. Slide the card home. If resistance is encountered the board is probably not aligned properly with the card guides. Now push firmly on the handles until the board is fully home. The front panel will fit against the card-cage. Lastly screw in the retaining bolts at the top and bottom of the front panel.

Remember that if you are using interrupts, the interrupt daisy-chain jumpers on the VMEbus should be configured correctly. Please consult the documentation for your VMEbus system to find out how to do this.

4 The IMS F008A support library

The IMS F008A is a package of software components, in INMOS ANSI C source format, that acts to document and to demonstrate the features of the IMS B016 VMEbus master card. It is written as a series of library modules whose routines focus on programming the devices and control registers of each sub–set of the board's functions.

Included with the library functions is an interactive monitor program which has command line options for setting up and exercising the I/O features and memory configuration. The monitor program comes in two configurations, one which is used via a development host interface, and another which can be burnt into the board EPROM and controlled via the serial ports.

For those who would like to use a Real–Time operating system in their IMS B016 application, the IMS F008A includes the source of a board support package for the VRTX32/T executive from Ready Systems Inc.

4.1 Package features

- Basic board setup procedures control address mapped access to VMEbus address space and byte–order swapping functions.
- VMEbus interface procedures control bus master and slave access and VME interrupt handling.
- RTC interface allows setting and reading of autonomous clock registers.
- Event library controls vectored interrupt handling capability.
- DUART support functions allow interrupt driven serial I/O over two channels.
- ROM support allows programming and reading of Flash ROM devices
- Command Line Interface allows easy experimentation and example application.
- VRTX32/T board support combines device access with pre–emptive scheduler regime.
- Compatible with INMOS ANSI C toolkit.

The following sections will describe the software installation and fully document the functions within each category together with their interfaces.

4.2 Installing and using IMS F008A

The release of the IMS F008A software is delivered on DOS format diskettes in 1.2Mb 5 1/4" format and 720Kbyte 3 1/2" format. To install and use the release about 2.5 Mb of disk space will be required.

To install the software:

- Insert Disk 1 of the release into your floppy disk drive.
- Run the command file 'install' on this diskette giving as parameters the letters of the source diskette drive and the destination hard disk drive. For example, if the floppy disk drive was identified as drive A and you wish to install the software on drive C then use the command

```
a:install a c
```

During the installation process a new directory tree is created, containing the software and associated files, at the top level of the destination drive.

You will need an INMOS ANSI C toolkit (Dx214B or later) to use the IMS F008A software. Although it is delivered in DOS format, the code can be used in conjunction with a toolkit hosted on any machine. You will also need a transputer interface of some kind, either a card inserted in your host machine or a transputer development node accessible across the network. The subsystem and link connections that come from this interface must be connected to the appropriate connector pins of the IMS B016 card in order to boot programs onto the IMS B016 from a development system. It must be ensured that the card is configured to boot from a link as opposed to boot from ROM by fitting jumper K1. A typical development example is shown in figure 4.1.

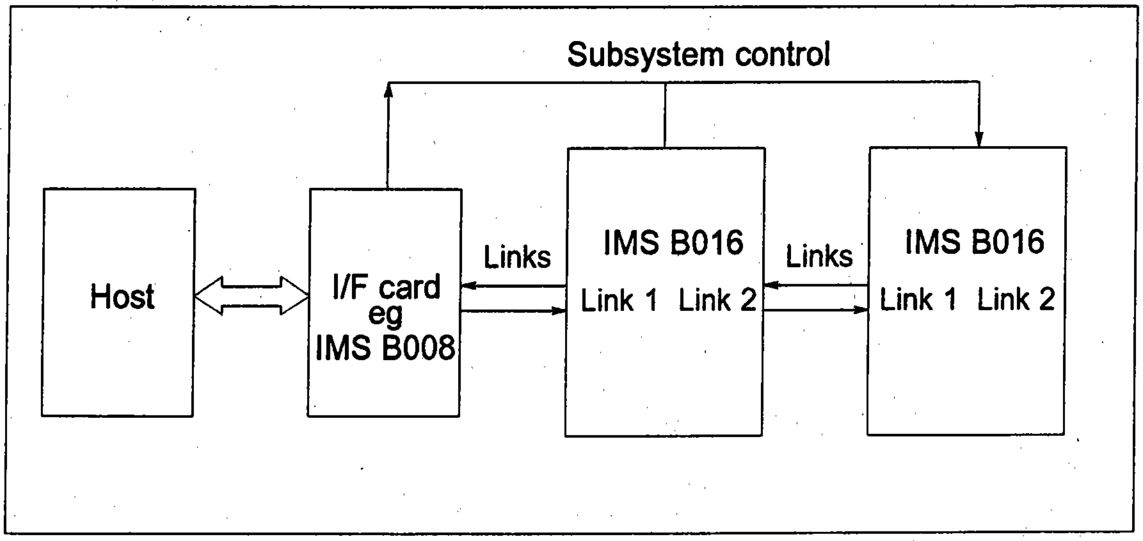


Figure 4.1

The IMS B016 is cabled to the development system by connecting a link cable between one set of serial link pins on connector P2 and the link ouput from the development system interface. Also connect a subsystem control cable between the services 'up' pins of connector P2 to the subsystem output from the development system interface. The nature of P2 connections varies on different VMEbus card cages but usually long wire–wrap tails are provided for pin rows A and C. Standard INMOS link connector cables can be plugged directly onto such pins provided the keyway is removed from the plug. As an example, using an IMS B008 interface installed into a PC development host, the link connection would typically appear on the 'pipetail' pins on the breakout board fitted to the IMS B008 rear connector. This should be connected to link 1 of a master IMS B016 (pins A5 through A9 of P2). Also connect the subsystem, labelled 'SS' on the breakout board, to pins C28 through C32 of P2. Ensure that the linkspeed selection of the interface board matches that of the IMS B016 configured using jumper K2.

With the equipment powered up, it should be possible to run a diagnostic tool such as 'ispy' on the transputer network and confirm the connections made. An example output is shown below:

```
Using #200 ispy 2.33.1
   # Part rate Mb Bt [  Link0  Link1  Link2  Link3 ]
   0 T800c-20 0.05 0 [   HOST    1:1    3:1    ... ]
   1 T2    -17 0.88 1 [    ...    0:1    ...   C004 ]
   3 T801b-25 0.88 1 [    ...    0:2    ...    ... ]
```

If the diagnostic program does not report having found the processor on the IMS B016, re–check all the connections and linkspeed settings. Some of the example programs require two IMS B016 cards for operation. The link connections required between the cards are specified in the configuration files for those programs.

The rest of this discussion will assume access to a C toolkit and a connection to one or more IMS B016 cards. It will also assume familiarity with the transputer development tools and transputer terminology. It will also assume that the software is installed in a directory/drive called `$INSTALL`.

To work with IMS F008A set your ISEARCH environment variable to include the paths `$INSTALL/include/` , `$INSTALL/libs/` and `$INSTALL/clilib/`. It should already contain references to the C toolkit include and library directories. Each directory of source should include one or more link control files and configuration source files which specify, in a host independent manner, the files required to build each target executable. There is also an existing makefile, but it may not be in a form suitable for your host environment. The toolkit utility program 'imakef' can be used to generate the appropriate makefile in all circumstances based on the search paths, link control files and configuration sources. It will also re–use the macro definitions and rules defined above the 'cut line' seen in the files. Once the the software installation is complete, 'imakef' should be run routinely to generate consistent makefiles for the local setup and environment.

4.3 Monitor code

Two monitor implementations are provided within the IMS F008A installation – one communicates via a host development system and the other is totally standalone, using the serial I/O facilities of the IMS B016. This second implementation can be programmed into the ROM of the IMS B016. In addition a number of examples of function usage are supplied that illustrate particular functions of VMEbus interface programming. The two sets of code can be combined if desired to give a comprehensive user experimentation interface across multiple boards.

The source code for the monitor implementations is in the files `$INSTALL/monitor/monitor.c` and `$INSTALL/monitor/boardmon.c`. See Appendix D for a description of the generic features of the monitor command language. The command set specific to the IMS B016 monitor is documented below:

To run a monitor program on the IMS B016, the following steps must be taken:

- Configure ISEARCH variable as previously described
- Change directory to `$INSTALL/F008/monitor`.
- Make sure connections from development system to IMS B016 are known and correspond to the host link descriptions in the `monitor.cfs` configuration source file.
- On a PC development host run

  ```
  imakef /d /c monitor.btl
  ```

  which should complete without error or warning if the search paths have been set up correctly.
- Run a make utility referencing the makefile

  ```
  make /x monitor.mak
  ```

  All referenced code should compile and link to produce `monitor.btl`
- Use iserver to boot the executable code onto the IMS B016

  ```
  iserver /ss /sb monitor.btl
  ```
- As the code runs, a prompt should appear

  ```
  IMS B016 board monitor v1.0
  B016>
  ```

  A set of monitor commands can be run using the command statement `source ''test.com''`

If compiling the ROMable monitor code is desired, the code, link and configuration files are supplied as `boardmon.c`, `boardmon.lnk`, `boardmon.cfs`. Note that it will be necessary to delete the object file for the `clilib` module in order to force a recompilation with a different compiler parameter. The binary file `boardmon.bin` that results from making this variant of the monitor code can be programmed into the ROM of the IMS B016 using the FLASH command of the hosted monitor.

The monitor CLI commands implemented for the IMS B016 are described below Note that commands that require host file system support are not included in the ROMable monitor implementation.

DISPLAY

```
DISPLAY <address><range>
```

Performs a formatted printout of memory words from the start address given for the specified number of bytes. Both Hex contents and an ASCII decode are shown. The address parameter defaults to the value of the BASEADDRESS variable and the range parameter defaults to the value of the RANGE variable. The command returns the value of the highest address reached.

SEARCH

```
SEARCH <address><range><value>
```

Searches for a given word value over a range of addresses. The range parameter specifies the length in bytes of the search area. All word values within this area are compared, including those that span word boundaries.The address parameter defaults to the value of the BASEADDRESS variable and the range parameter defaults to the value of the RANGE variable. The value parameter defaults to the value of the DATAVALUE variable.

FILL

```
FILL <address><range><value>
```

Fills a range of memory words with a specified value. The range parameter specifies the byte length of the filled area.The address parameter defaults to the value of the BASEADDRESS variable and the range parameter defaults to the value of the RANGE variable. The value parameter defaults to the value of the DATAVALUE variable.

COPY

```
COPY <address><range><newaddress>
```

Copies memory contents from one address to another. The range parameter specifies the number of bytes to be copied. The copy is done in such as way as to avoid problems with overlapping areas of memory.The address parameter defaults to the value of the BASEADDRESS variable and the range parameter defaults to the value of the RANGE variable.

MAPRAM

```
MAPRAM <address>
```

Displays the IMS B016 mapram entry for a given address. The given address is truncated to the Megabyte boundary below the value given.

MUXRAM

```
MUXRAM <address><range><mapcode><rswop><wswop>
```

Command to set up the mux function and corresponding mapram entry for an range of addresses. The mapcode is formed by ORing togther the control bits (defined as constant symbols) and the byte mux function number. The byte mux function is programmed according to the swop values given.The address parameter defaults to the value of the BASEADDRESS variable and the range parameter defaults to the value of the RANGE variable.

SLAVE

```
SLAVE <address><range>
```

The command sets up a slave decode area corresponding to the address and range passed as parameters.The address parameter defaults to the value of the BASEADDRESS variable and the range parameter defaults to the value of the RANGE variable. 32 bit addressing is assumed.

ROM

```
ROM
```

Displays the size and device codes of the Flash ROM devices on the IMS B016. The command will fail if the VPP generator is disabled.

FLASH

```
FLASH <filename>
```

Command will read the binary formatted ROM file (as produced from ieprom) and program its contents into the Flash ROM devices of the IMS B016. The data will be alligned such that its last byte is programmed at addresss 7fffffff in order that code bootstrap sequences will be correctly placed. The filename is a string parameter or symbol reference.

BWRITE

```
BWRITE <string>
```

The command uses the B channel of the DUART device on the IMS B016 to write out a string value. The default setup of the port is 9600 baud.

BREAD

```
BREAD
```

The command reads a string from the B channel of the DUART device. The input is terminated by CR or LF characters.

RTC

```
RTC <timecode>
```

The command will set or read the clock device on the IMS B016. If a time value is given, it is expressed in seconds since 1st Jan 1970, and this is programmed into the clock. If no value is given, the current clock value is returned and formatted into a date/time string. The command will fail if standby power is not applied to the clock device.

INIT

```
INIT <analyse><linkno>
```

Command resets and reads the error status of the transputer subsystem port. It is possible to change the link number associated with this port. The default is link 2. If the analyse flag is TRUE, then an analyse and reset cycle is performed.

PEEK

```
PEEK <address><range>
```

Performs a formatted printout of memory words from the start address given for the specified number of bytes. The data is obtained via transputer peek instructions given to a reset processor connected by a link. Both Hex contents and an ASCII decode are shown. The address parameter defaults to the value of the BASEADDRESS variable and the range parameter defaults to the value of the RANGE variable.

POKE

```
POKE <address><range><value>
```

Fills an area of remote transputer memory with the given value. The address parameter defaults to the value of the BASEADDRESS variable and the range parameter defaults to the value of the RANGE variable. The data defaults to the value of the DATAVALUE symbol.

4.4 Examples and BSP code

The source of the example programs is contained in the directory `$INSTALL/examples`.

The example code includes configurations that test multi–board operation. To build an example bootable, copy the appropriate master and slave source files to `master.c` and `slave.c` before running make. The bootable file in each case is called `vmetest.btl`.

- example 1 (`master1.c, slave1.c`) demonstrates simple setup of master and slave access capabilities between two IMS B016 cards.
- example 2 demonstrates VMEbus interrupt operation under software control.
- example 3 demonstrates both remote memory access and byte swapping functions.
- example 4 demonstrates the interfacing the IMS B016 to a ADC I/O card.

An additional implementation example is included in the `$INSTALL/vrtxbsp` directory. This code is the support package required to run tasks under the VRTX32/T real time executive product from Ready Systems Inc. and shows how the IMS F008A routines can be used in conjunction with this package. For further details of VRTX32/T, contact your INMOS/SGS–THOMSON sales office.

4.5 Basic library

The basic library forms the core of the IMS F008A software. It should be linked with all programs which use any other parts of the IMS F008A libraries. It contains the fundamental setup routines to initialise the IMS B016 for use. Note that most of these routines are below the level of abstraction normally required to program the IMS B016 and therefore will only be used for special low level programming requirements. The routines in the VMEbus library use these basic functions, but the overall functions are expressed in a higher level manner.

The source code of these routines is in the file **`$INSTALL/libs/basic.c`** The header file associated with use of this library is **`$INSTALL/include/basic.h`**.

basic_board_init

```
void basic_board_init()
```

This routine must be called before any other routine in the libraries can be used. Other parts of the library check to make sure that this routine has been called and, where possible, return an error if initialisation has not been done.

