USER'S GUIDE TO THE GENERIC BUS INTERFACE

June 1986

USER'S GUIDE TO THE GENERIC BUS INTERFACE

June 1986

This document was prepared by the Graphics Division of Symbolics, Inc.

Printed in the USA.

TABLE OF CONTENTS

APPENDIXES

LIST OF ILLUSTRATIONS

Figures

Tables

INTRODUCTION

This document describes the *Generic Bus Interface* (GBI). Both hardware and software are covered as well as specific examples and suggestions covering several kinds of applications. This document also includes installation instructions for both hardware and software.

This document is divided into the following chapters:

1. "GBI Architecture" provides a physical description of the GBI and includes a block diagram to show the logical location of devices on the *Generic Bus* (Gbus).

2. "Installation" is divided into hardware and software installation; hardware installation includes jumper connections and DIP switch settings. Diagrams of the interface boards are included.

3. "Programming the Gbus" describes the Lisp messages and flavors used to control devices on the Gbus.

4. Several appendixes are included. Appendix A describes the available hardware registers and the values to which they can be programmed. Appendix B lists the pinouts used for connecting the GBI to serial or parallel devices other than UNIBUS or MULTIBUS. Appendix C describes how to use the 68000 assembler provided on the Gbus system tape. Appendix D gives specific information about pin assignments on the Symbolics-manufactured MULTIBUS backplane.

In this document, the term *3600* refers to any member of that family of computers unless specifically contrasted or compared with other specific model numbers (such as 3670).

Although the term *Gbus* refers specifically to the external Gbus (the 60-conductor ribbon cable that connects to the UNIBUS or MULTIBUS interface board), it is also used to refer to the entire package of GBI boards and cables.

GBI ARCHITECTURE

Interface

The architecture of the Generic Bus Interface package can be more easily understood by making a logical separation at the rear bulkhead of the 3600. Installed within the 3600 backplane is the *Generic Bus Interface* (GBI), consisting of an Lbus board and a paddle board. Extending from the bulkhead is the *Generic Bus* (Gbus, or sometimes *external Gbus*), a pair of 60-conductor ribbon cables that connects the GBI to a UNIBUS or MULTIBUS interface board installed in the backplane of an external system.

The GBI forms the interface between the high-speed Lisp Machine world and the slower, asynchronous world supported by devices with UNIBUS or MULTIBUS backplanes. Through this interface, which supports UNIBUS and MULTIBUS backplane interface boards, the Symbolics 3600 can originate cycles on the Gbus, and both the 3600 and bus masters on the external bus have read-write access to a dual-ported memory buffer on the GBI.

The GBI consists of two boards that are installed in the Lisp Machine: a 16" x 18" Lbus board and a paddle board. Software is included for the GBI. An optional 68000 microprocessor is available for the GBI and can be used to manage Gbus devices when fast response to external events is required. An assembler is provided with the Gbus software for use in programming this optional 68000.

The external Gbus consists of a pair of 60-conductor ribbon cables that connects the GBI to the external device. It is an asynchronous bus functionally similar to that specified by the MULTIBUS standard. (For background on the MULTIBUS, refer to Intel MULTIBUS specification, Intel part no. 9800683-4.) Although the internal circuits of the GBI are buffered electrically from the external Gbus by the paddle board, the external Gbus is logically the same as the bus internal to the GBI.

The data path is 16 bits wide; the Gbus can handle either 8- or 16-bit transfers (the number of bits is specified by the bus master when the cycle is initiated). Like the MULTIBUS, Gbus address space is divided into memory and I/O: memory address space is 1 Mbyte (optionally 16 Mbytes) and I/O address space is 64 Kbytes.

The block diagram on the next page shows the major functional components of the Gbus. As shown in this diagram, the Gbus has *master* and *slave* devices. A master device can request and gain control of the bus; a slave is accessible for read or write operations by a bus master.

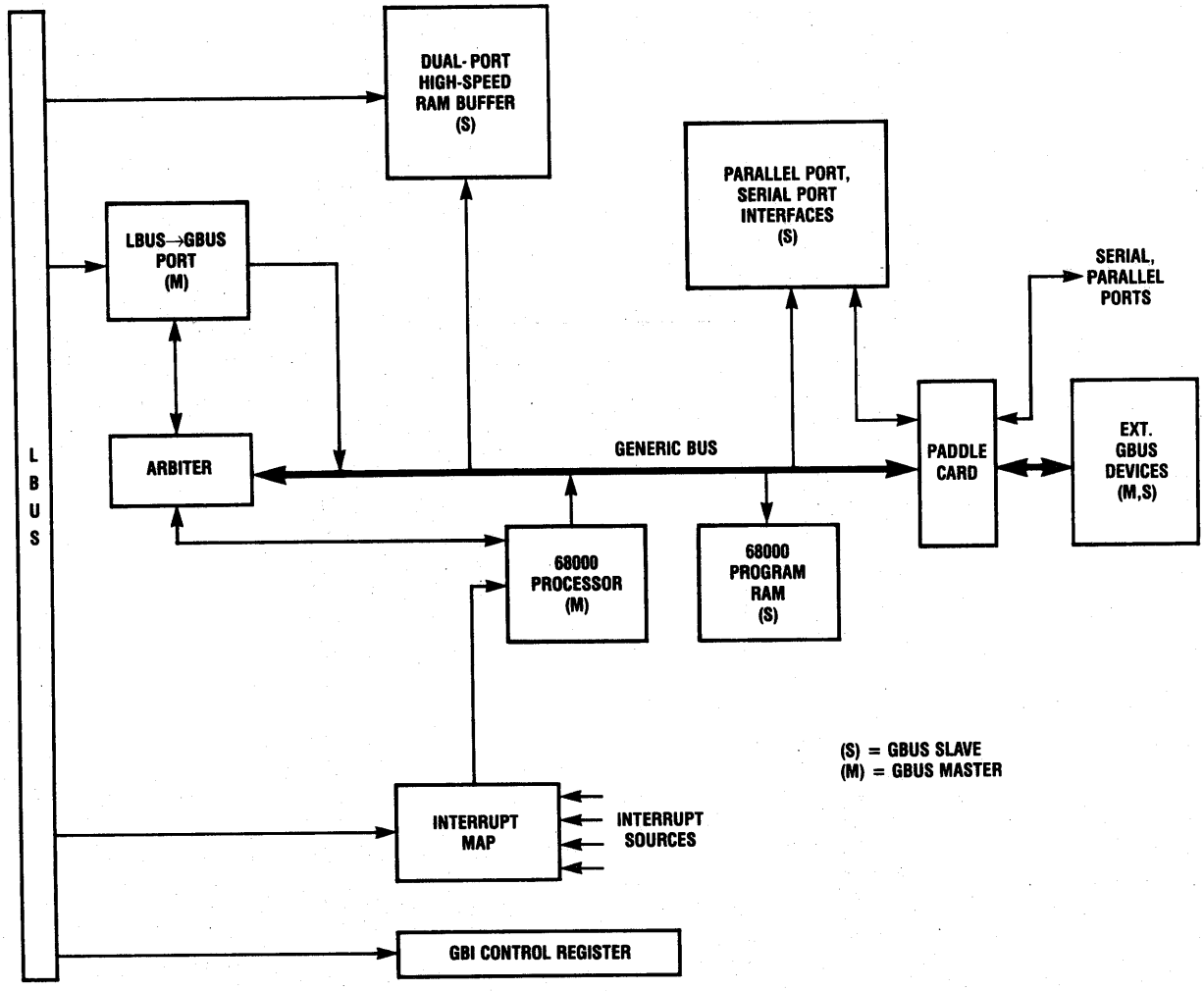

Figure 1.1 *Logical relationship of devices on the Gbus*

A master can initiate either a read or a write cycle in memory space or I/O space, using the messages **:memory-read, :memory-write, :io-read,** and **:io-write**. The master devices on the Gbus are the Lbus→Gbus port and the optional 68000 processor. The other devices, including the 68000 program RAM as well as parallel- and serial-type interface devices, are slaves. The UNIBUS interface board has slave devices on it. Each slave is assigned a specific address on the bus. A slave can be either some sort of memory or an I/O device such as a UART.

Both slave and master devices can exist within the external UNIBUS or MULTIBUS backplane. However, master devices on the external backplane must observe the following two restrictions:

1. The 3600 does not permit external processors to have direct memory access (DMA) to 3600 memory. Communication between the 3600 and the external device must take place in one of two ways: by either the Gbus or the external device using the dual-ported memory on the Gbus interface board; or by the 3600 doing bus cycles.

2. The Gbus cannot interrupt the 3600. Neither 3600 hardware nor software supports interrupts. The 3600 must poll the state of the Gbus and/or external devices to determine interrupt status. The 68000, however, does field interrupts and is recommended for real-time response to external stimuli.

How the 3600 Sees It

The 3600 can use the Lbus→Gbus port to initiate cycles on the Gbus by executing functions provided in the support software. In this way, the 3600 can read from or write to any slave device on the Gbus and any attached external bus. The 68000's program memory is accessed in this manner.

The 3600 can also directly read from and write to the dual-port high-speed RAM buffer as its own memory. A displaced array can be made to point to the buffer. This array can be used as any normal array would be. High-speed data transfers between external bus masters and the 3600 are made using the buffer. In the example of a UNIBUS option and a DR11-W DMA controller, the 3600 sets the device's registers using the Lbus→Gbus port, causing it to transfer data to or from the buffer.

Interrupts

Although it is not possible to interrupt the 3600 or to access 3600 memory directly from the Gbus, interrupts can be handled by interrupting the 68000 processor. Interrupting the 68000 processor is the preferred method for dealing with devices or problems that require very fast response. The 68000 can respond to an interrupt request in approximately 6 microseconds, from assertion to executing the first instruction of the service routine.

Fifteen interrupt sources are on the Gbus: seven "external bus" interrupts, plus eight from devices on the board. Interrupt requests from external devices can be stored in a 2K x 8-bit RAM called the *interrupt map*, where the 3600 can then read and determine the type and priority of the incoming interrupt request. A macro for defining this interrupt map is provided in the Gbus software and is described on pages 35 and 36.

MULTIBUS Option

With the MULTIBUS interface board, the Gbus can be connected to a MULTIBUS backplane, which can be either in an existing computer system or in a card cage with various MULTIBUS devices.

Since Gbus address space is logically the same as MULTIBUS address space, the MULTIBUS interface board performs only electrical buffering. With a MULTIBUS interface board, it is as if the Gbus is directly connected to the backplane, with no address mapping or restriction. Bus masters in the MULTIBUS backplane can access any slave on the Gbus board, including the buffer and 68000 program memory. Both the 3600 (through the Lbus→Gbus port) and the 68000 can access any slave on the MULTIBUS backplane. The MULTIBUS interface board can be jumpered to supply the bus-clock signals if necessary (see step 4 of MULTIBUS installation on page 15).

When designing or installing a Gbus/MULTIBUS system, be especially careful of address assignment conflicts between slave devices on both sides of the bus. Do not assign MULTIBUS devices to any of the addresses assigned to Gbus devices or designated "reserved" in the section "Devices and Addresses" (see page 7). In addition, caution must be exercised when interfacing MULTIBUS devices that use only 8 bits of I/O addressing. The MULTIBUS interface board does not decode any of the I/O address bits, so MULTIBUS devices that use only 8 I/O address bits should be assigned only addresses 0 through 177 (octal).

UNIBUS Option

The UNIBUS interface board interfaces the Gbus with a UNIBUS backplane in either an existing computer system or a card cage containing only I/O devices.

An 18-bit range of Gbus addresses is mapped directly into UNIBUS, and access of an address in that range initiates a UNIBUS cycle. A 15-bit (16-Kword) range of UNIBUS addresses can be mapped via a set of software-settable mapping registers to either Gbus memory or I/O space. The mapping registers can be set from either side. The actual UNIBUS address of the mapped area is set by switches on the interface board, and the size of the mapped area is likewise selected.

You can configure the UNIBUS interface board to be the *primary CPU* in the backplane. The primary CPU fields interrupts and arbitrates bus mastership. The UNIBUS cannot interrupt the 3600, but it can interrupt the 68000 processor. When the UNIBUS interface board is not the primary CPU, it can, under the control of a Gbus master, generate interrupts of specified vectors and priorities.

The registers, interrupts, and priorities of the UNIBUS interface board are described in Appendix A--"UNIBUS Registers, Interrupts, and Priorities."

Address Map

The address map is accessible in UNIBUS address space at a location that can be switch-programmed anywhere above 760,000(8). The Map Register Address DIP switch on the interface board is provided to select the address offset. (See page 24 for the settings.)

The map must be accessed in word mode.

Switches 7 and 8 (size 1 and size 0, respectively) select the size of the mapped area. The choices are 4, 8, 16, and 32 Kbytes. Each 2-Kbyte segment has a map register. When less than the full mapped area is selected, only the first map registers are used, for example, when 4 Kbytes are mapped, the first two map registers are used. See page 25 for instructions on setting these switches.

