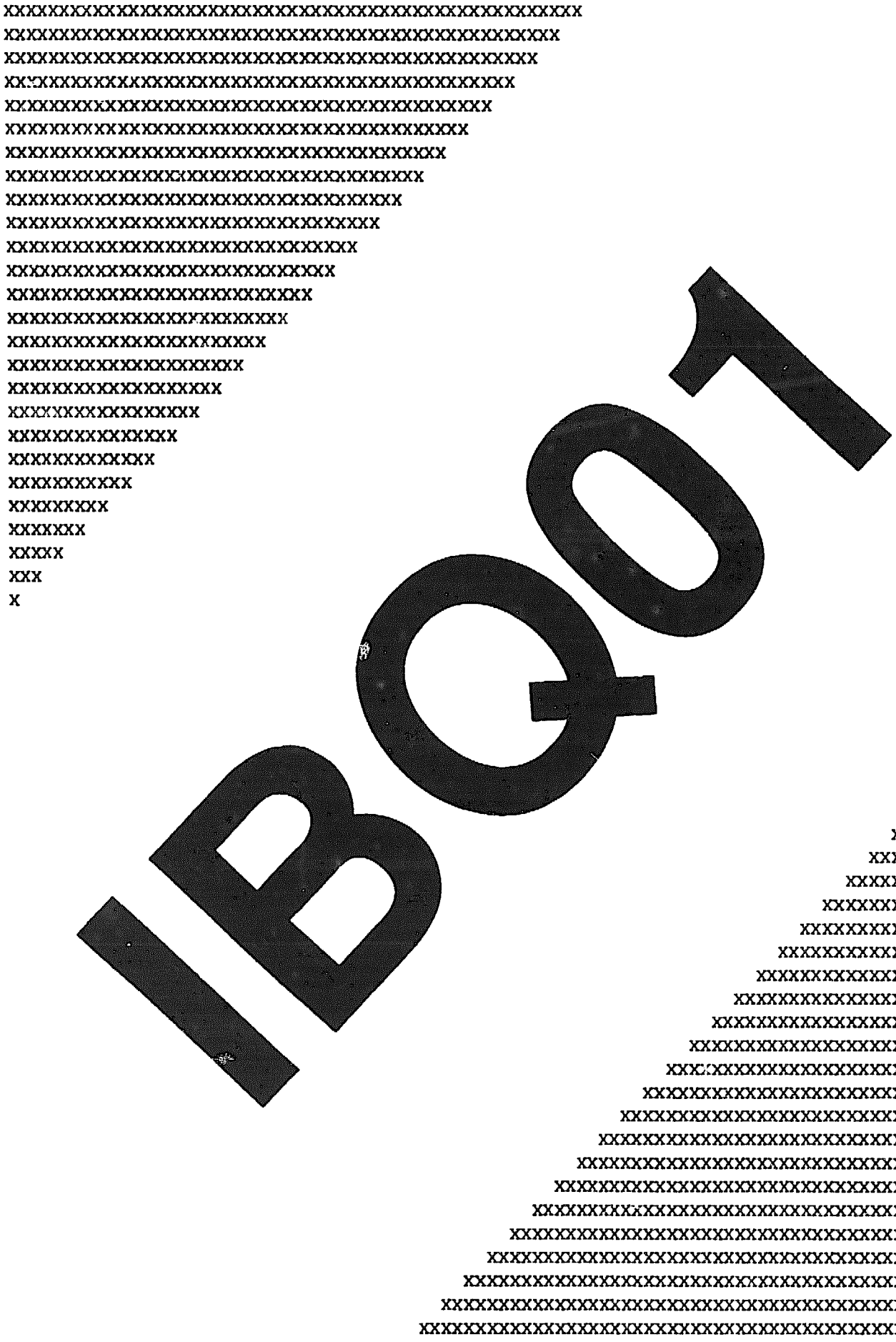

IBQ01

EK-IBQ01-UG-002

# IBQ01
# BITBUS Controller

## User's Guide

Prepared by
Computer Special Systems

2nd Edition, March, 1988



The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

Printed in U.S.A.

The software described in this document is furnished under a license and may be used or copied with the terms of such license. Book production was done by Educational Services Development and Publishing in Merrimack, N.H.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Digital Equipment Corporation.

The following are trademarks of Digital Equipment Corporation:

digital™
DATATRIEVE
DEC
DECmate
DECset
DECsystem-10
DECSYSTEM-20
DECUS
DECwriter
DIBOL
MASSBUS
PDP
P/OS
Professional
Rainbow
RSTS
RSX
Scholar
ULTRIX
UNIBUS
VAX
VMS
VT
Work Processor

# Table of Contents

## Figures

## Tables

PAGE x INTENTIONALLY LEFT BLANK

# Preface

This manual is written to support the IBQ01 BITBUS Controller programming and operating personnel. It assumes that the programmer is familiar with the entries from the following list which relate to his application:

- VAX/VMS operating system
- QIO System Services
- VAXELN
- Intel™ 8044 BITBUS enhanced microcontroller
- Intel DCX51 Distributed Control Executive operating system
- BITBUS Remote Access and Control (RAC) interface.

This manual contains five chapters, a glossary, and an index:

- CHAPTER 1, OVERVIEW — Briefly describes the IBQ family of equipment, software, and utilities.
- CHAPTER 2, BITBUS CONTROLLER — Describes the BITBUS controller, the BITBUS interface, and the RAC commands.
- CHAPTER 3, CROSSASSEMBLER — Contains a detailed discussion of the 8044 Crossassembler.

---

Intel™ is a trademark of Intel Corporation.

- CHAPTER 4, VMS SOFTWARE — Contains a detailed discussion of the operator to BITBUS Control Service (BCS) utility interface and the VMS program interface. The related driver calls and the related interface tools are described.
- CHAPTER 5, VAXELN SOFTWARE — Contains a detailed discussion of the VAXELN driver calls.
- GLOSSARY
- INDEX

## RELATED DOCUMENTS

In this manual you will find it helpful to refer to the following related manuals. They can help you use and understand the IBQ01 BITBUS Controller.

*IBQ01 BITBUS Controller Software Installation Manual* (AA-JQ52A-TN)

*IBQ01 BITBUS Controller Hardware Installation Manual* (EK-IBQ01-IN)

*IBQ01 BITBUS Controller Technical Manual* (EK-IBQ01-TM)

*Distributed Control Modules Handbook* (Intel 230973-001)

*Intel Embedded Microcontroller Data Book* (210918-005)

*VAX/VMS System Services Reference Manual* (AA-Z501B-TE)

*IEEE BITBUS Specification* (IEEE P1118)

*VAXELN Release notes*

*VAXELN Installation Manual*

*Introduction to VAXELN*

*VAXELN Host System Guide*

*VAXELN Run-Time Facilities Guide*

*VAXELN Application Design Guide*

*VAXELN PASCAL Language Reference Manual, Parts 1 and 2*

*VAX Language Sensitive Editor VAXELN PASCAL Guide*

*VAX/VMS DCL Dictionary*

*VAX/VMS Run-Time Library Routines Reference Manual*

*VAX Architecture Handbook*

*VAX Hardware Handbook*

*LSI-11 Analog System User's Guide*

## CONVENTIONS

This section describes the special symbols used in this manual.

| | |
|---|---|
| ... | Horizontal or vertical ellipses in text means that the information not directly related to the description has been omitted. |
| [info] | The information inside the square bracket is optional in the command. The command will complete without it. |
| COMMAND | The verb portion of all commands are shown in upper case. |
| param | The parameters in the BCS command set are shown in lower case. |
| PARAM | The parameters in the QIO commands are shown in upper case. |

## NOTES, CAUTIONS, AND WARNINGS

Notes, cautions, and warnings used in this manual are defined as follows:

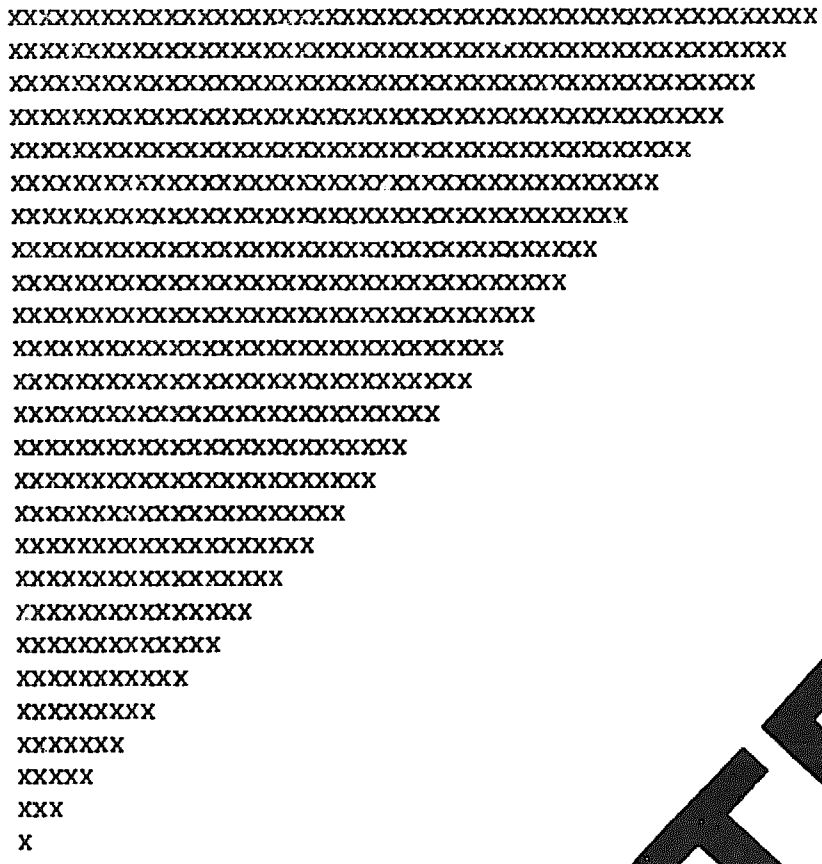
| | |
|---|---|
| NOTE | The information is important to the understanding of the process being described. |
| CAUTION | The information describes a process that can damage the equipment or software. |
| WARNING | The information describes a process that can harm the user. |

Corrections and suggestions for improving this publication are welcome.

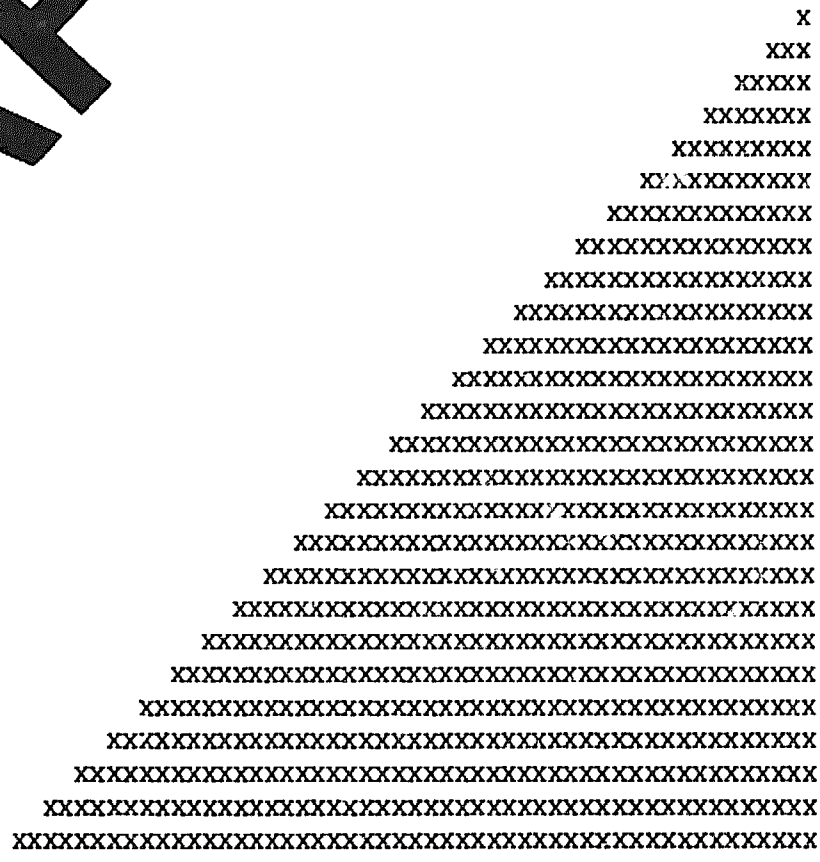
## FCC USER STATEMENT

**NOTICE**

This equipment generates, uses, and may emit radio frequency energy. The equipment has been type tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC rules, which are designed to provide reasonable protection against such radio frequency interference. Operation of this equipment in a residential area may cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.



# CHAPTER 1



# CHAPTER 1
# OVERVIEW

## INTRODUCTION

This chapter contains an overview of the IBQ01 BITBUS family of equipment. It includes a brief description of the software and the utilities.

## BITBUS CONTROLLER

The IBQ01 BITBUS Controller is the hardware, firmware, and software link between a MicroVAX II and the serial control BITBUS. The IBQ01 BITBUS Controller hardware consists of the BITBUS Controller board, the connector panel, and the interconnecting cable. The firmware is resident on the BITBUS Controller board. The software includes items in the host processor.

The IBQ01 BITBUS Controller board firmware includes:

- Diagnostics for power-up tests and self-test
- Message handling
- Initialization of the BITBUS network
- Maintenance of the BITBUS network

The host processor software includes the following.

VMS Software

- Device driver
- User's program
- BITBUS Control Service (BCS) configuration utility for VMS applications
- Task download utility

ELN Software

- System unique to the application. This system was built on a development system which includes the VAXELN Toolkit and the IBQ VAXELN driver. This system may then be downloaded to a target machine, which acts as host to the IBQ01 BITBUS controller board.
- VMS utility to facilitate task downloading under ELN.

The IBQ01 BITBUS Controller is the master device that supports computer integrated manufacturing in whatever form your application defines as part of a node controller (slave device) or any system requiring BITBUS communications. BITBUS communications include:

- Message passing between the host processor and all configured nodes.
- Automatic polling of specified nodes.
- Asynchronous reporting of network events to user applications.

## PROGRAM INTERFACE

The Program Interfaces are the IBQ01 Drivers for VAX/VMS and VAXELN. The format of the driver calls for these device drivers are described in Chapter 4 for VMS applications, and Chapter 5 for VAX ELN applications.

## BITBUS

Each host MicroVAX II can support up to eight IBQ01 BITBUS Controllers (processor dependent), and each IBQ01 can support up to 250 slave devices, depending on speed selection to take full advantage of the IBQ01 functionality each slave device must have an Intel remote access and control compatible controller.

## USER'S PROGRAM

Control of the operation of the slave nodes on the BITBUS is based on your program in conjunction with the configuration file.

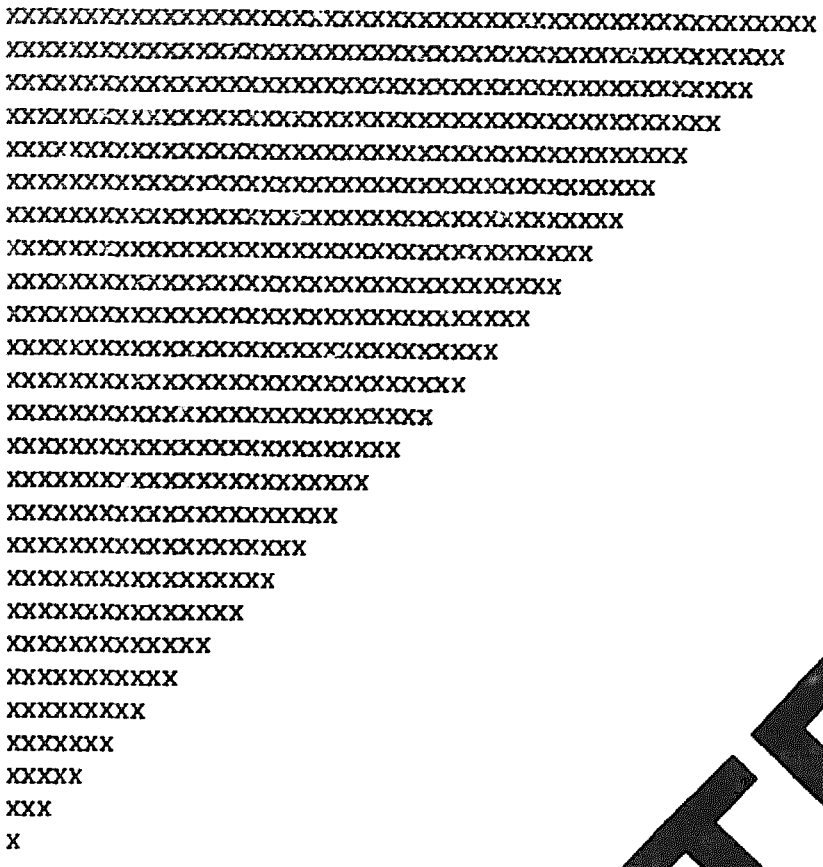
Your program can be written in any of the VMS supported languages by using the INCLUDE command or as an environment file in the case of PASCAL. Also, you can define any language with the SYSLIB command. The VMS supported languages include the following.

| Language | Include File |
|---|---|
| BASIC | IBQ$FUNC.BAS |
| C | IBQ$FUNC.H |
| FORTRAN | IBQ$FUNC.FOR |
| Macro | IBQ$FUNC.MAR |
| | IBQ$DEF.MAR |
| PASCAL | IBQ$FUNC.PAS (environment module) |
| | IBQ$FUNC.PEN (environment file) |

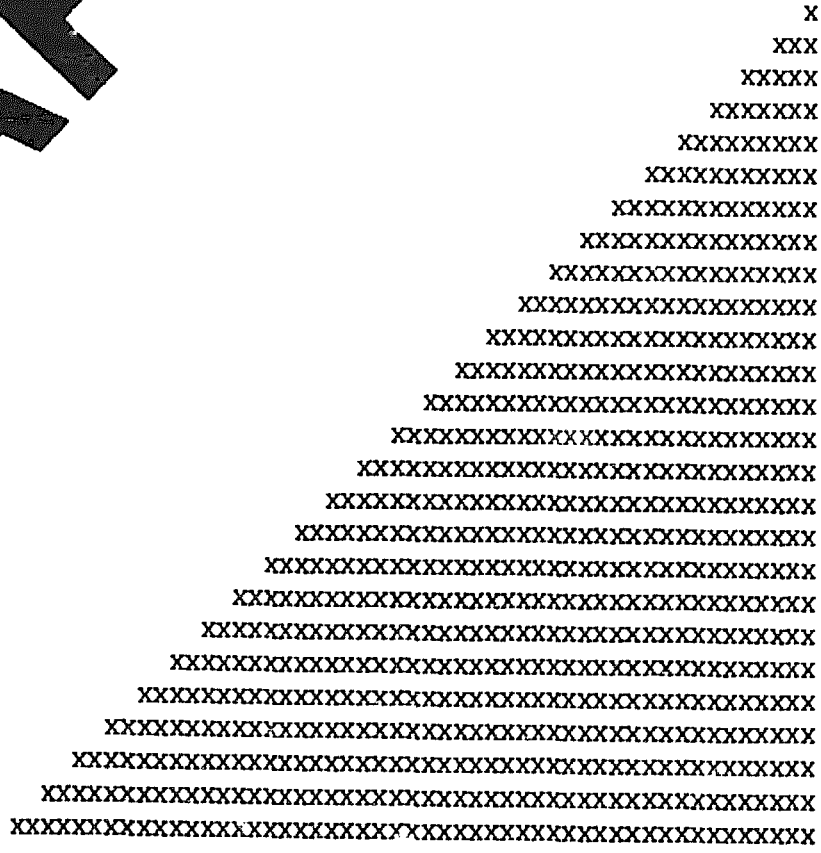
## INTERACTIVE INTERFACE

The Interactive Interface is a BITBUS Control Service (BCS) command line facility. It is available when using the VAX/VMS driver only. You can select any one of the VMS editors to generate a configuration file. You can use the BCS command line facility to generate and modify the configuration file. With either, you can define the BITBUS nodes (devices/positions connected to the serial BITBUS line). Also, with the command line facility you can interact with the Program Interface.