When this routine is called, the board is initialised, the mapram and decode ram are cleared to zero, the byte multiplexer is disabled and low–level routines used to write to the mapram and the decode ram are moved into the internal memory of the transputer. This process assumes that the stack space allocated to the run-time code has been configured to be at the lowest memory addresses (normally done using the order statement in the configuration description) and that sufficient space has been allowed on the stack such that the area where these routines are placed does not get reached in the normal expansion of stack at runtime. In practice the routines occupy about 22 words of memory and therefore do not run any real risk of being corrupted by the expansion of the stack.

check_board_init

```
int check_board_init()
```

This routine tests to see if **`basic_board_init()`** has been called. TRUE is returned if this is the case and FALSE is returned otherwise.

clear_map_ram

```
void clear_map_ram()
```

This routine clears all the IMS B016's mapram entries to zero. On the IMS B016, mapram entries cannot be read back. They can only be written. To get around this, the mapram routines store all mapram entries in a shadow array from which they can be read and examined. These shadow entries are also cleared by this routine.

shadow_write_map_ram

```
void shadow_write_map_ram(void *address,int data)
```

`*address`	pointer to the memory region whose map ram entry you wish to set.
`data`	the mapping value for this region

A region of memory is defined to be a 1 Megabyte page extending upwards from the given address or the 1 Megabyte page containing the given address. Since these addresses will be related to the VMEbus address space, it will be usual to give positive valued addresses (MSB zero) which can relate most readily to the addresses available on the other cards in the racked system.

The map RAM entry is a bitfield whose elements determine the mapping function required for a region of memory. Constants are provided in the `$INSTALL/include/mapram.h` file to generate this value. These are:

```
MASTER_PERMIT
AM_UDATA
AM_UPROG
AM_SDATA
AM_SPROG
ASIZ_AMSOURCE
ASIZ_32BIT
ASIZ_24BIT
ASIZ_16BIT
LADD00
LADD01
LADD10
LADD11
DSIZ_32BIT
DSIZ_16BIT
DSIZ_8BIT
DSIZ_THREE
```

Constants with the same prefix (e.g. AM_ or ASIZ_) are mutually exclusive. The appropriate constants should be OR'd together to make a valid mapram entry describing the function required for this region.

For example, to access a region on the VMEbus using 24 bit address and 32 bit data with the lower address bits always equal to 00 and specifying supervisor data access the following code would be used.

```
    data = (MASTER_PERMIT | AM_SDATA | ASIZ_24BIT |
                          DSIZ_32BIT | LADD00);
```

A mapram entry will need to be set up for each region of memory to be accessed. If the address space to be accessed is more than 1 Megabyte in extent, then multiple entries will need to be set up.

shadow_read_map_ram

```
int shadow_read_map_ram(void *address)
```

`*address`	pointer to the memory region whose mapram entry you wish to read.

This routine returns the mapram entry of the memory region specified by the address given.

turnon_byte_mux

```
void turnon_byte_mux()
```

This routine enables the byte multiplexor function of the IMS B016.

turnoff_byte_mux

```
void turnoff_byte_mux()
```

This routine disables the byte multiplexor function. The multiplexor should be disabled when updating a swap function.

set_mux_0_transparent

```
void set_mux_0_transparent()
```

This routine programs swap function 0 of the byte multiplexor to be transparent (that is a null mapping function). This will typically be done so that the boards registers and memory can be accessed without mapping operations being performed on them while the byte multiplexor is turned on.

program_byte_mux

```
void program_byte_mux(int which_fn, int read_swap, int
                      write_swap)
```

`which_fn`	is a number from 0 to 7 that specifies the swap function to be programmed.
`read_swap`	is a 32 bit integer where each nibble is a number from 0 to 7, specifying the number of the nibble on the boards data bus which will be mapped into that nibble position, as seen from the transputer, when a read is performed.
`write_swap`	performs the same function as above but define the mapping which will take place on a write operation.

This routine programs one of the 8 available byte swapping functions of the byte multiplexor. Setting a read swap value of 0x76543210 for a given function would be a null mapping operation for the word being read, since each nibble is mapped into the its original position. A mapping code of 0x32103210 would map the lower 16 bits of a 32 bit word on the data bus into both the upper and lower 16 bits as seen by the transputer.

program_mux_map_ram

```
void program_mux_map_ram(int which_fn,void *address)
```

`which_fn`	a number from 0 to 7 which specifies the swap function used in conjunction with this mapram entry.
`void *address`	pointer to the memory region whose map ram entry is to be set.

This routine programs the multiplexor select bits of a mapram entry. These bits define which swap function will be invoked when a region of memory is being accessed. This function must be used in conjunction with the other mapram entry capabilities described above.

clear_decode_ram

```
void clear_decode_ram()
```

This routine clears all the IMS B016's decode ram to zero. This effectively disables slave access to the boards memory.

write_decode_ram

```
void write_decode_ram(void *address,int val)
```

`*address`	pointer to the memory region whose decode ram entry is to be set.
`val`	the value for the decode ram control bit.

This routine allows setting the contents of the decode ram. Setting the decode ram control value to 2 enables slave mode access to that region. Setting it to zero disables slave accesses.

Decode memory regions are specified by the upper 16 bits of the address value given and thus slave access can be enabled or disabled over regions 64Kbytes in size as seen from the VMEbus address space. See also the VMEbus library for additional decode ram programming functions.

4.6 VMEbus library

The VMEbus library provides a set of functions which can be used to configure the interface of the IMS B016 to the VMEbus. Since the board can function as a slot one controller or a general purpose master/slave processor and memory card, the interfaces allow a variety of control options. This is the normal library of routines that have usefully abstract functions, enabling rapid and error free setup of board functionality. It is therefore recommended that this is the primary library used for master and slave programming.

All routines in this library return zero for success and a non–zero value for failure.

The source code for this library is in the file `$INSTALL/libs/vme.c` and the header file associated with its use is `$INSTALL/include/vme.h`.

Since much of the operation of these routines is concerned with the mapping of VMEbus address space to transputer address space in multiple cards, the reader is referred to Appendix A for a full discussion of this subject.

init_vme_master

```
int init_vme_master(int request_level, int re-
                    quest_mode, int writepost)
```

`request_level`	a number from 0 to 3, which specifies on which bus grant chain the bus requests will be made.
`request_mode`	identifies the mode of bus request made.
`writepost`	may be TRUE or FALSE which enables or disables master mode writeposting.

This routine sets up the parameters for master mode accesses from the IMS B016 across the VMEbus. It should be called early in the board initialisation process and certainly before remote memory access is required. The bus request mode can be one of:

RELMODE_ROR for release on request mode.
RELMODE_RWD for release when done mode.
RELMODE_ROC for release once complete mode.
RELMODE_BCAP for bus capture and hold.

init_slot_one

```
int init_slot_one(int arbitration, int bus_timeout)
```

`arbitation`	can be ARBITER_PRI for prioritized arbitration or ARBITER_RRS for round robin arbitration.
`bus_timeout`	identifies bus error time out strategy:

This routine enables the system controller functions of the IMS B016. These will take effect if the board is located in slot one of a VMEbus backplane, and should be called early in a system initialisation process. The parameters of the function set the mode of operation for the bus arbiter and the timeout time for VMEbus accesses. The timeout strategy value should be one of the following constants:

VMETIMEOUT_4US for 4us timeout
VMETIMEOUT_16US for 16us timeout
VMETIMEOUT_32US for 32us timeout
VMETIMEOUT_64US for 64us timeout
VMETIMEOUT_128US for 128us timeout
VMETIMEOUT_256US for 256us timeout
VMETIMEOUT_512US for 512us timeout
VMETIMEOUT_INF for no timeout

prog_slave_access

```
int prog_slave_access(void *decode_address, int de-
                      code_window, int address_type, int wri-
                      tepost)
```

`decode_address`	the starting address of the desired memory region as seen from VME address space.
`decode_window`	is the size of the slave memory region in bytes. This value is rounded up to the next 64kbyte boundary.
`address_type`	may be either 24 or 32 to indicate the number of valid address bits that will be used on the VMEbus.
`writepost`	may be either TRUE or FALSE and enables or disable writeposting on slavemode write accesses.

This routine sets up slave mode access and enables a region of memory to respond as slave memory. Regions of memory accessible as slave memory across the VMEbus are multiples of 64Kbytes. The base location of any region of slave memory, in terms of an offset from the base address of the transputer memory, is given by bits 16–21 of the region address. The upper region address bits (16–31) determine the base address of the region within the VMEbus address space.

prog_master_access

```
int prog_master_access(void *address, int address_win-
                       dow, int data_type, int address_type,
                       int low_bits, int mux_fn)
```

`address`	Start address of memory region, expressed as a VMEbus address
`address_window`	the size of the memory region to be accessed.
`data_type`	may be 8, 16 or 32 signifying the expected data value width.
`address_type`	may be 16, 24 or 32 signifying the expected address width.
`lower_address`	may be 0,1,2 or 3 to set lower address bits for this access.
`mux_fn`	The byte mux function to be applied to data values from this region

This routine sets up a memory region for master accesses by programming the mapram appropriately. It also enables the byte multiplexor function. Regions of memory accessible via master access across the VMEbus are multiples of 1Mbyte

turnon_vme_master

```
int turnon_vme_master()
```

This routine enables mastermode accesses for all memory regions whose mapram master enable bit is set.

turnoff_vme_master

```
int turnoff_vme_master()
```

This routine globally disables mastermode accesses, overriding the mapram master enable bits.

send_vme_interrupt

```
int send_vme_interrupt(int int_level)
```

`int_level`	VMEbus interrupt number between 1 and 7 inclusive.

This routine generates an interrupt on the VMEbus interrupt line specified in the parameter.

prog_vme_interrupt_vector

```
int prog_vme_interrupt_vector(int int_level,int vector)
```

`int_level`	is the level of VMEbus interrupt between 1 and 7 whose vector you wish to program.
`vector`	the vector number between zero and 255 to be associated with this interrupt. Not all vectors are available for re–allocation.

This routine is used to specify the interrupt vector number used when an specific VMEbus interrupt is acknowledged. This vector number is used to associate a specific handler with the interrupt using routines in the event library.

4.7 RTC library

One of the support devices on the IMS B016 card is a Real Time Clock chip. This is an autonomous time–keeping device with a resolution of 1/100 second. If the card is supplied with stand–by power, then this device will maintain timekeeping even during power and programming cycles of the other hardware. The routines in this library allow setting and reading of the clock registers as though they were expressed in ANSI C time format. The routines therefore operate with parameter types of `time_t` as defined in `<time.h>`. Time in this regime is expressed as the integer number of seconds since 1st Jan. 1970.

The source code for this library is in the file `$INSTALL/libs/rtc.c` and the header file associated with its use is `$INSTALL/include/rtc.h`.

rtc_settimeofday

```
time_t rtc_settimeofday(time_t *time);
```

`time`	pointer to time value

The time value given is programmed into the clock chip and the time cycle is started. The clock will be maintained from this point unless stand–by power is removed.

rtc_gettimeofday

```
time_t rtc_gettimeofday(time_t *time);
```

`time`	pointer to time value

The current time value maintained by the clock is returned.

4.8 Event library

Although the transputer does not directly support a vectored interrupt scheme, the large number of interrupt sources in the VMEbus environment make such a scheme very desirable. The routines in this library support a implementation of a vectored scheme using a single high priority transputer process acting as a dispatcher. This mechanism gives an easy way to assign event handlers at run–time to each source of interrupt. The source of interrupts is determined using the VIC support features. The dispatcher function clears the generic control registers associated with the generation of an interrupt, but specific event handlers must clear the individual source of interrupt associated with their vector.

VMEbus errors are handled by a default bus error handler once the event handler has been initialised, however a function is provided to substitute this for another handler if required. Interrupt vectors with no associated handler routine are passed to a default handler and the event dispatcher attempts to clear the event by generic actions. This should be avoided though, since if the interrupt does not clear, the system may be held in a tight service loop in response to repeated events.

The source code for this library is in the file `$INSTALL/libs/event.c` and the header file associated with its use is `$INSTALL/include/event.h`.

start_event_handler

```
int start_event_handler();
```

This routine initialises the interrupt vector table and installs the high priority dispatch process. It also initialises some default handlers in pre–specified vector slots. The pre–defined vector numbers for particular handler functions are:

VECTOR_PEX 11
VECTOR_DUART 12
VECTOR_BERR 9
VECTOR_RTC 10
VECTOR_ACFAIL 16
VECTOR_WRITEPOST 17
VECTOR_ARBTIME 18
VECTOR_SYSFAIL 19
VECTOR_INTACK 20
VECTOR_ICMS0 4
VECTOR_ICMS1 5
VECTOR_ICMS2 6
VECTOR_ICMS3 7
VECTOR_ICGS0 24
VECTOR_ICGS1 25
VECTOR_ICGS2 26
VECTOR_ICGS3 27
VECTOR_INT_1 28
VECTOR_INT_2 29
VECTOR_INT_3 30
VECTOR_INT_4 31
VECTOR_INT_5 32
VECTOR_INT_6 33
VECTOR_INT_7 34

install_vme_int_handler

```
int install_vme_int_handler(int level,int vector,void
                   (*routine)())
```

`level`	The VMEbus interrupt level to be serviced.
`vector`	The vector number which will be generated by interrupt when issued.
`(*routine)()`	Function pointer to the interrupt handler routine to be called.

This routine is called to install an interrupt handler routine for a VMEbus interrupt and programs the VIC to receive interrupts at that interrupt level. If the vector is not available, the routine will return a non–zero value. Locally generated interrupts can require some different setting up to VMEbus interrupts. Therefore separate handler install routines are provided for the local DUART and bus error interrupts.

install_vme_int_ack_handler

```
int install_vme_int_ack_handler(void (*routine)())
```

`(*routine)()`	Function pointer to the interrupt handler routine to be called on the occurrence of the interrupt.

This routine is used to install an interrupt handler routine which is called when a remote board acknowledges a locally generated VMEbus interrupt. A default handler is installed initially that simply resets the appropriate VIC register. This routine allows the installation of a handler that adds additional user functionality.

install_duart_int_handler

```
int install_duart_int_handler(void (*routine)())
```

`(*routine)()`	Function pointer to the interrupt handler routine to be called when the DUART interrupts.

This routine is used to install the interrupt handler for local DUART interrupts.

install_berr_handler

```
int install_berr_handler(void (*routine)()
```

`(*routine)()`	`Function pointer to the interrupt handler routine to be called on local bus errors.`

This routine is used to install a handler routine for bus error interrupts. Bus error control registers are in fact cleared by the default berr handler, therefore this routine allows setup of explicit user handling of the fact that a bus error has occurred. As an alternative, the default berr handler can be used and user software may poll the public variable `berr_happened` which gets set to TRUE on a berr. This flag should be explicitly reset each time it is found set by user code.

give next_vector

```
int give next_vector(int vector, void (*routine)())
```

`vector`	`Start vector number for search.`
`(*routine)()`	Function pointer to the interrupt handler routine to be installed.

This routine installs the handler function in the next available vector above the given value and returns the vector number. If this is not possible it returns a negative value.

ask_to_use_vector

```
int ask_to_use_vector(int vector, void (*routine)())
```

`vector`	vector number to test.
`(*routine)()`	Function pointer to the interrupt handler routine to be installed.

This routine will program the function pointer into the vector table at the specified position unless that vector number is already taken, when the routine returns a non–zero value.

4.9 DUART library

The IMS B016 supports serial I/O using a DUART device. The routines in this library can be used to program this device and support its use in interrupt driven I/O operations with or without flow control. The library therefore depends on the operation of the event library previously described.