The UNIBUS Select DIP switch locates the mapped portion of UNIBUS address space. Six switches select the high 6 bits of the address of the mapped area (see page 25). The mapped area can be placed anywhere in UNIBUS address space.

68000 Processor

The 68000 processor on the Gbus works like any other 68000 processor, with the following peculiarities:

- The very top 64 Kbytes of the 68000's 24-bit address space map into Gbus I/O space. The base address of 68000 I/O space is 77,600,000(8), or FF0,000(16).

- Byte addressing is reversed from standard 68000 practice. That is, when doing a byte transfer when address bit <A0> is cleared, data bits <0-7> are selected; when <A0> is set, data bits <8-15> are selected. This byte-addressing scheme is compatible with DEC processor practice, as opposed to Motorola (and IBM-360) practice and should be noted when trying to port programs from other 68000 processors.

- All interrupts are auto-vectored. The three interrupt lines to the 68000 come from the three high-order outputs of the interrupt map.

- The 68000 processor uses an 8-MHz clock. Reads of the 68000 program memory occur with no wait states; writes cause one wait state. Reads and writes of all other Gbus devices cause at least one wait state; the exact number varies from device to device.

A watchdog timer is included on the GBI; the timer can be enabled and disabled under control of the 3600. When enabled, the 68000 processor must perform a write to a certain I/O space address at least once every 8 seconds, or the timer asserts 68000 Reset.

Devices and Addresses

The Gbus address space, for historical reasons, supports a 20-bit memory-space addressing mode. In this mode, the high 4 bits of any Gbus memory-space address are ignored. A setting in the control register enables or disables 20-bit mode. If full 24-bit addressing is used, the high-order 4 bits for the address comparison are taken from a field of the control register. If the MULTIBUS option is used with an existing computer system, you almost certainly want to use full 24-bit addressing. Gbus masters always generate 24-bit addresses, so care should be taken in assigning addresses of I/O devices if the MULTIBUS option is used in a standalone backplane. The functions in the supplied software always take high-address bits into account when addressing the on-board slave devices.

Gbus Memory Space

Note that the following addresses are 20-bit addresses. If 24-bit addressing is enabled, the following are offsets within the 24-bit address space. In that case, the 4 high-order bits are determined by the high-address-bit field in the control register (see **:enable-hi-addr** message on page 34). Addresses are in octal.

0 - 377,777	128 Kbytes reserved for 68000 program RAM.
400,000 - 1,377,777	Reserved.
1,400,000 - 1,777,777	Dual-port high-speed buffer.
2,000,000 - 2,777,777	UNIBUS option, if present. Memory cycles made to this block of addresses are performed on the UNIBUS.

Note: The UNIBUS option decodes only the low 20 address bits and so does not function correctly when 24-bit addressing is enabled.

When the MULTIBUS interface board is used, Gbus and MULTIBUS address spaces are the same. When the Gbus is used in a currently operating computer system, it might be wise to set the high-address-bit field of the control register to some nonzero value to avoid collision between the computer's low-address memory and the 68000 program RAM. In such an installation it seems unlikely to be using the on-board 68000, so moving the 68000 program memory should not be a problem.

Gbus I/O Space

The location of the I/O device registers on the GBI depends on the setting of the high-address-bit field of the control register. The base address for the devices is 140,000(8) plus 1,000(8) times the high-address bits. The following addresses are in octal.

High-Address Bits <A20-A23>	I/O Base Address
0	140,000
1	141,000
2	142,000
:	:
15	155,000

The offsets given on the next page are added to the base address in order to obtain the device-specific address. Hence, when high-address bits contain 0, the first serial port device register is at 140,200(8).

See Appendix A--"UNIBUS Registers, Interrupts, and Priorities" for addresses and bit assignments of I/O devices on the UNIBUS interface board.

Serial Ports (Byte mode only)

Refer to Intel 8274 Data Sheet for programming information for the Multi-Protocol Serial Controllers (MPSC).

Offset(8)	Register
200	MPSC A Channel A Data
201	MPSC A Channel B Data
202	MPSC A Channel A Control
203	MPSC A Channel B Control
204	MPSC B Channel A Data
205	MPSC B Channel B Data
206	MPSC B Channel A Control
207	MPSC B Channel B Control
342	Serial Clock Control
	Bit <4> MPSC A Channel A Ext Clock Enable (1 = Enable)
	Bit <5> MPSC A Channel B Ext Clock Enable
	Bit <6> MPSC B Channel A Ext Clock Enable
	Bit <7> MPSC B Channel B Ext Clock Enable
354	Serial Baud Rate (word mode only; write only)
	Bits <0-3> MPSC A Channel A Baud
	Bits <4-7> MPSC A Channel B Baud
	Bits <8-11> MPSC B Channel A Baud
	Bits <12-15> MPSC B Channel B Baud

Each four-bit field is coded as follows:

Value(8)	Baud Rate
0	50
1	75
2	110
3	134.5
4	150
5	300
6	600
7	1200
10	1800
11	2000
12	2400
13	3600
14	4800
15	7200
16	9600
17	19,200

Parallel Port (Word mode only)

Offset(8)	Register	
370	Data bits	
	16-bit interface	
	Bits <0-15>	Parallel Port Data 0-15 In/Out
	8-bit interface	
	Bits <0-7>	Parallel Port Data 0-7 In/Out
	Bits <8-13>	Not used
	Bit <14>	Select (read only)
	Bit <15>	Paper Empty (read only)
372	Control bits	
	Bit <0>	Data Out Accepted (read only)
	Bit <1>	Data In Ready (read only)
	Bit <2>	Start Write Transfer
	Bit <3>	Not used
	Bit <4>	Data In Ready Interrupt Enable
	Bit <5>	Need Data Interrupt Enable
	Bit <6>	Spare Write Signal
	Bit <7>	Spare Read Signal

Other Registers

Offset(8)	Register	
210-213	Reserved	
240 - 277	Paddle Board ID Prom (read only)	
340	68000 Control Register	
	Bit <0>	68000 Interrupt Enable
	Bit <1>	Lisp→68000 Doorbell
	Bit <2>	Doorbell Interrupt Enable
600	Calendar Address Register	
	Bits <8-11>	Address input to the calendar chip
602	Calendar Data Register	
	Bits <8-11>	Data to/from the calendar chip. To access the calendar, the 4-bit address must first be written into the Address Register, and the data then read from or written into the Data Register. (Also refer to the data sheet for the National Semiconductor MM58274 Clock/Calendar chip.)

Offset(8)	Register
604	Timer (word mode only; write only) (Timer rate is selected by jumpers on the paddle board.) Bit <8> Enable Timer Interrupts Bit <9> Clear Timer Interrupt Bit <10> Reset Timer
612	Keep-Alive Register The 68000 should perform a write to this address at least once every 8 seconds when the watchdog timer is enabled. See the message **:enable-timeouts** on page 33.
614	Reserved

INSTALLATION

MULTIBUS

1. **Set the 68000 interrupt timer.**

   The Gbus paddle board is wired at the factory to support 31.25 KHz. To select a different rate, cut the wire in R6 and solder a wire into one of four pins on the paddle board. (See page 14 for the location of devices on the Gbus paddle board.)

   The following frequencies are available:

Jumper position	Frequency	Period
R6	31.25 KHz	32 usec (factory frequency)
R7	62.5 KHz	16 usec
R8	125 KHz	8 usec
R9	250 KHz	4 usec

2. **Configure the 8-bit parallel port.**

   If the 8-bit parallel port option is included and you intend to use it, you must configure the GBI paddle board for the type of electrical interface you want. The four types of interface are Symbolics Enhanced Line Printer (ELP), Centronics, Data Products Long Lines, and Data Products TTL.

   - ELP and Data Products Long Lines interfaces:

     Load SIP resistor networks RN12, RN13, and RN14, and remove SIP resistor network RN20.

   - Centronics and Data Products TTL interfaces:

     Remove SIP resistor networks RN12, RN13, and RN14, and load SIP resistor network RN20.

   (See Appendix B for more information about parallel port cabling.)

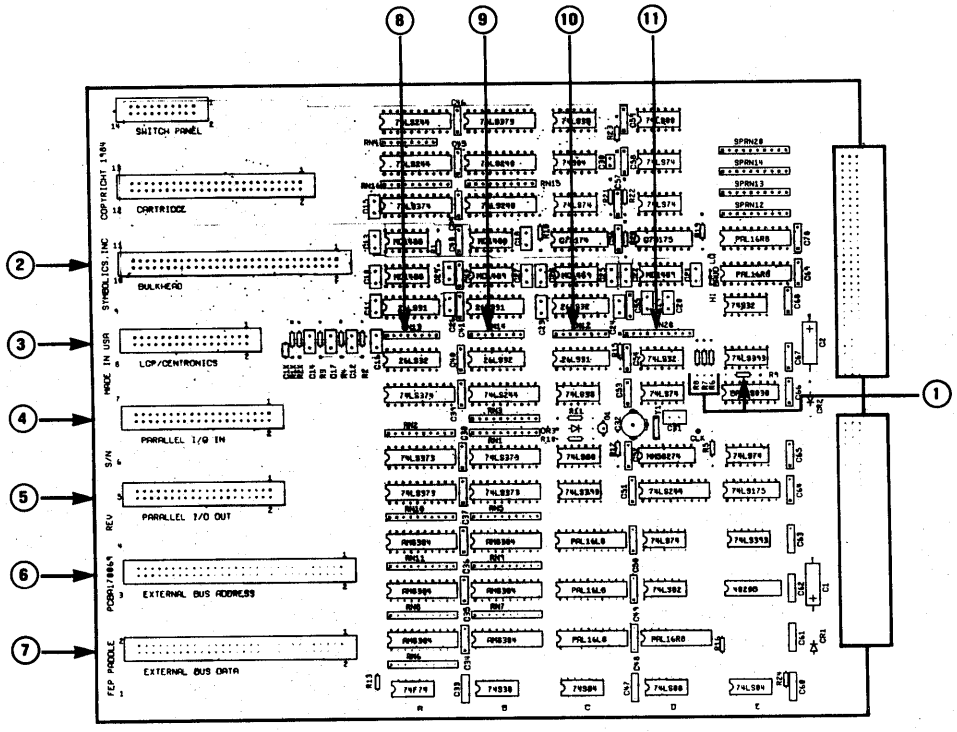

Figure 2.1 *Gbus paddle board*

1	Interrupt timer
2	Bulkhead connector
3	LGP/Centronics connector
4	Parallel I/O input connector
5	Parallel I/O output connector
6	External bus address connector
7	External bus data connector
8	RN13 SIP resistor network
9	RN14 SIP resistor network
10	RN12 SIP resistor network
11	RN20 SIP resistor network

3. Install the GBI boards.

The GBI Lbus board can be installed in any Lbus backplane slot that is not wired for the color bus. This wiring is accomplished by either backplane hard-wiring or the installation of the baby backplane.

The paddle board installs behind the Lbus board.

4. Set the Bus Clock jumpers.

The MULTIBUS interface can be configured to drive the BCLK and CCLK signals if no other device in the MULTIBUS card cage generates these clocks.

To enable driving the clocks, install the jumpers between the following pins in the group marked MBUS CTRL:

LOC BCLK to BCLK
LOC CCLK to CCLK

Do not install these jumpers if another device in the MULTIBUS card cage generates these clocks or if the MULTIBUS interface is used in existing computer systems, since microprocessor boards generate these signals.

Note: Certain jumpers in the MBUS CTRL group might not be labeled in early versions of the board. The following diagram shows the correct placement of jumpers:

EXT INIT	o	o	INIT
PWR RST	o	o	INIT
LOC BCLK	o	o	BCLK
LOC CCLK	o	o	CCLK
GND	o	o	BPRN
CBRQ OUT	o	o	CRBQ
CBRQ OUT	o	o	CRBQIN
HI	o	o	CBRQIN

5. Set the MULTIBUS INIT configuration.

The MULTIBUS interface can be configured to assert the MULTIBUS INIT signal under one or both of the following conditions:

- When the card cage containing the MULTIBUS is powered up.
- When, under 3600 program control, the MULTIBUS INIT line is asserted by the Gbus.