PAGE 4 INTENTIONALLY LEFT BLANK



# CHAPTER 2



# CHAPTER 2
# BITBUS CONTROLLER

## INTRODUCTION

This chapter contains a description of the BITBUS Controller, the BITBUS interface, and the RAC command set.

## IBQ01 BITBUS CONTROLLER (M3125)

The IBQ01 BITBUS Controller contains a supervisor microprocessor and a communications microcontroller. Under the control of the microprocessor, the BITBUS controller has a full functionality Q-bus interface to the MicroVAX II. The microcontroller is in control of the BITBUS multidrop line and initiates all communications activity.

## BITBUS CONTROLLER CONTROLS

The following sections describe *how* the IBQ01 BITBUS Controller controls the intelligent slave nodes through message packets. From your instructions through the driver commands, the IBQ01 BITBUS Controller forms BITBUS messages.

**NOTE**

**This is an overview of the BITBUS Controller. You must reference the applicable manuals on slave nodes you have in your BITBUS.**

## BITBUS INTERFACE

The interface between the IBQ01 BITBUS Controller and the BITBUS provides access to the slave nodes through message-passing utilities in the operating system of the 8051 CPU. This is iDCX, the Intel operating system for the 8051 CPU, which is a part of the Intel 8044 microcontroller. The Remote Access and Control (RAC) interface for the BITBUS interconnect defines a set of high level commands and responses to perform general purpose operations at a slave node.

See the Intel *Distributed Control Modules Databook* (23097-001) and *Distributed Control Modules* (146312-001) for details on BITBUS, iDCX, and RAC.

## MESSAGE AND DATA LINK PROTOCOLS

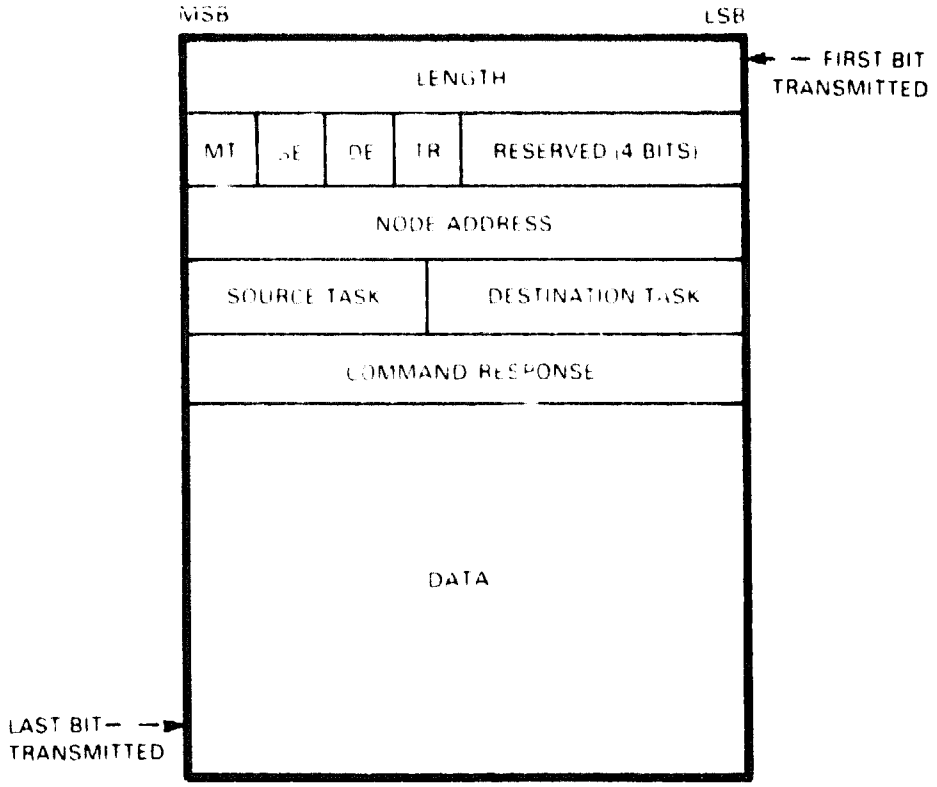
Protocol in the BITBUS message provides a task-to-task message interface between a master node (IBQ01 BITBUS Controller) and multiple slave nodes using an order/reply structure. The master node issues orders to the slave nodes which respond with replies. Every order on the BITBUS requires a reply.

This structure is built on top of the data link protocol using information frames for the message transfer. The data link protocol is a subset of the IBM™ Synchronous Data Link Control standard. The BITBUS data link protocol connects a master device to multiple slave devices in a multidrop topology.

# REMOTE ACCESS AND CONTROL (RAC) COMMANDS

The BITBUS Controller controls the remote nodes through RAC I/O and memory commands. All slave nodes must be configured with Intel-compatible software. This task must reside at task 0 on the node. If the remote node does not support the RAC commands, an error message is returned when certain RAC commands are addressed to the node.

The RAC interface is built on top of the standard message protocol. The message format is shown below.




CS-5642

## Message Format

All fields in a message, except the data field, are required in all messages. A description of the fields in a message follows:

| | |
|---|---|
| LENGTH | This eight-bit field specifies the total length of the message and can be between seven and 255 bytes. This is equal to the total number of data bytes plus seven. |
| MESSAGE TYPE (MT) | This bit describes the message as an order (0) or a reply (1). All message from the IBQ01 BITBUS Controller are orders, and all messages from slave nodes are replies. |
| SOURCE EXTENSION (SE) | This bit indicates the source of the order or the destination of the reply. It is set to 0 for the IBQ01 BITBUS Controller and it is set to 1 for the IBQ01 BITBUS Controller extension. |
| DESTINATION EXTENSION (DE) | This bit indicates the destination of an order or the source of a reply. It is set to 0 for slave, and it is set to 1 for a slave extension. |
| TRACK (TR) | This bit is used to provide message control at a master or slave which may be required by some implementations. It is set to 0 when sending a message, and it is set to 1 when receiving a message from the IBQ01 BITBUS Controller. |
| RESERVED | These four bits are reserved and are cleared when sending a message. |
| NODE ADDRESS | This eight-bit field specifies the destination node for orders and the source node for replies. Valid entries are 1 through 250. |
| SOURCE TASK | This four-bit field identifies the task that has generated an order or is to receive a reply. |
| DESTINATION TASK | This four-bit field identifies the task that is to receive an order or has generated a reply. |
| COMMAND/RESPONSE | This eight-bit field is used by both the user tasks and the message protocol. The message protocol uses the field for reporting errors. |
| DATA | This field can be a maximum of 13 bytes, and it is the only optional field in the message. |

## RAC Control Commands

The RAC control commands are:

| | |
|---|---|
| CREATE TASK | Initializes and begins execution of a task at a slave node. It assumes that the task to be created already exists in the slave node memory. |
| DELETE TASK | Stops a particular task from running in the slave node. The task number associated with the deleted task can be used again. |
| GET FUNCTION IDs | Causes the slave node to respond with a list of function ID codes for the tasks currently in existence on the node. |
| RAC PROTECT | Suspends or resumes RAC functions at a slave node. When suspended, the RAC functions only recognize the remote control commands. |
| RESET | Returns the slave node to its original (after power-up) state. |

## RAC Access Commands

In the RAC access commands there are three command groups:

1. Memory commands — These commands (MEMORY UPLOAD and MEMORY DOWNLOAD) control the flow of data between the master node and a slave node.
2. I/O commands — These commands (READ I/O, WRITE I/O, UPDATE I/O, OR I/O, AND I/O, and XOR I/O) allow the master node to access up to 256 I/O ports on each slave device.
3. Status commands — These commands (READ STATUS and WRITE STATUS) allow the master node to access up to 256 bytes of information on each slave device.

The RAC access commands are:

| | |
|---|---|
| XREAD I/O | Causes the slave node to read the specified I/O ports. |
| XWRITE I/O | Causes the slave node to write to the specified I/O ports. |
| XUPDATE I/O | Causes the slave note to write the data byte fields to the specified I/O ports and then reread the ports. |
| OR I/O | Causes the slave node to read the specified I/O port. and OR the data with the contents of the data byte field, write the data back to the I/O port. and reread the I/O port. |
| AND I/O | Causes the slave node to read the specified I/O port and AND the data with the contents of the data byte field. write the data back to the I/O port, and reread the I/O port. |
| XOR I/O | Causes the slave node to read the specified I/O port and XOR the data with the contents of the data byte field, write the data back to the I/O port, and reread the I/O port. |
| READ INTERNAL MEMORY | Causes a slave node to read to the specified memory locations. |
| WRITE INTERNAL MEMORY | Causes a slave mode to write to the specified memory locations. |
| DOWNLOAD EXTERNAL MEMORY | Write the data starting at the external slave memory. |
| UPLOAD EXTERNAL MEMORY | Read the data starting at the external slave memory location. |

# TROUBLESHOOTING

For problems encountered with the BITBUS Controller, follow the procedures contained in the *IBQ01 BITBUS Controller Technical Manual* (EK-IBQ01-TM).

## BITBUS Errors

A list of errors in hexadecimal returned by the BITBUS in IOSB[mode reply] follow.

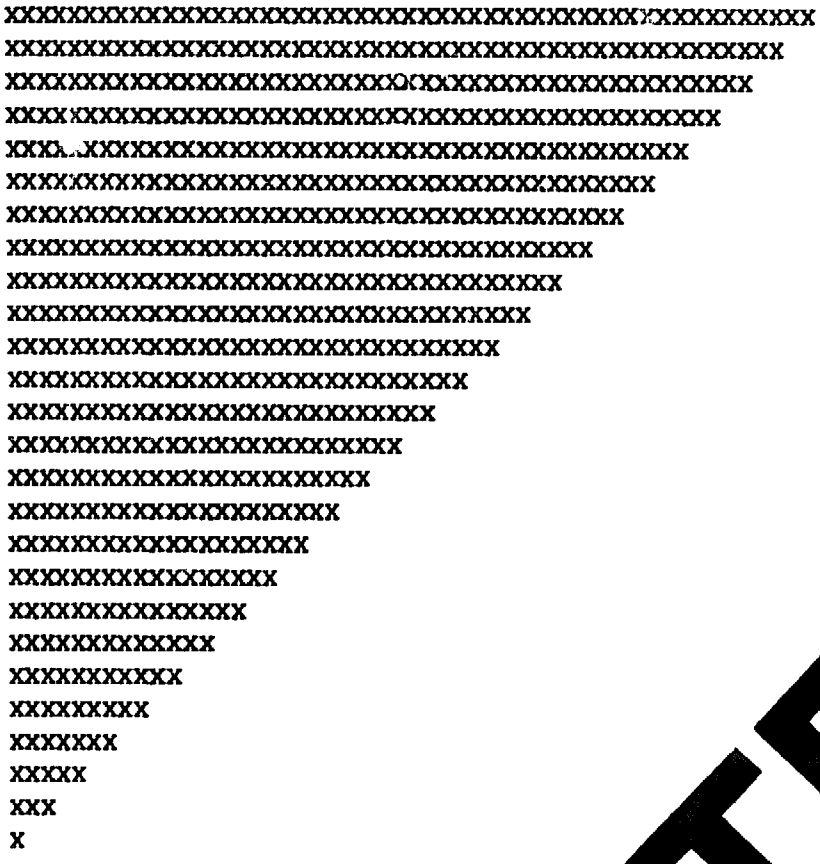
| | |
|---|---|
| 0 | No error |
| 80 | No destination task |
| 81 | Task overflow |
| 82 | Register bank overflow |
| 83 | Duplicate function ID |
| 84 | No buffers |
| 91 | Protocol error |
| 93 | No destination device |
| 95 | RAC protected |
| 96 | Unknown RAC command |
| 97 to FF | Reserved |

## IBQ01 BITBUS Controller Errors

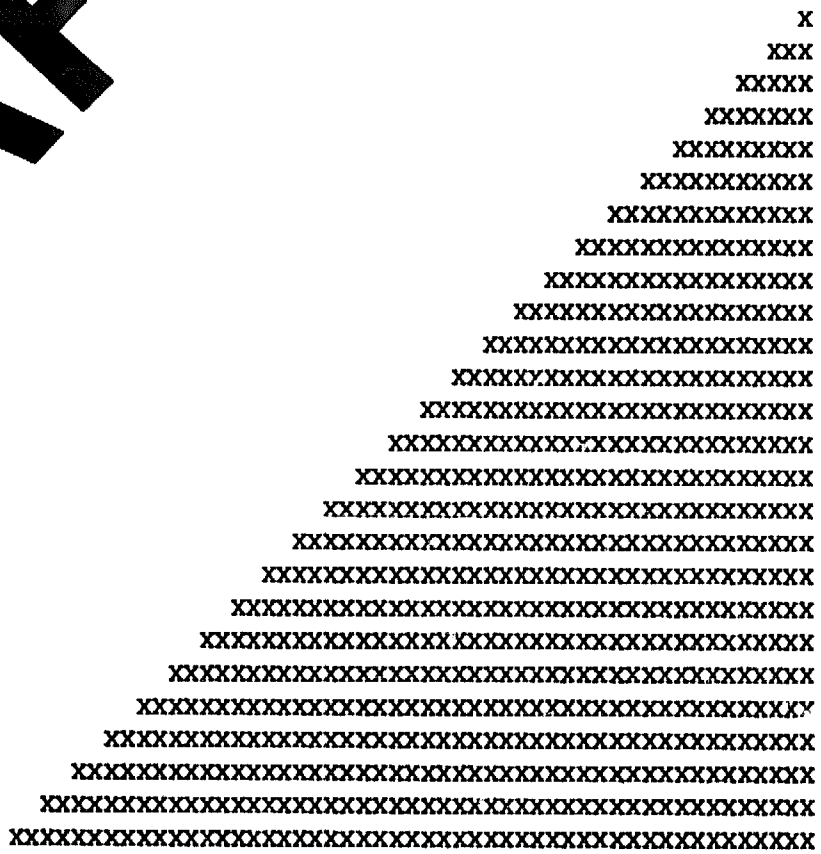
The following errors are returned by the IBQ01 BITBUS Controller in IOSB[IBQ01 code].

| | |
|---|---|
| 0 | No error |
| 1 | Invalid command |
| 2 | Node offline |
| 3 | DMA failure |
| 4 | Command failed to complete |
| 5 | Data buffers |
| 6 | Transmission failure |

PAGE 12 INTENTIONALLY LEFT BLANK



# CHAPTER 3



# CHAPTER 3
# CROSSASSEMBLER AND DOWNLOAD UTILITIES

## ASSEMBLER ASM44 OVERVIEW

The DATEM dDCM844 absolute assembler (ASM44) for the Intel 8044 microcontroller provides a tool for the generation of RMX51 application code. The ASM44 supports the needs of computer specialists.

## OPERATING ENVIRONMENT

ASM44 may be used with the download utilities (IBQ$LD44V and IBQ$IBQ44E) supplied by DEC™. This combination lets you generate application code, which in turn may be installed on BITBUS slave nodes. A review of the operating environment of such application software follows.

Slave nodes have a local 8044 microcontroller for network communications and local processing. Each slave node has the Intel RMX51 real-time operating system, a Remote Access and Control (RAC) task, and one or more software tasks. Although a slave node can support remote access to the I/O functions through command/response messages, it is often desirable to have software at the slave nodes provide application specific processing.

Slave node software operates under the control of the RMX51 operating system. This control program provides facilities which automatically schedule application tasks, preprocess asynchronous interrupts, provide intertask message support, and allows you to redefine the operating characteristics in realtime. Table 3-1 shows the available system commands. Entry points for these functions are in the file RMX51I.EXT, which is included in this release.

DEC™ is a trademark of Digital Equipment Corporation.

Table 3-1 RMX51 System Commands

| | |
|---|---|
| RQCREATETASK | - Create a new task |
| RQDELETETASK | - Delete an existing task |
| RQALLOCATE | - Allocate message buffer |
| RQDEALLOCATE | - Deallocate message buffer |
| RQDISABLEINTERRUPT | - Disable interrupt source |
| RQENABLEINTERRUPT | - Enable interrupt source |
| RQGETFUNCTIONIDS | - Return list of active tasks |
| RQSENDMESSAGE | - Send a message |
| RQWAIT | - Wait for message, interrupt or timeout |
| RQSETINTERVAL | - Set system clock interval |

You can use the RMX51 system calls to provide additional capability at slave nodes by generating new RMX51 tasks. These tasks are written in assembly language, and the purpose of the ASM44 and the download utilities is to aid in the generation of these tasks.

## ASSEMBLY LANGUAGE PROGRAMMING

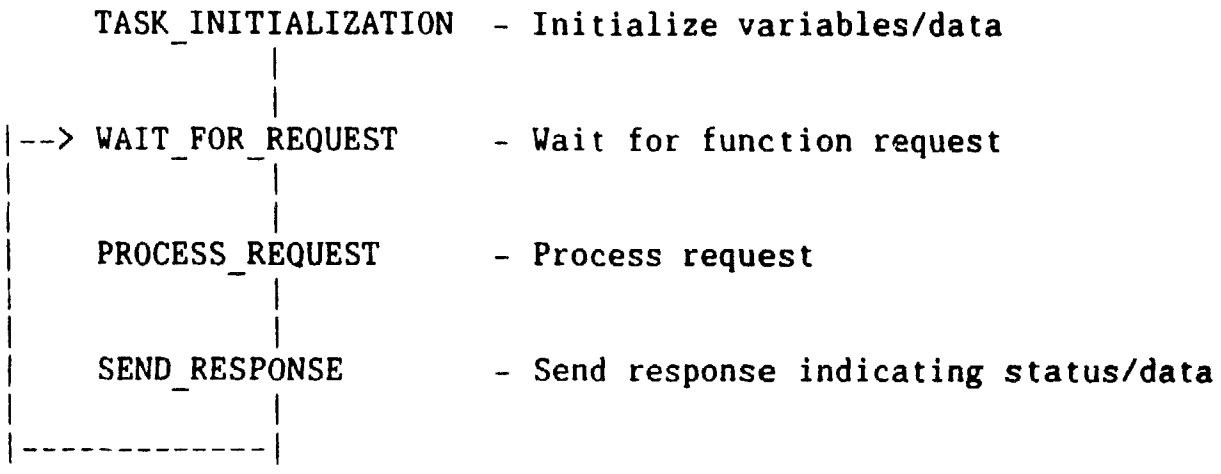
The 8044 microcontroller executes instructions fetched from a program memory. Each instruction directs the machine to either fetch or store a data variable, modify a variable, or transfer control to another string of machine instructions. Normally, instructions are executed sequentially from the program memory. The 8044 is an 8-bit processor, the data manipulation instructions operate on bytes (8-bit quantities).

With ASM44 you specify the instruction sequence using a set of mnemonics which are then translated into bit patterns. After you create a source file, you instruct the assembler to translate this source file into an object file for loading into the processor ready for execution.

ASM44 uses a hexadecimal file format to store the machine instructions which the download utilities software may then transfer across the BITBUS to the slave node processor. In addition, you may specify that a list file be generated which contains the mnemonics, the resulting object code, and any erroneous instructions.

# RMX PROGRAMMING ENVIRONMENT

ASM44 supports programming RMX51 application tasks, which usually have the following structure.




In general, each task should be fully standalone, each task should be self-contained, and code should not be shared by tasks.

# 8044 ARCHITECTURE

The 8044 microcontroller architecture is optimized for control applications.

## Memory Addresses

The processor recognizes five address spaces.

1- CODE (0-64K)

2- Directly and indirectly addressable on-board RAM (0-127)

3- External DATA (0-64K), indirect addressing only

4- Bit addressable space (0-255)

5- Special Function Registers (128-255)

Also, the processor supports special instructions, which allow reading the code space to fetch constants. There is no instruction that allows writing to the code space.

External data memory may only be accessed indirectly through either the DPTR register or a combination of the P2 port, and either R0 or R1. All such transfers must use the A register as the second operand.

The bit-addressable address space overlaps the on-board RAM memory from location 20H to 3FH (0-127) and the selected special function register bits. When using bit addressing, the CARRY flag (C) is treated as a bit accumulator.