The source code for this library is in the file `$INSTALL/libs/duart.c` and the header file associated with its use is `$INSTALL/include/duart.h`.

Incoming and out–going characters for the I/O routines are stored in circular buffers of lengths defined in the header file. The flow control characteristics, given by XON–XOff handshakes, are also set by margin values, also in the header file.

Most routines in this library come in pairs – one for the A port and one for the B port. The two channels work completely independently after initialisation is complete.

duart_init

```
void duart_init()
```

This routine initialises the DUART handler, installs the interrupt handler routines and sets up the buffer management processes. It must be called after the event handler code is initialised, but before any other use of the DUART library routines.

duart_use_xon_a/b

```
void duart_use_xon_a(int use)
void duart_use_xon_b(int use)
```

This routine is used to switch on and off Xon/Xoff flow control for the ports. If the parameter value is TRUE then flow control will be used, otherwise not.

duart_write_a/b

```
void duart_write_a(char *text)
void duart_write_b(char *text)
```

This routine outputs a null terminated string given as the parameter. The maximum length for this string is 255 characters. The routine will block if there is insufficient room in the output buffer.

duart_putchar_a/b

```
int duart_putchar_a(char c)
int duart_putchar_b(char c)
```

This routine outputs a single character on the serial port. The routine will block if there is insufficient space in the output buffer.

duart_getchar_a/b

```
char duart_getchar_a()
char duart_getchar_b()
```

This routine accepts a character from a serial port and will wait until one is received. The character is not echoed or interpreted in any way.

duart_setup

```
int duart_setup(int parameter, int value)
```

`parameter`	key value to sub–define function to be setup.
`value`	value of option

This routine is used to setup the various parameters of the IMS B016 serial ports. After the initialisation function is called, the ports are in a reset state. In calling this setup routine the 'parameter' value specifies what function is set and what significance, if any, should be attached to the value parameter.

'Parameter' may take any of the following values:

```
DUART_TX_BAUD_A
DUART_TX_BAUD_B
DUART_RX_BAUD_A
DUART_RX_BAUD_B
DUART_DATA_BITS_A
DUART_DATA_BITS_B
DUART_STOP_BITS_A
DUART_STOP_BITS_B
DUART_PARITY_A
DUART_PARITY_B
DUART_ENABLE_TX_A
DUART_ENABLE_TX_B
DUART_ENABLE_RX_A
DUART_ENABLE_RX_B
DUART_RESET_ERROR_A
DUART_RESET_ERROR_B
DUART_RESET_TX_A
DUART_RESET_TX_B
DUART_RESET_RX_A
DUART_RESET_RX_B
```

The defined values for setting the baud rates, stop bits and parity functions are provided in the header file. The reset and enable commands take no value parameter.

4.10 ROM library

The IMS B016 is fitted with re–programmable ROM devices – Flash ROMs. The board includes all control circuitry to re–program these devices in situ, and this library supports this process. The programming voltage required for this (VPP) must be controlled explicitly and must be enabled via the appropriate board jumper. There is insufficient power to program all 32 data bits at once and so the programming routines select an individual byte–wide ROM (0–3) as well as an address of the 32 bit word in which this byte is found.

Note that the ROM address space on the IMS B016 is based at transputer address 0x7fe00000. If the physical ROMs fitted occupy less address space than this, then the data will repeat through this address space to the top of transputer memory. If you are programming an IMS B016 to boot from ROM, then the topmost word in the memory space must contain a jump vector as described in the toolkit manual.

The source code for this library is in the file `$INSTALL/libs/rom.c` and the header file associated with its use is `$INSTALL/include/rom.h`.

reset_rom

```
void reset_rom()
```

This routine initialises all ROM devices to a known read only state and turns off any programming voltage.

blank_check_rom

```
int blank_check_rom()
```

This routine searches the contents of the ROMs and returns a code indicating their programming state. 0xff is returned if they are blank (i.e. all bytes are unprogrammed), zero is returned if they are fully programmed (i.e all locations are zeroes). If the ROMs are found with any other value, the function returns with the address of the first inconsistent address.

turnon_vpp

```
void turnon_vpp()
```

This routine turns on the 12 volt VPP power generator. There is a small delay built into this routine after turning the supply on in order to allow voltage to stabilise.

turnoff_vpp

```
void turnoff_vpp()
```

This routine turns off the 12 volt VPP power supply.

find_rom_size

```
int find_rom_size()
```

This routine returns the size in bytes of the ROMs. It does this by reading the device code of the chips to establish their capacity. Intel and AMD parts are recognised.

read_rom_device_code

```
int read_rom_device_code(int which_rom)
```

`which_rom`	selects the ROM (0–3) for this operation

This routine returns the single byte device code each ROM on the board. VPP must be on for this to work.

read_rom_manf_code

```
int read_rom_manf_code(int which_rom)
```

`which_rom`	selects the ROM (0–3) for this operation

This routine returns the single byte manufacturer code of each ROM on the board. VPP must be on for this to work.

rom_read_mode

```
void rom_read_mode(int which_rom)
```

`which_rom`	selects the ROM (0–3) for this operation

This routine sets the selected ROM into read mode. VPP must be on for this to work.

program_byte

```
int program_byte(int which_rom,int address,int data)
```

`which_rom`	selects which byte wide ROM to be programmed
`address`	address of word to be programmed
`data`	byte of data to be programmed

This routine programs the byte in the lower 8 bits of the data parameter into the selected ROM at the specified address. A non–zero value is returned if the operation fails. VPP must be on for this to work.

erase_rom

```
int erase_rom(int which_rom)
```

`which_rom`	selects the ROM (0–3) for this operation

This routine erases the selected ROM. A non–zero value is returned if the operation fails. VPP must be on for this to work.

4.11 CLI library

To support the creation of a useful command line driven monitor program, a library of utility functions is included and used within this software package. The library implements a command language as described in Appendix D and is extensible and configurable as shown within the IMS B016 monitor.

The source code for this library is in the file `$INSTALL/cli/clilib.c` and the header file associated with its use is `$INSTALL/include/clilib.h`.

The example monitor code demonstrates how the basic language can be used in different environments and with different command sets. All commands required in a given environment can be built out of library procedures offered by this module and described below.

The command line parsing routines use a structure of type `parse_t` which carries information about the current line and the state of parse operations. The structure is defined in the header file. The implementation also defines various types of symbolic variable, and keeps a table of such variables organised as a stack.

init_pars

```
void init_pars(parse_t *pars_p, char *new_line_p);
```

`pars_p`	pointer to structure which will be initialised to perform parsing of a new line.
`new_line_p`	new string to be decomposed.

Initialises parse structure with a fresh input line. By declaring and initialising multiple parse structures several different lines can be parsed together.

get_tok

```
short get_tok(parse_t *pars_p, char *delim_p);
```

`pars_p`	pointer to structure defining current line of interest and parse position.
`delim_p`	string containing a set of characters that will be allowed to terminate a token.

This is the fundamental parsing routine within the library. It allows the decomposition of the current line into tokens delimited by specified characters. The call is made with an initialised or previously used parse structure and the function extracts the next set of characters on the line ending in one of the given delimiter characters. A token is also terminated by reaching the end of current line. All details about the fresh token are returned in the updated parse structure. The function also returns the length of the token found, not including the delimiter Tokens found can span pairs of matched double quotes, and matched numbers of opening and closing brackets.

assign_value

```
boolean assign_integer(char *token_p, long value, bool-
                ean constant);
boolean assign_float(char *token_p, double value, bool-
                ean constant);
boolean assign_string(char *token_p, char *value, bool-
                ean constant);
```

`token_p`	a name for a new or existing symbol.
`value`	a new value for the symbol (of each type)
`constant`	for a new symbol, makes the value un–modifiable if true.

These routines create and assign values to symbols of integer, floating point and string types. The routine will return TRUE if the assignment cannot be performed. This could be the case if a symbol exists but is a constant, or a new symbol is to be created but the name would clash with an existing symbol.

register_command

```
boolean register_command(char *token_p, boolean (*ac-
                tion)(parse_t*, char*), char *help_p);
```

`token_p`	name of new command procedure
`(*action)()`	routine to be called when command is recognized
`help_p`	short help string associated with the command

This routine allows the programmer to add command procedures to the language and provide simple help facilities for them. The command procedure will be called with a parse structure containing the line in which it is referenced and a possible symbol name to which it can assign a value.

evaluate_value

```
boolean evaluate_integer(char *token_p, long *value_p,
                short default_base);
boolean evaluate_float(char *token_p, double *value_p);
boolean evaluate_comparison(char *token_p, boolean *re-
                sult_p, short default_base);
boolean evaluate_string(char *token_p, char *string_p,
                short max_len);
```

`token_p`	token containing possible value or expression
`value_p`	pointer to variable in which to return result
`max_len`	maximum length of string result
`default_base`	default number base with which to interpret integer numbers

This set of routines allows an arbitrary string to be examined and decomposed in an attempt to resolve a single value from its structure. The input may vary from a simple constant number to a complex nested expression involving symbol references. Each routine will look for unique characteristics to determine the type of the possible value. The functions will return TRUE if the string could not be resolved to a value.

cget_value

```
boolean cget_string(parse_t *pars_p, char *default_p,
                    char *result_p, short max_len);
boolean cget_integer(parse_t *pars_p, long default,
                    long *result_p);
boolean cget_float(parse_t *pars_p, double default,
                    double *result_p);
boolean cget_item(parse_t *pars_p, char *default, char
                    *result_p, short max_len);
```

`pars_p`	pointer to structure defining current line of interest and parse position.
`default`	A value to be returned if the parameter is missing or in error
`result_p`	A pointer to a variable in which to return the result
`max_len`	The maximum length for string results

Building on the evaluation routines described above, these functions attempt to find a token on the current line and resolve a value from it. The current position on the line is advanced during each call. The delimiters used for these functions are a default set including white space, commas and backslashes. If a particular type of parameter is expected for a command, the appropriate routine is called and it returns a value. This value could be the given default, if the parameter was not present, or the result of an evaluation operation as above. The routines return TRUE if a parameter was found on the line but it could not be evaluated. If no parameter was found, then no error is returned.

print

```
void print(const char* format, ...);
```

`format`	format string as used by 'printf' etc.
`...`	variable list of printed values

This function performs formatted output using standard facilities, but does not write to the standard I/O stream. It uses instead a user supplied function `io_write(char *string)` which can be supplied for any I/O device.

tag_current_line

```
void tag_current_line(parse_t *pars_p, char *mes-
                      sage_p);
```

`pars_p`	pointer to structure defining current line of interest and parse position.
`message_p`	error message string

The explicit way to generate an error message for the current input line. The input line is echoed and the current position, assumed in error, highlighted.

cli_main

```
void cli_main(boolean (*setup_r)(), long max_symbols,
              short default_base, char *ip_prompt_p,
              char *file_p);
```

`(*setup_r)()`	pointer to a routine that adds user commands and variables.
`max_symbols`	defines the size of the symbol table that will be needed at run–time.
`default_base`	the initial number base for integer input and output.
`ip_prompt_p`	the prompt string to use for interactive input
`file_p`	the name of an initial file of commands to be executed or NULL.

This is the entry point of the command line code. It allows for the calling of a user setup routine which can have any content, but is expected to register commands and create variables. It also optionally allows for a named file of commands to be executed at start–up. The size of the symbol table should include an estimate for the depth of macro calls which will be employed.

5 Detailed hardware description

5.1 Using the IMS B016

For the majority of users it is intended that IMS F008A should provide all the facilities required for applications programming. This chapter gives low level hardware programming details for users who are either modifying IMS F008A code for their own purposes or writing configuration code from scratch.

5.2 Primary control registers

The primary control registers are only accessible by the IMS T801 and appear in two possible places in the memory map. The primary position is from #7FDC0000, while a secondary address at location zero is provided for compatibility with other transputer boards and TRAMs (table 5.1). This region of the address space is often required for VMEbus addressing and on the IMS B016 can be disabled by writing a one to the register as indicated in table 5.1.

The primary control registers are accessed as memory locations as shown in table 5.1. Bit 0 is the only active bit in these registers. The other bits are undefined when read and must be written as zero. These registers contain the traditional "subsystem" registers as found on other transputer cards; control bits for the MAP-RAM and decode ram; enable control bits for VMEbus master and slave cycles and Byte Multiplexor control registers.

Address	Register	Function
#7FDC0000	Subsystem Reset/Error	Write '1' to assert notSubReset signal on P2. Read bit 0 to sense level of notSubError signal on P2
#7FDC0004	Subsystem Aalyse	Write '1' to assert notSubAnalyse signal on P2.
#7FDC0040	Enable VMEbus Slave Accesses	Write bit 0 to allow other VMEbus MASTERs to access IMS B016's dual access DRAM as SLAVE memory. Note that decode RAM and VIC must be setup first.
#7FDC0044	Enable Byte Multiplexor	Write bit 0 to enable the byte multiplexor function. When disabled, all data passes through byte–mux unaffected.
#7FDC0048	Next Cycle is to the Map RAM	Indicates a write to MAP–RAM is to occur next.
#7FDC004C	Next Cycle is to the Slave Decode RAM	Indicates a write to slave VMEbus decode RAM is to occur next.
#7FDC0050	Do not Map these registers to address zero	Disable decode of these primary registers at address #000000 –useful for accessing that address region as VMEbus memory instead.
#7FDC0054	Enable VMEbus Master Accesses	Write '1' to allow VMEbus master operation.

Table 5.1 T801 Primary Control Registers

5.3 VMEbus MASTER configuring

Configuring for VMEbus MASTER access requires the MAP-RAM, Byte Multiplexor, VIC and primary control registers to be setup correctly. Generally one would program the MAP-RAM first, then the Byte mux (if required), then the VIC and finally enable master operation by setting bit 0 in the register at address #7FDC0054.

5.3.1 MAP–RAM

The MAP-RAM contains control information for every address which can be generated by the IMS B016's IMS T801 transputer. For every IMS T801 cycle, the MAP-RAM produces a set of control information which is used to determine how the various parts of the board's circuitry behave. For instance the MAP-RAM determines what kind of VMEbus cycle will be performed when the transputer attempts to perform a VMEbus master cycle.

The MAP-RAM has an entry for each memory *page* . Pages are one megabyte in size and so it is impossible to have different MAP-RAM entries for two addresses unless they lie on different megabyte pages.

The MAP-RAM needs to be initialised at system start–up to contain the correct information for the application. Because of the way the MAP-RAM is written, it is unwise to attempt to change its contents after this point.

MAP-RAM entries are written as follows:

1. Write a “one” into the “Next Cycle is to MAP-RAM” control register.
2. The *very* next cycle must be a write to the address of the MAP-RAM page to be programmed. The data for the 12 MAP-RAM bits must be located in bits 4–15.

Note that the address written to is in fact the address to which the MAP-RAM entry refers. For instance, if you wished to set up the MAP-RAM entry for addresses `#01000000`–`#01100000` (one megabyte), then you could write your MAP-RAM contents to any address in that range. Note that addresses `#80000000`–`#80001000` are the IMS T801's internal RAM and registers and so no MAP-RAM configuration cycles can be performed to these addresses. For consistency, it is recommended that the last address in the page is always used.