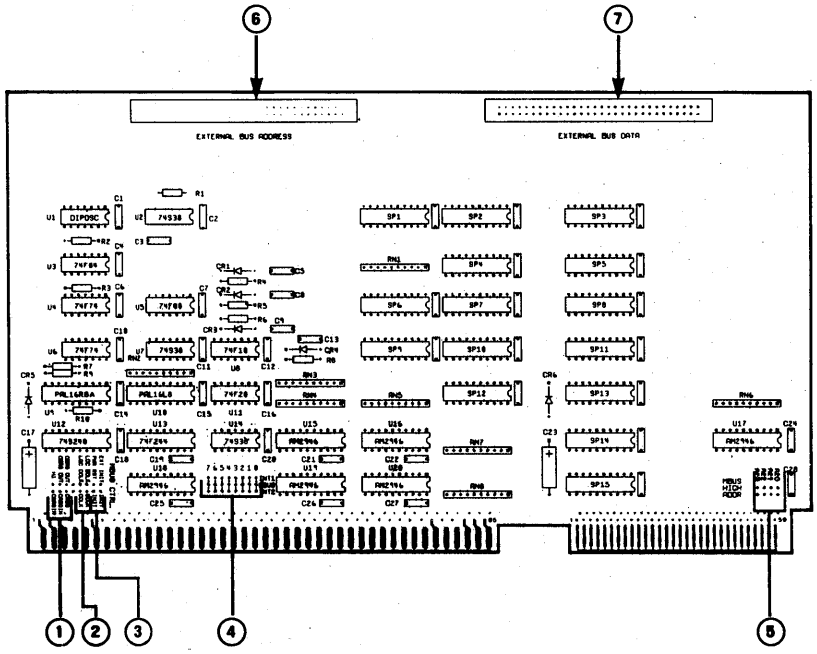

Figure 2.2 *MULTIBUS interface board*

1	Bus Arbitration jumpers
2	Bus Clock jumpers
3	INIT configuration jumpers
4	Interrupt jumpers
5	High-Address-Bit jumpers
6	External bus address connector
7	External bus data connector

To enable an INIT signal when the MULTIBUS card cage is powered up, install a jumper between the following pins in the MBUS CTRL group:

PWR RST to INIT

To enable INIT under the control of the Gbus in the 3600, install a jumper between the following pins:

EXT INIT to INIT

If you want the MULTIBUS interface to assert the INIT signal when the card cage is powered up and under control of the GBI in the 3600, install both jumpers.

Note: It is essential that a device in the MULTIBUS environment generate INIT at power-up.

6. **Set the Bus Arbitration jumpers.**

The way in which the arbitration jumpers are configured depends on the type of other MULTIBUS devices present.

a. When the MULTIBUS interface board is the only master-type device, install jumpers between the following pins in the MBUS CTRL group:

HI to CBRQIN
GND to BPRN

(These pins might not be marked in early versions of the board. Refer to step 4.)

Make sure jumpers are not installed on the CBRQIN and CBRQ OUT pins.

b. When one or more other master-type devices are in the MULTIBUS card cage and none of them uses the Common Bus Request (CBRQ) signal, install a jumper between the following pins:

HI to CBRQIN

Make sure jumpers are not installed on the CBRQ, BRPN, GND, and CBRQ OUT pins.

c. When some master-type devices that use Common Bus Request are in the MULTIBUS card cage, install jumpers between the following pins:

CBRQ OUT to CBRQ
CBRQ OUT to CBRQIN

Make sure nothing is connected to the HI pin.

d. When the MULTIBUS interface is to be the highest priority device in a serial-priority resolution scheme, add a jumper between the following pins:

GND to BPRN

(These pins might not be marked in early versions of the board. Refer to step 4.)

7. Set the High-Address-Bit jumpers.

The MULTIBUS interface supports MULTIBUS systems using either 20- or 24-address bits. The four jumpers in the MBUS HI ADDR group connect P2 edge connector pins 55, 56, 57, and 58 to the address transceivers on the MULTIBUS interface board.

The High-Address-Bit jumpers should always be installed in 24-address-bit systems. They can also be installed in 20-address-bit systems unless those P2 connector pins are used for some other purpose.

Note: In early versions of the MULTIBUS interface, the silkscreened labels for the pairs of jumper pins might be incorrect. When the board is viewed from the component side with the cable connectors at the top, A20 is the rightmost jumper.

8. Set the Interrupt jumpers.

The MULTIBUS interface can be jumpered so that any two of the eight MULTIBUS INT lines generate interrupts to the Gbus. The jumper group labeled INT1 MBUS INT2 determines which MULTIBUS interrupt lines affect the Gbus.

The 8 MULTIBUS INT lines (numbered 0 through 7) terminate in the jumper group labeled INT1 MBUS INT2. Two of these interrupts can be selected by

using a jumper to connect them to any of the pins under the labels INT1 and INT2. All pins under the label INT1 connect to Gbus Interrupt 1; all pins under INT2 connect to Gbus Interrupt 2.

For example, setting the jumpers indicated in the following diagram connects MULTIBUS INT0 to Gbus Interrupt 1 and connects MULTIBUS INT5 to Gbus Interrupt 2:

```
Interrupt
 Lines      INT1   MBUS   INT2

   0        o----------o     o
   1        o        o       o
   2        o        o       o
   3        o        o       o
   4        o        o       o
   5        o        o----------o
   6        o        o       o
   7        o        o       o
```

9. Configure the MULTIBUS backplane jumpers for priority resolution.

The MULTIBUS interface can be used with either serial- or parallel-priority resolution schemes.

- For serial-priority resolution, the Bus Priority In (BPRN) and BPRO signals must be chained correctly between boards in the card cage through use of backplane jumpers and/or ensuring that no empty slots exist between boards. Figure 2.3 shows the location of the jumpers on the Symbolics 4-slot MULTIBUS backplane (as seen from the front):

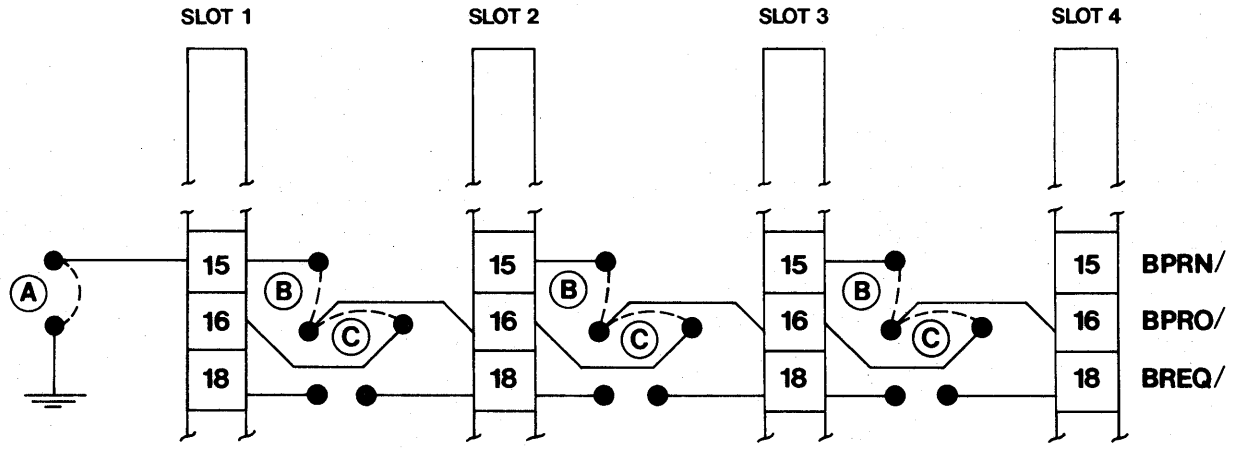

Figure 2.3 *Symbolics 4-slot MULTIBUS backplane, front view*

a. Make sure the jumper labeled A is installed.

b. Install jumpers labeled B when the slot to the left does not contain a master-type device.

c. Install jumpers labeled C when the slot to the left contains a master-type device.

- A parallel-priority resolution scheme must be accomplished by a resolution circuit external to the Symbolics MULTIBUS interface. If you are using a parallel-priority scheme, remove serial-priority jumpers from the backplane and all wirewrap connections between BREQ and BPRN (Bus Priority In) pins and the parallel-priority resolver.

10. Configure the MULTIBUS backplane for 24-bit addressing, if necessary.

Some backplanes, such as the Symbolics 4-slot backplane, require additional wiring to support 24 bits of memory addressing. For 24-bit addressing, the following four pins in each slot must be bussed across the P2 connectors of the backplane:

Pin 55 (ADR16/)
Pin 56 (ADR17/)
Pin 57 (ADR14/)
Pin 58 (ADR15/)

In the case of other MULTIBUS backplanes, refer to the relevant manufacturer's documentation.

11. Install the MULTIBUS interface board.

Install the MULTIBUS interface board into one slot in a standard MULTIBUS card cage. The slot that the board is plugged into is determined by the number and type of other MULTIBUS devices in the card cage. The MULTIBUS environment can fall into one of the following three categories:

a. The Symbolics MULTIBUS interface is the only master-type device in the MULTIBUS card cage, that is, all other MULTIBUS boards are either memory or non-DMA I/O devices.

   In this case, plug the MULTIBUS interface board into any MULTIBUS slot.

b. Other master-type devices are in the MULTIBUS card cage; none of them uses the Common Bus Request (CBRQ) signal.

   In this case, plug the MULTIBUS interface board into the slot that has lowest bus priority. If a serial-priority resolution scheme is being used, the MULTIBUS interface must be the master-type board located furthest from the board whose BPRN input is grounded.

c. Other master-type devices are in the MULTIBUS card cage; at least one of them uses Common Bus Request.

   In this case, install the MULTIBUS interface board at a priority that is either lower or higher than other boards that use Common Bus Request but lower than all master-type devices that do not use Common Bus Request.

12. Connect the MULTIBUS-Gbus external cables.

The interface between the MULTIBUS interface board and the Gbus paddle board uses two 60-conductor cables. One cable is used for address, the other for data.

Plug one end of one cable in the external bus address connector on the paddle board. Plug one end of the other cable in the external bus data connector on the paddle board.

Plug the free end of the cable connected to the external bus address connector into the address connector on the MULTIBUS interface board. Plug the free end of the cable connected to the external bus data connector into the data connector on the MULTIBUS interface board.

13. Connect the serial and parallel port cables (if included).

a. Connect the Gbus serial port cable to the bulkhead connector on the Gbus paddle board.

b. Connect the 8-bit parallel port cable to the LGP/Centronics connector on the Gbus paddle board.

14. Load the software.

See the section "Software" starting on page 27.

UNIBUS

1. Set the 68000 interrupt timer.

The Gbus paddle board is wired at the factory to support 31.25 KHz. To select a different rate, cut the wire in R6 and solder a wire into one of four pins on the paddle board. (See page 14 for the location of devices on the Gbus paddle board.)

The following frequencies are available:

Jumper position	Frequency	Period	
R6	31.25 KHz	32 usec	(factory frequency)
R7	62.5 KHz	16 usec	
R8	125 KHz	8 usec	
R9	250 KHz	4 usec	

2. Configure the 8-bit parallel port.

If the 8-bit parallel port option is included and you intend to use it, you must configure the Gbus paddle board for the type of electrical interface you want. The four types of interface are Symbolics Enhanced Line Printer (ELP), Centronics, Data Products Long Lines, and Data Products TTL.

- ELP and Data Products Long Lines interfaces:

  Load SIP resistor networks RN12, RN13, and RN14, and remove SIP resistor network RN20.

- Centronics and Data Products TTL interfaces:

  Remove SIP resistor networks RN12, RN13, and RN14, and load SIP resistor network RN20.

(See Appendix B for more information about parallel port cabling.)

3. Install the GBI boards.

The GBI Lbus board can be installed in any Lbus backplane slot that is not wired for the color bus. This wiring is accomplished by either backplane hard-wiring or the installation of the baby backplane.

The paddle board installs behind the Lbus board.

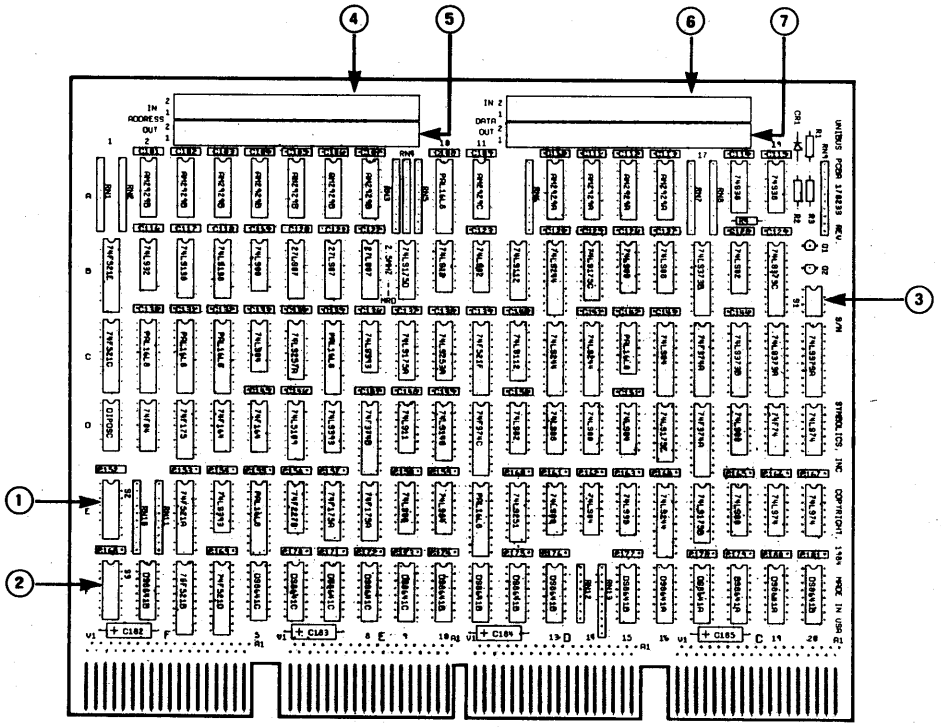

Figure 2.4 *UNIBUS interface board*

1	UNIBUS Select DIP switch
2	MAP Register Address DIP switch
3	4-unit DIP switch
4	Address-In connector
5	Address-Out connector
6	Data-In connector
7	Data-Out connector

4. Set the 4-unit DIP switch on the UNIBUS interface board.

The DIP switch in position B20 uses only switches 2, 3, and 4.