## General Purpose Registers

There are four banks of general purpose registers, with each bank containing eight registers. Registers R0 and R1 may be used to indirectly access either on-board memory or external data-memory. Two bits within the Program Status Word (PSW) select the active register bank. The registers are mapped to the on-board data memory locations 0-1FH.

## Stack

The stack pointer points to an on-board stack area (the last location used) which, in turn, must map to the on-board data memory. Typically, the stack is used to save intermediate results (PUSH and POP) as well as return addresses during subroutine CALLS.

## Symbolically Addressable Hardware Registers

All special function registers may be accessed through predefined assembler names. Table 3-2 lists these registers.

**Table 3-2 Special Function Registers**

| Symbol | Data Address | Function |
|---|---|---|
| ACC | E0H | ACCUMULATOR |
| B | F0H | MULTIPLICATION REGISTER |
| DPH | 83H | DATA POINTER (HIGH) |
| DPL | 82H | DATA POINTER (LOW) |
| IE * | A8H | INTERRUPT ENABLE REGISTER |
| IP * | B8H | INTERRUPT PRIORITY |
| NSNR * | D8H | XMIT/RECV COUNT |
| P0 * | 80H | I/O PORT 0 |
| P1 | 90H | I/O PORT 1 |
| P2 | A0H | I/O PORT 2 |
| P3 * | B0H | I/O PORT 3 |
| PSW | D0H | PROGRAM STATUS WORD |
| RFL * | CDH | RECEIVE FIELD LENGTH |
| RBS * | CCH | RECEIVE BUFFER START |
| RBL * | CBH | RECEIVE BUFFER LENGTH |
| RCB * | CAH | RECEIVE CONTROL BYTE |
| SBUF * | 99H | SERIAL BUFFER |
| SIUST * | D9H | SIU STATUS BYTE |
| SMD * | C9H | SERIAL MODE BYTE |
| SP * | 81H | STACK POINTER |
| STD * | CEH | STATION ADDRESS |
| STS * | C8H | STATUS BYTE |
| TBS * | DCH | TRANSMIT BUFFER START |
| TBL * | DBH | TRANSMIT BUFFER LENGTH |
| TCB * | DAH | TRANSMIT CONTROL BYTE |
| TCON * | 88H | TIMER CONTROL |

**NOTE**

**Registers marked with * are maintained by RMX51 and should never be modified by the user.**

Table 3-2 Special Function Registers (Cont)

| Symbol | Data Address | Function |
|---|---|---|
| TH0 | * 8CH | TIMER 0 HIGH BYTE |
| TH1 | * 8DH | TIMER 1 HIGH BYTE |
| TL0 | * 8AH | TIMER 0 LOW BYTE |
| TL1 | * 8BH | TIMER 1 LOW BYTE |
| TMOD | * 89H | TIMER MODE |

**NOTE**

**Registers marked with * are maintained by RMX51 and should never be modified by the user.**

## Function Flag Bit Location and Symbols

| | | |
|---|---|---|
| Carry Flag (C REG) | CY | PSW.7 |
| Auxiliary Carry | AC | PSW.6 |
| User Flag | F0 | PSW.5 |
| Register Select 1 | RS1 | PSW.4 |
| Register Select 0 | RS0 | PSW.3 |
| Overflow | OV | PSW.2 |
| Reserved | | PSW.1 |
| Parity Flag | $\overline{P}$ | PSW.0 |

## Bit Addressing

Extensive bit control instructions are available on the 8044 microcontroller. Any of the bit addressable registers, including the bits within the ACC and PSW registers, may be directly accessed using the format ACC.x (where x indicates the selected bit). Bit instructions use the C register as a bit accumulator. A full range of bit-test transfer-control instructions are available.

# RMX51 FLASH PROGRAM EXAMPLE

```
NAME LED_FLASH

;The following program has the structure of an RMX51
;task responsible for implementing a LED FLASH function.
;The function accepts 2 commands as shown below:
;
;       COMMAND         RESULT
        BYTE
;
;       0               Turn off the LED flash function
;
;       1               Turn on the LED flash function
;

;While the LED flash function is enabled, it will cause the RED
;indicator LED to flash at a 250 msec rate.
;
;The function will return an e_ok (0) response if the command is
;accepted or an e_invalid (011H) if the command is invalid.
;

include rmx51i.ext
wait_msg        equ     01          ;wait for message
e_ok            equ     0           ;command byte ok
e_invalid       equ     011h        ;invalid command byte
clr_cmd         equ     0           ;clears (disables) flash function
enb_cmd         equ     1           ;enables flash function

;The following bits are defined within the 8344 bit space

                bseg at 049h

led_flag:       dbit    1           ;flag tracks status of LED
enb_flag:       dbit    1           ;set to 1 if flash enabled
ledlit          equ     P1.0        ;bit address of led
```

```
;The program is ORG'D at the beginning of SOCKET 3 of a DATEM
dDCM900

                cseg at 4000h

task_entry:                                     ;enter here when task starts
        clr     led_flag                        ;led initially off
        setb    enb_flag                        ;but enabled for flashing
        clr     ledlit                          ;turn off the LED

comm_fnc:                                       ;forms main program loop
        mov     a,#wait_msg                     ;wait for a message
        mov     b,#0feh                         ;wait for 250 msec
        lcall   rqwait                          ;go wait for it
        jb      acc.0,comm_0                    ;jmp if message received
        acall   flash_fnc                       ;go flash LED if enabled
        ajmp    comm_fnc                        ;then wait for next event

comm_0:
        mov     a,r7                            ;get buffer (assume onchip)
        add     a,#6                            ;point to command
        mov     r0,a                            ;set pointer
        cjne    @r0,#clr_cmd,comm_1             ;jump if not clear command
        acall   clr_fnc                         ;clear the flash enable function
        ajmp    send_rsp                        ;go send response, all okay

comm_1:
        cjne    @r0,#enb_cmd,comm_2             ;jump if not enable command
        acall   enb_fnc                         ;enable the flash function data
        ajmp    send_rsp                        ;and exit

comm_2:
        mov     b,#e_invalid                    ;invalid command
```

```
send_rsp:
            mov     a,r7            ;get message pointer
            mov     r0,a            ;set up pointer register
            inc     r0
            inc     r0              ;adjust to point to length field
            mov     @r0,#7          ;force length to 7 bytes
            inc     r0              ;point to flag byte
            mov     a,@r0           ;get flag byte
            orl     a,#80h          ;set to response message
            mov     @r0,a           ;store it in message
            inc     r0              ;skip node address field
            inc     r0              ;skip task id field
            inc     r0              ;point to response field
            mov     @r0,b           ;save response byte
            mov     dpl,r7          ;set up dptr with message pointer
            mov     dph,#0
            lcall   rqsendmessage   ;go send response message
            ajmp    comm_fnc        ;and go wait for next...
clr_fnc:
            clr     enb_flag        ;clear flash enable flag
            mov     b,#e_ok         ;return e_ok status
            ret
enb_fnc:
            setb    enb_flag        ;enable flash flag
            mov     b,#e_ok         ;return e_ok status
            ret
flash_fnc:
            jnb     enb_flag,flash_1    ;exit if not enabled
            jnb     led_flag,led_on     ;if led off, turn it on
            clr     ledlit              ;else turn it off
            clr     led_flag
            sjmp    flash_1             ;go back and wait
```

```
led_on:
            setb    ledlit          ;turn led on
            setb    led_flag

flash_1:
            ret                     ;exit
            org 5ff0h

itd1:                               ;allow for autoload
            dw 0AA55h               ;flags
            dw itd1-task_entry      ;entry point
            db 6                    ;stack size
            db 3                    ;FUNCTION ID
            db 03h                  ;register = don't care, priority 3
            dw 0                    ;no interrupt processing allowed
            dw itd1-itd1+1          ;point to invalid ITD

end         itd1                    ;indicate itd address
```

# ASM44 INSTRUCTIONS

The following sections describe the instruction set of the 8044 microcontroller. For further information on the 8044 instructions and hardware functions, consult the manufacturer's reference material.

# INSTRUCTION SUMMARY

The 8044 instruction set is provided as shown below, in alphabetical order. This table contains the mnemonic, the function, and a description of operation.

| Mnemonic | Function | Description |
|---|---|---|
| ACALL | code_add | -save address of next instruction on stack and transfer control to specified address. Target must be INBLOCK. |
| ADD | A,#data | -add immediate data |
| | A,@Rr | -add indirect |
| | A,Rr | -add register |
| | A,data_add | -add to direct address |
| ADDC | A,#data | -add with carry immediate data |
| | A,@Rr | -add with carry indirect |
| | A,Rr | -add with carry register |
| | A,data_add | -add with carry to direct address |
| AJMP | code_add | -transfer to INBLOCK target address |
| ANL | A,#data | -bitwise AND A with immediate data |
| | A,@Rr | -bitwise AND A with indirect address |
| | A,Rr | -bitwise AND A with register contents |
| | A,data_add- | -bitwise AND A with direct address |
| | data_add,#data | -bitwise AND direct address immediate |
| | data_add,A | -bitwise AND direct address with A |
| | C,bit_add | -bitwise AND C with bit address |
| | C,/bit_add | -bitwise AND C with complement of bit |
| CJNE | #Rr,#data,code_add | -compare/jmp not equal |
| | A,#data,code_add | -target must be relative (+/- 127) |
| | A,data_add,code_add | |
| CLR | A | -clear A register |
| | C | -clear C bit |
| | bit_add | -clear contents of bit address |

| Mnemonic | Function | Description |
|---|---|---|
| CPL | A | -1's complement of A register |
| | C | -complement C bit |
| | bit_add | -complement contents of bit address |
| DA | A | -decimal adjust A register |
| DEC | @Rr | -decrement contents of memory |
| | A | -decrement contents of accumulator |
| | Rr | -decrement contents of register |
| | data_add | -decrement direct address byte |
| DIV | AB | -divide A/B |
| DJNZ | Rr,code_add | -decrement, jmp non zero |
| | data_add,code_add | |
| INC | Rr | -increment register r |
| | A | -increment accumulator |
| | DPTR | -increment 16-bit DPTR register |
| | @Rr | -increment indirect location |
| | data_add | -increment direct address byte |
| JB | bit_add,code_add | -jmp if bit is set (relative) |
| JBC | bit_add,code_add | -jmp if bit and clear it |
| JC | code_add | -jmp if C bit set (relative) |
| JMP | @A+DPTR | -jmp through table address |
| JNB | bit_add,code_add | -jmp if bit not set (relative) |
| JNC | code_add | -jmp if C bit cleared (relative) |
| JNZ | code_add | -jmp if Z flag cleared (relative) |
| JZ | code_add | -jmp if Z flag set (relative) |
| LCALL | code_add | -save address of next instruction on stack and transfer control to specified address. |
| LJMP | code_add | -transfer to code address |

| Mnemonic | Function | Description |
|---|---|---|
| MOV | @Rr,#data | -move immediate to indirect address |
| | @Rr,A | -move accumulator to indirect address |
| | @Rr,data_add | -move direct data to indirect address |
| | A,#data | -move immediate to accumulator |
| | A,@Rr | -move indirect to accumulator |
| | A,Rr | -move register contents to accumulator |
| | A,data_add | -move direct address to accumulator |
| | C,bit_add | -move bit variables to C register |
| | DPTR,#data | -16-bit data allowed |
| | Rr,#data | -move immediate to register |
| | Rr,A | -move accumulator to register |
| | Rr,data_add | -move direct address to register |
| | bit_add,C | -move C register to bit variable |
| | data_add,#data | -move immediate data to direct address |
| | data_add,@Rr | -move indirect to direct address |
| | data_add,A | -move accumulator to direct address |
| | data_add,Rr | -move register to direct address |
| | data_add1,data_add2 | -move direct address to direct address |
| MOVC | A,@A+DPTR | -load A from code space through DPTR |
| | A,@A+PC | -load A from code space through PC |
| MOVX | @DPTR,A | -move accumulator to external memory |
| | @Rr,A | -move accumulator to external indirect |
| | A,@DPTR | -move external memory to accumulator |
| | A,@Rr | -move external indirect to accumulator |
| MUL | AB | -multiply A*B |
| NOP | | -do nothing |
| ORL | A,#data | -bitwise OR A with immediate data |
| | A,@Rr | -bitwise OR A with indirect address |
| | A,Rr | -bitwise OR A with register contents |
| | A,data_add | -bitwise OR A with direct address |
| | data_add,#data | -bitwise OR direct address immediate |
| | data_add,A | -bitwise OR direct address with A |
| | C,bit_add | -bitwise OR C with bit address |
| | C,/bit_add | -bitwise OR C with complement of bit |
| POP | data_add | -pop byte from stack |
| PUSH | data_add | -push onto stack |

| Mnemonic | Function | Description |
|---|---|---|
| RET | | -return from CALL |
| RETI | | -return from interrupt |
| RL | A | -rotate left |
| RLC | A | -rotate left through carry |
| RR | A | -rotate right |
| RRC | A | -rotate right through carry |
| SETB | C | -set carry bit |
| | bit_add | -set bit address |
| SJMP | code_add | -relative jump |
| SUBB | A,#data | -subtract with borrow |
| | A,@Rr | -subtract indirect |
| | A,Rr | -subtract register |
| | A,data_add | -subtract to direct address |
| SWAP | A | -swap upper/lower nibbles |
| XCH | A,@Rr | -exchange accumulator indirect |
| | A,Rr | -exchange accumulator with register |
| | A,data_add | -exchange accumulator with data address |
| XCHD | A,@Rr | -exchange low nibbles indirect |
| XRL | A,#data | -EXCLUSIVE OR A with immediate data |
| | A,Rr | -EXCLUSIVE OR A with indirect address |
| | A,data_add | -EXCLUSIVE OR A with direct address |
| | data_add,#data | -EXCLUSIVE OR direct address immediate |
| | data_add,A | -EXCLUSIVE OR direct address with A |

## ASM44 OPERANDS AND EXPRESSIONS

The following sections describe the format of ASM44 instructions and subfields. In ASM44, the general form of an instruction line is:

<label:> Mnemonic <operand> <,operand> <,operand> <;comment>

The number and types of operands depend on the instruction or directive mnemonic. Operands fall into one of six classes and may be represented as a constant or expression.

## SEGMENT SPACES (ADDRESS SPACES)

ASM44 supports the five address spaces in the 8044 as separate segments. Each segment space has a location counter which is updated when addresses are assigned as storage location. Also, in the code segment through the generation of object code. Each location counter is initially set to 0. When a segment is activated using one of the segment selection directives, the location counter is set to either the last location used or to the value specified by the optional 'AT expression' clause.

| Segment | Characteristics |
|---|---|
| CODE | Contains executable object code (0-65535) |
| DATA | Onboard direct address space (0-255) |
| BIT | Onboard bit addressable address space (0-255) |
| IDATA | Onboard indirectly addressable addresses (0-255) |
| XDATA | Offboard indirectly addressable addresses (0-65535) |

IDATA and XDATA types may only be used as operands in the IDATA (XDATA) directives or in immediate mode addressing while loading the indirection registers RO, RI, and DPTR.

The active location counter may be set to an arbitrary value using the ORG expression directive. Note that when you modify the location counter using ORG expression directive, the address range is restricted to addresses greater than or equal to the location pointer value, and less than the hardware defined maximums.

The current position of the location counter is represented by symbol $, which is used in expressions as any other symbol. The $ represents the first byte of storage of the instruction or directive.

# SYMBOLS

Symbols must begin with a letter or with one of the special characters ? or _, and may be up to 255 characters in length. Only the first 31 characters are significant. Each symbol must be stored in an internal table; this restricts you to approximately 10000 characters spaces. (Assuming an average of eight characters per symbol, this represents 1200 user defined symbols.) All characters are converted to upper case. Refer to the RESERVED SYMBOL NAMES section in this chapter for the list of all predefined symbols.

All symbols are defined with four attributes.

1. The type attribute defines the symbol as a register, number, or an address.
2. The segment-type attribute defines the segment address type (DATA, BIT, XDATA, CODE, an IDATA).
3. ASM44 is absolute, forcing all symbols to have local attribute.
4. The value attribute is either the numeric value assigned, the address location assigned, or, if the symbol is a register, a value representing the register.

Once a symbol is defined, it may not be redefined unless the SET directive was used. ASM44 performs checks on the attribute fields for the suitability of the symbol, and reports errors if the symbol cannot be used.

ASM44 also defines a group of special assembler symbols to represent special registers within an instruction opcode. These symbols, listed below, may be used as part of the operand field within an instruction. The special assembler symbols A and R0-R7 may also appear as the operand in the EQU and SET directives. Symbols defined as special assembler symbols may only be used as operands in instructions and subsequent EQU and SET directives.

| Special Symbol | Description |
|---|---|
| A | Accumulator |
| R0-R7 | Eight general-purpose registers of currently active register bank |
| DPTR | 16-bit data pointer used for indirect references to CODE and XDATA space |
| PC | Program counter |
| C | Carry flag/boolean accumulator |
| AB | Operand for MUL and DIV instructions |

The absolute addresses of the currently active registers may be accessed through the predefined symbols AR0-AR7. The data address value of these symbols is calculated as an offset from the value specified in the USING directive. Refer to the Assembler State Control Directives section in this chapter. If a symbol is defined with an EQU or SET directive to one of AR0-AR7, its value will not change following subsequent USING directives.

## STATEMENT LABELS

Labels are a form of symbol which are defined by specifying the symbol name immediately followed by a colon (:) as the first field on a line. The labels are defined with the value of the current location pointer and with the segment type of the active segment. Labels may be used in any expression which accepts a memory address. Label definitions may only appear on empty statements, statements which initialize data (DB and DW), storage allocation directives (DS and DBIT) and machine instructions.

# ASSEMBLY TIME EXPRESSIONS

Assembly time expressions consist of numbers, character strings, symbols and operators, which are evaluated during assembly to produce a single 16-bit value and associated type.

## Numbers

Numbers may be specified using a default radix (2-16) or an explicit radix by appending a radix specifier (as shown below). All numbers must begin with a digit (0-9) and will be assigned the type NUMBER.

| Radix Specifier | Radix |
|---|---|
| B | Binary (0,1) |
| O or Q | Octal (0-7) |
| D | Decimal (0-9) |
| H | Hexadecimal (0-9, A-F) |

If no radix specifier is provided, ASM44 uses the default radix (see radix control).

Numbers are entered as unsigned values in the range 0-65535, but all expression evaluation is performed using signed arithmetic rules. If a number greater than 65535 is entered, an error message will be generated. Arithmetic overflows are not trapped during expression evaluation.

## Character Strings

Character strings are specified by enclosing the string in single quotes ('). If the string is less than three characters long, it may form part of an expression and will be automatically converted to a number, the first character in the high byte position if two characters are present or the low byte position if one character is present.

Strings longer than two characters may only be used in the DB directive and each character (left to right) will be assigned to a consecutive memory location. Note that when used in a DB directive, a null string formed by two adjacent quote characters will not be assigned any locations. If used in an expression, a null string will evaluate to 0. A quote character may be embedded in a string by immediately following the quote with a second quote character.

## Operators

ASM44 supports arithmetic, logical, special and relational operators. All operators return to a 16-bit value with the type determined by the operands. Operators represented by alphanumeric strings (MOD, NE, AND, ...) must be separated from adjacent operands by at least one space or tab character, or be enclosed in parenthesis.

Expressions are evaluated left to right according to the precedence levels shown, unless specifically modified by use of the ( ) parenthesis operators. ASM44 maintains an independent arithmetic stack which allows operand nesting up to 25 levels.

Relational operators evaluate to either true (1) or false (0) and ASM44 will treat any non-zero value as true. Relational expressions may be used as numeric results, allowing the encoding of in-line, conditional assignments. The statement:

```
ABC EQU (X1 > X2)*DEF + (X1 <=X2)*GHI
```

assigns ABC the value of DEP if the variable X1 is greater than X2, or the value of GHI if X1 is less than or equal to X2. The relational operators treat variables as signed, 16-bit values in all calculations.