It is most important that there are no IMS T801 memory cycles performed between steps 1 and 2 above. Examples of situations which would break this rule are:

- Running multiple tasks on the transputer–execute this code at high priority.
- Running the code which does the initialisation of the MAP-RAM from ROM or any other memory apart from on-chip RAM. Note that this does not prevent ROM booted programs from configuring the MAP-RAM, one would arrange for the relevant code to execute from internal RAM.
- Compilers which perform “constant caching”–that is, code is written which specifies that a constant (usually located alongside the program code) is to be written to the MAP-RAM; but the compiler actually stores the constant into a temporary location because it is used multiple times in the same program-segment.

IMS F008A routine `write_map_ram()` achieves these conditions by copying a code fragment into internal RAM and jumping to it. Users may wish to examine the source code. This technique is tuned to the requirements of INMOS' toolset 'C' compiler and other compilers may need a different approach.

The MAP-RAM controls the following aspects of the board's behaviour:

- Whether the page is allocated as VMEbus address space. You can restrict which address regions will, when accessed by the IMS T801, cause VMEbus master cycles to be initiated. The MAP-RAM pages for addresses *definitely not* to be used for VMEbus access should have the VMEbus Master Access Enable bit cleared otherwise strange things can happen. The best way to ensure that this is the case is to have initialization code clear the complete MAP-RAM contents before programming any MAP-RAM pages with valid contents.

- The VMEbus address modifier codes produced when the board preforms VMEbus master cycles. This is controlled by two MAP-RAM bits, working in conjunction with other control bits in the VIC registers. It is possible to make the board generate all the usual address modifier codes and any user-defined ones[3].

For most applications, it is sufficient to program these two bits to 0,0 for 'User Data access".

- The value presented on the lower two VMEbus address lines during D08 and D16 cycles. Since the IMS T801 has no way of generating the byte and word cycles available on Motorola M68000 series processors, all VMEbus master cycles are seen by the transputer as 32-bit memory accesses. One consequence of this feature is that the IMST801 only produces 30 bits of address bus. The MAP-RAM supplies the one or two lower address lines for the VMEbus.

In use, these MAP-RAM bits allow different pages to be programmed such that the transputer can perform a BYTE(0–1) access using one page and a BYTE(2–3) access using another. The two pages would actually cause VMEbus transfers to the same slave. Similarly, four pages can be used to access all four different bytes from a D08 slave.

3.
VMEbus *address modifiers* are basically five extra address lines which are used to indicate certain things about the bus cycle in progress. The value present on the address modifier lines is called an *address modifier code*. The codes defined in the VMEbus specification specify the type of addressing (A32/24/16), whether a block-transfer is beginning, whether the CPU initiating the transfer is running in user or supervisor state and whether the access is a code or data access. Since the transputer does not currently have either a user or supervisor state and does not give an external indication of data and code accesses, the MAP-RAM can be used to generate this information.

This means that one can only usefully perform D08 and D16 transfers in the A16 and A24 address spaces. Hopefully this is not a practical restriction.

- The VMEbus data size. This can be selected to be D32, D16, D08 and THREE-BYTE. This is selected by MAP-RAM bits 10 and 9 as follows: 0,0 for 32-bit, 0,1 for 8-bit, 2,0 for 16-bit and 1,1 for 24-bit. For those interested in the full story, these bits are connected to the VIC SIZ1,0 pins.
- The VMEbus address space used for master transfers. This can be A16,A24 or A32. Bits 8 and 7 in the MAP-RAM control the address size as follows: 0,1 gives A32, 1,1 gives A24 and 1,0 gives A16. Again, for those interested in the full story, bits 8 and 7 are connected to the VIC's ASIZ1 and ASIZ0 respectively.
- The particular swap-function performed by the Byte Multiplexor for this page. Note that this field applies to all transputer accesses (except to the private SRAM and primary control registers), not just to VMEbus cycles. The three byte multiplexor control bits feed the set number selects to the multiplexor, provided that the byte multiplexor enable bit in the primary control registers has been set.

5.3.2 Configuring the byte–multiplexor

The behaviour of the byte-multiplexor is controlled by a bit in the primary control register and by a set of multiplexor control registers.

The register is at address #7FDC0044. Writing “1” enables the byte multiplexor. This bit is cleared on board reset and should be set after the multiplexor control registers have been configured.

For the purpose of configuring the byte multiplexor, the local and board databuses are subdivided into four-bit groups and numbered as shown in table 5.3.

Bit	Function
4	Byte Multiplexor control bit 0
5	Byte Multiplexor control bit 1
6	Byte Multiplexor control bit 2
7	VMEbus Address Size control bit 0
8	VMEbus Address Size control bit 1
9	VMEbus Data Size control bit 0
10	VMEbus Data Size control bit 1
11	VMEbus Address bit 0
12	VMEbus Address bit 1
13	VIC Function Control bit 1
14	VIC Function Control bit 2
15	VMEbus Master Access Enable bit

Table 5.2 MAP RAM Control Bits (Note bits 4—15)

Bus and Bit Group	Multiplexor Number
Local bits 0–3	0
Local bits 4–7	1
Local bits 8–11	2
Local bits 12–15	3
Local bits 16–19	4
Local bits 20–23	5
Local bits 24–27	6
Local bits 28–31	7
Board bits 0–3	8
Board bits 4–7	9
Board bits 8–11	10
Board bits 12–15	11
Board bits 16–19	12
Board bits 20–23	13
Board bits 24–27	14
Board bits 28–31	15

Table 5.3 Bus Numbering for Byte Multiplexor

The *Multiplexor Control Registers* (see table 5.4) contain a field for each of the four-bit groups as defined in table 5.3. The bit patterns stored in the Multiplexor Control registers represent the *source* bit groups for each of the *destination* bit groups.

There are eight sets of multiplexor control registers (one for each multiplexing set-up as selected by the MAP-RAM) and the "x" in the addresses in table 5.4 should be replaced by "8" plus the set number (0–7).

Address	Bits 28–31	Bits 24–27	Bits 20–23	Bits 16–19
#7FDC00x0	12	8	4	0
#7FDC00x4	13	9	5	1
#7FDC00x8	14	10	6	2
#7FDC00xC	15	11	7	3

Table 5.4 Multiplexor Control Registers

Note that the configuration registers occupy bits 16–31 of the local databus and that they are never subject to any multiplexing themselves. The registers are also readable although the data returned on bits 0–15 is undefined.

5.3.3 Bus Error

The card takes the bus error signal and converts it to a local interrupt. Circuitry latches a bus error and asserts VIC local interrupt number 1. Thus, with appropriate programming, the IMS T801 and the application program may be alerted to a bus error. The IMS T801 cycle which resulted in the bus error'ed VMEbus transaction completes. Data read will be meaningless.

After the application program has picked up the interrupt relating to a bus error, the latch circuit should be reset by writing `#100` (bit 8) to address (`#7FD98000`).

5.4 VMEbus SLAVE configuring (Dual–Access DRAM)

Configuring for SLAVE VMEbus accesses to the Dual-Access DRAM involves the decode-RAM, the VIC and the primary register at address `#7FDC0040` which enables slave operation.

5.4.1 Configuring the VMEbus slave decoder

The decoding of which addresses the IMS B016's dual-access DRAM appears at in the VMEbus address spaces is performed by a programmable circuit based on a very fast RAM. The RAM has 65536 locations of one bit each. Decoding is performed by presenting the top 16 VMEbus address bits to the decoder. If a "one" is stored in the RAM location corresponding to that address then the board responds to the access. A bit in the primary control registers enables/disables all decoding and this bit is cleared on board reset. This means that the random contents of the decoder RAM at power-on do not cause spurious decoding.

Programming of the decoder is similar to the programming of the MAP-RAM and the same conditions as to code placement apply. One should proceed as follows:

1 Ensure that the decoder is disabled by writing a zero into the *Enable VME Slave Decode* register.

2 For each of the 65536 64K byte Address pages perform the following operation:

- Write "1" into the *Next Cycle to VME decode RAM* register (address `#7FDC004C`).
- *Immediately* perform a write cycle at the decode address. Write "2" (bit 1) if that page is to be decoded or a zero if not.

3 After configuring the VIC (if necessary), enable VMEbus decoding by writing a one to the *Enable VME Slave Decode* register (address `#7FDC0040`).

Of course this is describing a program running on the IMS T801, it is not possible to program the decoder from the VMEbus. This decoding scheme allows multiple decoding regions in the address space as well as "holes" to avoid clashes with other cards. It also allows a VMEbus decode region which is much smaller than the dual-access DRAM (as small as 64K bytes). This can be useful when it is desired to provide some privacy and protection for most of the DRAM while still allowing intercard communication and data sharing.

5.4.2 VIC Programming for slave access

VIC registers should be programmed as follows for slave accesses:

1 Local Bus Timing Register (address `#7FD900A4`) should contain `#73`.

2 Slave select 1 Control Register 1 (address `#7FD900CC`) should contain `#11`.

3 Slave select 1 Control Register 0 should have bit 7 set to enable slave writeposting. Bit 5, when set restricts access to "supervisor" transfers. Bit 4 must be set. Bit 3 should be clear. Bit 2 should be set to recognise A24 addresses, clear for A32. Bit 1 should be clear and bit 0 should be set if BLT transfers are to be accepted. Note that bit 6 should be preserved as it relates to master operation.

User-defined address modifier codes can be recognised by programming the AM code into the Address Modifier Source Register (address `#7FD900B4`) and programming bits 2 and 3 of Slave select 1 Control Register 0 to 1,1.

5.5 System controller functions

The VIC can perform all the normal VMEbus system controller (or solt 1) functions. These include SYSCLK driving, bus arbitration, IACK daisy-chain driving and bus timeout. A card performing system controller functions (that is one which is installed in slot 1), should have jumper K3 installed.

Functions associated with arbitration and bus timeout are programmable via the VIC. Bit 7 in the "Arbiter and Requestor Configuration Register" (at address `#7FD900B0`) should be set to select *priority* arbitration. If cleared, *round-robin* arbitration will be selected. Note that the remaining bits in this register should be preserved as they relate to other master functions.

Bits 5–7 of VIC "Transfer Timeout" register select the VMEbus timeout delay according to the following table:

Bit 7	Bit 6	Bit 5	timeout (μs)
0	0	0	4
0	0	1	16
0	1	0	32
0	1	1	64
1	0	0	128
1	0	1	256
1	1	0	512
1	1	1	Infinite

5.6 VMEbus interrupter

VMEbus interrupts can be *generated* by writing to VIC "VMEbus Interrupt Request/ Status Register" (at address #7FD90080. Write a bitfield with bit 0 clear and bits 1–7 set if an interrupt should be generated on any of those VMEbus interrupt levels. The VIC will recognise the interrupt handler's status/ID read cycle (vector fetch) and clear the outstanding VMEbus interrupt. The status/ID byte returned by the VIC at this time is programmable in the VIC's "VMEbus Interrupt Vector Registers 1–7" (at addresses #7FD90084–#7FD9009C).

The VIC also provides the capability to interrupt the IMST801 when an interrupt handler performs a status/ID read cycle for a pending VMEbus interrupt which was generated by the card.

5.7 Interprocessor communications registers

There are eight *Interprocessor Communications Resigters* in the VIC. These are accessible from the VMEbus using the VMEbus data lines D07–D00 or from the IMS T801 using VIC register accesses in the address range #7FD90060–#7FD9007C. From the VMEbus, the ICR's respond to address modifiers #2D and #29 and with A05,A04 = 0,0. Address bits 06 and 07 are don't care and bits 08 to 15 are programmed according to the HEX switches on the card (see section 3.1)

Registers 0–4 are general purpose dual port registers. Register 5 is a dual port read-only ID register to identify the VIC and its revision level. Register 6 is a module status register which is read only from the VMEbus and contains bits as shown in table 5.5. Register 7 provides semaphores for registers 0–4 and several system control functions as shown in table 5.6.

Bit position	Function
7	This bit is read only from the VMEbus and IMS T801. On a VME-bus read a '1' indicates that the board is in reset. On an IMS T801 read, the value of VMEbus signal ACFAIL is returned.
6	This bit is read only from the VMEbus and can be read or written by the IMS T801. The VIC will set this bit upon assertion of card reset. If bit 7 of ICR7 is zero, the VIC will assert SYSRESET whenever this bit is set.
5--0	These bits are read only from the VMEbus and can be read or written from the IMS T801. The VIC will set these bits on certain error conditions which are not usually relevant to the IMS B016.

Table 5.5 Interprocessor Communications Register 6 Functions

Bit position	Function
7	This bit allows masking of VMEbus signal SYSFAIL. A '1' in this bit location inhibits the VIC from asserting SYSFAIL in response to a zero in bit 6 of ICR6.
6	This bit allows reset of the IMS B016 card from other VMEbus cards. Writing a '1' to this bit will assert card reset until this bit is cleared or until a VMEbus SYSRESET or the front panel reset button is pressed.
5	This bit indicates the status of the VIC as VMEbus master. This bit will have the value '1' if read while the VIC is VMEbus master.
4--0	These bits provide semaphores to interprocessor communications registers 4–0 respectively. Each bit is set when the corresponding ICR is written. These bits can be read or written from either the VMEbus or the IMS T801.

Table 5.6 Interprocessor Communications Register 7 Functions

The Interprocessor Communications Registers can be configured to cause local interrupts (events) to the IMS T801. This is achieved by programming the "ICMS Interrupt Control Register", the "ICGS Interrupt Control Register", and the associated "ICGS Interrupt Vector Base Register" and "ICMS Interrupt Vector Base Register". A full description is outside the scope of this manual and users are directed to read the VIC Specification, [7].

5.7.1 Interrupts

Many different things can cause an interrupt condition. All these are signaled through the VIC and cause the IMS T801 *event* pin to be asserted. Since only one process may wait on an event at a time, a suitable process needs to pickup the VIC interrupt vector to determine what caused the event. The vector is read from a 'pesudo register" at address `#7FDA0004`. The complete list of possible interrupt sources is as follows:

1 VMEbus interrupts being handled by the card. The vector is the status/ID byte supplied by the VMEbus interrupter card.

2 Bus Error. On the IMS B016, bus error asserts local interrupt number 1 to the VIC. The vector is sourced from the VIC and should be programmed in the local interrupt vector base register (address `#7FD90054`).

3 RTC interrupt signal which is connected to VIC local interrupt number 2.

4 PEX interrupt signal which is connected to VIC local interrupt number 3.

5 DUART interrupt signal which is connected to VIC local interrupt number 4.

6 Interrupts relating to Interprocessor Communications Switches, where the vector is sourced from the VIC.

7 Various error conditions such as bus error on a writeposted cycle, where the interrupt vector is sourced from the VIC.

A short explanation of the circuitry used to generate events on the IMS B016 will help users understand the requirements of service routines.

All interrupts, both local from the on-board peripherals and BERR logic, and VME-bus interrupts, plus error interrupts generated by the VIC chip itself, are processed within the VIC. The VIC chip produces one unified interrupt output which is used to event the IMS T801 as follows:

The IMS T801 has two event pins (EventRequest and EventAcknowledge). When EventRequest is asserted, the processor schedules the process which is waiting on event. When this process is started, the processor asserts EventAcknowledge. The external circuitry can then disassert EventRequest. A subsequent re-assertion of EventRequest will now result in the processor taking another event.