Switch 2:	Open -	Activating the Gbus INIT line automatically activates the UNIBUS INIT line.
	Closed -	Activating the Gbus INIT line does not activate the UNIBUS INIT line.
Switch 3:	Must be closed.	
Switch 4:	Open -	The 3600 is the UNIBUS primary CPU.
	Closed -	The 3600 is not the UNIBUS primary CPU. See the section "Address Map" on page 5.

5. Set the Map Register Address DIP switch.

This 8-unit DIP switch, which is in location F1 on the UNIBUS interface board, selects part of the address offset for the UNIBUS map registers in UNIBUS address space.

The address offset must fall between 760,000 and 777,760. Hence bits <A13-A17> of the offset are preset to 1s. The switches select bits <A5-A12> of the offset. Switch 1 selects bit <A12>, switch 2 selects bit <A13>. The other switches follows this pattern until switch 8, which selects bit <A5>. When the switch is open, the corresponding address bit is a 1.

For example, if you wanted to assign the starting location of the map registers to UNIBUS address 770,000 (octal), you would set the Map Register Address switches as follows:

1	Open (1)
2	Closed (0)
3	Closed
4	Closed
5	Closed
6	Closed
7	Closed
8	Closed

6. Set the UNIBUS Select DIP switch.

This 8-unit switch, which is in location E1 on the UNIBUS interface board, selects the size and offset for the FEP in UNIBUS address space. Switches 7 and 8 (size 1 and size 0, respectively) are used to select the size. They map as follows:

	Size 1 (Switch 7)	Size 0 (Switch 8)
4 Kbytes	Closed	Closed
8 Kbytes	Closed	Open
16 Kbytes	Open	Closed
32 Kbytes	Open	Open

The remaining 6 switches select the offset. Switch 1 selects bit <A17>, switch 2 selects bit <A16>; the switches follow this pattern until switch 6, which selects bit <A12>. If the switch is open, the corresponding address bit is a 1.

If the size is 8 Kbytes, only 5 of the switches are needed; if the size is 16 Kbytes, only 4 of the switches are needed, etc. All unused switches must be set closed.

For example, if you wanted to map 8 Kbytes of UNIBUS address space into Gbus space starting at UNIBUS address 100,000 (octal), you would set the UNIBUS Select DIP switch as follows:

1	Closed (0)
2	Closed
3	Open (1)
4	Closed
5	Closed
6	Closed
7	Closed
8	Open

7. Install the UNIBUS interface board.

If you are installing the UNIBUS interface board in an existing computer system, install it at a lower priority than the processor or UBA. Otherwise, install it in the highest priority slot of the UNIBUS backplane. This is the top slot in the Symbolics-supplied Wesperline backplane; use connectors C, D, E, and F (the rightmost connectors).

Be sure to refer to the manufacturer's documentation for specific installation instructions when installing the UNIBUS board into an existing computer system.

8. Connect the UNIBUS-Gbus external cables.

The interface between the UNIBUS interface board and the Gbus paddle board uses two 60-conductor cables. One cable is used for address, the other for data.

Plug one end of one cable in the external bus address connector on the paddle board. Plug one end of the other cable in the external bus data connector on the paddle board.

Plug the free end of the cable connected to the external bus address connector into the connector labeled "Address-In" on the UNIBUS interface board. Plug the free end of the cable connected to the external bus data connector into the connector labeled "Data-In" on the UNIBUS interface board.

9. Connect the serial and parallel port cables (if included).

a. Connect the Gbus serial port cable to the bulkhead connector on the Gbus paddle board.

b. Connect the 8-bit parallel port cable to the LGP/Centronics connector on the Gbus paddle board.

10. Load the software.

See the section "Software" starting on page 27.

Software

1. **Load the tape.**

   Software comes on a standard distribution format cart tape. To load it, use the function **dis:load-distribution-tape**. This creates the directory **SYS:Gbus;** on the system host, and loads the files for the systems Gbus and Gbus-ASSEMBLER.

2. **Load the system.**

   Type the command **:Load System Gbus** (or **:Load System Gbus-ASSEMBLER**) to bring the system into memory. When the Gbus system is loaded, the world, which now contains the Gbus system, can be saved to disk for transfer to other Lisp machines. The Gbus system facilitates this transfer by clearing itself of any machine-specific knowledge when the world is saved to disk.

   When a cold boot is executed, the Gbus system reconfigures itself to account for the current machine-specific information, ensuring up-to-date values for the variables representing the GBI boards. The world can be saved with the Gbus system and transferred to other machines; when the world is saved, the Gbus system clears all knowledge of the particular machine on which it is running and reconfigures itself upon cold boot.

3. **Update the namespace entry.**

   If a UNIBUS interface board is connected to the Gbus board, it must be declared in the namespace entry for the host because there is no way to detect the presence or absence of the UNIBUS option; the program must rely on the peripheral field of the host's namespace entry.

   Since the MULTIBUS is electrically identical to the Gbus and since all signals sent to the Gbus are therefore sent through buffers to the MULTIBUS, the MULTIBUS need not be declared as a separate item in the namespace. A MULTIBUS device should be declared using interface type GBUS <n> (for example, GBUS 0). A Gbus that connects to a MULTIBUS need only be declared as Gbus in the peripheral namespace entry.

For the purpose of the namespace entry, multiple Gbus boards are identified by number. Numbering starts at 0 at the leftmost Gbus board and increases by 1 for each Gbus board to the right. A lone Gbus board is always 0. This corresponds to the position of the corresponding instance in the list in the variable ***gbus-interfaces***, that is, the instance for unit <n> can be located by (**nth** <n> ***gbus-interfaces***).

Enter a peripheral entry in the following manner:

Peripheral: *Pair:* UNIBUS *Set: Pair:* GBUS 0
Pair: NUMBER 0
Pair: ACCESS-BASE 140000 *Pair:* ACCESS-SIZE 2048

You assign unit numbers to the UNIBUS board via the Number attribute. When other peripheral entries refer to a UNIBUS number, it is the Number attribute of the namespace entry that refers to the UNIBUS board.

ACCESS-BASE is the octal address of the beginning of the mapped area in the UNIBUS address space, as set in the switches on the UNIBUS board. ACCESS-SIZE is the decimal size of the mapped area in words as set in the switches and must be one of 2048, 4096, 8192, or 16384.

If you want the Gbus software to automatically configure peripheral devices connected to the UNIBUS, you can have entries in the namespace entry for their hosts.

An example entry might be:

Peripheral *Pair:* PS-300 *Set: Pair:* UNIBUS 0

More information using the namespace to set up interface software can be found on page 46 in the section "Defining Peripherals."

PROGRAMMING THE GBUS

GBI Software

The file **sys:gbus;gbus.lisp** provides a software model of the GBI and the UNIBUS interface board to make writing software applications more convenient. You are welcome to use this software and to modify it as you see fit.

All Lisp symbols mentioned in this section are implicit in the package GBUS (which uses GLOBAL) unless qualified with a package name.

Flavor gbus-interface

An instance of the flavor **gbus-interface** is the model of a single GBI board. It supports a set of messages relevant to the operation of the GBI. The GBI control word is accessed, in whole or by individual fields, through messages to a **gbus-interface** instance. Bus cycles are done similarly.

A **gbus-interface** can have one or more **external-interfaces**, which are models of the various option boards. The MULTIBUS interface board has no software model; its function is purely electrical.

gbus-interfaces *Variable*

Contains a list of **gbus-interface** instances, one for each board in the machine, sorted by ascending slot number (slot 0 being the slot furthest from the data path and other CPU boards in the backplane). In multiboard systems, the proper way to select the correct instance is to add peripherals entries to the host's namespace entry and use the function **gbus:find-peripheral-model**. The position in this list of each instance corresponds to the *unit number* of the board it represents. See page 27 for information on using the namespace system.

:unit *Message to* **gbus-interface**

Returns a fixnum, the number of the unit in which the corresponding Gbus board resides.

:external-bus-interfaces *Message to* **gbus-interface**

Returns a list of several elements, either an instance of flavor **external-interface**, a dependent, or **nil**.

The software does not automatically notice external interfaces; they must be declared on the local host's Peripheral attribute in its namespace record. This procedure is described on page 45 in the section "Defining Peripherals." Note that a MULTIBUS interface board has no software model; it is only an electrical extension of the Gbus.

:buffer-memory-size *Message to* **gbus-interface**

Returns a fixnum, the count of 32-bit words in the dual-port high-speed RAM buffer. Double this number for the count of 16-bit words.

:buf-base *Message to* **gbus-interface**

Returns a fixnum, the virtual address of the beginning of buffer memory in Lisp Machine address space. This number can be used as the target of a displaced array. An array of type **art-16b** is recommended since buffer memory is accessed as 16-bit words by the Gbus.

:device-io-base *Message to* **gbus-interface**

Returns a fixnum, the base Gbus I/O space address of the GBI on-board I/O devices. This fixnum is dependent on the high-address bits in use on the board; it is therefore a message and not a constant.

:memory-read *memory-address* &optional *byte-mode-p* *Message to* **gbus-interface**

Reads from memory location *memory-address*. A single fixnum is returned.

:memory-write *memory-address data* &optional *byte-mode-p* *Message to* **gbus-interface**

Writes *data* to memory location *memory-address*. **nil** is returned.

:io-read *i/o-address* &optional *byte-mode-p* *Message to* **gbus-interface**

Reads from location *i/o-address*. A single fixnum is returned.

:io-write *i/o-address data* &optional *byte-mode-p* *Message to* **gbus-interface**

Writes *data* to location *i/o-address*. **nil** is returned.

:device-read *device-offset* &optional *byte-mode-p* *Message to* **gbus-interface**

Reads from the I/O device located at the address derived from adding the on-board I/O device base address to *device-offset*. This is equivalent to using the message **:io-read**, adding *device-offset* to the value returned by the message **:device-io-base**.

:device-write *device-offset data* &optional *byte-mode-p* *Message to* **gbus-interface**

Writes to the I/O device located at the address derived from adding the on-board I/O device base address to *device-offset*. This is equivalent to using the message **:io-write**, adding *device-offset* to the value returned by the message **:device-io-base**.

Bus Cycle Error Flavors

When a bus error occurs when performing any of the mentioned bus cycles, the condition is signaled by the following flavors:

gbus-cycle-error *Flavor*

Base flavor of the following five flavors, built on the flavor **gbus-error**, which is built on **error**.

no-transfer-acknowledge *Flavor*

Signaled when a slave device does not respond to the bus cycle within 2 milliseconds.

bus-conflict *Flavor*

Signaled when an interface board cannot get access to its external bus (such as when the UNIBUS is continuously busy). The software automatically retries such a transfer five times before signaling this error.

bus-grant-timeout *Flavor*

Signaled when some other bus master had control of the Gbus and the Lbus→Gbus port was not granted the bus within 2 milliseconds of requesting it.

external-bus-not-present *Flavor*

Signaled when a cycle is done on some external interface that is declared in the namespace but not actually attached. Discovery of an unattached external interface is made by detecting recursive bus errors while trying to signal an external bus error.

external-bus-error *Flavor*

Base flavor for errors involving cycles on external bus interfaces. The UNIBUS errors are based on this flavor and are documented in section on the UNIBUS software on page 39.

Control Word Messages

The following messages are supported for manipulating fields of the control word.

:enable-slaves *Message to* **gbus-interface**
:disable-slaves *Message to* **gbus-interface**

Enable/disable the on-board slave devices, including the 68000 program memory, serial I/O, and buffer memory.

:enable-timeouts *Message to* **gbus-interface**
:disable-timeouts *Message to* **gbus-interface**

Enable/disable the 68000 watchdog timer. When the watchdog timer is enabled, the 68000 must write to the Keep-Alive Register in I/O device space at least once every 8 seconds, or the watchdog timer resets the 68000. The board is initialized with timeouts disabled.

:reset-68000 *Message to* **gbus-interface**

Halts and resets the 68000. The Reset line is asserted only momentarily, then cleared, as it resets the serial I/O ports also. 68000 program execution, when started, begins by taking the reset vector at address 0.

Note: Be careful using **:reset-68000** when high-address bits are nonzero. (See the message **:enable-hi-addr** (page 34). The high-address bits apply to the 68000 program memory as well, that is, setting that field changes the address of 68000 program memory. The 68000 always takes the reset vector at address 0 no matter the setting of the high-address bits, while the program memory might have been moved. Resetting the 68000 then causes it to halt from bus errors.

:halt-68000 *Message to* **gbus-interface**
:start-68000 *Message to* **gbus-interface**

Halt/start the 68000. 68000 program execution continues where it left off. The board is initialized with the 68000 halted.