Since many operands within ASM44 are restricted to byte values, it tests to ensure that the assigned values fit into a byte. Bytes are considered unsigned values and if contained within an expression are treated as positive integers in the range 0 to 255. When testing the results of an expression being assigned to a byte quantity, ASM44 checks that the upper byte is all zeros (range 0 to 255) and will generate an error message if the quantity cannot be represented as a byte.

| Operator | Precedence | Function |
|---|---|---|
| ( ) | 0 | modify expression precedence |
| HIGH a | 1 | return upper 8 bits of a |
| LOW a | 1 | return lower 8 bits of a |
| a * b | 2 | multiply a times b |
| a / b | 2 | divide a by b |
| a MOD b | 2 | return remainder of a/b |
| a + b | 3 | add a to b |
| a - b | 3 | subtract b from a |
| + a | 3 | unary plus of a |
| - a | 3 | unary minus of a |
| a EQ b | 4 | return TRUE if a equal b |
| a NE b | 4 | return TRUE if a not equal b |
| a LT b | 4 | return TRUE if a less than b |
| a LE b | 4 | return TRUE if a less or equal b |
| a GT b | 4 | return TRUE if a greater than b |
| a GE b | 4 | return TRUE if a greater than or equal b |
| a = b | 4 | return TRUE if a equal b |
| a < > b | 4 | return TRUE a not equal b |
| a < b | 4 | return TRUE if a less than b |
| a < = b | 4 | return TRUE if a less or equal b |
| a > b | 4 | return TRUE if a greater than b |
| a > = b | 4 | return TRUE if a greater than or equal b |
| NOT a | 5 | return TRUE if a = 0 |
| a AND b | 6 | return bitwise logical AND of a and b |
| a OR b | 7 | return bitwise logical OR of a and b |
| a XOR b | 7 | return bitwise exclusive OR of a and b |

## SEGMENT TYPES IN EXPRESSIONS

While the results of an expression is, in general, a number type, some expressions result in a typed value, according to the following rules:

1. Unary operations result in the same type as operand.
2. All binary operations except + and - will result in a number type.
3. For binary + and -, if one of the operands has a segment type, then the result will have the same type. Otherwise, the result is a number type.

# ADDRESSING MODES

The maintenance of separate address segments and the typing of all symbols and expressions allows ASM44 to ensure that the address references made in instructions and directives are to the correct address space. If an operand refers to a particular address space, the operand must be of type address and have the correct segment type, or it must be of type number.

## Code Addresses

Code addresses are used in operands to transfer instructions and and may be a number type or address type within the code segment.

When used in relative jumps (SJMP and conditional jumps), ASM44 calculates the correct offset from the user specified target address. The absolute range of target addresses is -128 to +127 from the first byte of the next instruction.

When used with block transfers (AJMP and ACALL), the target address must be within the block address of the first byte of the next instruction. A block address is defined to be contained within 11 bits, therefore the upper five bits of the target address and the address of the first byte of the next instruction must be identical.

The long jump and call instructions (LJMP and LCALL) accept a full 16-bit target address.

## Bit Addresses

Bit addresses represent the internal RAM bit space or the special function registers of the chip. They may be specified as an absolute bit address or in a base_address.offset form.

If specified explicitly, the expression must evaluate to an address in the bit segment or to a number. Locations 0-127 map to internal RAM locations 32-47, and locations 128-255 map to the special function registers.

The base_address.offset form of bit addresses requires that the base_address expression evaluates to either a number type or to an address in the data segment. It must be in the range 32-47 or 128-247. The offset must evaluate to a number in the range 0-7. If the base_address is less than 128, it has 32 subtracted from it and is then multiplied by 8. The offset is then added to form the BIT address and the resulting value is flagged as an ADDRESS type within the BIT segment.

## Data Addresses

Data addresses must be either numbers or address types in the data segment, which maps to the first 128 bytes of internal RAM and to the special function hardware registers. Valid ranges are 0 to +255. Access to the internal RAM above location 127 must be made through the IDATA space.

## Immediate Addresses

Immediate addressing, indicated by the pound sign (#) is encoded as part of the machine code. The expression may be any TYPE and its range is either 0 to 255 if a byte operand or 0 to 65535 if loading the DPTR register. The XDATA and IDATA address types may only be used as part of an immediate operand.

## Indirect Addresses

Indirect addresses provide access to memory through a register (either R0 and R1 for onboard RAM of DPTR) if accessing offboard memory or code memory. Access to onboard memory is made by specifying @R0 or @R1, where R0 or R1 must contain the address of the location to be accessed. Offboard memory is accessed using one of @R0, @R1 or @DPTR in the MOVX instruction. Code memory is accessed using either the MOVC @A+DPTR or MOVC @A + PC instruction. In all cases, the registers must be preloaded using an immediate value.

# ASM44 ASSEMBLER DIRECTIVES

The following sections describe the ASM44 directives for the definition of symbols, reservation of storage locations, and control of memory allocation.

With the exception of DW and DB, the directives do not produce object code and may not have a label associated with them. The five classes of directives are as follows.

- Symbol Definition Type

```
symbol      EQU    expr
symbol      SET    expr
symbol      DATA   data_address
symbol      XDATA  xdata_address
symbol      IDATA  idata_address
symbol      BIT    bit_address OR data_address.offset
symbol      CODE   code_address
```

- Storage Initialization type

```
<label:>    DS      expr
<label:>    DBIT    expr
<label:>    DB      expr <,expr...> OR 'ascii string'
<label:>    DW      expr
```

- State Control type

```
ORG     expr
END     <expr>
USING   expr
```

- Segment Selection type

```
CSEG    <AT expr>
DSEG    <AT expr>
XSEG    <AT expr>
ISEG    <AT expr>
BSEG    <AT expr>
```

In each of the preceding definitions, <...> indicates the field is optional and ???_address (where ??? is a segment class name) indicates the value must have either the indicated segment type or be typeless.

Expressions which define symbol values or which set the location counter may not contain forward references.

## SEGMENT SPACES

ASM44 supports the five address spaces in the 8044 as separate segments. Each segment space has a location counter which is updated when addresses are assigned as storage locations, or through the generation of object code when in the CODE segment. Each location counter is initially set to 0. When a segment is activated using one of the segment selection directives, the location counter will be set to either the last location used, initially 0, or to the value specified by the optional <expr>. The currently active location counter may be set to an arbitrary value using the ORG directive. Note that when modifying the current location counter using the ORG construct, the address range is restricted to addresses greater than or equal to the current value and less than the hardware defined maximums.

## SYMBOLS

Symbols must begin with a letter or with one of the special characters, ? or _, and may be up to 255 characters in lengths, of which 31 characters are significant. All characters are converted to uppercase, and unless defined using the SET directive, a symbol may not be redefined.

## STATEMENT LABELS

Labels are a form of symbol which are defined by specifying the symbol name immediately followed with a colon (:) as the first field on a line. The labels will be defined with the value of the active segment. Labels may be used in any expression which accepts a memory address. Label definitions may only appear on empty statements, statements which initialize data (DB and DW), storage allocation directives (DS and DBIT) and machine instructions.

# SYMBOL DEFINITION DIRECTIVES

Symbol definition directives allow defining new symbols. They may not be preceded with a label.

## EQU Directive

The EQU directive allows assigning a symbol with a numeric value or a special assembler type A or RO-R7. The expression must not contain forward references and the symbol will be the same TYPE as the expression. Symbols defined as A or R0-R7 will have the type REGISTER and may only be used as instruction operands or other EQU directives.

## Set Directives

The SET directive is similar to EQU, except that it allows the symbol to be redefined.

## BIT directive

The BIT directive creates a symbol of type BIT ADDRESS. The expression must evaluate to either a NUMBER or a BIT ADDRESS type, and must be in the range 0-255.

## Data Directives

DATA Directive creates a symbol of type DATA ADDRESS. The expression must evaluate to either a NUMBER or a DATA ADDRESS type, and must be in the range 0 to 255.

## XDATA Directive

The XDATA directive is used to create a symbol of type XDATA ADDRESS. The expression must evaluate to either a number or a XDATA ADDRESS, and may be in the range 0 to 65535.

## IDATA Directive

The IDATA directive is used to create a symbol of type IDATA ADDRESS. The expression must evaluate to either a number or an IDATA ADDRESS, and may be in the range 0 to 65535.

## CODE Directive

The CODE directive is used to create a symbol of type CODE ADDRESS. The expression must evaluate to either a number or a CODE ADDRESS, and may be in the range 0-65535.

# STORAGE INITIALIZATION AND RESERVATION DIRECTIVE

The following directives initialize or reserve storage in either word, byte or bit units, and may be preceded by a label. Note that only the DB and DW directives generate object code, and these directives may only be used in the CODE segment.

## DS Expression Directive

The DS expression directive reserves the number of bytes indicated by expression and may be used in any segment except the BIT segment. The expression may not contain forward references and the sum of expression plus the current value of the location counter must not exceed the address space of the segment.

## DBIT Expression Directive

The DBIT expression direction reserves the number of bits indicated by expression, and may only be used in the BIT segment. The expression may not contain forward references, and the sum of expression plus the current value of the location counter must not exceed the address space of the segment.

## DB Directive

The DB directive allows initializing code memory with a series of bytes or string constants. Each value must be separated by a comma. Null strings will not generate any data. The label, if present, will be assigned the address of the first byte.

## DW Directive

The DW directive allov s initializing code memory with a series of word (16-bit) constants. Each value must be separated by a comma. Null strings will generate a value of 0, and strings longer than two characters are not allowed. The label, if present, will be assigned the address of the first byte.

# ASSEMBLER STATE CONTROLS DIRECTIVES

The assembler state controls manipulate the environment or state of the assembly.

## END Directive

The END directive indicates to the assembler the end of the current program. END directives encountered within INCLUDE files are ignored. If an expression follows the END statement, it must be either a NUMBER or a CODE ADDRESS and will be included in the object file. Typically, within an ASM44 environment, the expression would reference the address of the initial task descriptor of an iRMX-51 task. If no expression is provided, a value of 0 will be used.

## ORG Expression Directive

The ORG expression directive adjusts the value of the location pointer. The expression should evaluate to a number and must be greater than or equal to the current location pointer and contain no forward references. The current location pointer will assume the value of expression.

## USING Directive

The USING directive defines which register bank is currently in use. The directive allows the use of AR0-AR7 to access the absolute registers, based on the most recent USING statement.

# SEGMENT SELECTION DIRECTIVES

The segment selection directives select which segment is to be active. If the optional AT expression is provided, the current location pointer of the selected segment will be set to the value of the expression. The expression must evaluate to a number and may contain no forward references.

The current active segment is first closed and then the selected absolute segment is opened. If no AT expression was provided, the current location pointer is set to the next available address within the new segment, initially 0.

# ASM44 ASSEMBLER CONTROLS

ASM44 operation is controlled by a series of assembler controls in the user's source file. The following sections describe how to invoke the assembler and the functions of the individual controls.

# INVOCATION

ASM44 may be invoked with a combination of command line and user prompt specifications. Unless the command line satisfies all the prompt items the prompts will be issued.

Define ASM44 with the following:

```
ASM44:==$SYS$SYSTEM:IBQ$ASM44"
```

The general form of invocation is:

ASM44 sourcefile <,objectfile> <,listfile> </switch /switch...>

The source file must be specified. If no extension is provided, the default is extension .A44. The object file defaults to the source file with the extension .HEX. The list file defaults to NUL.LST.

To skip a field, enter two consecutive commas ',,'. To force remaining unspecified fields to their default values, enter a semicolon ':'.

When switches are entered, each must be preceded with a slash '/', and allow toggling the default values of the binary controls described below. The switches may be entered at any point in a command or prompt line and include the following.

| Switch | Control | Default |
|---|---|---|
| L | LIST | ON |
| C | CLIST | OFF |
| M | MLIST | OFF |
| S | SLIST | ON |
| X | XLIST | OFF |
| P | PAGE | ON |
| E | EPRiNT | ON |
| 1 | PASS1 | OFF |
| D | DEBUG | OFF |

If the command line does not fully specify the three filenames, and does not contain a ';', the following prompts are issued:

```
SOURCE FILE [.A44]              <-IF SOURCE NOT SPECIFIED
OBJECT FILE [sourcefile.HEX]    <-IF OBJECT NOT SPECIFIED
LIST FILE [NUL.LST]             <-IF LIST NOT SPECIFIED
```

In response to a prompt, you may enter any or all of the remaining file specifications, any combination of switch values or a ';' to terminate the prompts. File names may be up to 44 characters long if no extension is specified, or 47 characters if the file extension is included.

If an invalid switch is specified, the system displays the message:

```
INVALID CONTROL SWITCH ?, IGNORED
```

where ? is the switch character.

After loading, ASM44 will sign on with the message:

```
DATEM ASM44 - 8044/8051 Cross Assembler, Version 1.1
Copyright © 1985, 1986, 1987 DATEM Ltd.
Copyright © 1987 Digital Equipment Corporation
```

ASM44 then processes the command and prompt lines, determines the invocation arguments, and then displays:

```
PASS1
```

Following PASS1 processing, the message:

```
PASS2
Assembly complete
Errors 0 Warnings 0
```

is displayed and the object and the list files are generated. The assembler terminates by displaying a summary of errors and warnings detected and returns to the DOS shell with the number of errors as the ERRORLEVEL return value.

# ASSEMBLER CONTROLS

The following section defines the various controls available for ASM44. All controls will default to a logical set of values.

## Binary Controls

The binary control directives set or clear a set of assembler flags which control the listing formats and the information contained in the output files. The default values may be redefined using /switch arguments during invocation. The general format is:

```
binary_control    <expr>
```

where the optional <expr>, if supplied, may contain any valid combination of symbols and numbers must evaluate to a number. If the <expr> evaluates to a non-zero value, the corresponding binary_control is enabled. If the <expr> evaluates to zero, the control is disabled. The predefined values ON (equal to 1) and OFF (equal to 0) may be used as the expression. If no expression is provided, the binary control will assume the value ON, enabling the associated control.

In the following control list, items marked with * are included for future compatibility. They have no effect in the current release of ASM44.

| Control | Default | Function |
|---|---|---|
| LIST | ON | Enables listing to the list file |
| * CLIST | OFF | Enables listing of false conditionals |
| * MLIST | OFF | Enables listing of expanded macros |
| SLIST | ON | Enables listing of symbol table |
| * XLIST | OFF | Enables generation of cref file |
| PAGE | ON | Enables formfeed/header at each page |
| EPRINT | OFF | Enables error line display |
| PASS1 | OFF | Enables error display during PASS1 |
| * DEBUG | OFF | Outputs debug symbol information |

The LIST control enables or disables the generation of the LIST file. If disabled, the CLIST and MLIST controls are implicitly disabled. The LIST control may be used to selectively enable the listing of all or part of a user's program.

[The CLIST control enables the listing of false conditional source lines. In normal operations, ASM44 will only include in the list file source which is responsible for generating or modifying the object code records. If CLIST is enabled, all source, including that which is embedded in false conditional segments of code, will be included in the list file.]

[MLIST enables the inclusion of all macro expansions within the list file. If disabled,only the macro invocation line will be included in the list file.]

SLIST controls the generation of a symbol table listing. If enabled, the symbol table is appended to the list file and will include all user defined symbols. If disabled, no user symbol is generated.

The PAGE control enables or disables the generation of the header lines at the top of each page. If enabled, the assembler generates a form feed and three header lines at the top of each new page. If disabled, only the first page will include the headers and no other paging functions will be in effect.

The EPRINT control enables or disables the display of errors on the user's console. While EPRINT is enabled, each error will generate an error message on the console consisting of the line number, error number and a short description of the error.

In normal operation, errors are only reported during PASS2 of the assembly. The PASS1 control enables the user to view the PASS1 errors on the system console. The EPRINT control must be enabled for PASS1 to have effect.

[The DEBUG control instructs the assembler to include all user defined symbols to the OBJECT file. Since dDCM844 only supports absolute HEX output files, this control will have no effect.]

## Parameter Controls

The parameter control directives establish the current parameters used for page formatting and the default RADIX. The general format is:

Parameter_control <expr>

where the optional <expr>, if supplied, may contain any valid combination of symbols and numbers must evaluate to a number. The value of <expr> is then used to established the corresponding value for the parameter. Each parameter has a minimum/maximum range, and if the optional <expr> is not supplied, the parameter will be reset to its default value.

| Control | Default | Range | Function |
|---|---|---|---|
| PAGELENGTH | 60 | 10-32767 | -Set page length in lines |
| PAGEWIDTH | 120 | 80-255 | -Set page width in characters |
| RADIX | 10 | 2-16 | -Default radix for constants |
| TAB | 8 | 1-32 | -Tab stops |

The pagelength and pagewidth controls set the format for each page and may be reset at any time. If the PAGE control is disabled, the pagelength control will set the corresponding parameter but will have no other effect.

The RADIX control sets the default radix for numbers. For radix between 11-16, the constant must begin with a digit and may consist to the standard hexidecimal characters 0-9 and A-F, subject to the digit range set by the radix value. The explicit radix overrides (T, O, Q, B, and H) will still override the default radix. The RADIX control does not take effect until the next source statement.

The TAB control will allow the user to set the tab stop length for use in the list file. Values range from 1 to 32 character spaces.

## General Controls

The general controls allow saving and restoring the binary control states as well as generating arbitrary page feeds. The format of the controls is:

general_control <expr>

where the optional <expr> may contain any valid combination of symbols and numbers but must evaluate to a number.

The value of <expr> is used as an iteration count to allow repetitions of the control. If no <expr> is specified, the value will default to 1.

| Control | Default | Range | Function |
|---|---|---|---|
| EJECT | 1 | 1-5 | Generate form feeds in list file |
| SAVE | 1 | 1-16 | Save state of binary controls |
| RESTORE | 1 | 1-16 | Recover previous binary controls |

The EJECT control allows the generation of form feeds (1 to 5) within th list file. If the PAGE switch is disabled, EJECT will have no effect. If the iteration count is greater than 1, only the last page will have the header information.

The SAVE and RESTORE commands allow saving and restoring the state of the binary controls. The maximum stack depth is 16.

## String Controls

The string controls allow the specification of various test strings used within the list file and the inclusion of external source files through the INCLUDE facility. The general format is:

```
string_control <{(} text {)}>
```

where the optional text may be any user specified text string. The delimiters for the text may be explicitly defined as parenthesis by having the first non-blank character a '('. If an opening parenthesis is used, the text must be balanced with a closing parenthesis. If no opening parenthesis is found, the text string begins with the first non-blank character and extends to the end of the line. If no text is specified, a null string (all blanks) will be used.

| Control | Max Length | Function |
|---|---|---|
| DATE | 9 | Defines date field in list file |
| INCLUDE | 47 | Specifies user include filename |
| NAME | 31 | Defines name field in list file |
| TIME | 6 | Defines time field in list file |
| TITLE | 60 | Defines title field in list file |
| SUBTITLE | 60 | Defines subtitle field in list file |

The DATE field is printed at the top of each page when the PAGE switch is enabled. Initially it is set to the system DATE maintained by VMS. This text string may be replaced by any user text string.

The INCLUDE command allows the inclusion of external files into the current assembly, with a maximum nesting depth of 4. The text string must be a valid VMS pathname, with a maximum length of NN characters. During the processing of an INCLUDE file the END directive will be ignored. If no INCLUDE file is specified, an indication of the current include nesting level and file name will be displayed on the standard error output device.

The NAME control allows specifying the NAME of the output module, which also appears in the list file. The NAME directive should only be used at the start of the program and should precede any object code generation or symbol definition directives. The default name will be the source file name.

The TIME field is printed at the top of each page when the PAGE switch is enabled. Initially it is set to to the system TIME maintained by VMS. This text string may be replaced by any user test string.

The TITLE field is printed at the top of each page when the PAGE switch is enabled The text string may be replaced by any user defined string. The default TITLE is the name of the source file.

The SUBTITLE field is printed at the top of each page when the PAGE switch is enabled. Initially it is set to all blanks, and may be replaced by any user defined text string.

## ASM44 ERROR MESSAGES

The following sections describe the errors which are reported by ASM44. The error number, line number and brief error message will be included in the list file using the following format:

*** ERROR ##, LINE LL (PP), ERROR MESSAGE

where

- ## is the error number
- LL is the current line number
- PP is the last line which contained an error.

In addition, if EPRINT is active, the error number, line number, and error message will be displayed on the user console.

# SOURCE FILE ERRORS

Errors in the range 0-99 result from errors detected when evaluating the user source program.

1 SYNTAX ERROR

A syntax error is reported whenever the assembler cannot determine the meaning of a source line. The remainder of the line is ignored.