The IMS B016 has a state-machine circuit which generates EventRequest. This circuit is present to convert the VIC interrupt signal, which is level sensitive, to an edge sensitive signal for EventRequest. After an idle period, or reset, an interrupt will cause EventRequest to be asserted. Once EventRequest is asserted, the circuit then looks for EventAcknowlege and when it becomes asserted it dis-asserts EventRequest. Next the circuit waits for EventAcknowlege to be dis-asserted. Only then will it wait for the clear-event signal (caused by writing the value `#100` the relevant control register at address `#7FD90004`). Finally, after the clear-event signal, the circuit reverts to the inactive state and the next interrupt may be processed.

This four-state interlocked scheme makes sure that each interrupt can only cause one T801 event, provided that the user correctly removes the interrupt source before writing the "Clear event" register. The setup requirements for local interrupts are as follows:

1. Program the VIC interrupt control registers.
2. Program all VIC and VMEbus vector registers such that every distinct interrupt source has a distinct vector.
3. Program the peripheral chips and possibly a PEX board to generate their interrupts correctly.
4. Write to the *Clear Event* register to clear any spurious event.
5. Enable all interrupts by writing to the VIC.

Now, when an interrupt is generated and therefore an event, the event process should perform the following tasks:

1. Determine the source of the interrupt by reading the register at address `#7FDA0004` and disable or remove that interrupt by writing to the relevant control registers.
2. Write `#100` to the *clear event* register to clear the event circuitry.
3. Perform any other actions necessary for handling the interrupt.
4. Wait on the event again.

Note that MAP-RAM control bits 11 and 12 must be programmed to 1,1 for the address region corresponding to the VIC registers for interrupts to function.

Users may find it useful to examine the source code for IMS F008A event library routines as well as the DUART library code.

5.7.2 DRAM refresh

The VIC "DRAM Refresh" function *must* not be enabled. It is turned off at power-on and reset and if enabled will cause the board to stop working.

5.8 Serial ports and DUART

The IMS B016's serial ports are provided via a Philips SCN2681 DUART device. This part provides two independent asynchronous serial channels and on-chip baud-rate generation. It is addressed between `#7FD80000` and `#7FD8003C` and appears in the lower byte of the databus. IMS F008A contains predefines for all the DUART registers in the file `$INCLUDE/include/duart.h`.

DUART serial transmit and receive signals, along with uncommitted pins OP0,1 and IP0,1 are buffered to EIA-RS232 levels (no filtering) and routed to pins on P2. The buffering uses VMEbus ±12V supplies, which must be present for the serial port signals to function.

The DUART interrupt signal is connected to VIC local interrupt number 4. It should be programmed to be level sensitive, active low.

Spare input and output bits available on the DUART chip are used to create three extra "subsystem" ports. These signals are available on connector P2 (see figures 3.1 and A.1). Table 5.7 shows which DUART control bits correspond to which subsystem port. Note that these three extra subsystem ports control bits in the DUART use the logic level on the actual subsystem signals directly rather than performing an inversion as in the "traditional" subsystem port. For instance, to assert notAReset, one would write a *zero* into the relevant control bit in the DUART.

DUART control Bit	Function
`IP4`	notAError
`IP5`	notBError
`IP6`	notCError
`OP2`	notAReset
`OP3`	notAAnalyse
`OP4`	notBReset
`OP5`	notBAnalyse
`OP6`	notCReset
`OP7`	notCAnalyse

Table 5.7 Three Extra Subsystem Ports

For DUART programming without the functions provided in IMS F008, the user is directed to consult IMS F008A duart library functions source code and to read the SCN2681 Datasheet (Philips/Signetics).

5.9 Real–time clock

The IMS B016's real–time clock function is provided via a National Semiconductor DP8572 RTC device.

The RTC contains a total of 61 8-bit registers. These are addressable as byte 1 (bits 8–15) of the 32-bit words at addresses `#7FD88000`–`#7FD88080`. An indirection scheme using bits in control registers allows more registers to be addressed than are available in the memory map. A total of 33 bytes of non-volatile RAM are available in the RTC.

RTC interrupt output is connected to VIC local interrupt number 2 and should be programmed active low, level sensitive.

The RTC clock frequency is adjusted by means of the trimmer located adjacent to the DP8572 chip. Local temperature and supply variations mean that it may be necessary to re–calibrate the clock. This is done by first using a frequency counter connected to pin 11 of the DP8572 and adjusting for 32.768kHz. Next, for exact adjustment, run the clock for a period and note whether it runs fast or slow Make fine adjustments to the trimmer to achieve accurate timekeeping. Note that the clock may run at a slightly different frequency depending upon whether it is supplied from the main or standby power supply. Most accurate timekeeping will be achieved by performing the calibration when the clock is powered from the supply most commonly used. That is to say, if the equipment is usually switched off at night then it would be best to calibrate using the standby power supply.

Address	Register
#7FD88000	Main status
#7FD88003	Real–time mode
#7FD88007	Output mode
#7FD8800C	Interrupt control 0/Periodic flag
#7FD88010	Interrupt control 1/Time save control
#7FD88013	100th second counter
#7FD88017	Seconds counter
#7FD8801C	Minutes counter
#7FD88020	Hours counter
#7FD88023	Day of the month counter
#7FD88027	Months counter
#7FD8802C	Years counter
#7FD88030	Units julian counter
#7FD88033	100's julian counter
#7FD88037	Day of week counter
#7FD8804C	Seconds compare
#7FD88050	Minutes compare
#7FD88053	Hours compare
#7FD88057	Day of month compare
#7FD8805C	Months compare
#7FD88060	Day of week compare
#7FD88063	Seconds time save
#7FD88067	Minutes time save
#7FD8806C	Hours time save
#7FD88070	Day of month time save

continued

Address	Register
#7FD88073	Months time save
#7FD88077	RAM
#7FD8807C	RAM/TEST

Table 5.8 Real–Time Clock Registers

Note that correct RTC operation is only possible with a supply connected to the VMEbus standby power rail on the backplane. This supply must be available at all times, including when the main +5v supply is present.

For RTC programming without the functions provided in IMS F008, the user is directed to consult IMS F008A RTC library functions source code and to read the DP8572 Datasheet (National Semiconductor).

5.9.1 F–ROM

Four 32K x 8-bit Flash Memory devices, arranged as a 32-bit word, are addressed between `#7FE00000` and `#7FFFFFFF` (repeated eight times within that region). This addressing is compatible with the IMS T801's BootFromROM behaviour, which starts fetching code from `#7FFFFFFC` on reset. Jumper K1 is connected to the CPU BootFromROM pin such that when it is removed the IMS T801 will attempt to boot from ROM.

The devices are normally Intel 28F512 parts, although it is possible that larger memories (up to 2Mbit) and devices from other manufacturers could be fitted. The flash memories are soldered to the card and in-system programming is achieved with the on-card VPP generator and a program running on the IMS T801 (which for this purpose would be booted from one of its links).

Users wishing to program the flash memories without using the facilities offered in IMS F008A should consult the device datasheet (28F512).

The programming voltage to the Flash memories is turned off at board reset. To enable the VPP, write `#100` (bit 8) into the "Enable Vpp" register (address `#7FD98008`). Similarly, VPP can be turned off again by writing zero into this register. Note that this register is readable but that only bit 8 is valid on reading. Note also that bit 8 reads the inverse of the value written.

When the VPP generator is enabled, read the "Read Vpp voltage sensor" register (address `#7FD9800C`). If bit 8 is set then the VPP generator is functioning correctly. Otherwise something is wrong with the VPP rail and Flash device programming should not be attempted. Note that this status bit will not read a valid status unless the VPP generator has been enabled first.

At least 100ms should be allowed for VPP to stabilise after being enabled under software control. Jumper K4 allows VPP to be permanently disconnected for extra

Flash memory contents security. However, damage to the devices or loss of contents is extremely unlikely even with VPP permanently enabled. This is because the correct programming enable sequences must be sent to the memories before they will carry out programming operations. This means that it is entirely feasible to use the devices for non-volatile data storage. Note that the programming voltage generator on the IMS B016 only provides sufficient power to program one FROM at a time. Programming software must be written such that each FROM is erased and programmed separately.

IMS F008A contains routines and tools which support programming and erasing from a host development system.

5.10 PEX boards

Connector P3 allows one PEX card to be fitted to the IMS B016. The PEX interface is addressed between `#7FDB8000` and `#7FDB803C` and provides a set of byte registers on the lower byte of the 32-bit word. The PEX interrupt signal is connected to VIC local interrupt number 3. The PEX interface is very similar to a 68000 bus. Full details of the interface are available in the "PEX Bus Specification" from Radstone Technology Ltd.

5.11 Mechanical and thermal details

The IMS B016 is designed to accord with DIN 41494 and IEC 297 standards. The board is nominally 160mm by 233.35mm. Nominal board thickness is 1.6mm. The supplied front panel width is 4HP (aprox 20mm). This is compatible with a board-to-board pitch in a card cage of 0.8". M2.5 fastening bolts are provided on the front panel, these mate with tapped holes in the card cage and fix the board securely. Front panel handles allow the board to be removed from the card cage (by unscrewing the retaining bolts and pulling hard on the handles). Note that the front panel is *required* when operating the IMS B016 in a card cage, both for mechanical rigidity and to give correct cooling air flow.

No components protrude more than 2.47mm below the surface of the board. To fit in a 0.8" pitch card-cage, no component should protrude more than 13.7mm above the surface of the board.

Adequate cooling air flow must be provided to maintain the components on the board within their operating temperature. Air flow should run parallel to the board surface and parallel to the front panel. The IMS B016 dissipates 25W maximum. This means that a J1/J2 backplane[4] must be used. The cooling air flow required for a particular application will probably need to be determined empirically.

4.
J1 is the minimum VMEbus backplane and mates with P1 connectors on VMEbus boards. J2 mates with P2 connectors and is sometimes called a 32-bit backplane because it is needed for 32-bit VMEbus operations. Combined J1/J2 backplanes mate with both P1 and P2 and are needed for reliable operation of fast 32-bit VMEbus transfers.

A single board operating in static air at room temperature (and not in a card-cage) will usually not need forced air cooling. This kind of set-up should only be used for lab and development work. High reliability is not to be expected from boards which are not provided with adequate cooling.

The two DIN 41612 (603-2-IEC-C096Mx-xxx) connectors (P1 and P2) have class 2 contact finish (1 micron gold) and are specified to give 400 mating cycles minimum.

5.11.1 Mating Connectors

Connectors to mate with the connectors on the IMS B016 are as follows:

P1/P2: 96-way DIN41612 female connectors which are available from most connector manufacturers. These must be gold plated to at least commercial class II in order to ensure contact reliability.

P3: 0.025" square post pin header. Pins 8mm long, gold plated. Connector on the IMS B016 is Berg (DuPont) type 76342-320.

Jumpers: 0.1" pitch jumpers or programming shunts, again available from many manufacturers. Some types are very stiff-fitting and suit fixed configurations, others are slacker and are more useful when configurations are changed often.

Note that INMOS does not guarantee that these connector descriptions and part numbers remain correct.

5.11.2 Environmental details

	Operating	**Storage**
Temperature	0 to 50°C ambient air	–55 to 85°C
Relative humidity	95% non condensing	95% non condensing
Thermal shock	< 0.08°C/s	< 0.15°C/s
Altitude	–300 to +3000m	–300 to +16000m

Table 5.9 Environmental details

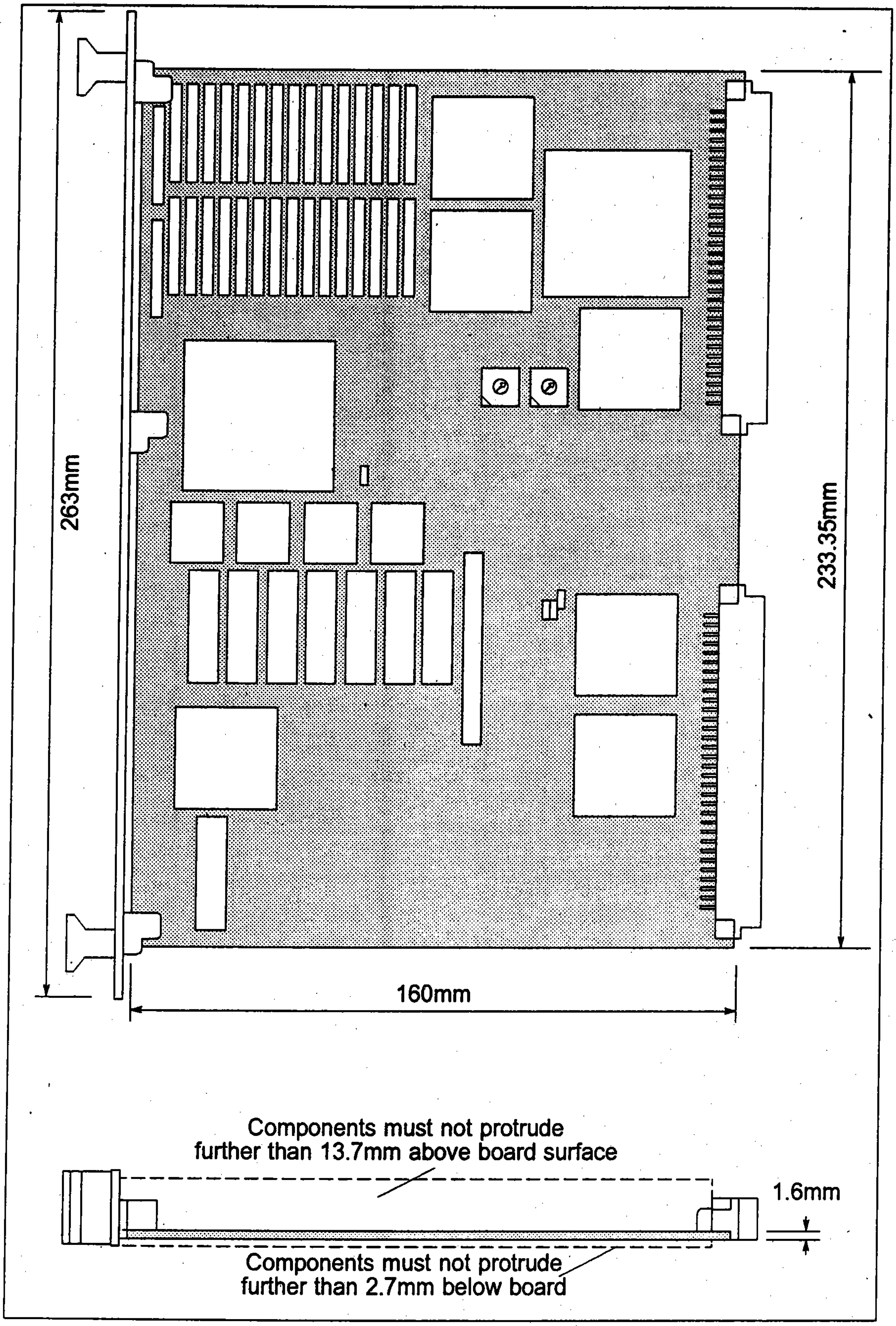

263mm
233.35mm
160mm
Components must not protrude further than 13.7mm above board surface
1.6mm
Components must not protrude further than 2.7mm below board

Figure 5.1 Board dimensions

6 References

1 *Microcomputer Interfacing*, Harold S. Stone, Addison Wesley 1982.