:reset-and-start-68000 *Message to* **gbus-interface**

Halts, resets, and starts the 68000. 68000 program execution, when started, begins by taking the reset vector at address 0.

:gbi-auto-cmd *Message to* **gbus-interface**
:gbi-manual-cmd *Message to* **gbus-interface**

Affect the way the Lbus→Gbus port handles bus cycles for the 3600. You do not need to use these messages unless the timing of bus cycles must be handled manually. The board is initialized to auto-cmd and must be in that state when the bus-cycle messages are used.

:ring-doorbell *Message to* **gbus-interface**

Asserts the 68000's Lbus interrupt request line. Interrupt requests must be enabled by the 68000 control register to actually interrupt the 68000. This bit is cleared only by the 68000.

:hog-gbus *Message to* **gbus-interface**
:dont-hog-gbus *Message to* **gbus-interface**

Assert/deassert Bus-Request at the Lbus→Gbus port, locking out other bus masters. Useful when doing many bus cycles on a busy bus.

:gbi-init *Message to* **gbus-interface**

Momentarily asserts Bus-INIT. Resets and disables/halts all devices including the 68000.

:enable-hi-addr *high-address-bits* *Message to* **gbus-interface**

Enables 24-bit addressing mode, setting the high-address bits to the 4-bit fixnum value supplied. 20-bit vs. 24-bit mode applies to on-board slave devices in memory space, although the on-board devices in I/O space are affected by high-address bits as well. See the section "Devices and Addresses" on page 7. Be careful when using the 68000 when the high-address bits are set to other than 0, since the reset vector might not come from the currently assigned 68000 program memory.

:ignore-hi-addr *Message to* **gbus-interface**

Changes to 20-bit addressing mode. This message also clears the high-address bits. See the section "Devices and Addresses" on page 7.

:turn-on *Message to* **gbus-interface**

Enables local arbitration, local devices, buffer memory, and halts the 68000.

:wait-for-interrupt *Message to* **gbus-interface**
&optional (*bit-mask* **#o17**) *timeout*

Waits for one of the Gbus→Lisp Machine interrupts to be asserted. Four different signals can be selected from the 15 interrupt signal sources via the Gbus map. The optional argument *bit-mask* allows testing of a subset of the four. *timeout* is given multiples of 1/60th of a second and defaults to no timeout. The method for this uses **process-wait**; therefore, the response time is likely to be longer than 1/60th of a second. The message **:wait-for-interrupt-hard** can be used when it is necessary to spend time in a tight loop awaiting the assertion of an interrupt line.

:wait-for-interrupt-hard *Message to* **gbus-interface**
&optional (*bit-mask* **#o17**) (*timeout-in-msec* **1000.**)

Waits in a tight loop with scheduling inhibited for one of the four Gbus→Lisp Machine interrupts to be asserted. **Note:** The timeout argument is now in milliseconds and cannot be **nil**.

construct-interrupt-map &rest *specs* *Macro*

Makes an interrupt-map array that can be loaded into the Gbus interrupt map using the **:install-interrupt-map** message. The arguments should all be lists of the form *source-name destination-name*. The space of names is as follows:

Sources	Destinations
:external-bus-<1 through 7>	:Lbus-<0 through 3>
:mpsc-<a or b>	:68000-<1 through 7>
:paddle-timer	
:pio	
:lisp-flag	
:doorbell	

Note: The Lbus interrupt requests signified in this mapping scheme consist of four independent integers representing the four possible Lbus interrupt requests. The seven 68000 interrupt request lines, however, are determined by reading the bit-weighted value of three separate interrupt request lines. When no lines are asserted, the value is zero. When one or more lines are asserted, the combination of interrupt requests makes up one of the integers 1 through 7. For more information, see page 51 in Appendix A--"UNIBUS Registers, Interrupts, and Priorities."

Furthermore, external bus interrupts are "priority-coded," as are the 68000 interrupt inputs. This means that asserting one 68000 interrupt might mean that another interrupt is blocked. In the case of simultaneous interrupt requests, the order in which arguments to this macro are specified is significant, and priority proceeds in ascending order. That is, the later specification(s) have priority over the earlier one(s). For example, given the following specifications in the following order:

```
(:pio :68000-7)

(:doorbell :68000-3)
```

the doorbell interrupt preempts the PIO interrupt even though the PIO interrupt has a numerically higher priority. The order in which they are declared is the deciding factor.

construct-interrupt-map is used as the initial value in a defconst or defvar form, as shown:

```
(defconst foo-interrupt-map  (construct-interrupt-map (:mpsc-a :68000-1)
                                                      (:mpsc-b :68000-1)))
```

Use the **:install-interrupt-map** message to actually load the map into the interface.

:install-interrupt-map *interrupt-map-array* *Message to* **gbus-interface**

Loads the supplied interrupt-map array into the Gbus interface. The previous contents of the interface's interrupt map are lost.

:set-interrupt-map *slot value* *Message to* **gbus-interface**

Changes the contents of one slot of the interface's interrupt map. This is a primitive interface and is included for compatibility only. New programs should use the facilities above.

Use of Buffer Memory

The dual-port high-speed RAM buffer provides fast communication between the two worlds. While not really "dual-ported" in the original sense, access to the memory is arbitrated between the two busses. It is both readable and writable from the Lbus and Gbus, and its cycle time is fast enough that even with both sides doing block transfers, wait states are kept to a minimum. When both sides request access simultaneously, the Gbus side has priority.

Each **gbus-interface** instance has a displaced array pointing to its buffer memory. This array is available by sending the **:buffer-memory-array** message. It is an **art-fixnum** array and is as long as the actual buffer memory in the board. This array can be used by itself, or it can be used as the host for other indirect and index-offset arrays.

In applications in which a DMA peripheral or external processor must communicate blocks of data with the 3600, the buffer memory can be used as the staging area. Blocks of several hundred words or a few Kwords can be set aside for transfers, and blocks of several words can be used as flags and simulated "control registers."

Since the array returned is an **art-fixnum**-type array, writing IEEE single-precision, floating-point numbers to the buffer memory is not possible using simple **aref** and **aset** calls. It is recommended that you either indirect a like-sized **art-q**-type array to it or use the subprimitive **%fixnum** to coerce the data type of the flonum to fixnum before using **aset**. Be careful, however, using an **art-q** pointing to buffer memory; objects that are neither fixnums nor single-floats will not be stored meaningfully because the buffer memory is only 32 bits and does not record the type field.

All data read from the buffer memory array using **aref** are fixnums, regardless of the type of array used to read them. The subprimitive **%flonum** can be used to coerce the data type in the cases where the data in the buffer memory are actually IEEE single-precision, floating-point numbers.

Buffer Memory Messages

The buffer memory messages are provided for using the buffer memory. They are not required to use the buffer memory, but might be useful when different activities are sharing the Gbus.

:buffer-memory-array *Message to* **gbus-interface**

Returns an **art-fixnum**-type array, displaced to the board's buffer memory.

:buffer-base-address *Message to* **gbus-interface**

Returns, as a fixnum, the Gbus address of the base of buffer memory. If the high-address bits are in use, they are included in the address returned. This gives the program the address needed by external bus DMA devices or the UNIBUS→Gbus map.

:allocate-buffer-memory *elements array-type* &key (*wait-p* **t**) *less-is-ok* *Message to* **gbus-interface**

Assigns a portion of the buffer memory for use. This message is to be used when concurrent processes are using the Gbus. *elements* should be the number of array elements to allocate, of the specified array type. *array-type* should be either an array-type symbol (such as **art-8b** or **art-string**) or an array (or string) object. Conversion from array elements to words is done internally. *wait-p* is **t** when the message waits for space to be available, or is **nil** when the message returns **nil** immediately if the requested space cannot be given. *less-is-ok* allows the largest available portion to be allocated, even if it is smaller than requested.

The three values returned are *buffer-index*, *gbus-address*, and *elements-allocated*. *buffer-index* is the index into the buffer memory if it were accessed through an array of type *array-type*. *gbus-address* is the Gbus byte address of the first element of the allocated area. *elements-allocated* is the number of elements allocated, which might not be the number requested if the message was passed with *less-is-ok* being other than **nil**.

:deallocate-buffer-memory *buffer-index array-type* *Message to* **gbus-interface**

Frees the allocated portion of buffer memory. An error is signaled when the specified portion was not allocated.

:copy-array-to-buffer *array start-index n-elements buffer-index* *Message to* **gbus-interface**

Copies the specified portion of *array* to the buffer memory, starting at *buffer-index*. *buffer-index* should be the first value returned from **:allocate-buffer-memory**.

:copy-buffer-to-array *array start-index n-elements buffer-index* *Message to* **gbus-interface**

Copies the specified portion of the buffer memory to *array*, starting at *buffer-index*. *buffer-index* should be the first value returned from **:allocate-buffer-memory**.

Flavor unibus-interface

The flavor **unibus-interface** models the UNIBUS interface board. Instances of this flavor can be found using **gbus:find-peripheral-model**, passing a device type of **:unibus** and 0 in the keyword argument *number*.

:read-register *offset* *Message to* **external-interface**

Reads the I/O-space register located at the interface's base address plus *offset*. This message and the following one are useful for manipulating the registers of the UNIBUS interface.

:write-register *offset data* *Message to* **external-interface**

Writes *data* to the I/O-space register located at the interface's base address plus *offset*.

:unibus-read *address* *Message to* **unibus-interface**

Performs a read cycle to the address specified. The read cycle is done via a **:memory-read**. **:unibus-read** is available in word-mode only. The result of the cycle is returned as a fixnum, or an error is signaled.

:unibus-write *address data* *Message to* **unibus-interface**

Performs a write cycle to the address specified. The write cycle is done via a **:memory-write**. **:unibus-write** is available in word-mode only and either returns **nil** or signals an error.

:unibus-interrupt *vector* *Message to* **unibus-interface**

Initiates an interrupt. An interrupt can be done only when the UNIBUS interface board is configured not to be the primary CPU. *vector* should be a fixnum in the range 100(8) to 476(8).

:interrupting-unibus-p *Message to* **unibus-interface**

Returns **t** if an interrupt has been initiated with the **:unibus-interrupt** message and has not yet been handled by the CPU. Returns **nil** otherwise.

:set-priority *level* *Message to* **unibus-interface**

Sets the interrupt and bus mastership priority of the interface board. Priority depends on whether the board is configured as the primary CPU. See page 51 for information about the UNIBUS Priority Register.

:priority *Message to* **unibus-interface**

Returns the current priority level.

:interrupt-vector *Message to* **unibus-interface**

Returns the vector for the most recent UNIBUS interrupt. This message is useful only when the UNIBUS interface board is configured as the primary CPU, since it does not field interrupts otherwise.

:interrupt-request-p *Message to* **unibus-interface**

Tells whether a device is requesting an interrupt. This message is useful only when the UNIBUS interface board is configured as the primary CPU.

:clear-interrupts &optional (*mask* **#o17**) *Message to* **unibus-interface**

Clears any interrupts that are currently being passed into the Gbus. The optional argument *mask* can be used to selectively clear the interrupts.

:enable-interrupts &optional (*mask* **1**) *Message to* **unibus-interface**

Allows the specified interrupt signals to be passed into the Gbus. The default *mask* is 1 since 1 corresponds to UNIBUS Interrupt, which is the interrupt most likely to be in use.

:disable-interrupts &optional (*mask* **1**) *Message to* **unibus-interface**

Disables the passing of UNIBUS interface board interrupts into the Gbus. The bits specified in *mask* are cleared in the Gbus Interrupt Enable/Disable Register.

The UNIBUS interface board does not have a separate **:wait-for-interrupts** method. It can generate an external bus priority 7 interrupt, which can be mapped by the

interrupt map into one of the Gbus→Lbus interrupt request lines. This can be tested much more efficiently than repeatedly checking the Gbus Interrupt Read/ Clear Register.

:write-map *UNIBUS-address Gbus-address* *Message to* **unibus-interface**
&optional *io-space-p byte-mode-p* (*byte-number* **0**)

Writes the proper slot in the UNIBUS→Gbus access map. The Gbus address can be in memory space or I/O space. Since there is no byte-mode read cycle on the UNIBUS and some Gbus registers require being addressed in byte mode, the map entry can specify byte-mode access and the low bit to be used. The 11 low-order bits of the two addresses must be the same. An error is signaled when the low-order bits are not the same, when the address is outside the mapped area of UNIBUS space, or when the mapped area was not specified in the namespace entry.

The map is composed of 16 registers, each one corresponding to 1 Kword (2 Kbytes). Only one use of this message is needed for each 1-Kword segment.

Note that a CPU on the UNIBUS can set the map itself, because it is mapped into the UNIBUS address space as well.

:read-map-register *register-number* *Message to* **unibus-interface**

Returns the 12-bit fixnum stored in the addressed map register. *register-number* should be a fixnum in the range 0 to 15. The low 9 bits of this fixnum are bits <19:12> of the Gbus address. The other bits are documented on page 56 in Appendix A--"UNIBUS Registers, Interrupts, and Priorities." Three other values are also returned:

1. **t** if the register maps to I/O space or **nil** if it maps to memory space;
2. **t** if the register specifies byte-mode accesses or **nil** if it specifies word-mode accesses;
3. the low-order bit used for byte-mode accesses.