2 SOURCE FILE PROCESSING TERMINATED AT 255 CHARACTERS

The maximum length of a source line is 255 characters. The line will be ignored, and the remainder of the line will be treated as the next line.

3 ARITHMETIC OVERFLOW IN NUMERIC CONSTANT

The maximum value of a constant is 65535. All values will be truncated.

4 ATTEMPT TO DIVIDE BY ZERO

An attempt to divide by 0 was detected. Note that undefined symbols will evaluate to 0, possibly causing this error during PASS1. If the error occurs during PASS2, the expression is in error.

5 FORWARD REFERENCE IN EXPRESSION NOT ALLOWED

Forward references are not allowed in any expression which defines the value of a symbol or the current location pointer of a segment.

6 SYMBOL PREVIOUSLY DEFINED MAY NOT BE SET

The symbol has been previously defined. Only symbols defined using the SET directive may be redefined.

7 SYMBOL PREVIOUSLY DEFINED

The symbol has been previously defined. Only symbols defined using the SET directive may be redefined.

8 ATTEMPT TO ADDRESS NON-BIT ADDRESSABLE BIT

The expression used to reference a bit address does not evaluate to a valid bit address.

9 BAD BIT OFFSET IN BIT ADDRESS EXPRESSION

The offset value used in base_address.offset structures must evaluate to a NUMBER in the range 0-7.

10 TEXT FOUND BEYOND END STATEMENT - IGNORED

This is a warning only.

11 PREMATURE END-OF-FILE NO END STATEMENT

The assembler did not detect an END statement. This is treated as a warning only.

12 ILLEGAL CHARACTER IN NUMERIC CONSTANT

An illegal character was detected in a numeric constant. Check the the default RADIX, if applicable, corresponds to the requirements of the constant characters.

13 ILLEGAL USE OF REGISTER NAME IN EXPRESSION

Registers, and symbols EQUATED or SET to registers, may not be combined in arithmetic expressions.

14 SYMBOL IN LABEL FIELD ALREADY DEFINED

The symbol appearing in a label field can be previously defined, either as a symbol or as a label.

15 ILLEGAL CHARACTER

An illegal character was detected in the source program.

16 MORE ERRORS DETECTED NOT REPORTED

A maximum of nine errors are reported on a single source statement. The remaining errors are tallied in the ERROR count, but are not displayed.

17 LOCATION COUNTER EXCEEDS SEGMENT BOUNDARY

An attempt to increment beyond a segment boundary.

18 UNDEFINED SYMBOL

A symbol appears in an expression which has not been defined. Processing continues using a NUMBER with a value of 0, which may cause further errors.

19 VALUE WILL NOT FIT INTO A BYTE

The expression must evaluate to a value in the range -256 to +255.

20 ILLEGAL OPERATION IN THIS SEGMENT

An attempt was made to generate object code in a segment other than the CODE segment.

21 STRING TERMINATED BY END-OF-LINE

This warning indicates that no closing quote mark was detected on a string constant.

22 STRING CONSTANT GREATER THAN 2 CHARACTERS

String constants may be used as arithmetic values if they are less than three characters long. Strings three or more characters long may only be used in the DB directive.

23 STRING NUMBER OR IDENTIFIER EXCEEDS 255 CHARACTERS

The maximum length of a string constant is 255 characters.

24 DESTINATION ADDRESS EXCEEDS INBLOCK ADDRESS RANGE

The upper 5 bits of the target address the address of the NEXT instruction are not the same. Rewrite the program using a LJMP or LCALL.

25 DESTINATION ADDRESS EXCEEDS RELATIVE ADDRESS RANGE

The maximum address range of a relative transfer is -128 to +127 from the address of the NEXT instruction. Rewrite the program to use a combination of a relative transfer and an absolute or long transfer.

28 REFERENCE NOT TO CURRENT SEGMENT

The expression used in a segment select directive does not evaluate to the current segment type.

29 IDATA SEGMENT ADDRESS EXPECTED

An idata segment address was expected.

35 LOCATION COUNTER MAY NOT POINT BELOW SEGMENT BASE

An attempt to ORG below the value set using the segment select directive was detected. When an 'AT expression' is used, it sets the minimum value which the ORG directive may reference in the absolute segment.

36 CODE SEGMENT ADDRESS EXPECTED

37 DATA SEGMENT ADDRESS EXPECTED

38 XDATA SEGMENT ADDRESS EXPECTED

39 BIT SEGMENT ADDRESS EXPECTED

The operand must be a NUMBER or an ADDRESS in the indicated segment.

40 BYTE OF BIT ADDRESS NOT BIT ADDRESSABLE

The base_address of a base_address.offset structure is not in the range 0-255.

41 INVALID HARDWARE REGISTER

Reserved for future use. Contact DATEM if this error occurs.

42 INVALID REGISTER BANK NUMBER

The register bank specified in the USING directive must be in the range 0-3.

50 EXPRESSION STACK ERROR

An error was detected due to an invalid expression structure. This error should seldom occur.

51 OPERATOR STACK ERROR

A poorly formed expression such as MOV A,#5+ was detected.

52 SYMBOL TABLE OVERFLOW

The user has exceeded the symbol table space. Please contact DATEM if this error occurs.

53 PHASE ERROR BETWEEN PASS1 AND PASS2

The address assigned to a symbol or label has changed between PASS1 and PASS2 processing. This error is usually a by-product of other errors.

## ASSEMBLER CONTROL ERRORS

The following errors are generated while processing the assembler control directives.

100 CONTROL EXPRESSION NON-NUMERIC

A non-numeric value was detected in a directive expecting the form:

directive <expr>

101 CONTROL STRING TOO LONG

The <string> expression exceeds the allowable length for the associated directive.

102 UNBALANCED CONTROL STRING PARENTHESIS

The first non-blank character following the control was an opening parenthesis. No closing parenthesis was found. The string is terminated at the end of the line.

104 INVALID INCLUDE FILE

The user specified INCLUDE file could not be opened for reading. Processing continues, but multiple errors usually result.

105 INVALID PAGEWIDTH

The pagewidth specified is outside of the MIN or MAX range allowed.

106 INVALID PAGELENGTH

The pagelength specified is outside of the MIN or MAX range allowed.

107 INVALID RADIX

The default RADIX is outside of the MIN or MAX range allowed.

108 INVALID TAB SIZE

The tab size specified is outside of the allowable MIN to MAX range.

109 EJECT COUNT EXCEEDS MAXIMUM

The maximum EJECT iteration count has been exceeded.

110 CONTROL STACK OVERFLOW

The user has attempted to SAVE the control environment on too deep a level.

111 CONTROL STACK UNDERFLOW

The user has attempted to RESTORE the control environment more often than SAVING it.

112 INCLUDE FILE NESTING LEVEL EXCEEDED

The user has exceeded the maximum nesting level for INCLUDE statements.

## OTHER ERRORS

ASM44 may report errors in the range 1000-1999. If these occur, it indicates an internal error. These can occur as a secondary effect of the above errors, and are usually corrected when initial error is corrected.

## RESERVED SYMBOL NAMES

AB AC ACALL ACC ADD ADDC AJMP AM AND ANL ARO AR1 AR2 AR3
AR4 AR5 AR6 AR7 AT
B BIT BSEG BV
C CJNE CLIST CLR CODE CPB CPL CSEG CY
DA DATA DATE DB DBIT DEBUG DEC DIV DJNZ DPH DPL DPTR DS
DSEG DW
EA END EJECT EPRINT EQ EQU ES ET0 ET1 EX0 EX1 EXTRN
FO
GE GT
HIGH
IDATA IE0 IE1 IE IEC INCLUDE IP IT0 IT1
JB JBC JC JMP JNB JNC JNZ JZ
LE LCALL LIST LJMP LOW LT
MLIST MUL MOD MOV MOVC MOVX
NAME NE NOP NR0 NR1 NR2 NS0 NS1 NS2 NSNR NOT
OFF ON OR ORG ORL OV
P P0 P1 P2 P3 PAGE PAGEWIDTH PASS1 PC POP PS PSW PT0 PT1
PUBLIC
PUSH PXO PX1
R0 R1 R2 R3 R4 R5 R6 R7 RADIX RBL RBS RBP RCB RE RESTORE RET
RETI
RFL RL RLC RR RRC RS0 RS1 RSEG RTS
SAVE SER SES SET SETB SEGMENT SHL SLIST SHR SI SJMP SMD
SP STAD
STS SUBB SUBTITLE SWAP
TAB TBL TBS TBF TCB TCON TF0 TF1 THO TH1 TIME TITLE TL0 TL1
TMOD
TR0 TR1
USING
XDATA XCH XCHD SLIST XOR XRL XSEG

# DATEM HEX FILE FORMAT

The DATEM HEX file format is identical to the Intel 8051/8044 HEX file format, except that it uses a feature found in the 8080 HEX file format which allows the indication of the file start address.

Each record may consist of up to 256 data bytes. When data records are written, the will consists of 26 data bytes, representing two RAC task DOWNLOAD messages. A data record consists of the following fields:

:<length><load address><record type><record data><checksum>

| | |
|---|---|
| : | -indicates start of record |
| length | -length of record (BYTE) |
| load address | -data load address for data records (WORD)<br>start address for end-of-file records (WORD) |
| record type | -0 for data records<br>1 for end-of-file records |
| record data | -multiple data bytes for data record<br>ignored for end-of-file records |
| checksum | -2's complement of sum of all record bytes |

# MEMORY MAP REQUIREMENTS

The ability to dynamically load and execute 8044 software across the BITBUS network relies on a slightly modified memory partition at the destination node. The 8051 architecture, on which the 8044 BITBUS processor is based, supports a non Von-Neuman machine structure in which the DATA space is independent from the CODE space. This supports a more efficient control oriented address structure.

The download utilities make use of the RAC facility which supports transferring data into the DATA space of the remote node. Note that there is no instruction within the 8051/8344 which allows writing to the CODE space.

In order to support downloading RMX51 tasks, it is necessary that the CODE and DATA spaces physically overlap, allowing the RAC facilities to write to the DATA space and the processor to execute from the (same) CODE space.

DATEM modules which support downloadable code provide three JEDEC memory sites, configured with the following address decode.

| Memory Site | Code/Data | Address | Function |
|---|---|---|---|
| 1 | CODE | 0-3FFFH | -Datem/user firmware (ePROM) |
| 2 | DATA | 0-3FFFH | -RMX/Datem/user data (RAM) |
| 3 | COMBINATION | 4000-BFFFH | -Loadable data space |

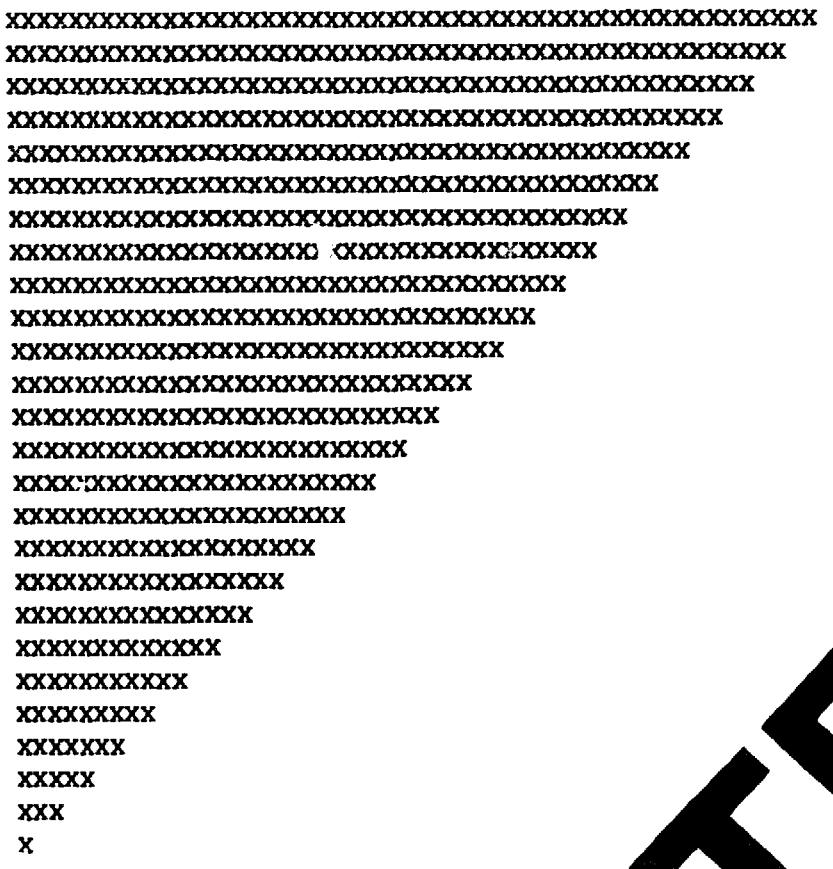
The appropriate users manual describe how to configure the third memory site as DATA only or CODE only. If the socket is not configured as COMBINATION, the download utilities will not operate correctly. If the socket is configured as CODE, it is not possible to write data into the RAM. If the socket is configured as DATA, it is possible to transfer files to the RAM, but it is not possible to execute from it.

Other manufactures modules must have equivalent COMBINATION memory to permit downloads.

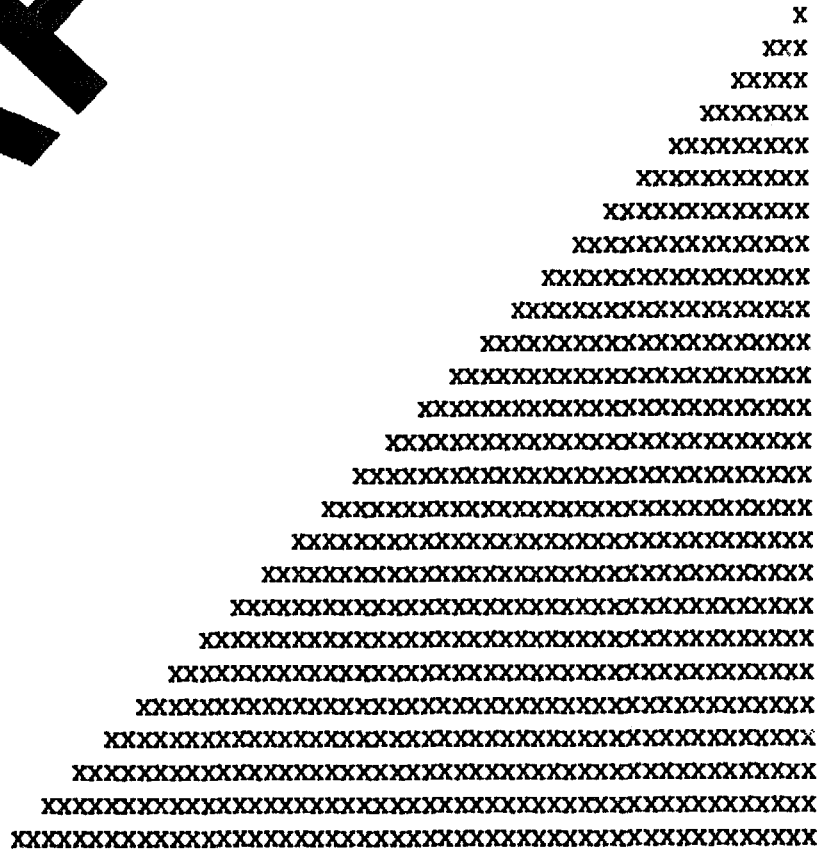
# RMX51I.EXT FILE CONTENTS

```
        dseg
        org     030h

rgtaskid:               ds  1   ;currently running task id
rgtaskpriority:         ds  1   ;priority of currently running task
rqclockunit:            ds  1   ;clock tick counts

        cseg
        org     083h

rqcreatetask:           ds  3   ;create task entry
rqdeletetask            ds  3   ;delete task entry
rqgetfunctionids:       ds  3   ;get function ids entry
rqdisableinterrupt:     ds  3   ;disable interrupt entry
rqenableinterrupt:      ds  3   ;enable interrupt entry
rqsendmessage:          ds  3   ;send message entry
rqsetinterval:          ds  3   ;set time interval
rqwait:                 ds  3   ;wait for event entry
rqallocate:             ds  3   ;allocate freespace
rqdeallocate:           ds  3   ;deallocate freespace
```

PAGE 58 INTENTIONALLY LEFT BLANK



# CHAPTER 4



# CHAPTER 4
# VMS SOFTWARE

## INTRODUCTION

This chapter describes the BCS interactive interface and command set and the BITBUS QIO set.

## INTERACTIVE INTERFACE

The Interactive Interface is a BITBUS Control Service (BCS) command line facility. You can select any one of the VMS editors to generate a configuration file. You can use the BCS command line facility to generate and modify the configuration file. With either, you can define the BITBUS nodes (devices/ positions connected to the serial BITBUS line). Also, with the command line facility you can interact with the Program Interface.

## BCS OVERVIEW

The following sections contain a detailed description of the BITBUS Control Service (BCS). This includes a description of the command set available to you for exchanging, adding or deleting modules, or for testing the modules.

The BITBUS Control Service is a command line facility. You can use BCS or any VMS editor to create and modify a program. With the command line facility, you have direct access to the Program Interface in an easy-to-use interactive format that allows you to test the system and to modify the configuration files.

# BITBUS CONTROL SERVICE CONFIGURATION UTILITY

This section discusses the BITBUS Control Service (BCS) configuration utility and the association of logical names to processes running at remote nodes. You can use BCS to change the definition of a BITBUS node or the entire BITBUS configuration.

There is a help file available for the BITBUS Control Service command set. To use BCS or to run the help file you type:

```
$ RUN SYS$SYSTEM:IBQ$BCS.EXE
```

Then enter

```
$ BCS <RETURN>
```

The host system responds with

```
$ BCS>
```

The command set for BCS follows:

| | | | |
|---|---|---|---|
| DEFINE | DELETE | EXIT | QUIT |
| READ | RESET | SHOW | WRITE |
| DIAGNOSE | DOWNLOAD | EXCHANGE | OFFLINE |
| ONLINE | RECEIVE | REPEAT | SCANON |
| SEARCH | SEND | | |

You use the first group of commands in the handling of the configuration file. These commands let you read and write files to VMS storage, add and delete information in the files, and display the contents of the files.

You use the second group of commands to test and exchange information with BITBUS slave nodes. These commands let you load programs into slave nodes, send data to and receive data from slave nodes, and place slave units on-line or off-line. Many of these BCS commands have a corresponding QIO command.

With BCS, you can change the operation to verbose mode or simulate mode. Also you can run in both verbose and simulate modes. The commands for the mode changes are:

| | |
|---|---|
| -V | enter verbose mode |
| -S | enter simulate mode |
| - -V | exit verbose mode |
| - -S | exit simulate mode |

The BCS commands are described on the following pages in alphabetical order. Included at the end of the BCS command set are descriptions of the BCS command and radix and range.

## DEFINE Command Line

**Syntax** DEFINE node task logical [comments]

**where** **node** is the number (1-250) of the slave node on the BITBUS

**task** is the task number (0-7) of the operation at the slave node on the BITBUS

**logical** is the name (1 to 31 characters) you associate with the task (it must be unique)

**comments** is any comment (1 to 25 characters) you want to relate to the task

**Description** The **DEFINE** command enters tasks, their logical names, and optional comments into the configuration file in BCS memory. You cannot repeat logical names in the other tasks. When you EXIT BCS after performing DEFINE operations, BCS WRITES a new configuration file to IBQ$DEFCONFIG.

*Example 1* **BCS> DEFINE 1 1 CHILLER_FAN TURN_ON_FAN**

This command associates the name CHILLER_FAN (logical) to task 1 on BITBUS slave node number 1 and enters it into BCS memory. The comment describes the task operation.

*Example 2* **BCS> DEFINE 1 2 CHILLER_OFF TURN_FAN_OFF**

This command associates the name CHILLER_OFF (logical) to task 2 on BITBUS slave node number 1 and enters it into BCS memory. The comment describes the task operation.

*Example 3* **BCS> DEFINE 100 7 HEATER_TEMP MEASURE_T**

This command associates the name HEATER_TEMP (logical) to task 7 on BITBUS slave node number 100 and enters it into BCS memory. The comment describes the task operation.