2 *The Transputer Databook*, second edition, 1989.

3 *The Transputer Development and iq Systems Databook*, second edition, INMOS Ltd 1991. (pp 191–208.)

4 *The Transputer Applications Notebook – Systems and performance*, INMOS Ltd 1989,

5 *MECL System Design Handbook*, William R. Blood Jr, Motorola Inc., 1983

6 *VMEbus Specification Manual Revision C.1*, VITA, Printex Publishing Inc., 1985

7 *Cypress, VMEbus Interface Controller Specification*, Cypress, VTC Inc., 1989.

Appendices

A Memory map of IMS B016

Bit	Function
4	Byte Multiplexor control bit 0
5	Byte Multiplexor control bit 1
6	Byte Multiplexor control bit 2
7	VMEbus Address Size control bit 0
8	VMEbus Address Size control bit 1
9	VMEbus Data Size control bit 0
10	VMEbus Data Size control bit 1
11	VMEbus address bit 0
12	VMEbus address bit 1
13	VIC Function Control Bit 1
14	VIC Function Control Bit 2
15	VMEbus Master Access Enable bit

Table A.1 MAP RAM Control Bits

Address	Function
#80400000–#FFFFFFFF	VMEbus address space
#80040000–#803FFFFF	Dual–Access DRAM (4–cycles)
#80001000–#8003FFFF	Private SRAM (2–cycles)
#80000000–#80000FFF	On–Chip RAM (1–cycle)
#7FE00000–#7FFFFFFF	ROM Address Space
#7FDC0000–#7FDFFFFF	Control Registers
#7FD80000–#7FDBFFFF	Peripheral Address Space
#00040000–#7FCFFFFC	VMEbus Address Space
#00000000–#0003FFFC	(optional) Control Registers

Table A.2 T801 Memory Map (4MByte DRAM)

Address	Function
#81000000–#FFFFFFFF	VMEbus address space
#80040000–#80FFFFFF	Dual–Access DRAM (4–cycles)
#80001000–#8003FFFF	Private SRAM (2–cycles)
#80000000–#80000FFF	On–Chip RAM (1–cycle)
#7FE00000–#7FFFFFFF	ROM Address Space
#7FDC0000–#7FDFFFFF	Control Registers
#7FD80000–#7FDBFFFF	Peripheral Address Space
#00040000–#7FCFFFFC	VMEbus Address Space
#00000000–#0003FFFC	(optional) Control Registers

Table A.3 T801 Memory Map (16MByte DRAM)

Address	Register
#7FDB8000	PEX–Daughterboard
#7FD98000	Secondary Control Registers
#7FD90000	VIC Programming Registers
#7FD88000	Real–Time Clock
#7FD80000	DUART

Table A.4 T801 Peripheral Address Map

Address	Function
#7FD90000	VMEbus Interrupter Interrupt Control
#7FD90004–#7FD9001C	VMEbus Interrupter Control 1–7
#7FD90020	DMA Status Interrupt Control
#7FD90024–#7FD9002C	Local Interrupt Control 1–7
#7FD90040	ICGS Interrupt Control
#7FD90044	ICMS Interrupt Control
#7FD90048	Error Group Interrupt Control
#7FD9004C	ICGS Interrupt Vector
#7FD90050	ICMS Interrupt Vector
#7FD90054	Local Interrupt Vector
#7FD90058	Error Group Vector
#7FD9005C	Interprocessor Comms. switch
#7FD90060–#7FD9007C	Interprocessor Comms. 0–7
#7FD90080	VMEbus interrupt request and status
#7FD90084–#7FD9009C	VMEbus interrupt vectors 1–7
#7FD900A0	Transfer timeout register
#7FD900A4	Local bus timing
#7FD900A8	Block transfer definition
#7FD900AC	VMEbus interface configuration 1
#7FD900B0	Arbiter and requester configuration
#7FD900B4	Address modifier source
#7FD900B8	Bus error status
#7FD900BC	DMA status (not used)
#7FD900C0	Slave select 0 control 0 (not used)
#7FD900C4	Slave select 0 control 1 (not used)
#7FD900C8	Slave select 1 control 0
#7FD900CC	Slave select 1 control 1
#7FD900D0	Release control
#7FD900D4	Block transfer control
#7FD900D8	Block transfer length 0
#7FD900DC	Block transfer length 1
#7FD900E0	System Reset

Table A.5 VIC Register Memory Map (bits 0–7)

Address	Register
#xx00	Subsystem Reset/Error
#xx04	Subsystem Analyse
#xx40	Enable VMEbus Slave Accesses
#xx44	Enable Byte Multiplexor
#xx48	Next Cycle is to the MAP RAM
#xx4C	Next Cycle is to the Slave Decode RAM
#xx50	Do not MAP these registers to address zero
#xx54	Enable VMEbus Master Accesses
#xx80–#xxFF	Byte Multiplexor Control Registers

Table A.6 T801 Primary Control Registers (bit 0)

Address	Register
#7FD98000	Clear BusError Interrupt
#7FD98004	Clear Event
#7FD98008	Enable Vpp to F–ROMs
#7FD9800C	Read Vpp voltage sensor
#7FDA0004	Read VIC Interrupt Vector

Table A.7 T801 Auxiliary Registers (bit 8)

B Connector diagrams and cables

B.1 Connector pin assignments

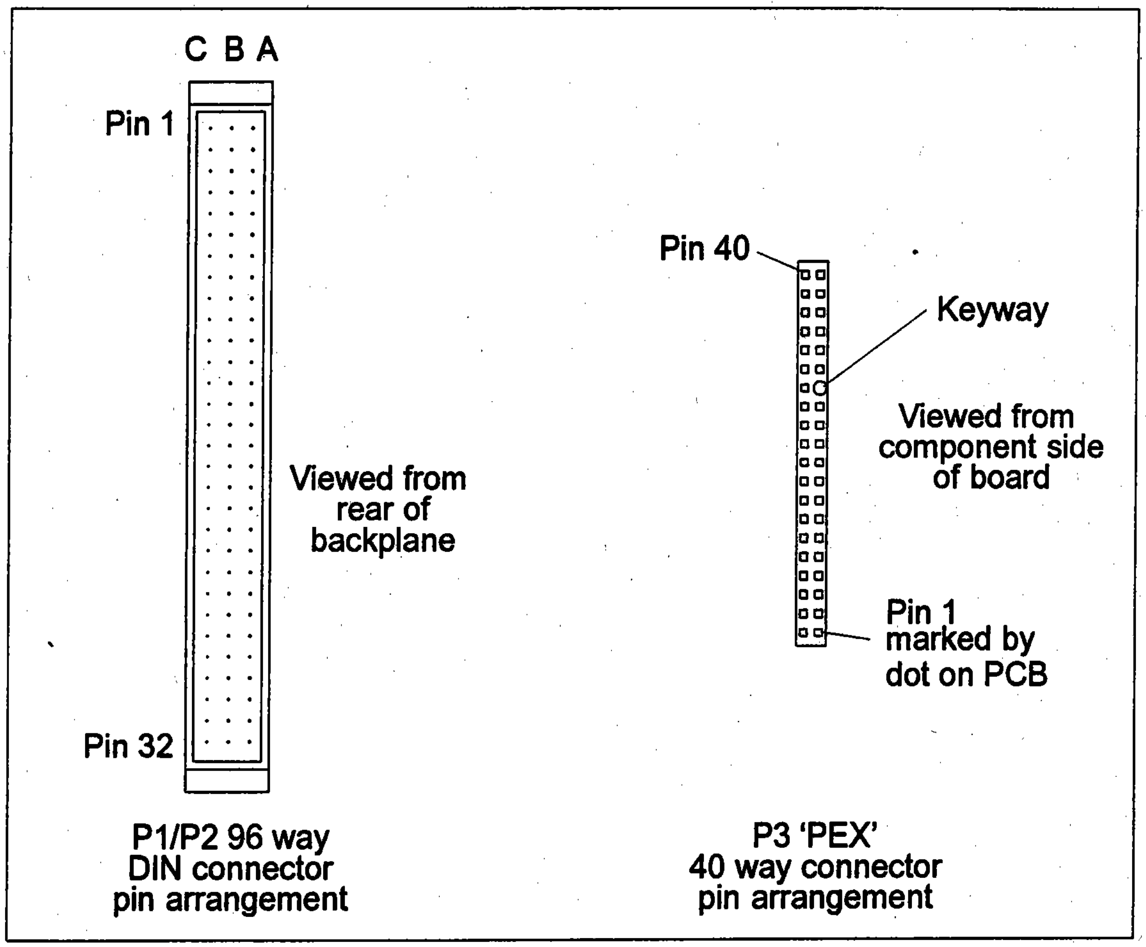

C B A
Pin 1
Pin 32
Viewed from rear of backplane
P1/P2 96 way DIN connector pin arrangement
Pin 40
Keyway
Viewed from component side of board
Pin 1 marked by dot on PCB
P3 'PEX' 40 way connector pin arrangement

Figure B.1 Connector arrangements

The IMS B016 has three connectors— P1, P2 and P3. P1 and P2 are the VMEbus connectors. Although only P1 is needed for a 16-bit VMEbus system, it is recommended that the power pins on P2 are connected. The outer two rows of pins on P2 are used to carry links and RS-232 signals off the card. Connector P3 is the 'PEX' connector and carries address, data and control signals to PEX daughter-boards. Pin assignments for all the connectors are shown in section the following tables.

Pin	Row C	Row B	Row A
1	D08	BBSY*	D00
2	D09	BCLR*	D01
3	D10	ACFAIL*	D02
4	D11	BG0IN*	D03
5	D12	BG0OUT*	D04
6	D13	BG1IN*	D05
7	D14	BG1OUT*	D06
8	D15	BG2IN*	D07
9	GND	BG2OUT*	GND
10	SYSFAIL*	BG3IN*	SYSCLK
11	BERR*	BG3OUT*	GND
12	SYSRESET*	BR0*	DS1*
13	LWORD*	BR1*	DS0*
14	AM5	BR2*	WRITE*
15	A23	BR3*	GND
16	A22	AM0	DTACK*
17	A21	AM1	GND
18	A20	AM2	AS*
19	A19	AM3	GND
20	A18	GND	IACK*
21	A17	SERCLK	IACKIN*
22	A16	SERDAT	IACKOUT*
23	A15	GND	AM4
24	A14	IRQ7*	A07
25	A13	IRQ6*	A06
26	A12	IRQ5*	A05
27	A11	IRQ4*	A04
28	A10	IRQ3*	A03
29	A09	IRQ2*	A02
30	A08	IRQ1*	A01
31	+12V	+5VSTDBY	–12V
32	+5V	+5V	+5V

Table B.1 Connector P1 pin assignments

Pin	Row C	Row B	Row A
1	TxA	VCC	TxB
2	RxA	GND	RxB
3	OpA	RESERVED (nc)	OpB
4	IpA	A24	IpB
5	GND	A25	GND
6	nc	A26	nc
7	P2Link0Out	A27	P2Link1Out
8	P2Link0In	A28	P2Link1In
9	GND	A29	GND
10	GND	A30	GND
11	nc	A31	nc
12	P2Link2Out	GND	P2Link3Out
13	P2Link2In	VCC	P2Link3In
14	GND	D16	GND
15	nc	D17	nc
16	notAReset	D18	notBReset
17	notAAnalyse	D19	notBAnalyse
18	notAError	D20	notBError
19	GND	D21	GND
20	GND	D22	GND
21	nc	D23	nc
22	notCReset	GND	notSubReset
23	notCAnalyse	D24	notSubAnalyse
24	notCError	D25	notSubError
25	GND	D26	GND
26	GND	D27	GND
27	nc	D28	nc
28	notUpReset	D29	notDownReset
29	notUpAnalyse	D30	notDownAnalyse
30	notUpError	D31	notDownError
31	GND	GND	GND
32	GND	VCC	GND

Table B.2 Connector P2 pin assignments

Pin	Signal	Pin	Signal
1	+12v	21	VCC
2	Vcc	22	VALWAYS
3	A7	23	PEXSEL*
4	A6	24	DSACK0*
5	A5	25	PROCLK*
6	A4	26	BERR*
7	A3	27	Reserved
8	A2	28	PEXINT*
9	A1	29	PEXRES*
10	A0	30	PEXIN*
11	GND	31	PEXOUT*
12	GND	32	GND
13	D7	33	GND
14	D6	34	SYSCLK
15	D5	35	R/W*
16	D4	36	AS*
17	D3	37	DS*
18	D2	38	PEXPRES*
19	D1	39	VCC
20	D0	40	–12v

Table B.3 Connector P3 pin assignments

B.2 Cables

INMOS has developed a standard cable set for evaluation boards. Link cables have 5-pin single-in-line connectors at each end with a key pin. Services cables have 5-pin single-in-line connectors at each end with two key pins. The keyed pins prevent the connectors being inserted incorrectly. The cables are available in a range of packs of various lengths. Using this system it is quick and easy to connect links and services signals from board to board and system to system. These cables must not be used between equipment on different mains circuits and ideally, to ensure noise-free operation, should only be used within one system operating from the same 5V power supply.

IMS B016 users may find it convenient to remove the key pins from cable connectors mating with a VMEbus J2 backplane, (P2 pins on the IMS B016).

C Electrical specifications

C.1 Electrical Details

C.1.1 Power supply

The IMS B016 requires power supply voltages in accordance with the VMEbus specification. That is, the +5V DC supply must be between 4.875V and 5.25V and have less than 50mV pk-pk noise and ripple between DC and 10MHz. The IMS B016 does not incorporate protection against incorrect power supplies. Major damage can result from operating the board outside its power supply range.

C.1.2 Board–to–board link connections

The INMOS serial links provided on the IMS B016's connectors may be used to communicate directly with other boards in the same electrical environment (same DC supply, ground reference and low electromagnetic noise). However, these link connections *must* be made with transmission lines of characteristic impedance 100Ω.

The following are examples of valid connection methods:

1 A printed circuit trace of about 0.008" width over a ground plane on a multi-layer board.

2 Twisted pair ribbon cable.

3 INMOS link cables.

4 Twisted pair wire-wrap wire.

Single wires, PCB traces without ground planes and coaxial cable should *not* be used. It is recommended that link connections are not longer than 1m. Although long connections will function, enhanced reliability will be gained by making longer connections using buffered connections.

Services connections are not subject to the same conditions and can be carried on simple wires.

C.1.3 Non–local link connections

Links may be used to communicate between TRAMs and transputers (and other boards containing transputers, even boards not manufactured by INMOS) wherever their location. However, when using links to communicate between TRAMs or transputers which are not powered from the same DC supply some special considerations need to be addressed:

1 Electrical noise may corrupt link data when ordinary INMOS link cables are used over long distances and between equipment which is powered from different supplies.

2 Since the INMOS cables and boards use single-ended, common ground signaling, earth loops between equipment can lead to signal corruption (not necessarily on the link signals but perhaps on the services signals causing spurious system resets).