The following conditions might be signaled while performing **:read-map-register**:

unibus-error *Flavor*

Base flavor for the following flavors. It is based on **external-bus-error**, which is based on **gbus-cycle-error**.

unibus-grant-timeout *Flavor*

The UNIBUS was continuously busy for 200 microseconds.

unibus-non-existent-memory *Flavor*

No slave responded within 6 microseconds.

unibus-parity-error *Flavor*

A parity error occurred reading from UNIBUS memory.

unibus-interrupt-still-pending *Flavor*

The **:unibus-interrupt** message was sent, but the last interrupt requested was still pending.

Serial Ports

:make-serial-stream &rest *keyword-args* — *Message to* **gbus-interface**

Makes a serial stream that connects to the serial hardware in the Gbus board. The same keyword arguments used in **si:make-serial-stream** apply.

Note: The **:unit** keyword value should be between 1 and 3, inclusive. Unit 1 corresponds to the EIA 5 serial connector, unit 2 corresponds to EIA 6, and unit 3 corresponds to EIA 7. Likewise, unit 1 corresponds to MPSC A, channel B; unit 2 corresponds to MPSC B, channel A; and unit 3 corresponds to MPSC B, channel B.

gbus:serial-binary-stream — *Flavor*
gbus:serial-character-stream — *Flavor*

Instances of these flavors control Gbus serial streams. They support the same messages as regular serial streams. They are not, however, handled by an external processor, the way the FEP handles the regular serial ports. Thus input from these streams is not as simple a thing, since characters might be missed. If an application cannot afford to have the 3600 spending much of its time polling the Gbus serial hardware, it is recommended that you use the 68000 processor to handle serial I/O.

:transmit-buffer-not-full — *Message to* **basic-gbus-serial-stream**

Returns **t** when the hardware for the port is ready to transmit another character or byte, and **nil** when it is not ready. This can be used in places where letting the **:tyo** method wait using **process-wait** takes too much time and the application needs more control of how it waits for the hardware to be ready. This message is supported in addition to the regular protocol for a serial stream.

Parallel Port

Since there is only one parallel port on a Gbus board, the **gbus-interface** instance handles its protocol directly.

:parallel-port-output *8-or-16-bit-value* — *Message to* **gbus-interface**
:parallel-port-set *8-or-16-bit-value* — *Message to* **gbus-interface**

Place the given value in the Parallel Port Output Register. The first message waits until any previous cycle has been acknowledged, as when a handshaking, stream-oriented device is connected. The second message just sets the register and returns.

Note: If the Gbus parallel port is configured as an 8-bit interface, only the lower-order 8 bits of the data value are relevant.

:parallel-port-input — *Message to* **gbus-interface**
:parallel-port-read — *Message to* **gbus-interface**

Read the data on the parallel port inputs. The first message waits until a data-ready signal appears, as when a handshaking, stream-oriented device is connected. The second message just reads the register. A 16-bit fixnum is returned.

If the parallel port is configured as an 8-bit interface, bits <0-7> of the value returned contain the data, and bits <14> and <15> contain the value of the Select and Paper Empty Status, respectively.

:parallel-port-listen — *Message to* **gbus-interface**

Returns **t** if the attached device is asserting data ready, **nil** if not.

:parallel-port-ready — *Message to* **gbus-interface**

Returns **t** when the hardware is ready to begin an output cycle, **nil** when a cycle is in progress.

:parallel-start-write — *Message to* **gbus-interface**

Forces **:parallel-port-ready** to return **t**. This message is to be used at the beginning of any handshaking output interaction when the state of the handshaking is unknown.

Defining Peripherals

By using some utilities provided in the namespace system, you can automatically configure device interfaces upon loading the application system and cold booting.

The peripherals field in a host's namespace entry allows you to declare peripheral devices on the host. The UNIBUS is declared in this way, as shown in the chapter "Installation" (page 27). The kind of connection between peripherals and the machine is declared by specifying an *interface type*. The interface type should be the first attribute declared for the peripheral.

The following interface types are significant to the Gbus user:

Type	Significance
UNIT	The peripheral uses some type of serial interface.
GBUS	The peripheral uses a Generic Bus Interface. (Select this interface type if your Gbus connects to a MULTIBUS.)
UNIBUS	The peripheral uses a UNIBUS interface board.

Other external interfaces can define their own interface types.

The kind of peripheral is declared by the device type, which is the first token in the peripheral entry. The following is an example of a peripheral entry in the namespace:

```
Peripheral: MODEM UNIT 1 MODEL CDS-224 PHONE-NUMBER 2134780681 AUTO-ANSWER Yes
```

Here, the device type is MODEM and the interface type is UNIT, which implies a serial interface. Other options are present as well.

The namespace system allows you to declare the recognized options to devices and their interface types, so that the system can parse the options (into numbers, for example) for the application. In the above example, the UNIT option is declared an integer by the UNIT interface type, and the AUTO-ANSWER option is declared to be boolean by the MODEM device type.

For more information, see the source code of the Gbus system, in uses of **neti:define-peripheral-interface-type** and **neti:define-peripheral-device-type**.

The Gbus system uses these utilities to automatically configure the Gbus system by the information on the local host's namespace entry, both when the Gbus system is loaded, and at cold-boot time if the Gbus system was previously saved in a world.

If you are programming Gbus applications, you can also use the namespace utilities and the Gbus system's hooks to initialize the state of your own software. This way, the Gbus software notes that your peripheral is declared on the local host and either calls a user-defined function or instantiates a user-defined flavor.

If you anticipate having more than one possible interface for a device, you should have setup functions or flavors for each interface type.

Each **gbus-interface** instance, when being initialized, looks over the host's peripherals for peripherals that use the Gbus interface type, with the unit number for each instance (counting from 0 at the leftmost). For each interface type, it looks for two properties on the device-type keyword symbol, either **:gbus-peripheral-setup** or **:gbus-peripheral-flavor**. These properties do not apply to UNIBUS devices; UNIBUS devices, of course, have their own interface type--UNIBUS. The names of the properties used for UNIBUS devices are **:unibus-peripheral-flavor** and **:unibus-peripheral-setup**.

On the one hand, if **gbus-interface** finds the keyword symbol **:gbus-peripheral-setup**, it expects the value of the property to be a function. **gbus-interface** then calls the function it finds, passing it two arguments:

- the value of **gbus-interface** itself
- the structure **neti:peripheral**

The software then expects the function called by **gbus-interface** to return either an instance that is a software model of the peripheral, or **nil**. If the returned value is **nil**, the instance is collected into the **gbus-interface**'s peripherals instance variable.

On the other hand, if **gbus-interface** finds the property **:gbus-peripheral-flavor**, **gbus-interface** calls **make-instance**, using the property's value as the flavor. **gbus-interface** also passes its own value in the INIT keyword argument **:interface**, followed by the rest of the peripheral's options as listed in the namespace entry. The flavor must handle all the INIT keywords presented, or the instance is not made (the error is trapped with **catch-error**). Use of **:init** methods (or **make-instance** methods in the new scheme) can smooth out differences between the flavor's instance variables a nd the namespace's entry option names.

The following example shows how to define the peripheral Slithy-Tove.

```
(neti:define-peripheral-device-type :slithy-tove
        (:gyre :boolean)
        (:gimbling-index :integer :range 0 9)
        (:frabjousity :integer))

(defprop :slithy-tove gbus-slithy-tove :gbus-peripheral-flavor)
```

Now when the software sees a Slithy-Tove peripheral attached to a Gbus interface, flavor **gbus-slithy-tove** is instantiated.

You could, instead of declaring that flavor, define an arbitrary function:

```
(defun (:property :slithy-tove :gbus-peripheral-setup) (interface peripheral)
  ...)
```

Note that a peripheral on the Gbus does not need to be a board on the external bus; for example, it can be something attached to one of the serial or parallel ports. A peripheral can be modeled only by a serial stream, if that is the logical thing.

Other Kinds of Interfaces

If you have your own kind of interface, you can also use **peripherals-mixin** to define the interface.

peripherals-mixin *Flavor*

Provides the instance-variable **peripherals** and the mechanism for using the name-space peripheral entry of the local host. The flavor containing this mixin must provide a method for **:interface-type**, returning the same keyword symbol as used in the **neti:define-peripheral-interface-type** definition for the interface type in question. For Gbus, the interface type is **:gbus**; for UNIBUS, the interface type is **:unibus**.

In an **:after :init** (or **make-instance :after**) method, the local host's peripherals are scanned. For each one with the correct interface type, the peripheral's device-type symbol is checked for the properties :<interface type name>-**peripheral-setup** and :<interface type name>-**peripheral-flavor**. For each one found, either the function is called or the flavor is instantiated.

The following steps demonstrate how to set up a software interface.

1. Declare the interface type to the namespace. For this exercise, assume the interface type is GPIB.

```
;;; For devices using the mythical GPIB interface:
(neti:define-peripheral-interface-type :gpib :gpib
   (:gpib :integer))                ;number of the GPIB in question
```

2. Declare the device type of the interface box itself, so the Gbus knows to initialize it. Also declare the flavor to make the interface model.

```
;;; For the (mythical) GPIB interface option to the Gbus.
(neti:define-peripheral-device-type :gpib)        ;no device options.

(defprop :gpib gbus-gpib-interface :gbus-peripheral-flavor)
```

3. Mix **peripherals-mixin** into the interface flavor, so that when the (mythical) GPIB interface is instantiated, any peripherals declared on the GPIB interface are set up.

```
(defflavor gbus-gpib-interface (interface)     ;a Gbus-interface
(peripherals-mixin))
```

4. Make a (mythical) device called an audio-oscillator and presume it lives on the GPIB. Make only a GPIB interface for it.

```
(defprop :audio-oscillator gpib-audio-oscillator-interface :gpib-peripheral-flavor)
```

Presuming that a flavor was called **gpib-audio-oscillator-interface** and that both the GPIB and audio-oscillator were on the local host's namespace entry, the software is set up accordingly.

This setup takes place when the Gbus system is loaded and when the world is cold-booted. The application software, since it is most likely to be loaded after the Gbus system, must decache the state of the Gbus system for new peripherals to be noticed.

init-gbus-interfaces &optional *forget-old-interfaces* *Function*

This function should be called, with argument *forget-old-interfaces* given **t**, after defining new peripheral types as described above. This function resets the state of the Gbus system, making new instances and initializing them. It is automatically done at cold-boot.

find-peripheral-model *device-type* *Function*
&rest *options* &key &allow-other-keys

Searches through the local host's peripheral entries and the known peripheral model instances to find the corresponding instance. If there is more than one device of *device-type*, more of the device's options can be specified in *options*. This returns the model of the first peripheral description found for the given specification.

APPENDIX A

UNIBUS Registers, Interrupts, and Priorities

The registers on the UNIBUS interface board are in Gbus I/O space.

UNIBUS Priority Register (I/O address 1000(8))

This 4-bit register performs two functions.

1. When the interface board is configured as the primary CPU (and hence is arbitrating the UNIBUS), this register selects the arbitrator priority. The following table explains the meaning of the contents of this register:

0000-0011	Any priority level can access the UNIBUS
0100	Only BR5 priority level and above can access the UNIBUS
0101	Only BR6 priority level and above can access the UNIBUS
0110	Only BR7 priority level and above can access the UNIBUS
0111	Only NPR can access the UNIBUS
1000	Only the Gbus can access the UNIBUS
1001-1111	No priority level

2. When the interface board is not configured as the primary CPU, this register selects the level of its interrupts to the UNIBUS. The following table explains the meaning of its contents:

0000-0011	No interrupts are executed; they just remain pending until cleared.
0100	BR4
0101	BR5
0110	BR6
0111	BR7
1000-1111	No interrupts are executed; they just remain pending until cleared.

This register is set to 0111 (decimal 7) upon power-up, a Gbus or UNIBUS initialization, or any UNIBUS interrupt if the UNIBUS is configured as the primary CPU (consistent with PDP-11 processors.)

INIT and Power Register (Write only) (I/O address 1002(8))

This register has three bits:

<D0>	Setting this bit activates the UNIBUS initialization line. The line remains activated until <D0> is cleared.
<D1>	Setting this bit activates the UNIBUS DC LO line. The line remains activated until <D1> is cleared.
<D2>	Setting this bit activates the UNIBUS AC LO line. The line remains activated until <D2> is cleared.

These bits are not readable. To find out whether the UNIBUS INIT, DC LO, or AC LO line is set, read the FEP Interrupt Read/Clear Register.

Gbus Interrupt Enable/Disable Register (I/O address 1004(8))

This 4-bit register enables and disables the four possible interrupts to the Gbus.

Note: Even if a particular interrupt is disabled, the appropriate bit in the Gbus Interrupt Read/Clear Register is set if the correct conditions for an interrupt occur. Clearing the interrupt via the Gbus Interrupt Read/Clear Register also clears the bit.