## DELETE Command Line

**Syntax** DELETE node [task]

where node is the number (1-250) of the slave node on the BITBUS

task is the task number (0-7) on the slave node

**Description** The DELETE command removes nodes and tasks from the BITBUS configuration file in BCS memory. The number is available for reassignment. When you EXIT BCS after performing DELETE operations, BCS WRITES a new configuration file to IBQ$DEFCONFIG.

*Example 1* **BCS> DELETE 1**

Removes node 1 and its associated tasks from the configuration file in BCS memory.

*Example 2* **BCS> DELETE 1 2**

Removes task 2 of node 1 from the configuration file in BCS memory.

## DIAGNOSE Command Line

**Syntax** DIAGNOSE [/IBQ=A]

where **/IBQ=A** is optional and is the identity (A through H) of the BITBUS Controller. The default value is A.

**Description** The DIAGNOSE command causes the IBQ to run its self diagnostics. Allow up to one minute for this command to complete.

*Example 1*

```
BCS> DIAGNOSE
BCS>
```

Runs the BITBUS self-diagnostics for IBQ A and indicates that they were successfully completed.

*Example 2*

```
BCS> DIAGNOSE /IBQ=C
BCS> ERROR (error message)
BCS>
```

Runs the BITBUS self-diagnostics for IBQ C and indicates that there was a failure.

## DOWNLOAD Command Line

**Syntax** DOWNLOAD [/IBQ=A] logical address filename

where /IBQ=A is optional and is the IBQ identification (A through H) — the default value is A.

logical is the name of the task on the slave node.

address is the external address location (0 - 65535) in the slave node to load the file.

filename is the name of the VMS file to be downloaded to slave node.

**Description** The DOWNLOAD command sends a file to the selected logical device.

*Example 1* **BCS> DOWNLOAD CHILLER_FAN 200 AFILE.DAT**

This command loads the file AFILE.DAT to location 200 on IBQ A in the BITBUS slave nodes with a task CHILLER_FAN.

*Example 2* **BCS> DOWNLOAD /IBQ=B HEATER_TEMP 209 BFILE.DAT**

This command loads the file BFILE.DAT to location 209 on IBQ B in the BITBUS slave node with a task HEATER_TEMP.

# EXCHANGE Command Line

**Syntax** EXCHANGE [/IBQ=A] logical appcom (data1,,,data12)

**where** /IBQ=A is optional and is the IBQ identification (A through H). The default is A.

logical is the name of task on the BITBUS slave node the data is being sent to.

appcom is the application command — the command that is delivered to the slave node and is acted upon by the slave node.

data1 through data12 is the numeric data, in bytes, to be sent from BCS to the slave node — see radix description in the physical unit numbers section. There must be an even number of bytes.

**Description** The EXCHANGE command sends a data packet of up to twelve bytes to the named nodes on the selected IBQ and receives the data from the named nodes. If a data entry exceeds a byte length, BCS truncates the most significant bits.

*Example 1* **BCS> EXCHANGE /IBQ=B DRILL 9 (4,4,9,^XD)**

Received the following data back from logical DRILL

| Byte # | Data |
|---|---|
| 1 | 4 |
| 2 | 4 |
| 3 | 9 |
| 4 | D |

This command sends the formatted data to the node with task named DRILL on IBQ B. The named node then sends the data back to BCS (the display is hexadecimal).

*Example 2* **BCS> EXCHANGE WELD 12 (12,445,'XFF,'X99)**

Received the following data back from logical WELD

| Byte # | Data |
|---|---|
| 1 | C |
| 2 | 1B |
| 3 | FF |
| 4 | 99 |

This command sends the formatted data to the node on IBQ A with the task named WELD. The named node then sends the data back to BCS (the display is hexadecimal).

## EXIT Command Line

**Syntax** EXIT

**Description** The EXIT command returns you to the DCL prompt in VMS. If you have made any modifications to the configuration file in BCS memory during the current BCS session, EXIT writes a new VMS configuration file to IBQ$DEFCONFIG.

*Example 1*

```
$ BCS
BCS>READ
BCS>DIAGNOSE
BCS> EXIT
$
```

With this session, there were no changes to the file. BCS closes the session without creating a VMS file.

*Example 2*

```
$ BCS
BCS>READ
BCS> DEFINE 2 3 DRILL
BCS> ...
BCS> EXIT
$
```

With this session BCS creates a new VMS file named IBQ$DEFCONFIG with the adds and deletes you made through the DEFINE and DELETE commands.

## HELP Command Line

**Syntax** HELP [topic]

where topic is one of the topics covered in the HELP file.

**Description** The HELP command, without the optional topic, produces a display of all of the HELP topics. This includes the command-line set and related items. If you type in one of the topics, HELP displays information about that topic.

*Example 1*

```
BCS> HELP
Additional information

BCS       DEFINE  DELETE   DIAGNOSE  DOWNLOAD  EXCHANGE
EXIT      HELP    OFFLINE  ONLINE    QUIT      radix
range     READ    RECEIVE  REPEAT    RESET     SCANON
SEARCH    SEND    SHOW     WRITE

To obtain further help type 'HELP topic'.
```

This command displays the BCS commands and ancillary information on which there is a help file.

*Example 2*

```
BCS> HELP EXIT
EXIT

Usage:  EXIT

  Description:  The EXIT command returns to VMS

                • If the configuration in memory was
                  changed during the BCS session, the
                  new configuration will be written to
                  the configuration file prior to exiting.

  Example       EXIT return to VMS
                BCS>
```

# OFFLINE Command Line

**Syntax** OFFLINE [/IBQ=A] [node]

**where** /IBQ=A is optional and is the IBQ identification (A through H) — the default value is A.

node is optional and is the number (1 - 250) of the slave node on the BITBUS — the default value is the IBQ01 Controller itself.

**Description** The OFFLINE command places the selected slave node off-line. The node is not available for sending or receiving and the slave node does not report to the host.

*Example 1*

```
BCS> OFFLINE /IBQ=B 2
BCS>
```

This command places IBQ B slave node 2 off-line.

*Example 2*

```
BCS> OFFLINE 1
BCS>
```

This command places IBQ A slave node 1 off-line.

*Example 3*

```
BCS> OFFLINE
BCS>
```

This command places Controller IBQ A off-line.

## ONLINE Command Line

**Syntax** ONLINE [/IBQ=A] [node]

**where** /IBQ=A is optional and is the identification (A through H) of the IBQ — the default is A.

node is optional and is the number (1-250) of the slave node on the BITBUS — the default is the IBQ Controller itself.

**Description** The ONLINE command places the selected slave node on line. The slave node begins reporting to the host; it can send and receive.

*Example 1*

```
BCS> ONLINE /IBQ=B 2
BCS>
```

This command places slave node 2 for IBQ B on-line.

*Example 2*

```
BCS> ONLINE 1
BCS>
```

This command places slave node 1 for IBQ A on-line.

*Example 3*

```
BCS> ONLINE
BCS>
```

This command places controller IBQ A on-line.

# QUIT Command Line

**Syntax** QUIT

**Description** The QUIT command returns you immediately to the DCL level prompt in VMS. Nothing is modified in the configuration files.

*Example*

```
$ BCS
BCS> READ
BCS> DEFINE 12 7 TEST
BCS> EXCHANGE TEST 4 (8,6)

Received the following data back from the logical
'TEST'.

          Byte #      Data

            1          8
            2          6

BCS> QUIT
$
```

## READ Command Line

**Syntax** READ [filename]

**where** **filename** is optional and is a valid VMS file specification containing the following information:

**File format:** node task logical [comments]

**where:**

**node** is the BITBUS node number (1-250)

**task** is the task number (0-7)

**logical** is the task name (1 to 31 characters)

**comments** is any comment (1 to 25 characters)

**Description** The READ command reads the selected configuration file — loads the file in BCS memory. You can then display the file through the SHOW command and modify it through the DELETE and DEFINE commands. File IBQ$DEFCONFIG is the default value. You may have several configuration files, one or more for each of the IBQ BITBUS Controllers.

BCS defines the logical names based on the information in the file. Reading a file erases any data currently in BCS memory.

*Example 1*

```
BCS> READ
BCS memory contains valid data, do you want to over-
write it [Y/N]? <N> <Return>
WARNING, READ command not execute
```

The question response occurs when you have entered data in BCS memory through either another READ command or DEFINE commands. If you answer N (or <Return>), the file is not copied into BCS. If you answer Y, BCS clears the contents of memory as in the QUIT command and places a copy of the configuration file IBQ$DEFCONFIG in BCS. You can then use the file for test or you can modify the file using the other BCS commands.

*Example 2*

```
BCS> READ IBQ$DEFCONFIG2

BCS>
```

This command places a copy of the configuration file IBQ$DEFCONFIG2 in BCS. You can then use the file for test or you can modify the file using the other BCS commands.

## RECEIVE Command Line

**Syntax** RECEIVE [/noscreen] [/IBQ=A] logical appcom [count]

**where** **/noscreen** is the optional VTxxx mode disable switch, and is specified when running a non-VTxxx terminal. The default value is VTxxx mode.

**/IBQ=A** is optional and is the IBQ identification (A through H). The default is A.

**logical** is a task name on the BITBUS slave node BCS is to receive from.

**appcom** is the application command — the command that is delivered to the slave node and is acted upon by the slave node.

**count** is optional and is the number of bytes (2 to 12) to receive. The default is 2. There must be an even number of bytes.

**Description** The RECEIVE command sets BCS to receive from a logical device (slave node) on the BITBUS. When you have two or more slave nodes with the same logical, the highest numbered of those slave nodes sends to BCS.

*Example 1*

```
BCS> RECEIVE CHILLER_FAN 3 2

** BITBUS CONTROL SERVICE **

        DATA MONITORING SCREEN

Node  Task  Logical-Name     Command  Byte #  Data
 22     5   CHILLER_FAN         3       1     78
                                        2     9A
```

This command sets up BCS to receive 2 bytes from the slave node on IBQ A which has a task with the logical CHILLER_FAN. The application command is 3.

*Example 2*

```
BCS> RECEIVE /noscreen /IBQ=B HEATER_TEMP
14 6

** BITBUS CONTROL SERVICE **

DATA MONITORING SCREEN

Node  Task  Logical-Name     Command  Byte #  Data
5     3     HEATER_TEMP      14       1       78
                                     2       F0
                                     3       43
                                     4       09
                                     5       E0
                                     6       02
```

This command sets up BCS to receive 6 bytes from the slave node node on IBQ B which has a task with the logical HEATER_TEMP. The received data will be formatted for a non-VT device. The application command is 14.

## REPEAT Command Line

**Syntax** REPEAT count command

where count is the number of times to repeat the selected command.

command is any valid BCS BITBUS control command with its parameters.

**Description** The REPEAT command repeats the valid BITBUS control command count times.

*Example 1*

```
BCS> REPEAT 10 RECEIVE HEATER_TEMP 14 6
```

With this command, the receive command for HEATER_TEMP node task on IBQ is repeated 10 times. See the RECEIVE command for details.

*Example 2*

```
BCS> REPEAT 6 SEND CHILLER_FAN 10 (1,2)
```

With this command, the send command for CHILLER_FAN node task on IBQ is repeated six times. See the SEND command for details.

## RESET Command Line

**Syntax** RESET [/IBQ=A] node

where /IBQ=A is optional and is the IBQ identification (A through H). The default is A.

node is the number (1-250) of the slave node to be reset.

**Description** The RESET command resets the selected slave node to the initialized state.

*Example 1*

```
BCS> RESET/IBQ=B 2
BCS>
```

This command resets slave node 2 on IBQ B to the initialized state.

*Example 2*

```
BCS> RESET 1
BCS>
```

This command resets slave node 1 on IBQ A to the initialized state.

# SCANON Command Line

**Syntax** SCANON [/noscreen] [/IBQ=A] logical appcom [countn].

**where** **/noscreen** is the optional VTxxx mode disable switch specified when running on a non-VTxxx terminal. The default is VTxxx mode.

**/IBQ=A** is optional and is the IBQ identification (A through H). The default is A.

**logical** is the task name on the BITBUS slave node BCS is to receive from.

**appcom** is the application command — the command that is delivered to the slave node and is acted upon by the slave node.

**countn** is optional and is the number of bytes (2 to 12) BCS is to receive, the default is 2. There must be an even number of bytes.

**Description** The SCANON command puts BCS in the continuous receive operation from a slave node with the named task. To turn off SCANON, press the <Return> key.

*Example 1*

```
BCS> SCANON CHILLER_FAN 1 4

** BITBUS CONTROL SERVICE **

DATA MONITORING SCREEN

Node   Task   Logical-Name      Command   Byte #   Data
 22      5    CHILLER_FAN                   1      78
                                            2      96
                                            3      E3
                                            4      1A
```

This command sets the BCS to receive 4 bytes from the slave node with task CHILLER_FAN on IBQ A and sends the application command 1 to the slave node. BCS displays the memory locations once every second. The data is displayed in hexadecimal.

*Example 2* **BCS> SCANON /noscreen /IBQ=B HEATER_TEMP 6 2**

| Node | Task | Logical-Name | Command | Byte # | Data |
|---|---|---|---|---|---|
| 22 | 5 | HEATER_TEMP | 6 | 1 | AF |
| | | | | 2 | 00 |

This command sets the BCS to receive 2 bytes from the slave node with task HEATER_TEMP on IBQ B and sends the application command 6 to the slave node. The data is formatted for display on a non VTxxx device. BCS displays the data once every second. The data is displayed in hexadecimal.

## SEARCH Command Line

**Syntax** SEARCH [/IBQ=A]

where /IBQ=A is optional and is the identification (A through H) of the IBQ. The default value is A.

**Description** A SEARCH command instructs the selected IBQ to search all BITBUS addresses for devices at those addresses. The SEARCH command will display all the nodes that respond. Please allow up to one minute for this command to complete.

You should do a SEARCH at system startup to verify the configured devices. Also, you should do a SEARCH after you modify the BITBUS connections.

*Example 1*

```
BCS> SEARCH

                                        BITBUS Addresses

Device Code (16): 99, Version Number (16): 1
Node List (10):

    0     1     5     6     7     12    13    15
    16    17    19    20

BCS> Exit
```

BCS performs a search of the 250 node addresses and displays those that were found on the BITBUS network.

## SEND Command Line

**Syntax** SEND [/IBQ=A] logical appcom (data1, ..., data12)

**where** **/IBQ=A** is optional and is the identification (A through H) of the IBQ BITBUS Controller. The default is A.

**logical** is a task name on the slave node BCS is to send data to.

**appcom** is the application command — the command that is delivered to the node and is acted upon by the slave node.

**data1 through data12** are the numeric data bytes to be sent from BCS to the slave node. There must be an even number of bytes.

**Description** The SEND command sends data to the BITBUS slave node with the named task.

*Example 1*

**BCS> SEND CHILLER_FAN 10 (1,2)**

This command sends the data (1,2) to the slave node with a task named CHILLER_FAN on IBQ A with the application command 10.

*Example 2*

**BCS> SEND HEATER_TEMP 6 (2,4,6,'XE8)**

This command sends the data (2,4,6,E8) to the slave node with a task named HEATER_TEMP on IBQ A with the application command 6.

## SHOW Command Line

**Syntax** SHOW [range|logical]

where **range** is any valid node address (physical unit number) or range of addresses — see physical unit numbers description (if you use range BCS ignores logical).

**logical** is any task name — when you append an * to the logical name, BCS performs a wild card search for all nodes with logical names matching the first characters given.

**Description** The SHOW command shows the configuration of the selected slave nodes from the current active file. SHOW does not affect any operations on the BITBUS.

*Example 1* **BCS> SHOW COMPRESSOR_RPM**

This command displays the configuration of the slave node containing a task with the logical COMPRESSOR_RPM.

CONFIGURATION

| Node | Task | Logical Name | Comments |
|---|---|---|---|
| 22 | 1 | DRILL | TWO CM |
| | 3 | COMPRESSOR_RPM | SIXTY |

*Example 2* **BCS> SHOW COMPRESSOR_***

This command displays the configuration of all slave nodes that have tasks that begin with the logical COMPRESSOR_.

CONFIGURATION

| Node | Task | Logical Name | Comments |
|---|---|---|---|
| 22 | 1 | DRILL | TWO |
| | 3 | COMPRESSOR_RPM | SIXTY |
| 29 | 4 | COMPRESSOR_PRESSURE | EIGHTY |

*Example 3* **BCS> SHOW**

This command displays the configuration of all of the BITBUS slave nodes.

*Example 4* **BCS> SHOW 3 5**

This command displays the configuration of slave nodes 3, 4, and 5.

CONFIGURATION

| Node | Task | Logical Name | Comments |
|---|---|---|---|
| 3 | 1 | DRILL | ONE |
| | 2 | TEMP_ON | SEVENTY |
| 4 | 1 | DRILL_THREE | THREE |
| | 3 | COMPRESSOR_RPM | SIXTY TWO |
| 5 | 2 | TEMP_OFF | EIGHTY |

## WRITE Command Line

**Syntax** WRITE [filename]

**where** filename is a valid VMS file specification.

**Description** If you specify the filename, BCS writes the file generated during this session into the VMS file system. If you do not specify the filename BCS writes to the default filename, IBQ$DEFCONFIG.

**File format:** node task logical [comments]

where

node - is the BITBUS node number (1-250)

task - is the task number (0-7) on the node

logical - is the name (1 to 31 characters) assigned to a task on the node - without spaces

comment - is any comment (1 to 25 characters) about the task - spaces are permitted

*Example 1* **WRITE**

This command writes the contents of the configuration file in BCS memory to VMS file IBQ$DEFCONFIG.

*Example 2* **WRITE CONFIG.DAT**

This command writes the contents of the configuration file in BCS memory to VMS file CONFIG.DAT.

# BCS Command Line Facility

**Syntax** BCS

**Description** Places you at the BCS level prompt (BCS>) if you have entered in your LOGIN.COM file the following:

**$BCS :== $SYS$SYSTEM:IBQ$BCS.EXE**

Before you run BCS, you can define the logical name IBQ$DEFCONFIG as the default configuration file. BCS uses this as the default file for the READ and WRITE commands.

*Example 1*

```
$BCS
BCS>
```

This command places you in the BCS utility in the normal operating mode and opens a configuration file.

*Example 2*

```
$BCS -V
BCS>
```

This command places you in the BCS utility verbose mode. In the verbose mode, BCS prints status information on your terminal as the commands you select are being executed. You can enter verbose mode at any time.

*Example 3*

```
BCS> - -V
```

This command places you in the non verbose mode of operation (either normal or simulate).

*Example 4*

```
$BCS -S
BCS>
```

This command places you in the BCS utility simulate mode. In the simulate mode, BCS simulates the execution of the commands without actually accessing the hardware. You can enter simulate mode at any time.

*Example 5*

```
BCS> - -S
```

This command places you in the non-simulate mode of operation (either normal or verbose).

# PHYSICAL UNIT NUMBERS

## Range Parameter

**Range**

**Description** A physical unit number or range of numbers from 1 through 250.

| *Example* | | |
|---|---|---|
| | 1 | specifies the physical unit 1 |
| | 200 | specifies the physical unit 200 |
| | 8 10 | specifies the physical units 8, 9, and 10 |

**Radix**

**Description** In BCS, all user input numbers are assumed to be in base 10. All displayed data is in hexadecimal unless otherwise specified in the output as NUM(base), [15(10) is the number 15 in the base 10]. You can override the radix for input data by using one of the following:

| | |
|---|---|
| ^X | The next number input is in base 16 |
| ^O | The next number input is in base 8 |
| ^D | The next number input is in base 10 |

| *Examples* | | |
|---|---|---|
| | 9 | Specifies a base 10 number of 9 |
| | ^X01 | Specifies a base 16 number of 16 decimal |
| | ^O77 | Specifies a base 8 number of 63 decimal |

# PROGRAM INTERFACE OVERVIEW

The following sections describe the interface between your BITBUS program and the IBQ01 BITBUS Controller through the VMS QIO system service. It describes the 14 VMS QIOs used in this interface and the parameters of these QIOs.