3 The common ground in the cables can contravene electrical safety regulations which prevent earth signals being connected between equipment more than a certain distance apart. These regulations prevent, among other things, fire damage during electrical storms.

INMOS suggests that when such inter-box connections are required that differential, double-ended balanced connections are used. These may be constructed from ordinary 26LS31 and 26LS32 driver/receiver devices for 10Mbits/s link operation and from ECL drivers/receivers for 20Mbits/s links. New driver/receiver devices (DS892xA) designed for high-speed disk drives allow non-ECL 20Mbits/s operation. Signals should be receiver terminated to a floating ground and not to the ground at the receiving end. The services signals must be treated in a similar way although the timing constraints are not critical. Background information on this subject is to be found in [NO TAG], [NO TAG] and [NO TAG].

Connections driven by DS892xA (National Semiconductor) devices and terminated in 100Ω are compatible with the IMS B415 Differential-Link TRAM, the IMS B019 rack–mounted differential link card and the IMS B300 TCP/IP gateway.

D Monitor command language

As there is an ever present need to provide a textual user interface capability for programs and testbeds, a generalised module has been created to make this kind of implementation easier and, as a by–product, encourage a degree of uniformity between interfaces created for different uses. The module implements a configurable Command Line Interface; in other words a prompted, command driven form of interaction, with a language like format that enables the interface to be extended at run time by the scripting of complex interpreted macro commands in a natural syntax.

The basic capabilities of the language and the CLI commands are described below.

D.1 Language basics

The CLI is an interpreted environment – that is the lines of the language are analysed and executed individually as presented. This is somewhat slower than the used of a compiled language at execution time but offers many advantages for an interactive environment. The general characteristics of the language are defined by the interpreter that accepts the lines of input and some of these characteristics are important in understanding the syntax that the language features possess and the errors that may be encountered when using the tool.

D.1.1 Components

Executable language consists of a series of commands, assignments or control flow statements. Commands can be built in — i.e. the code that determines their action has been compiled and linked with the interpreter or they can be macros, which are collections of other language statements. No terminator is required for a language statement — they are all assumed to be contained on one line and the line–feed or carriage–return character terminates the statement.

D.1.2 Delimiters

White space (spaces, tabs and CR/LF) is significant and generally used to delimit components of the command statements, but the amount of space is not significant. In particular, any number of spaces at the beginning of the language statement will not affect its meaning or interpretation. Other command delimiters that can be used interchangeably are commas and backslash characters. The number of these is significant and can be used, for instance, to introduce place–holders into command statements.

D.2 Abbreviations

A name abbreviation scheme is used which applies to commands, control flow statements, symbols and macro names, as follows:

- Case is not significant
- Symbols are internally stored and printed in upper case for consistency.
- Any number of characters that uniquely identifies the correct object can be used.
- In the case of ambiguity, the symbol name which most closely matches the given string is used; i.e. the shortest of the declared names.
- Where multiple variable types can be accepted, the possible variable names are searched in the order integer, floating point, string. Thus abbreviations which could match variables of more than one type will match the first appropriate definition found.

For example, given the symbols "fred" and "fredsMum", the following abbreviations are valid:

Name typed	Symbol matched
fr	fred
fred	fred
freds	fredsMum
fredsmum	fredsMum

This abbreviation strategy has been designed to give flexibility, especially in interactive use, to reference fully descriptive names by some sub–string. Almost all situations of abbreviated reference will be resolved un–ambiguously. This includes usage of symbol names declared as substrings of other symbol names such as above. However, if a symbol "fredsDad" is introduced, then "freds" will match either "fredsDad" or "fredsMum" depending on the order of declaration (since they are the same length). In general, specify the full name of a symbol to be sure of matching the correct one.

D.2.1 Defaulting

To ease the use of the language in an interactive mode, a general policy of defaulting has been adopted. Typical command statements will have a set of position sensitive parameters with the most frequently used parameters appearing first. The user can then specify only those parameter values which are explicitly required and omit others whereupon they will default to 'sensible' values. In particular commands will often default to the most general mode of usage or adopt a default action that has been explicitly set by a previous statement.

D.2.2 Error reporting

Syntax errors from all command are reported as they are interpreted and, on finding an error, the interpretation and execution will be aborted. Errors are reported as meaningfully as possible with an explicit identification of the position of the error on the supplied line. All macro and control flow execution will be terminated by any such error, even at a nested level.

D.3 Variables and expressions

The implementation of extensive programming features has been sidestepped in the command language, but there still exists a powerful underlying mechanism for mathematical manipulation and expression handling.

D.3.1 Numbers

An integer type of basic variable is provided. The number range is limited by the machine's natural 32 bit representation of integers, but this allows considerable scope for operations on large numbers. For more precision in calculation, a floating point type is also provided. Constants and variables of both types can be used freely. Constant integers can be used at any point where a value is required in the language. The constant can be expressed in hex, decimal octal or binary notation. A floating point value is signified by including a decimal point in the notation and can be given with no decimal places, or any number of digits after the decimal point.

```
40 - integer
#4f - hex integer
o67 - octal integer
$010 - binary integer
+2 - decimal integer
4.0 - floating point
0.2 - floating point
3.2e5 - floating point
```

D.3.2 Numeric expressions

In place of any explicit number, a mathematical expression of arbitrary complexity can be used.The expression must not contain embedded spaces, since these might be interpreted as value terminators, but may include brackets to make precedence clear. Precedence will be exercised from right to left in normal evaluation.

The available mathematical operators are:

- + integer and floating point addition
- − integer and floating point subtraction
- * integer and floating point multiplication
- / integer and floating point division
- % integer remainder
- & bitwise AND
- | bitwise OR
- ^ bitwise XOR
- ~ bitwise NOT (this is a unary operator)
- ! logical NOT (this is a unary operator)

Examples of the use of these operators in the context of expressions are given below:

```
23+44.0
1002*3
24/(34+2)
(23-22)*(400.0+0.1)
```

Note that there is no facility to include function calls inside an expression – this is precluded by both syntax and convention since the language is purely procedural.

D.3.3 Numeric symbols

In place of any constant number or expression a symbol can be used. Symbols can take constant or variable numeric values and can be used freely within the language. Constant symbols are established by the programmer and pre–declared within the interpreter – this is used for command constants such as TRUE, FALSE and PI. Symbol variables can be pre–declared or freely created and deleted by the interactive user. A symbol is declared using the assignment statement as shown here:

```
apples=23
oranges=0.1
```

The symbol is declared and takes on the initial value given using the C style syntax. Thereafter the symbol can be used by referencing either its full or abbreviated name:

```
apples/34.2
oranges+apples-1
(or*app)-22
```

Symbols can be used either as variables are used in programming languages or simply as a meaningful name for a commonly used numeric parameter value. In this way constants, for instance, can be abbreviated or made symbolic (e.g. PI). Care must be taken when using abbreviations to reference the correct symbol i.e. use as full a name as possible. The abbreviations are convenient when using interactive commands but can be misinterpreted in longer macro usage.

The rules for symbol names are typical of many programming languages: no leading numeric characters are allowed, only alphanumeric digits and underscore characters can be used within names and only 32 characters are significant.

D.3.4 Assignments

A variable assignment can be used to hold the results of an expression evaluation. The syntax of the assignment is that of C, and the target of the assignment may or may not already have been declared. Since some symbols may have been created as constants, not all assignments can succeed even in the case where the syntax is correct.

```
apples=oranges+(4/1.2)
bored=TRUE
```

Commands can and do return values of various sorts and thus can be used within an expression statement. Due to limitations of the syntax, it is not possible to mix commands and generalised expressions so the format is limted to statements of the type

```
temp=pr "a" 1 2 3
a123
sho temp
a123
```

The type and value of return values from commands should be documented with the command.

D.3.5 String symbols

Variables can also be created to hold string values. The strings can be used by commands as parameters where appropriate. Identical rules apply about declarations and usage, but the contents of a string are case preserved. All string constants are represented by enclosing text within double quotes.

```
card="the string is CARD"
paper="thinner than CARD"
```

The double quotes are necessary to identify a string constant, since all other features of the commands syntax are the same.

D.3.6 String assignment

Once a string has been declared it may be reassigned freely and concatenated with other strings using the '+' operator.

```
temp="hello"
temp=temp+" world"
```

The quotes symbol may be included within a string by preceding it with the escape '\' character.

D.4 Intrinsic commands

The operation of the language interpreter is simplified by the inclusion of some basic facilities which bind the individual command functions into a cohesive whole. The following pages give some details of these commands and their use.

Intrinsic commands may have special limited properties which restrict their use in general circumstances. It isn't necessarily very useful to be able to print help within a repeat loop for instance. In all other ways they are model examples of the type of behavior found in all commands.

EXIT or END

```
EXIT
END <value>
```

The EXIT statement is used to terminate the entry of macro bodies and to terminate the interpreter when at the interactive level. Both EXIT and END can be used interchangeably. When a value is given in an END statement in a macro definition, this value can be assigned to a variable by the caller of the macro.

HELP

```
HELP <command>
```

The HELP command provides an on–line and rapid aide–memoir facility for the language environment. If a particular command name is given as a parameter (in full or abbreviated form) then a (usually) one line summary of that commands' function and parameters are given. This is not intended as a substitute for full documentation of the command function but a quick and easy way to remember parameter order and type.

SHOW

```
SHOW <symbol>
```

This utility command allows the display of selected symbols in a somewhat controllable format. The type and value of the symbol, if recognised, is displayed. If the symbol is a macro definition, then its contents are shown. If no parameters are given, the current symbol definitions are listed.

SOURCE

```
SOURCE <fileString> <echoflag>
```

This command allows the execution of streams of commands that reside in ASCII files. The streams can be echoed as input using the optional flag which defaults to FALSE. There is no restriction on the commands that can be contained in the files, but they are often used to define macro commands which can later be executed in interactive mode.

It is important to realize that the filename is a string parameter and therefore must be given surrounded in double quotes. If no name is given a default string variable COMMAND_FILE is used as the source for the filename. Any other string variable can of course be used on the command line to specify the file.

LOG

```
LOG <fileString>
```

The LOG command allows definition of a file to which all command statements will be echoed before execution. When this command is used any already existing LOG file is closed and a new file opened according to the name and path given in the file string parameter. By default there is no LOG file opened at start–up.

DEFINE

```
DEFINE <macro> <>|<parameters>
```

The DEFINE command allows the creation of command macro procedures. The macros possess a name and a number of formal parameters which are supplied on the definition line. The content of the macro is a sequence of command statements, declarations, assignments, etc.

There is no restriction on the contents of a macro, but the validity of the contents is only checked at execution time and any error will terminate execution. Macros can be defined recursively (be careful!) and can contain other macro definitions. Entry of the definition of a macro will terminate when the number of END (or EXIT) statements equals the number of DEFINE statements in the flow.

Formal parameters are of undefined type and are only instantiated as symbols when the macro is executed with actual parameters. Any number of such parameters can be declared and given names which will override globals of the same name within the scope of the macro execution. To return values from execution some of the formal parameters may have to be used as reference symbols. This will also take place at the time of macro execution.

Macro names are subject to the same restrictions as other symbols and additionally may not clash with existing command names or other macro definitions. Again, if there is an existing macro of the same name in the current scope, it will be replaced by the new definition.

DELETE

```
DELETE {<variable>|<macro>}
```

DELETE is a command used for removing defined symbols and macro definitions. All symbols declared in the scope of a macro execution are deleted when it exits so this command will normally only be used in interactive mode. Use SHOW to establish the current symbol table contents.

VERIFY

```
VERIFY <true| false> <>
```

This command controls the automatic echoing of executed lines to the screen. This can be useful when debugging macros as it allows you to see which lines are being executed, which line causes an error and so on. It can be turned on or off at any point. The command VERIFY without a parameter reports the current state of the mode.

D.5 Macros

The language is considered to be user extensible. The interpreter contains a set of hard coded base routines which operate directly on programmatic structures. These base commands are set up and used in a particular fashion according to the implementers' idea of the functional usage. The functions are usually created to be simple in form and atomic in operation. When a more complex function is desired, or simply a base function with a different set of default parameters or options, it can be created by the user using macro definitions and thereafter used in an identical way to the original command. The use of macro commands is very similar to that of the base commands even down to the defaulting and error reporting procedures. If this facility were simply an aliasing function it would be quite powerful, but limited by the demands of textual substitution. Instead it has been implemented as a full procedural language function and offers a variety of control constructs to provide a rich functional development environment.

D.5.1 Formal and actual parameters

A macro is defined by specifying a name and any number of formal parameter names.

```
> define muldiv a b c
```

The formal parameter names can be considered to be equivalent to symbol declarations which are in effect during the execution of the macro. The types of these formal parameters symbols are, however, undefined until the macro is called as explained below. Additional declarations of symbols can be made using assignment statement in the body of the macro as required. All of these symbols will be created as the macro is executed and will cease to exist when the macro exits. Statements within the macro will also be able to access global variables, that is variables declared in the enclosing scope of the macro whose names do not clash with the variables declared locally. The body of the macro can contain any command statements that are used elsewhere and entry of body statements must be terminated using an EXIT or END statement. These statements are equivalent in function.

```
>> res=0
>> res:=(a*b)/c
>> sho "the result is " res
>> exit
```

Executing the macro defined above is very simple:

```
> muldiv 3 2 1
the result is 6
> mu 5 4 2
the result is 10
```

Note that the macro is invoked by simply giving its name, in full or abbreviated form, on the command line along with its parameters. Any value parameters given will be fully evaluated before being passed into the macro. If a string parameter is given to the macro above, a string symbol will be created under the name of the formal parameter and will probably result in an error when evaluating the macro's internal expression. If a value is left unspecified on the invocation line, a numeric symbol is created for the actual parameter, with a value of zero.

D.5.2 Returning results

Clearly the example given above has only a limited use and at some stage it will be necessary to return results from a macro execution. The could be achieved by storing the result in a global variable of course but this is inflexible and only allows operation of the macro in certain ways. The other alternative is to have the value calculated and returned into the calling scope by means of an assignment.

```
> define muldiv a b c
>> num= (a*b)/c
>> exit num
```

At first sight this does not appear to achieve anything. However when called, the target of the assignment in the macro invocation RES is given the value that results from the calculation within the macro and referenced in the END statement.

```
> res= muldiv 3 2 1
> sho res
6
```

Again, note that the use of macros is purely procedural in nature. It would be impossible to use a macro call in the context of an expression due to both the syntax involved (the use of spaces as parameter separators) and the fact that no function or command value concept is exhibited.

D.5.3 Control flow

Macros are given real programming power with statements that control conditional execution. These are presented in a strictly structured fashion and, in fact, are implemented using a parameterless macro system themselves. The functions available are IF/ELSE, WHILE and FOR, and their formats are described on the following pages. The logical expression which controls these statements can be formed as a comparison between two numeric values or a numeric expression which evaluates to a logical result. In this sense a value of zero equates to logical FALSE. The functions available for comparison are:

- > greater than
- < less than
- ==equal to
- != not equal to
- >= greater or equal
- <= less or equal
- <> not equal

The following illustrates the use of this mechanism.

```
> if a>=b
> if a
> if a<>b
> if (a*2)/3>=1
```

Note that comparisons cannot be nested and that comparisons between floating point values will in fact be performed in the integer domain.