Setting the appropriate bit enables an interrupt; clearing the appropriate bit disables an interrupt. The bits are assigned as follows:

<D0>: Interrupt A	UNIBUS Interrupt	Gbus Priority 7
<D1>: Interrupt B	UNIBUS INIT	Gbus Priority 6
<D2>: Interrupt C	UNIBUS DC LO	Gbus Priority 3
<D3>: Interrupt D	UNIBUS AC LO	Gbus Priority 4

Interrupt A is the interrupt used for UNIBUS interrupts of the Gbus. It is activated only when the Gbus is the primary CPU.

Interrupt B is the interrupt used when you have activated the UNIBUS INIT line. As long as this interrupt is set (whether enabled or disabled), the Gbus receives a bus error when it tries to access the UNIBUS.

Interrupt C is the interrupt used when you have activated the UNIBUS DC LO line. As long as this interrupt is set (whether enabled or disabled), the Gbus receives a bus error when it tries to access the UNIBUS.

Interrupt D is the interrupt used when you have activated the UNIBUS AC LO line. Unlike the previous two interrupts, the Gbus can still access the UNIBUS when this interrupt is set.

All interrupts are disabled by a Gbus or UNIBUS initialization.

Gbus Interrupt Read/Clear Register (I/O address 1006(8))

Reading this register indicates what interrupts to the Gbus are set regardless of whether they are enabled or disabled. Writing a 1 to any bit of this register clears the corresponding interrupt when it is set and has no effect when it is clear. Writing a 0 to any bit of this register has no effect.

The bits are assigned similarly to the Gbus Interrupt Enable/Disable Register:

<D0>: Interrupt A	Gbus Interrupt Priority 7
<D1>: Interrupt B	Gbus Interrupt Priority 6
<D2>: Interrupt C	Gbus Interrupt Priority 3
<D3>: Interrupt D	Gbus Interrupt Priority 4

UNIBUS Interrupt Request/Vector Read Register (I/O address 1010(8))

This 16-bit register serves two purposes:

1. When the Gbus is fielding interrupts from the UNIBUS, this register contains the interrupt vector from the last UNIBUS interrupt. Writing to this register when the Gbus is the primary CPU does nothing.

2. When the Gbus is not the primary CPU, writing to this register initiates an interrupt to the UNIBUS. The value written to this register is the vector used for the interrupt. A bus error is generated and the write not executed if the Gbus already has an interrupt to the UNIBUS pending and it attempts another.

UNIBUS Interrupt Clear Register (Write only) (I/O address 1012(8))

A write of anything to this register clears any interrupt to the UNIBUS that the Gbus has pending.

Bus Error Read Register (Read only) (I/O address 1014(8))

This register performs two functions: it indicates the reasons the last Gbus bus error occurred and it indicates whether a Gbus interrupt to the UNIBUS has been serviced yet. The bits are assigned as follows:

<D0> This bit is set when the last bus error occurred because the Gbus was trying to make a UNIBUS access and the NPG did not arrive within 200 microseconds.

<D1> This bit is set when the last bus error occurred because the Gbus was either accessing the UNIBUS or performing an interrupt to the UNIBUS and SSYN was not received within 6 microseconds after MSYN or INTR were originally asserted.

<D2> This bit is set when the last bus error occurred because the Gbus was performing a read of the UNIBUS and the UNIBUS parity lines indicated an error.

<D3> This bit is set when the last bus error occurred because the Gbus tried to interrupt the UNIBUS and its last interrupt to the UNIBUS was still pending.

<D4> Not used.

<D5> This bit is set when a Gbus interrupt of the UNIBUS is pending.

Not all bus error conditions are indicated by the Bus Error Read Register. An INIT or DC LO condition is indicated by reading the Gbus Interrupt Read/Clear Register.

UNIBUS Map Registers (I/O address 1040(8)-1076(8))

These sixteen 12-bit registers are used to map UNIBUS addresses to Gbus addresses. This address map is also accessible by Gbus masters and masters on the UNIBUS.

Each 12-bit register supplies information for mapping a 2-Kbyte (11-bit) sector of UNIBUS address into a 2-Kbyte sector of Gbus address. If the total mapped UNIBUS address space selected by the UNIBUS select DIP switch is 4 Kbytes, only the first two map registers are significant. If the total UNIBUS address space selected by the DIP switch is 8 Kbytes, only the first 4 map registers are significant, etc.

Within each register the bits have the following significance:

<0-8> Indicate the 9 most significant Gbus address bits (<A19-A11>) of the 2-Kbyte sector to which that register corresponds. The low 11 bits of the address come directly from the UNIBUS master.

<9> When this bit is set, the corresponding sector maps to I/O space. When the bit is clear, the corresponding sector maps to memory space. Note that I/O space is only 16 bits, hence only bits <0-4> in the above-mentioned field are meaningful when this bit is set.

<10> When this bit is set, all accesses to the Gbus in this sector appear to the Gbus as 8-bit accesses. When this bit is clear, all accesses to the Gbus in this sector appear as either 8- or 16-bit accesses, whatever the UNIBUS access is.

Note: If this bit is set, the UNIBUS should be performing 16-bit accesses. Bit <10> is provided to help when a UNIBUS master accesses a Gbus or MULTIBUS slave that requires byte-mode access, since UNIBUS has no byte-read cycle.

<11> This bit is used only when bit <10> is set. Bit <11> supplies the value that is used for <A0> for an 8-bit access of the Gbus resulting when bit <D10> is set.

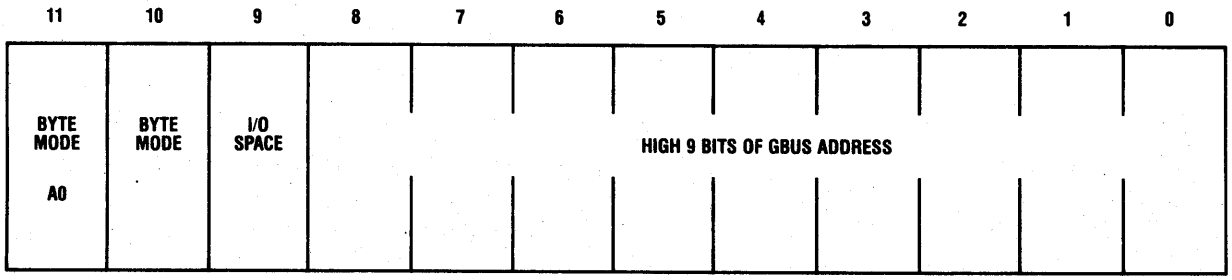

Figure A.1 *UNIBUS address map register*

The first UNIBUS map register is for the first 2K memory locations after the UNIBUS address offset selected by the UNIBUS Select DIP switch. The second UNIBUS map register is for the second 2K memory locations after the UNIBUS address offset selected by the DIP switch, etc.

APPENDIX B

Parallel and Serial Port Connectors

External devices that have serial or parallel interfaces can be connected to 3600-family machines through the Gbus interface paddle board. This appendix describes the serial and parallel ports that connect to peripheral devices that do not use either the MULTIBUS or UNIBUS interface option boards.

8-Bit Parallel Port Connectors

The 8-bit parallel port option, if included, connects to a 36-pin CHAMP-type connector on the 3600 bulkhead. Cabling for the most commonly used configuration, the Centronics interface, is given on the next page. If you use the Centronics-type interface, you must remove SIP resistor networks RN12, RN13, and RN14 from the paddle board and install SIP resistor network RN20.

Use Belden 9515 or an equivalent cable, that is, a shielded round cable with at least 12 twisted pairs, 24 AWG stranded.

Make no external connection to pins 14, 15, 19-27, 32, and 33 of the Gbus connector.

Table B.1 *Gbus-to-Centronics device pin connections*

Gbus (Bulkhead) AMP-Shielded CHAMP Screw-Lock Connector*		Centronics Device AMP-Shielded CHAMP Bail-Lock Connector*
Pin	**Signal**	**Pin**
1	Data Strobe (active LOW)	1
35	Return	19
2	Data 1	2
35	Return	20
3	Data 2	3
35	Return	21
4	Data 3	4
35	Return	22
5	Data 4	5
35	Return	23
6	Data 5	6
35	Return	24
7	Data 6	7
35	Return	25
8	Data 7	8
35	Return	26
9	Data 8	9
35	Return	27
10	Acknowledge (LOW)	10
35	Return	28

***AMP connector component part numbers:**

	GBI	Centronics Device
Housing	554257-1	554259-1
Shield	554255-1 & 554256-1	554254-1 & 554256-1
Strain relief	554263-1	554263-1
Cover	554264-1 & 554265-1	554264-1 & 554265-1
Inner ferrule	554266-6	554266-6
Screws	229996-4	------

Table B.1 *Gbus-to-Centronics device pin connections (continued)*

Gbus (Bulkhead)		Centronics Device
AMP Shielded CHAMP Screw-lock Connector*		AMP Shielded CHAMP Bail-lock Connector*
Pin	**Signal**	**Pin**
12	PE (paper empty = 1)	12
13	SLCT	13
11------+	RD REQ DIFF +	
16------|	ELP (Centronics = LOW)	
17------|	ELP ENBL L (Gbus printer interface = LOW)	
35------+	GND	
28------+	ACK DIFF -	
29------|	RD REQ DIFF -	
30------|	PE DIFF -	
31------|	SLCT DIFF -	
34------+	ELP TTL BIAS	

16-Bit Parallel Port Connectors

The 16-bit parallel interface is present at a pair of 40-pin ribbon cable connectors on the Gbus paddle board. (The connectors are labeled Parallel I/O In and Parallel I/O Out.) All signals are single-ended TTL lines. All input signals are terminated on the Gbus paddle board with a 220-ohm resistor to +5V and a 330-ohm resistor to GND.

Make sure all outputs are terminated in the same manner on the external device.

Cabling to these connectors is probably not supplied by Symbolics. You can, however, construct your own cables by following the pinouts in tables B.2 and B.3.

Note: If the 16-bit parallel port is used, the 8-bit parallel port cable must be removed from the LGP/Centronics connector on the Gbus paddle board.

Table B.2 *16-bit parallel-in cable pins*

Signal	Paddle Board Connector Pin	Direction (Relative to Gbus Board)
DATA IN 0	2	Input
DATA IN 1	4	Input
DATA IN 2	6	Input
DATA IN 3	8	Input
DATA IN 4	10	Input
DATA IN 5	12	Input
DATA IN 6	14	Input
DATA IN 7	16	Input
DATA IN 8	18	Input
DATA IN 9	20	Input
DATA IN 10	22	Input
DATA IN 11	24	Input
DATA IN 12	26	Input
DATA IN 13	28	Input
DATA IN 14	30	Input
DATA IN 15	32	Input
DATA IN ACCEPTED	34	Output
EXT WR L	36	Input
SPARE READ OUT	38	Output
SPARE READ IN	40	Input
GND	All odd-numbered pins	

Table B.3 *16-bit parallel-out cable pins*

Signal	Paddle Board Connector Pin	Direction (Relative to Gbus Board)
DATA OUT 0	2	Output
DATA OUT 1	4	Output
DATA OUT 2	6	Output
DATA OUT 3	8	Output
DATA OUT 4	10	Output
DATA OUT 5	12	Output
DATA OUT 6	14	Output
DATA OUT 7	16	Output
DATA OUT 8	18	Output
DATA OUT 9	20	Output
DATA OUT 10	22	Output
DATA OUT 11	24	Output
DATA OUT 12	26	Output
DATA OUT 13	28	Output
DATA OUT 14	30	Output
DATA OUT 15	32	Output
ACCEPT DATA OUT	34	Input
DATA OUT AVAIL	36	Output
SPARE WRITE IN	38	Input
SPARE WRITE OUT	40	Output
GND	All odd-numbered pins	

Serial Port Connectors

The Gbus serial port option, if included, connects to three 25-pin male DB25-type connectors on the 3600 bulkhead. These connectors are labeled EIA 5, EIA 6, and EIA 7. The EIA 5 connector corresponds to MPSC A, channel B; the EIA 6 connector corresponds to MPSC B, channel A; and the EIA 7 connector corresponds to MPSC B, channel B. Likewise, EIA 5 corresponds to unit 1, EIA 6 corresponds to unit 2, and EIA 7 corresponds to unit 3.

All signals use RS-232 levels.

The pinouts for these three connectors are as follows:

EIA-5	Pin	Signal Name	Direction (Relative to Gbus Board)
	1	GND	
	2	Serial TXD 1	Output
	3	Serial RXD 1	Input
	4	Serial RTS 1	Output
	7	GND	
	5	Serial CTS 1	Input
	20	Serial DTR 1	Output
	8	Serial CD 1	Input

EIA-6	Pin	Signal Name	Direction (Relative to Gbus Board)
	1	GND	
	2	Serial TXD 2	Output
	3	Serial RXD 2	Input
	4	Serial RTS 2	Output
	7	GND	
	5	Serial CTS 2	Input
	20	Serial DTR 2	Output
	8	Serial CD 2	Input
	15	Serial Ext TX Clk 2	Input
	17	Serial Ext TX Clk 2	Input

EIA-7	Pin	Signal Name	Direction (Relative to Gbus Board)
	1	GND	
	2	Serial TXD 3	Output
	3	Serial RXD 3	Input
	4	Serial RTS 3	Output
	7	GND	
	5	Serial CTS 3	Input
	20	Serial DTR 3	Output
	8	Serial CD 3	Input

APPENDIX C

Assembler for the Gbus MC68000 Processor

About this Appendix

This appendix describes how to construct, within the Lisp environment, segments of code that have meaning to the 68000 assembler program provided with the Gbus software. Invoked by executing the message **:assemble**, the 68000 assembler described in this appendix compiles the defined source code and loads the resulting object code into 68000 program memory in Gbus memory space.