## QIOs

The VMS Queue I/O Request system service queues an I/O request to a channel associated with a device. The QIO service completes asynchronously; that is, it returns to the caller immediately after queuing the I/O request. The QIO service performs device-independent preprocessing of the request. Each QIO call consists of up to 12 parameters; six device-independent and six device-dependent (the IBQ01 device driver uses up to six dependent parameters).

The format for a QIO call is:

SYS$QIO [EFN],[CHAN],[FUNCT],[IOSB],[ASTADR],[ASTPRM],
[P1],[P2],[P3],[P4],[P5],[P6]

where The six device-independent parameters are (see the *VMS System Service Manual* for details on how to address QIOs and use the device-independent parameters):

- EFN — An event flag that is set when the QIO completes. The default is event flag 0.
- CHAN — The I/O channel assigned to the device.
- FUNCT — The device-specific function codes and function modifiers specifying the operation to be performed. (For a listing of the functions, see the section on QIO functions.)
- IOSB — The I/O status block to receive the final completion status of the I/O operation and is derived from codes returned from the driver. The IOSB contains the condition value, the transfer count, and device-specific information which includes IBQ01 reply, node reply, and application command. (See Chapter 6 for a description of the contents of IOSB.)

ASTADR The AST service routine to be executed when the I/O completes. The ASTADR argument is the address of a longword value that is the entry mask to the AST routine.

ASTPRM The AST parameter to be passed to the AST service routine. The ASTPRM argument is a longword value containing the AST parameter.

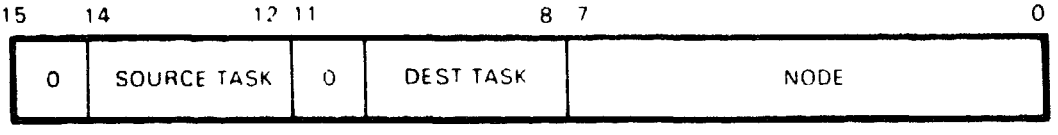
The six device-dependent parameters are (all 32-bit values):

P1 The address of the user DMA buffer. This is used for all functions which can do a DMA transfer — SEND, RECEIVE, XCHANGE, TRADE, UPLOAD, DOWNLOAD, SCANON, and ONLINE. Must be an even address.

Address of the AST routine to be called upon network event. Used by the SETAST command.

P2 The value of the length of DMA transfer in even-numbered bytes by value. This is used by SEND, RECEIVE, XCHANGE, TRADE, UPLOAD, DOWNLOAD, SCANON, and ONLINE. The maximum length is 60 kbytes normally. For SCANON and ONLINE to nodes other than 0, the maximum length is 12 bytes.

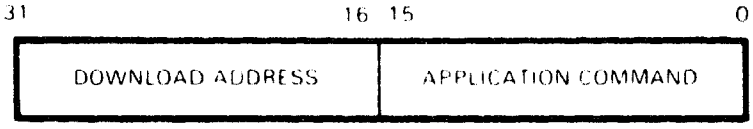
P3 A combination of the physical node address (0 through 250 - where 0 is the IBQ01) and the task number (0 through 7) which identify a node on the BITBUS. Used by OFFLINE, ONLINE, RESET, SEND, RECEIVE, DOWNLOAD, TRADE, UPLOAD, SCANOFF, and SCANON when the node number is not 0. Used by SCANOFF, SCANON, RESET, ONLINE, and OFFLINE when the node number is 0. The format for the word is:




CS-5735

Node is in the lower byte of the lower word; destination is in the lower half of the upper byte of the lower word; and, source is in the upper half of the upper byte of the lower word. You can use the source field (which can be used for anything) to further identify the source of the request (as when the same function is sent to the same task).

P4 The value of the download address. An address in the node from which the IBQ01 BITBUS Controller will read from for the UPLOAD function and write to for the DOWNLOAD command. The address is in the upper word.




CS-5639

Application command - Identifies function to destination task. The application function is in the lower word. See Chapter 2 for application command examples.

P5 The value of the timeout parameter in seconds. Specifies the length of time in which the operation must be completed. A timeout error is generated if the operation does not complete in this time. A parameter which is used for SEND, RECEIVE, EXCHANGE, TRADE, UPLOAD, and DOWNLOAD. When you specify P5=0, you specify an infinite timeout. Timeouts are generated on one-second boundaries; therefore, do not use a time of one second.

P6 Optional input buffer. Valid only for the TRADE and XCHANGE function. A write buffer protects the data being sent. When P6 is specified, P1 only refers to the output buffer. When P6 is not specified, P1 refers to both the input and output buffer. A value of 0 indicates no optional input buffer.

**NOTE**

**An improper parameter entry returns an error message, SS$_BADPARAM.**

## DATA FORMATS

For most communications functions, data contained in the DMA buffer must correspond to certain formats. In particular, RAC defines constructs for message data (see Chapter 2). In order to communicate with this and other protocols, you must be careful to format data in the DMA transferred buffers to conform to the protocol.

Since the IBQ01 does not recognize any particular format it does not change the data received (either from the host or a slave node). However, the partitioning performed by the IBQ01 for messages larger than the standard Intel message size of 13 bytes varies with different IBQ01 function types. To aid the user in formatting data buffers for remote nodes. A description of this partitioning is contained in the function descriptions.

## QIO FUNCTIONS

There are 14 QIO function codes used by the BITBUS driver. With virtual I/O privileges, you can use all of these commands except the TESTIBQ and UPLOAD commands, for which you must have physical I/O privileges to use. The function codes, which are described in the following sections, are shown in the following table in alphabetical order.

**IBQ01 I/O Functions**

| Function Code | Arguments | Function |
|---|---|---|
| IBQ$_BUSSEARCH | | Search bus |
| IBQ$_DOWNLOAD | P1,P2,P3,P4,P5 | Sends data to node memory |
| IBQ$_OFFLINE | P3 | Sets nodes inactive |
| IBQ$_ONLINE | P1,P2,P3,P4 | Sets nodes active |
| IBQ$_RECEIVE | P1,P2,P3,P4,P5 | Receive node information |
| IBQ$_RESET | P3 | Resets nodes |
| IBQ$_SCANOFF | P3 | Sets nodes off to receive |
| IBQ$_SCANON | P1,P2,P3,P4 | Sets nodes on to receive |
| IBQ$_SEND | P1,P2,P3,P4,P5 | Sends a message to a node |
| IBQ$_SETAST | P1 | Inform process by AST on network event |
| IBQ$_TESTIBQ | | Tests the IBQ01 board |
| IBQ$_TRADE | P1,P2,P3,P5,P6 | Does a SEND and RECEIVE |
| IBQ$_UPLOAD | P1,P2,P3,P4,P5 | Loads data from node memory |
| IBQ$_XCHANGE | P1,P2,P3,P4,P5,P6 | Does a SEND and RECEIVE |

## IBQ$_BUSSEARCH

Performs a search of the BITBUS and makes a table in the IBQ01 listing all of the nodes available. Allow one minute for this command to complete. Use an IBQ$_ONLINE to node 0 to obtain this table.

## IBQ$_DOWNLOAD

Sends information to be loaded into the internal memory of a slave node (8044 external).

The QIO arguments for this function are:

P1 - the address of the host VAX data area

P2 - the length of the host VAX data area

P3 - the address of the node/task

P4 - the address of the external RAM in the receiving node which is the upper word and the application command in the lower word

P5 - the value of the timeout parameter

**DOWNLOAD** makes use of the application-specific field in the PIO for downloading data to a slave node. The application-specific field holds the download start address (16 bits) (within the slave node). The IBQ01 delivers data to the slave node up to 13 bytes, with the starting address in the first two message data bytes. For data lengths greater than 13 bytes, the IBQ01 loads a start address (16 bits) into the first two message bytes of each packet and fills the remaining bytes (11 for a full packet) with the data in its buffer. The slave task must read the first two bytes as starting offset and write the data to its RAM space. All subsequent packets sent within the same download request will have an updated start address in the first two bytes of the message.

This command type is patterned after RAC DOWNLOAD although other commands can be issued as long as the receiving task recognizes the command and can respond appropriately. The IBQ01 checks the BITBUS command field and inserts a RAC DOWNLOAD command if a 0 is loaded.

## IBQ$_OFFLINE

Selects one node or the IBQ01 BITBUS Controller to be inactive.

The QIO argument for the function is:

P3 - the address of the node (0 to 250)

## IBQ$_ONLINE

Selects one slave node or the IBQ01 BITBUS Controller node to be active.

The QIO arguments for this function are:

P1 - the address of the host VAX data area

P2 - the length of the host VAX data area - DMA length must be <= 12 for node ≠ 0; for ONLINE 0 length must be <= 60 kbytes, only the first 254 bytes contain meaningful data

P3 - the address of the node (0 to 250)

P4 - the number of the application command

**ONLINE** performs a single packet transmission to the designated node to determine if it is present on the BITBUS. The IBQ01 uses an exchange format with the exception that no DMA occurs from the host to the IBQ01. The IBQ01 will format a single packet and pack zeroes into the specified buffer length. The command field is definable and will default to '0F' hex if the field is left open. Any return data is loaded into the buffer and DMA transferred back to the host VAX.

A special ONLINE case occurs when the node address is set to 0. The IBQ01 transfers the contents of its node table into the allocated buffer, and DMA transfers it back to the host VAX. Data returned includes the device code and revision number (first two bytes) and the chronological sequence of node addresses (byte integers) that are online. Unused bytes are packed with zeros.

**NOTE**

**When a remote node loses power and recycles back up, issue the ONLINE command twice. Since the controller and the node will not be synchronized, the first ONLINE will return an error. Ignore the first error. The first ONLINE will resynchronize the nodes and the second ONLINE should then proceed normally.**

## IBQ$_RECEIVE

Receives information from a slave node.

The QIO arguments for this function are:

P1 - the address of the VAX data area

P2 - the length of the host VAX data area

P3 - the address of the node/task

P4 - the number of the application command

P5 - the value of the timeout parameter

A **RECEIVE** commands a slave to return a message with data. No data is sent on transmit to the slave node, only the packet header and user-defined command field. This function assumes a user-defined command that instructs the slave task to return data. The length of the data returned is dependent on the responding slave and is captured as follows:

- The PIO DMA length defines the maximum data length receivable during a command. If a 100-byte buffer is allocated, no more than 100 bytes are requested (implied by number of command packets).
- The IBQ01 calculates the number (n) of 13-byte packets needed to retrieve the buffer length and delivers a command packet (no data) n times to the slave node.
- The slave node response can return up to 13 data bytes per command packet (up to the DMA buffer size). The IBQ01 collects and buffers the slave node responses and delivers them to the host after the final command packet is finished.

## IBQ$_RESET

Resets slave node or the IBQ01 BITBUS Controller (node 0).

The QIO argument for this function is:

P3 - the address of the node (0 to 250)

No data is handled on this command. IBQ01 delivers an 8-byte command packet to the node and does not wait for an acknowledge. A RESET 0 takes 20 seconds, cancels any current activity and locks out any new activity during that period. Any QIO issued during a RESET 0 operation will return with the error code SS$_DEVOFFLINE. Any RESET 0 issued during operation of any other command will return with error code SS$_INTERLOCK.

A RESET 0 puts the controller back into its power-up state and clears any hung nodes. If there are pending QIO operations, a RESET 0 should be preceded by a call to the System Services $CANCEL.

## IBQ$_SCANON

Receives information from a node continuously.

The QIO arguments for this function are:

P1 - the address of the host VAX data area

P2 - the length of the host VAX data area

P3 - the address of the node/task

P4 - the number of the application command

SCANON sends a command packet with no data to the designated node/task. The frequency of this continuous scan is determined by the response time of the slave node. Data is returned in the slave node's reply packet and DMA transferred to the host. The data cannot be longer than 12 bytes and is in the format of the replying slave node. Only one task-per-node can be scanned. Data will be placed continuously in the host data area and the QIO will remain active until a SCANOFF is issued to that node/task.

When P3 is 0 , SCANON turns the master scanning function on initiating or resuming any node/task scans in progress.

## IBQ$_SCANOFF

Turns off the SCANON command.

The QIO arguments for this function are:

P3 - the address of node/task

**CAUTION**

**SCANON remains active until you issue a SCANOFF to the same destination. A QIOW call will not return a until corresponding SCANOFF is issued from another program. When only one program is being run, it is necessary to use the QIO call so that further processing, including the SCANOFF, can continue.**

When P3 is 0, **SCANOFF** turns the master scanning function off, halting any data update until a SCANON to node 0 is issued. Pending SCANONs will remain in queue.

## IBQ$_SEND

Sends a message to a node.

The QIO arguments for this function are:

P1 - the address of the host VAX data

P2 - the length of the host data area

P3 - the address of the node/task

P4 - the number of the application command (see Chapter 2)

P5 - the value of the timeout parameter

Data is partitioned on 13-byte boundaries such that a 50-byte data buffer requires four message packets; the first three have 13 data bytes and the last one 11 data bytes. No Data replies are expected for **SEND**. The IBQ01 indicates completion to the host upon receipt of the fourth packet acknowledgment received from the slave.

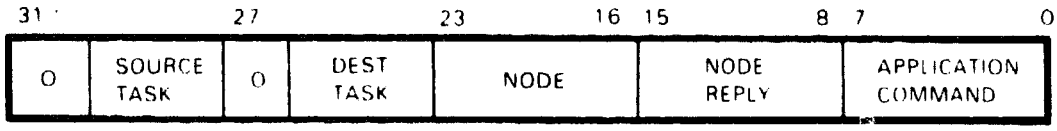
## IBQ$_SETAST

Informs the process by AST on network event (an unsolicited message from a node/task). For example, when a node is unable to respond a network event is generated.

The QIO arguments for this function are:

P-1 - the address to be executed upon AST delivery

With SETAST, a longword parameter passed by value to the AST routine will contain the node/task address in the upper word, the node application command in the lower half of the lower word, and the node reply in the upper half of the lower word.




## IBQ$_TESTIBQ

Tests the IBQ01 board by performing a series of tests:

- testcpu - test the 80186 CPU
- testram - tests the controller ram
- testclock - tests the on-board clock
- testmaster - tests the 8044 CPU

# IBQ$_TRADE

Combines the functionality of the SEND and RECEIVE command and allows you to retain the SEND data.

The QIO arguments for this function:

P1 - the address of the host data area

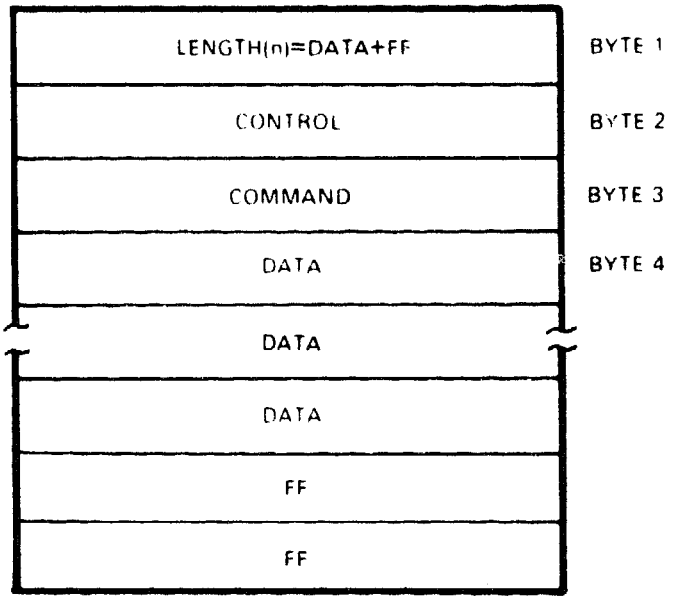
P2 - the length of the host data area

P3 - the address of the node/task

P5 - the value of the timeout parameter

P6 - the address of the optional input buffer in the host VAX

**TRADE** is similar to EXCHANGE except that the data field is formatted differently. For single commands, the following format must be used:



The parameter definitions are as follows.

**LENGTH:** Total number of data bytes to transmit excluding CONTROL, COMMAND, LENGTH.

**CONTROL:** Four MSBs of the byte as follows:

a. **TYPE:** A single bit (bit 7) that indicates whether message is an order or reply.

b. **SRC:** A bit value (bit 6) that indicates whether the source task resides on an extension or BITBUS controller.

c. **DEST:** A bit value (bit 5) that indicates whether the destination task resides on an extension or BITBUS controller.

d. **TRK:** A bit value (bit 4) used to track a message during a TRADE transfer. TRK must be set to 0 before sending an order message.

The typical value for the CONTROL Byte is 40H.

**COMMAND:** BITBUS application command.

**DATA:** Up to 13 bytes of data to be delivered to a remote node.

The IBQ01 reads the LENGTH field and formats a TRADE message packet with a data length of (LENGTH) bytes. This length can be less than 13 bytes, thus allowing for transmissions of unequal lengths within the same command sequence.

Response data is returned as follows:

a. **LENGTH:** This field contains the return data length.

b. **CONTROL:** The IBQ01 response control field is returned here.

c. **COMMAND:** The node response status is returned here.

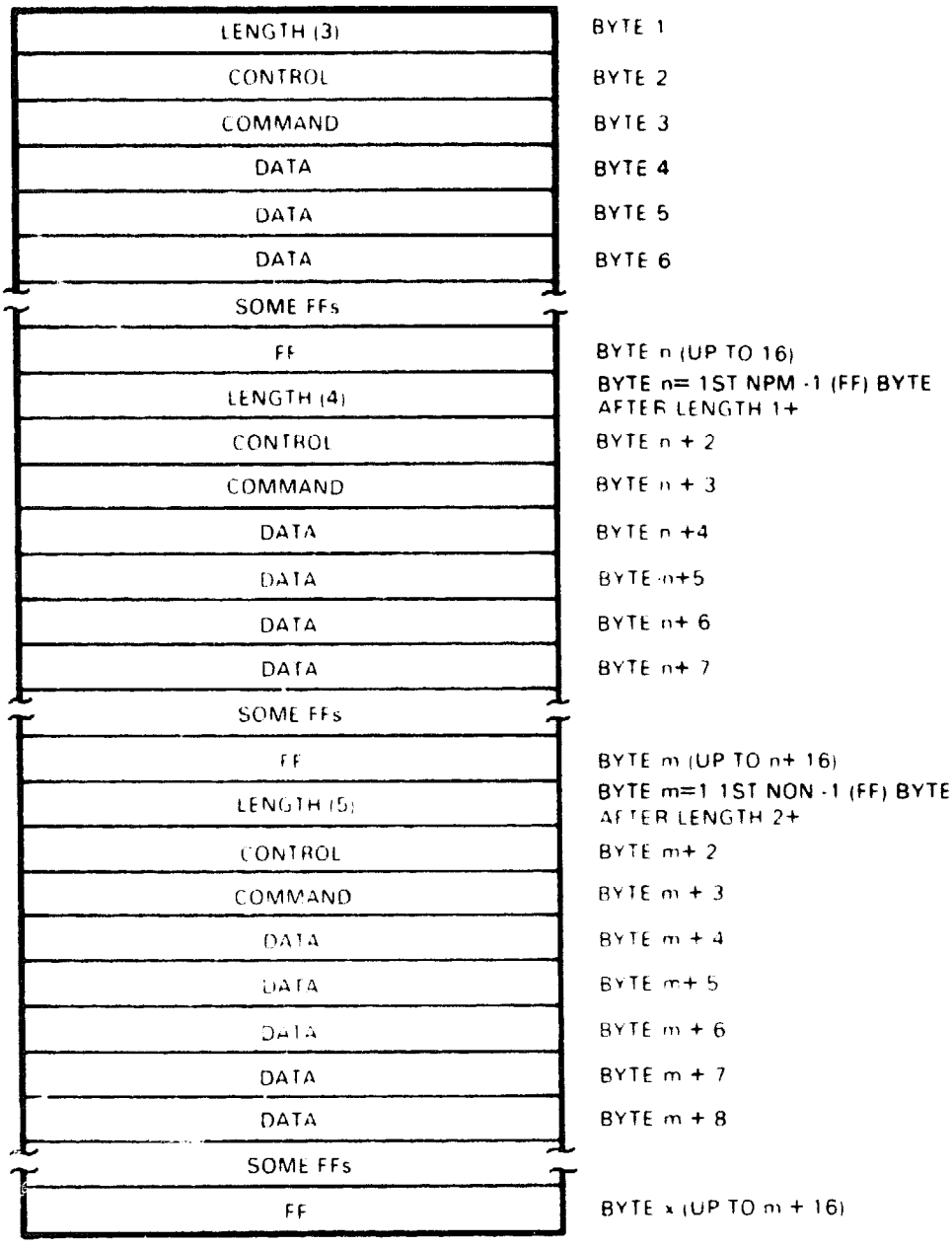
d. **DATA:** The response data (up to 13 bytes) is returned here.