WHILE

WHILE <comparison>

```
> while a<b
>> a:=a+1
> end
```

The execution of the macro body statements is controlled by the evaluation of the logical expression before each execution. The comparison must be formed from currently declared variables and expressions, at least one half of which is assumed to be changed in value by statements within the macro body The macro body can contain other statements as desired including other control flows and macro definitions but must be terminated by an END or EXIT statement.

FOR

```
FOR <control variable> <initial> <final> <step>
```

The execution of the body statements is repeated over the range of values given for the control variable. The step defaults to unity and can be negative. The initial and final values default to unity as well. The control variable symbol name need not exist before the statement, but will exist after the loop has terminated with an undefined value. The macro body can contain other statements as desired including other control flows and macro definitions but must be terminated by an END or EXIT statement.

IF and ELSE

```
IF <comparison>
ELSE
```

The result if the logical expression is used to determine which of the two parameterless statement bodies is executed. The body statements can be of any form, including other macro definitions and control flows, but must be terminated with an END or EXIT statement.

WHILE

```
WHILE <comparison>
```

The execution of the macro body statements is controlled by the evaluation of the logical expression before each execution. The comparison must be formed from currently declared variables and expressions, at least one half of which is assumed to be changed in value by statements within the macro body The macro body can contain other statements as desired including other control flows and macro definitions but must be terminated by an END or EXIT statement.

E IMS B016 design revisions

The IMS B016 along with other INMOS *iq* Systems products is subject to periodic design review. In the case of the IMS B016 there has been one major design change–instigated to allow faster operation and reduce component count. While the two designs implement the same functions and are intended to be transparently compatible, there are some minor differences which some users should be aware of. In addition, there are some enhancements in the newer design which were not present in the original.

Physical inspection of the card will show a set of identification numbers adjacent to jumpers K1–K3. Original design cards show the ID:

```
IMS B016 221-CBRD-218-02
```

and have the circuit revision below that at `CT Rev 5` .

Newer cards have the ID:

```
IMS B016 221-CBRD-218-04
```

and `CT Rev 1` .

A small number of intermediate cards exist with the ID:

```
IMS B016 221-CBRD-218-03
```

and `CT Rev 2` –these are *identical* in operation to the normal "newer" cards. Still newer cards, with revisions "05" and higher, incorporate manufacturing improvements and will be identical in operation.

Software can distinguish between the two design revisions by reading the register at `#7FD98000` (Clear BusError Interrupt). On an original card, bit 8 reads '0'. A newer card will read '1' in bit 8.

Documentation and software for the original design instructs the user to disable VIC DRAM refresh. In the newer design, DRAM refresh is automatically disabled at reset and *must not* be enabled. If refresh is enabled the card will stop working with the middle LED lit brightly (Slave Access Indication). Since the original design in fact worked with DRAM refresh enabled it is possible that users' application software which inadvertently enabled VIC DRAM refresh would work on an original card and fail on a newer card. The IMS B016 uses its own refresh circuitry to ensure Dual-Access DRAM refresh.

The original design did not support the use of the PEX connector while the newer design does. The original design did not function correctly with IMS T801 byte writes to VMEbus D32 regions (the card's specification did not include this feature).

The older cards were fitted with revision F4 and F5 VIC chips. These device revisions feature a number of bugs – most of which are not relevant to the IMS B016.

However some have proved to be inconvenient such as the lack of software reset for the card from the VMEbus. The newer cards are fitted with VIC068A devices (F9 revision) which do not suffer from the bugs which are present in the older parts. Users wishing to do so may change the VIC068 device in an old card for a new part (available from Cypress Semiconductor, part number VIC068A-BC).

The newer design allows functional testing of the MAP-RAM devices by read–back of MAP-RAM contents. This is achieved by proceeding as per a MAP-RAM programming cycle, except that data is read rather than written and the data appears on bits 20–31 of the word read. After reading the data, a subsequent write cycle must be performed to reset the circuitry.

F Compatibility with the IMS B011

Other INMOS VMEbus boards include the IMS B014 TRAM Motherboard and the IMS B011 Master/Slave board. The IMS B011 is older than the IMS B016 and was designed to fill many of the same application areas. Consequently many customers who currently use the IMS B011 will be interested in moving to the IMS B016. This section outlines the relevant differences between the two boards and how they would affect users moving from the IMS B011 to the IMS B016.

F.1 General compatibility issues

Since the technology used to implement the IMS B016's VMEbus interface is significantly different to that used in the IMS B011, the two boards are *not* software compatible. Rather they are *functionally* compatible. That is, all functions performed by the IMS B011 (with a few exceptions) are also available on the IMS B016. The IMS B016 of course also has many additional features and is significantly faster than the IMS B011.

F.1.1 VMEbus interface

The programming and operation of the VMEbus interfaces are completely different between the two boards. The software routines which configure the interface must be totally re-written.

F.1.2 Memory map

The IMS B016's memory map is broadly compatible with the IMS B011–memory and VMEbus address spaces are compatible. Peripherals and registers are placed in different regions. F-ROM is located in the same region as the ROM on the IMS B011. The IMS B016 adds the ability to address VMEbus address zero (by mapping the IMS T801 registers to a different address). The IMS B011 lacked this ability but compatibility for certain transputer software which address the subsystem registers is ensured by having the registers mapped to address zero at board reset. The MAP-RAM feature of the IMS B016 removes the need for fixed VMEbus address regions for the different VMEbus cycle-types. Users should be able to configure the MAP-RAM to reflect the IMS B011's VMEbus memory-map. Note however that it may be necessary to configure the byte multiplexor in order to obtain the data bits in the same part of the transputer's 32-bit word as found on the B011.

The IMS B011 will initiate VMEbus cycles when its transputer makes accesses to any address allocated to the VMEbus in the board's memory map. This means that runaway software can make erroneous accesses to the VMEbus. The IMS B016 allows VMEbus MASTER accesses by the IMS T801 to be restricted on a page-by-page basis using the MAP-RAM. It is recommended than use is made of this feature when moving applications to the IMS B016. Also, for applications which should *never* make VMEbus MASTER accesses, all MASTER cycles are disabled on reset and need to be explicitly enabled using the appropriate control register.

F.1.3 Byte multiplexor

The IMS B011 does not have a byte multiplexor like the IMS B016's. The IMS B016, on reset, disables its byte multiplexor so applications which do not require this feature should not need to configure it.

F.1.4 TRAM slots

The IMS B011 has two TRAM slots. The IMS B016 has no TRAM slots and also lacks the IMS C004 used to switch links on the IMS B011. The function of TRAM motherboard is provided by the IMS B014, which was designed after the IMS B011. Applications which have TRAMs installed in an IMS B011 will need to add an IMS B014 to the card-cage.

F.1.5 Serial ports

The IMS B016 implements it's serial ports using the same DUART IC as used on the IMS B011. This means that software for the IMS B011 serial ports should need minimal modification in order to work on the IMS B016. Due to mechanical constraints, the IMS B016 lacks the serial port connector on the front panel. Also, because of the increased number of connections to P2, the IMS B016 serial port connections are on different pins on P2 when compared to the IMS B011.

G VMEbus performance

Unfortunately the data-transfer performance of VMEbus systems is difficult to estimate without performing characterisation experiments on the actual system. However, some specific timings can give the user an idea as to what performance to expect from the board.

The IMS B016 will respond to slave accesses with a DSx to DTACK delay of 70ns for writeposted transfers and 290ns for reads and non-writeposted transfers.

As a master, the IMS B016, when transferring to and from another IMS B016 is capable of the following transfer cycle times (measured while performing IMST801 block transfers from SRAM):

Transfer cycle times	Speed (ns)
VMEbus read	700
VMEbus write	700
VMEbus slave writeposting	520
VMEbus master writeposting	480
VMEbus master and slave writeposting	400

Peak transfer rates of 12.5Mbytes/s are achievable using master and slave writeposting with IMS T801 block moves from internal RAM. Due to the benefits of writeposting, higher performance is always obtained by moving data *from* where it originates, rather than *to* its destination. That is, the VMEbus transfers should be writes not reads.

These figures assume no VMEbus aquisition overhead which would be system-dependent.

Index

Sales Offices

EUROPE

DENMARK

2730 HERLEV
Herlev Torv, 4
Tel. (45–42) 94.85.33
Telex:35411
Telefax: (45–42) 948694

FINLAND

LOHJA SF–08150
Karjalankatu, 2
Tel. 12.155.11
Telefax: 12.155.66

FRANCE

94253 GENTILLY Cedex
7, Avenue Gallieni – BP 93
Tel. (33–1) 47.40.75.75
Telex: 632570 STMHQ
Telefax: (33–1) 47.40.79.10

67000 STRASBOURG
20, Place des Halles
Tel. (33) 88.75.50.66
Telex: 870001F
Telefax: (33) 88.22.29.32

GERMANY

6000 FRANKFURT
Gutleutstrasse, 322
Tel. (49–69) 237492
Telex: 176997 689
Telefax: (49–69) 231957
Teletex:6997689=STVBP

8011 GRASBRUNN
Bretonischer Ring, 4
Neukerloh Technopark
Tel. (49–89) 46006–0
Telex: 528211
Telefax: (49–89) 4605454
Teletex:897107=STDISTR

5000 HANNOVER 51
Rotenburgerstrasse, 28A
Tel. (49–511) 615960
Telex: 175118418
Telefax: (49–511) 6151243

8500 NURNBERG 20
Erlenstegenstrasse, 72
Tel. (49–911) 59893–0
Telex: 626243
Telefax: (49–911) 5980701

5200 SIEGBURG
Frankfurter Str. 22a
Tel. (49–2241) 660 84–86
Telex: 889510
Telefax: (49–2241) 67584

7000 STUTTGART
Oberer Kirchhaldenweg, 135
Tel. (49–711) 692041
Telex: 721718
Telefax: (49–711) 691408

ITALY

20090 ASSAGO (MI)
V.le Milanofiori – Strada 4 – Palazzo A/4/A
Tel. (39–2) 89213.1 (10 lines)
Telex: 330131 – 330141 SGSAGR
Telefax: (39–2) 8250449

40033 CASALECCHIO DI RENO (BO)

00161 ROMA
Via A. Torlonia, 15
Tel. (39–6) 8443341
Telex: 620653 SGSATE I
Telefax: (39–6) 8444474

NETHERLANDS

5652 AM EINDHOVEN
Meerenakkerweg, 1
Tel. (31–40) 550015
Telex: 51186
Telefax: (31–40) 528835

SPAIN

08021 BARCELONA
Calle Platon, 6, 4th Floor, 5th Door
Tel. (34–3) 4143300 – 4143361
Telefax: (34–3) 2021461

28027 MADRID
Calle Albacete, 5
Tel. (34–1) 4051615
Telex: 27060 TCCEE
Telefax: (34–1) 4031 134

SWEDEN

S–16421 KISTA
Borgarfjordsgatan, 13 – Box 1094
Tel. (46–8) 7939220
Telex: 12078 THSWS
Telefax: (46–8) 7504950

SWITZERLAND

1218 GRAND–SACONNEX (GENEVA)
CheminFrançois-Lehmann18/A
Tel. (41–22) 7986462
Telex: 415493 STM CH
Telefax: (41–22) 7984869

United Kingdom And Eire

MARLOW, BUCKS SL7 1YL
Planar House, Parkway
Globe Park
Tel. (44–628) 890800
Telex: 847458
Telefax: (44–628) 890391

AMERICAS

BRAZIL

05413 SAO PAULO
R. Henrique Schaumann 286–CJ33
Tel. (55–11) 883–5455
Telex: (391) 11–37988 "UMBR BR"
Telefax: 11–551–128–22367

CANADA

BRAMPTON, ONTARIO
341, Main St. North
Tel. (416) 455–0505
Telefax:416–455–2606

USA

NORTH & SOUTH AMERICAN MARKETING HEADQUARTERS
1000, East Bell Road
Phoenix, AZ 85022
(1)–(602)867–6100

SALES COVERAGE BY STATE

ALABAMA

ARIZONA

1000, East Bell Road
Phoenix, AZ 85022
Tel. (602) 867–6100

CALIFORNIA

200 East Sandpointe,
Suite 120,
Santa Ana, CA 92707
Tel. (714) 957–6018

2055, Gateway Place,
Suite 300
San José, CA 95110
Tel. (408) 452–9122

COLORADO

1898, S. Flatiron Ct.
Boulder, CO 80301
Tel.(303)449–9000

FLORIDA

902 Clint Moore Road
Congress Corporate Plaza II
Bldg. 3 – Suite 220
Boca Raton, FL 33487
Tel.(407)997–7233

GEORGIA

6025, G.Atlantic Blvd.
Norcross, GA 30071
Tel. (404) 242 –7444

ILLINOIS

600, North Meacham
Suite 304,
Schaumburg, ILL 60173–4941
Tel. (708) 517–1890

INDIANA

1716, South Plate St.
Kokomo, IN 46902
Tel. (317) 459–4700

MASSACHUSETTS

55, Old Bedford Road
Lincoln North
Lincoln, MA 01773
Tel. (617) 259–0300

MICHIGAN

17197, N. Laurel Park Drive
Suite 253,
Livonia, MI 48152
Tel. (313) 462–4030

MINNESOTA

7805, Telegraph Road
Suite 112
Bloomington, MN 55438
Tel. (612) 944–0098

NEW JERSEY

Staffordshire Professional Ctr.
1307, White Horse Road Bldg. F.
Voorhees, NJ 08043
Tel. (609) 772–6222

NORTH CAROLINA

4505, Fair Meadow Lane
Suite 220
Raleigh, NC 27607
Tel. (919) 787–6555

TEXAS

1310, Electronics Drive
Carrollton, TX 75006
Tel. (214) 466–7402

ASIA/PACIFIC

AUSTRALIA

NSW 2027 EDGECLIFF
Suite 211, Edgecliff Centre
203–233, New South Head Road
Tel. (61–2) 327.39.22
Telex: 071 126911 TCAUS
Telefax: (61–2) 327.61.76

HONG KONG

WANCHAI
22nd Floor – Hopewell Centre
183, Queen's Road East
Tel. (852–5) 8615788
Telex: 60955 ESGIES HX
Telefax: (852–5) 8656589

INDIA

NEW DELHI 110001
Liaison Office
62, Upper Ground Floor
World Trade Centre
Barakhamba Lane
Tel. 3715191
Telex: 031–66816 STMI IN
Telefax: 3715192

KOREA

SEOUL 121
8th Floor Shinwon Building
823–14, Yuksam-Dong
Kang-Nam-Gu
Tel. (82–2) 553–0399
Telex: SGSKOR K29998
Telefax: (82–2) 552–1051

MALAYSIA

PULAU PINANG 10400
4th Floor, Suite 4–03
Bangunan FOP, 123D Jalan Anson
Tel. (04) 379735
Telefax: (04) 379816

SINGAPORE

SINGAPORE 2056
28 Ang Mo Kio – Industrial Park, 2
Tel. (65) 4821411
Telex: RS 55201 ESGIES
Telefax: (65) 4820240

TAIWAN

TAIPEI
12th Floor
571, Tun Hua South Road
Tel. (886–2) 755–4111
Telex:10310 ESGIE TW
Telefax: (886–2) 755–4008)

JAPAN

TOKYO 108
Nissaki Takanawa Bld. 4F