This appendix does not purport to teach how to write assembly language programs for the 68000. For information about 68000 assembly language programming and the 68000 instruction set, see the *MC68000 User's Manual.*

About the Assembler

The assembler is provided for users of the Gbus 68000 processor. It accepts programs written in a list-structured assembly language based on the 68000 instruction set and loads the assembled programs directly into the processor's memory in Gbus address space. It can also generate listings into Zmacs editor buffers. It has no macro or cross-reference facilities. The source code for the assembler is distributed on the Gbus software tape in order for you to customize the assembler program to your own specifications.

68000 Assembly Language Program Structure

The assembly language accepted by the assembler is a list-structured language that looks similar to Lisp. Programs are defined using the special Lisp form **define-program**, which is described in greater detail on page 68.

Assembly language programs written for the optional 68000 on the Gbus are expected to have the following three types of elements, all of which are formatted as Lisp forms:

- **Pseudo-operations**, or pseudo-ops. Pseudo-ops take the form of lists (non-atomic forms). Pseudo-ops are instructions to the assembler and do not literally convert to object code, although they might control the format, type, or memory location of the object code. The assembler interprets pseudo-ops as having the structure *(function* &rest *argument(s))*, where *function* represents the mnemonic for the operation performed by the pseudo-op, and *argument(s)* represents one or more arguments or operands that can be passed to the pseudo-op.

- **Machine instructions**. Machine instructions also take the form of lists (non-atomic forms). The assembler converts them into 68000 object code. The assembler interprets machine instructions as having the structure *(function)* &rest *argument(s))*, where *function* represents the mnemonic for the operation performed by the instruction, and *argument(s)* represents one or more arguments or operands that can be passed to the instruction. Machine instruction operands can also take one of several addressing modes, which are described on page 69.

- **Labels**. Any atomic form contained in the Lisp special form **define-program** is interpreted by the assembler as a label.

The following example shows how the Lisp special form **define-program** is used to provide source code for the assembler:

```
(define-program foo ()
  (org 40)                    ;a pseudo-op
  (alloc junk)                ;another pseudo-op
  start                       ;a label
  (move  /# 3  + junk)        ;a machine instruction with two operands
  )
```

In this example, the 68000 program "foo" is defined; it uses one label, one instruction, and two pseudo-operations.

Addressing Modes

In the 68000, all instructions that take operands have an associated *addressing mode*, which directs how the operand's location is computed. Instructions can take either one or two operands. In this assembler, each operand is expressed by two elements, which take the following form:

1. the addressing mode, and

2. an *expression* that evaluates to a numeric value.

The following symbols are used to express addressing modes:

Mode Symbol	Mode Name	Traditional Form
D	Data register direct	D5
A	Address register direct	A5
/#	Immediate	#foo
@#	Absolute (short address)	@#foo
@@#	Absolute (long address)	@#foo (assembler deciding which)
+	PC relative	foo
+X	PC relative, indexed	foo(D7)
@	Address register indirect	@A7 or (A7)
@++	Indirect postincremented	(A7)+
--@	Indirect predecremented	-(A7)
@+	Indirect displaced	A7(foo)
@+X	Indirect indexed	A7(foo,D5)

Examples of instructions that take one operand:

```
(stop /# 44)                ;Stops immediate 44.
(addq # 1 + foo)            ;Increments (relative) FOO.
(subq # 1 --@ 7)            ;Decrements A7 then decrements the word
                              ;it points to.
```

Examples of instructions that take two operands:

```
(add + foo d 7)             ;Adds (relative) FOO to D7
(add d 3 @++ 2)             ;Adds D3 to the word to which A2 points,
                              ;then increments A2.
```

An expression is interpreted differently for each mode:

Mode	Expects
D, A, @, @++, --@	A register number (interpreted as a data or address register depending on the mode). A warning is given if the value of the expression is not suitable as a register number (that is, in the range 0 to 7).
+, /#, @#, @@#	A datum or address.

The remaining three modes expect a list with more information:

+X	A list: *(expression (D-or-A reg-number)* &optional *WORD-or-LONG)*
@+	A list: *(address-reg-number expression)*
@+X	A list: *(address-reg-number expression (D-or-A reg-number)* &optional *WORD-or-LONG)*

Register numbers are actually treated as expressions and evaluated.

For **+X** and **@+X**, *D-or-A* means either the symbol **D** or the symbol **A**, denoting which kind of register the index should come from. *WORD-or-LONG* means either the symbol **WORD** or the symbol **LONG**, depending on the size of indexing desired (the default is **WORD**).

The following are examples of the latter three cases:

```
(incf +X foo (d 3))                ;Increments the word at FOO+(D3)
(move @+x (7 0 (d 3)))             ;Increments the word at (A7)+0+(D3)
(move d 4 @+ (0 foo-slot))         ;Moves the value in D4 to the word
                                     ;(A0)+foo-slot (presumably a slot offset
                                     ;in some structure).
```

The relative addressing modes (+ and +X) always treat the effective address as absolute; therefore, subtract the current PC value from it. Be careful. This method of addressing works correctly for Branch and Jump, as well as for referring to variables using relative mode (for example, . . .+ **foo-variable**. . . gets the value of that variable), but it does not work to say . . .+ **3**. . . when you mean "3(PC)". You need to say . . .+ **(+ 3 /.)** . . ., that is, you need to "add the PC back in" so it comes out correctly. This applies to relative-indexed mode **(+X)** as well.

Expressions

Expressions are evaluated much like Lisp expressions, except that any symbols (aside from the names of functions) are interned in the assembler's symbol table. It therefore works to use an expression like **(+ some-label 32)**. Expressions can be nested; however, some special forms do not work, because the expression-parsing function only knows about lists that are function forms (for example, **let** does not work because it has clauses that are lists but not function forms).

One symbol is reserved. That symbol is **/.** (a period, but it must be "quoted" with the slash), which always references the current location. It is updated whenever the current location counter is changed. It does not work to set it, because the assembler always sets it back. Use **org** to reference the current location.

Machine Instructions

All the instruction names listed in the *MC68000 User's Manual* are supported in all the appropriate length modes (byte, word, and long). Length modes are specified by appending the letter for the mode to the instruction name (for example, **MOVE** in byte mode becomes **MOVEB**).

Some instructions take one operand and therefore expect a form with three elements (the instruction name [or *op-code*], address mode, and expression). Others take two operands and therefore expect a form with five elements (the instruction name, followed by two sets each of address mode and expression). The remaining instructions (save one) take no operands; therefore, the form should just be a list containing the op-code. Refer to the *MC68000 User's Manual* for the list of instructions and usage.

The one exception to this rule is **MOVEM**. It works as follows:

(MOVEM(L) *mode expression IN-or-OUT*
&optional *list-of-data-registers list-of-address-registers*)

Moving to or from the Status Register, Condition Code Register, or User Stack Pointer is accomplished by using the special symbols **.SR.**, **.CCR.**, and **.USP.**, respectively.

mode and *expression* express the place to put or get the data, *IN-or-OUT* specifies the direction of the transfer (into or out of the processor), and the lists of registers specify which ones are to be saved or restored.

Pseudo-ops

Pseudo-ops control the assembler itself and are not part of the 68000 program. They are listed here:

org *expression*

Sets the current location to the value of expression.

define *symbol expression*

Permanently defines *symbol* to have the value of *expression*. In essence it makes the symbol a label with an arbitrary value.

assign *symbol expression*

Temporarily defines *symbol* to have the value of *expression*. The value of *symbol* can be changed at any time. **assign** is the equivalent of the Lisp function setq.

alloc *symbol* &optional (*size* **1**) (*units* **words**)

Defines *symbol* to be the current location and leaves *size units* of room. In essence, it makes a variable called *symbol*. *units* should be one of the symbols **bytes**, **words**, or **longs**. The size is rounded to an integral number of words.

bytes &rest *byte-values*

Deposits the byte values, rearranged for 68000 byte order preference. A trailing 0 can be added to make an integral number of words.

word &rest *word-values*

Deposits the word values.

long &rest *long-values*

Deposits the long values.

char-string *string*

Deposits the character string, rearranged for 68000 byte order preference. A trailing 0 can be added to make an integral number of words.

char-string-and-length *string*

Deposits the character string as above, with the length of the string (word value) first.

vector *vector-number address*

Places the value of *address* (long value) at the specified *vector-number* (specifically at address 4**vector-number*). A warning is given if *vector-number* is not valid as a vector number (in the range 0 to 255).

Labels

Labels are pseudo-ops in a way. They permanently define the symbol that represents the value of the current location. Any attempt to redefine or reassign the symbol is an error.

Defining Programs

The special form **define-program** defines an assembler program, rather like defconst defines variables. It is in the ASSEMBLER package and so must be used with the qualifier **asm:**.

Note: Symbols in the program itself (except for imported ones) are not affected by the package they are in, since they are re-interned in other packages anyway.

asm:define-program *name options* &rest *forms*

name is declared special and set to a flavor instance which contains the *forms*. instance can later be told to assemble the program. *options* is a list of keyword-value pairs that can contain the following:

:import-symbols &rest *symbols*

Declares the specified symbols to be imported and not be re-interned when used in the program. Be careful; if one is used as a label or otherwise assigned, its value is changed. Example: **:import-symbols sys:page-size**. Imported symbols are dependent on the package they are in.

Once the program is defined, you can assemble it and load it into the Gbus processor by using the **:assemble** message.

:assemble &key *(destination* **:ignore***)* *(listing* **nil***)* *Message to* **assembler-program**

The argument *destination* can be **:gbus** to load the program into the processor. When the argument *listing* is not **nil**, the assembler generates a listing in an editor buffer (which is called "Assembler program <name> listing").

APPENDIX D

Symbolics 4-Slot MULTIBUS Card Cage and Backplane

This appendix contains information about the P1 and P2 connectors located in the Symbolics-manufactured 4-slot MULTIBUS card cage and backplane that is installed inside the 3600 cabinet.

P1 Connectors

Table D.1 lists the signals present on the P1 (top) connector of each slot. Note that not all signals are used by the Symbolics MULTIBUS interface board.

Table D.1 *Pin assignments on MULTIBUS backplane P1 connectors*

Component Side		Circuit Side	
Pin	Signal	Pin	Signal
1	GND	2	GND
3	+5V	4	+5V
5	+5V	6	+5V
7	+12V	8	+12V
9	-5V	10	-5V
11	GND	12	GND
13	BCLK/	14	INIT/
15	BPRN/	16	BPRO/
17	BUSY/	18	BREQ/
19	MRDC/	20	MWTC/
21	IORC/	22	IOWC/
23	XACK/	24	INH1/
25	LOCK/	26	INH2/
27	BHEN/	28	AD10/
29	CBRQ/	30	AD11/
31	CCLK/	32	AD12/
33	INTA/	34	AD13/

Table D.1 *Pin assignments on MULTIBUS backplane P1 connectors continued*

35	INT6/	36	INT7/
37	INT4/	38	INT5/
39	INT2/	40	INT3/
41	INT0/	42	INT1/
43	ADRE/	44	ADRF/
45	ADRC/	46	ADRD/
47	ADRA/	48	ADRB/
49	ADR8/	50	ADR9/
51	ADR6/	52	ADR7/
53	ADR4/	54	ADR5/
55	ADR2/	56	ADR3/
57	ADR0/	58	ADR1/
59	DATE/	60	DATF/
61	DATC/	62	DATD/
63	DATA/	64	DATB/
65	DAT8/	66	DAT9/
67	DAT6/	68	DAT7/
69	DAT4/	70	DAT5/
71	DAT2/	72	DAT3/
73	DAT0/	74	DAT1/
75	GND	76	GND
77	(Bussed)	78	(Bussed)
79	-12V	80	-12V
81	+5V	82	+5V
83	+5V	84	+5V
85	GND	86	GND

P2 Connectors (24-bit addressing memory)

The P2 (bottom) connector of each slot is a wirewrap-type connector, with no factory-installed wiring. You can add wiring to suit your application.

To accommodate 24 bits of MULTIBUS memory addressing, the following four pins in each slot must be bussed across the backplane:

Pin 55 (ADR16/)
Pin 56 (ADR17/)
Pin 57 (ADR14/)
Pin 58 (ADR15/)

Note: The pin numbers stamped on the P2 connector housings might not match the actual MULTIBUS P2 numbering scheme. Viewed from the rear (wiring side) of the backplane, pin 1 is on the top right of the P2 connector and pin 60 is on the bottom left.

INDEX

C

D

E

F

G

R

S

T

U