These fields will change as shown on response, and must be reinitialized if the buffer is to be reused in another TRADE command.

**NOTE**

**The DMA length determines the maximum buffer size and hence the maximum message response. When a DMA length of 10 bytes is requested, 7 data bytes can be returned to the IBQ01. Responses larger than the DMA buffer are truncated to that DMA buffer length and the remaining data is lost!**

Extra space allocated for return data bytes (up to 13-per-packet) must be filled with -1 (FF). For multiple BITBUS commands in one trade operation, the following format must be used.


LENGTH (3)
CONTROL
COMMAND
DATA
DATA
DATA
SOME FFs
FF
LENGTH (4)
CONTROL
COMMAND
DATA
DATA
DATA
DATA
SOME FFs
FF
LENGTH (5)
CONTROL
COMMAND
DATA
DATA
DATA
DATA
DATA
SOME FFs
FF
BYTE 1
BYTE 2
BYTE 3
BYTE 4
BYTE 5
BYTE 6
BYTE n (UP TO 16)
BYTE n= 1ST NPM -1 (FF) BYTE AFTER LENGTH 1+
BYTE n + 2
BYTE n + 3
BYTE n +4
BYTE n+5
BYTE n+ 6
BYTE n+ 7
BYTE m (UP TO n+ 16)
BYTE m=1 1ST NON -1 (FF) BYTE AFTER LENGTH 2+
BYTE m+ 2
BYTE m + 3
BYTE m + 4
BYTE m+ 5
BYTE m + 6
BYTE m + 7
BYTE m + 8
BYTE x (UP TO m + 16)

CS-5641

## IBQ$_UPLOAD

Sends the information from a node's internal memory (8044 external) to the MicroVAX.

The QIO arguments for this function are:

P1 - the address of the host VAX data area

P2 - the length of the host VAX data area

P3 - the address of the node/task

P4 - the address of the external RAM of the sending node which is the upper word and the application command in the lower word

P5 - the value of the timeout parameter

**UPLOAD** uses the application-specific field in the PIO for uploading data from a slave node. The application-specific field holds the start address (16 bits) for the upload. The IBQ01 collects data starting at this address up to the DMA length specified — up to 13 bytes. For data lengths greater than 13 bytes, the IBQ01 will load a start address (16 bits) into the first two message bytes of each packet and fill the remaining bytes (11 for a full packet) with zeros. The slave task must read the first two bytes as offset and collect the data needed to fill the remaining message area. All subsequent packets sent within the same upload request will have an updated start address in the first two bytes of the message.

This function is patterned after RAC UPLOAD although other commands can be issued as long as the receiving task recognizes the command and can respond. The IBQ01 checks the BITBUS command field and inserts a RAC UPLOAD command if a 0 is loaded.

## IBQ$_XCHANGE

Combines the functionality of the SEND and RECEIVE commands.

The QIO arguments for this function are:

P1 - the address of the host VAX data area

P2 - the length of the host VAX data

P3 - the address of the node/task

P4 - the number of the application command

P5 - the value of the timeout parameter

P6 - the address of the optional input buffer in the host VAX

XCHANGE assumes a bilateral data transfer of commands from the master node to a slave node and back to the master node. The commands include slave addressing directives within the message data field (RAC — see Chapter 2). The IBQ01 assumes the same message size is returned in the slave's response. For data lengths greater than a single packet size, the IBQ01 divides the entire buffer into data packets 13 bytes long. This partition may cause the need for byte stuffing in order to obtain data alignment from packet to packet in a multi-packet exchange.

The major difference between the TRADE and the XCHANGE command is the format of the data buffers. IBQ$_XCHANGE is potentially easier to use as the header data for each BITBUS packet is constructed by the IBQ01 Controller from information in the QIO parameters. IBQ$_TRADE is constructed from information in the DMA input buffer.

# EXAMPLES

This section contains programming examples of device communication commands.

## Example 1

In this example, SCANON/SCANOFF functions are tested. The IBQ01 scans slave node 10 for a period equal to 100,000 loops (about six seconds) and displays the output generated.

```
c
c IMPLICIT INTEGER*4 (A-Z)

      INCLUDE 'SYS$SYSROOT:[SYSHLP.EXAMPLES]IBQ$FUNC.FOR'
      !IBQ constants
      INTEGER*2 BQ_CHAN !stores channel assigned
      INTEGER*2 IOSB(4) !stores return status
      BYTE DBUF(6) !buffer area in which data is received
      PARAMETER LENGTH=6 !number of bytes to input
      PARAMETER NODE=10 !node number + task since
                              !number is <256, task=0
      PARAMETER               !time in which node should
      TIMEOUT=6
                              !respond + 1
c
c
c     Assign the channel number
c
      STAT = SYS$ASSIGN ('_BQA:',BQ_CHAN,,)
      IF (.NOT. STAT) CALL LIB$STOP (%VAL(STAT)) !if can't
                                                 !assign stop
c
c
c Scan node 10 for a period of time, then turn the scan off
c
      STAT= SYS$QIO (%VAL(1),%VAL(BQ_CHAN),%VAL(IBQ$_SCANON)
      1 ,IOSB,,,DBUF(1),%VAL(LENGTH),%VAL(NODE)
      1 ,%VAL(IBQ$_RAC_WRITE),%VAL(TIMEOUT),,)
c
```

```
c Output data received, loop symbolizes additional processing to
c be performed.
c
      DO 100 I=1,100000
      TYPE 1000,DBUF !continually output scanned data
100   CONTINUE
      STAT= SYS$QIOW (%VAL(1),%VAL(BQ_CHAN),%VAL(SCANOFF)
      1 ,IOSB,,,DBUF(1),,%VAL(10),,,,)
      IF (.NOT. STAT) CALL LIB$STOP (%VAL(STAT))
      IF (.NOT. IOSB(1)) CALL LIB$STOP (%VAL(IOSB(1)))
1000 FORMAT ('DATA = ',I2)
      END
```

## Example 2

In this example, ONLINE function is tested, the IBQ01 places slave node 34 online.

```
c
c
c

      IMPLICIT INTEGER*4 (A-Z)
      INCLUDE 'SYS$SYSROOT:[SYSHLP.EXAMPLES]IBQ$FUNC.FOR'
      !IBQ constants
      INTEGER*2 BQ_CHAN !stores channel assigned
      INTEGER*2 IOSB(4) !stores return status
      BYTE DBUF(6) !buffer area in which data is received
      PARAMETER LENGTH=0 !no data exchanged
      PARAMETER NODE=34 !node number + task since
                                !number is <256, task=0
      PARAMETER TIMEOUT=10 !time in which node should respond + 1
c
c     Assign the channel number
c
      STAT = SYS$ASSIGN ('_BQA:',BQ_CHAN,,)
      IF (.NOT. STAT) CALL LIB$STOP (%VAL(STAT)) !if can't
                                                      !assign stop
c
c Place node 34 on line
c
      STAT= SYS$QIOW
      (%VAL(1),%VAL(BQ_CHAN),%VAL(IBQ$_ONLINE)
      1 ,IOSB,,,DBUF(1),%VAL(0),%VAL(NODE)
      1 ,%VAL(IBQ$_RAC_WRITE),%VAL(TIMEOUT),,)
c
c Verify correct return
c
      IF (.NOT. STAT) CALL LIB$STOP (%VAL(STAT))
      IF (.NOT. IOSB(1)) CALL LIB$STOP (%VAL(IOSB(1)))
      END
```

## Example 3

In this example, the **EXCHANGE** function is tested with slave node 10.

```
c
c
c

      IMPLICIT INTEGER*4 (A-Z)
      INCLUDE 'SYS$SYSROOT:[SYSHLP.EXAMPLES]IBQ$FUNC.FOR'
      !IBQ constants
      INTEGER*2 BQ_CHAN !stores channel assigned
      INTEGER*2 IOSB(4) !stores return status
      BYTE DBUF(6) !buffer area in which data is received
      PARAMETER LENGTH=4 !number of bytes to input
      PARAMETER NODE=10 !node number + task since
                                !number is <256, task=0
      PARAMETER TIMEOUT=6 !time in which node should
                                !respond + 1
c
c
c     Assign the channel number
c
      STAT = SYS$ASSIGN ('_BQA:',BQ_CHAN,,)
      IF (.NOT. STAT) CALL LIB$STOP (%VAL(STAT)) !if can't
                                                      !assign stop
c
c     Transfer 4 bytes of data to node 10 task 0 and receive that 4
c     bytes of data from that node
c
      STAT= SYS$QIOW
      (%VAL(1),%VAL(BQ_CHAN),%VAL(IBQ$_EXCHANGE)
      1 ,IOSB,,,DBUF(1),%VAL(LENGTH),%VAL(NODE)
      1 ,%VAL(IBQ$_RAC_WRITE),%VAL(TIMEOUT),,)
      IF (.NOT. STAT) CALL LIB$STOP (%VAL(STAT))
      IF (.NOT. IOSB(1)) CALL LIB$STOP (%VAL(IOSB(1)))
c
c Do something with the data received in DBUF
c
      END
```

# IBQ$LD44V DOWNLOAD UTILITY

IBQ$LD44V is a VMS utility to download to a node tasks created with the ASM44 8044 assembler. You invoke IBQ$LD44V as an image and run it once for each task you want to send to the node. IBQ$LD44V requests the device name of the IBQ01 controller, the file name of the ASM44 hex output file containing the assembled task code, and the BITBUS node number, which will receive the download. IBQ$LD44V prints the hexa-decimal lines as it processes them and displays any errors (a sample is shown below). Your 8044 source file should have a start address in the END statement (address of the initial task descriptor (ITD)). If no start address is given, the download operation will not complete.

```
$ RUN SYS$SYSTEM:IBQ$LD44V

                    LD44V Version V1.1
        Copyright (C) 1987, Digital Equipment Corporation

Enter controller name [default = 'BQA0']: <CR>
Enter the name of the hex file: [ .DEMO]FLASH.HEX
Enter the downloadable node number (decimal): 250
Beginning data download (hex file lines displayed) ...
:1a400000c249d24ac290740175f0fe12009820e00411500106ef2406f8b678
:1a401a0000041144012ab60104114a012a75f011eff808087b0708e644802b
:1a403400f6080808a6f08f827583001200920106c24a75f00022d24a75f006
:13404e000022304a0d304906c290c2498004d290d24922b7
:0b450000aa550500060503000000019d
:00450001ba
Download complete, starting task...
Task started, assigned task number 2
$
```

# TROUBLESHOOTING

Failures you may encounter in using the IBQ01 BITBUS Controller and the corrective steps are described in the following sections.

# DEVICE COMMUNICATIONS FAILURE

This section describes the two types of device communications failures that can occur:

- System failures - these are failures that occur in the system.
- Operator failures - these are failures that are related to operator fault.

## System Failures

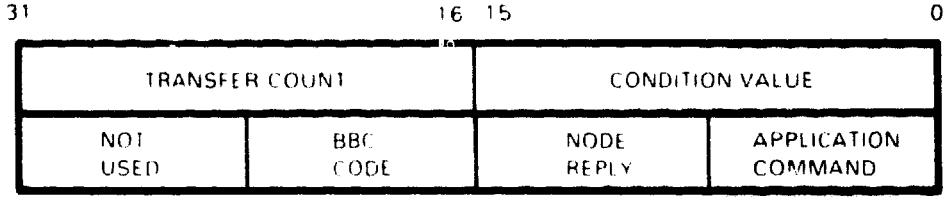
SS$PROTOCOL - VMS Driver/IBQ01 controller communication failure. May indicate controller problem. If this happens contact your DIGITAL representative.

## Operator Failures

There are two major sources of errors which can happen from the programmer generating a QIO call to the IBQ01 BITBUS Controller.

1. A bad device-independent parameter. The QIO system service rejects the call by returning an error code as the function value (see *VMS System Service Manual*).
2. All other errors. By returning an error code in the function value or I/O status block (see codes below).

The I/O status block is a quad word (64 bits) with the following information.




CS-5643

## IBQ01 VMS Driver Return Codes

The QIO system service returns errors based on whether the QIO call is set up correctly or due to run time I/O errors (see the *QIO System Service Manual*). The IBQ01 Driver can return the following codes:

1. SS$_ACCVIO - The DMA function is specifying an inaccessible buffer for reading or writing.
2. SS$_EXQUOTA - The SETAST function exceeds the quota allotted to the process for specifying AST requests.
3. SS$_INSFMEM - There is not enough system memory to complete the operation. This can happen during any operation.
4. SS$_DEVOFFLINE - Device offline, communications could not be initiated.
5. SS$_POWERFAIL - Power fail. Power fail condition detected on attempted write to device. This may effect all outstanding messages.
6. SS$_BADPARAM - Bad parameter, such as invalid DMA length (too large, or odd).
7. SS$_DEVOFFLINE - A QIO was issued during a RESET 0 operation.

8. SS$_INTERLOCK - A RESET 0 was issued during another QIO operation.

9. SS$_CANCEL - Cancel. Cancel is invoked before request completes.

10. SS$_TIMEOUT - A timeout occurred while waiting for a response from the device.

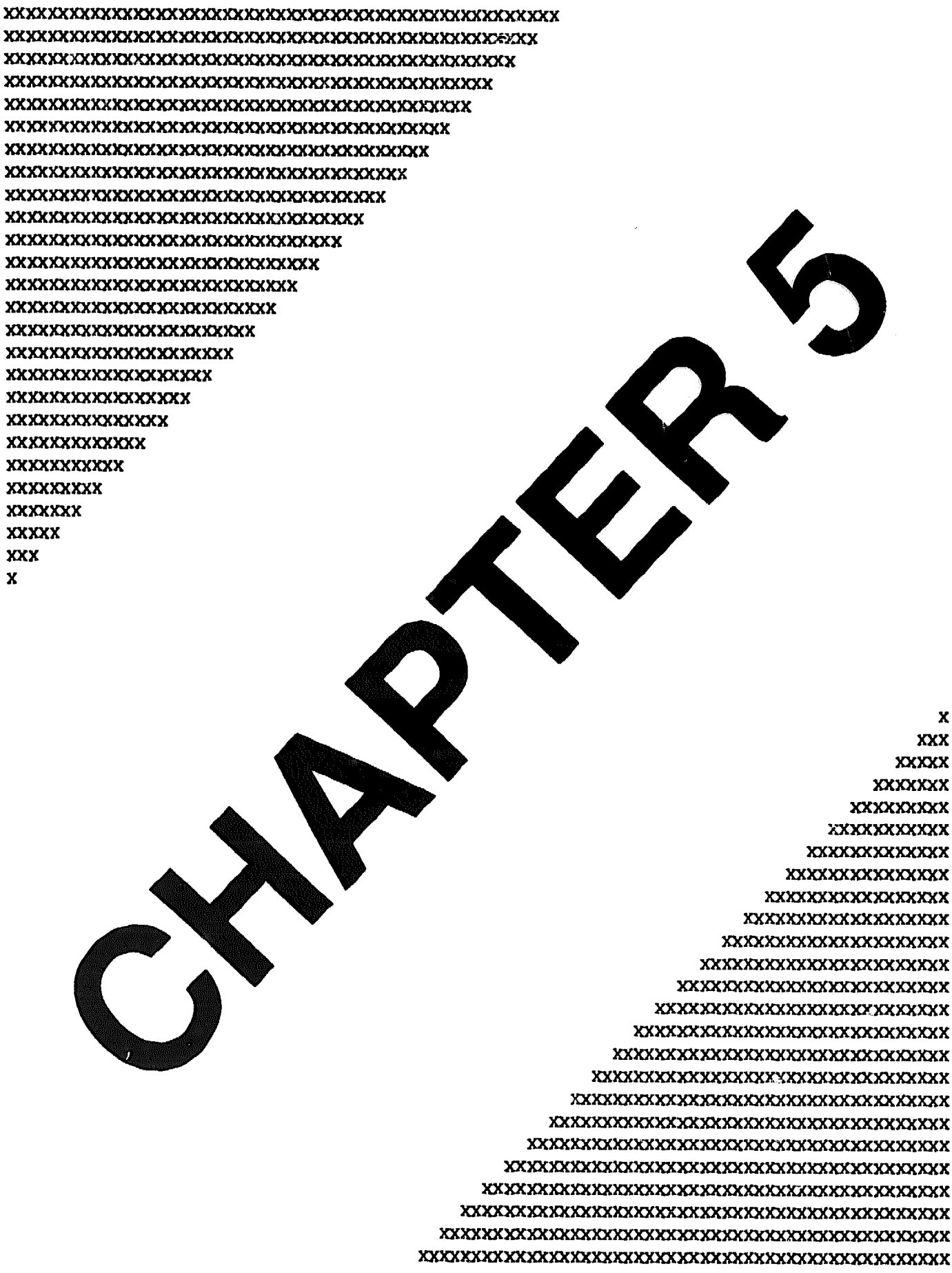
11. SS$_UNREACHABLE - The IBQ01 BITBUS Controller returned a transmission problem code.

12. SS$_PROTOCOL - Control information received from the slave node does not match the data sent.

13. SS$_NORMAL - Normal. Normal return.

## BITBUS Errors and IBQ01 BITBUS Controller Errors

See Chapter 2 for a list of these errors.


CHAPTER 5

# CHAPTER 5
# VAXELN SOFTWARE

## INTRODUCTION

This chapter describes the interface between your program and the IBQ01 BITBUS Controller through the VAXELN service. It describes the related commands used in this interface and how they are used.

## OVERVIEW

The VAXELN driver for the IBQ01 provides the functionality of the DECscan IBQ01 Q-bus-BITBUS board with a friendly program interface. In using this manual the following conventions are used.

1. All parameters are required unless enclosed with brackets [PARAMETER:=].
2. All parameters requiring byte values should be of type IBQ$_BYTE_TYPE.
3. All parameters requiring word values should be of type IBQ$_WORD_TYPE.
4. All parameters input to the IOSB field should be of type IBQ$_IOSB.

To build your application please observe the following rules:

1. When compiling, include in the compilation statement: ELN$:IBQ$ELNLIB/LIB. The names of any IBQ$ subroutines used should be specified by way of the INCLUDE option with the compiler or in your code.
2. When linking, include in the link statement: ELN$:IBQ$ELNLIB/LIB before the normally required ELN$:RTLSHARE/LIB, ENL$_RTL/LIB.
3. The program being built should be in the kernel mode.
4. The .DAT generated should have the following information if the hardware is installed as shipped:

```
DEVICE BQA0/REGISTER=%O760770/VECTOR=%O300/PRIORITY=1
/NO AUTOLOAD
```

All variables are passed by reference unless otherwise noted.

## IBQELN I/O Functions

| Function Code | Function |
|---|---|
| IBQ$_ABORT | Aborts last command |
| IBQ$_BUSSEARCH | Search bus |
| IBQ$_DOWNLOAD | Sends data to node memory |
| IBQ$_INIT | Assigns and initializes device |
| IBQ$_LOAD44 | Downloads a task to node memory |
| IBQ$_MAP | Maps user area DMA buffers to Q-bus |
| IBQ$_NONODEINFO | Loads entry index of pending requests |
| IBQ$_NOTIFY | Signals a network event |
| IBQ$_OFFLINE | Sets node offline |
| IBQ$_ONLINE | Sets node online |
| IBQ$_RECEIVE | Receive node information |
| IBQ$_RESET | Resets nodes |
| IBQ$_SEND | Sends a message to a node |
| IBQ$_SHUTDOWN | Deassigns node |
| IBQ$_SYSINFO | Returns info about pending requests |
| IBQ$_TESTIBQ | Tests the IBQ board |
| IBQ$_TRADE | Does a SEND and a RECEIVE |
| IBQ$_UNMAP | Unmaps user DMA buffers |
| IBQ$_UPLOAD | Sends from slave node memory |
| IBQ$_XCHANGE | Does a SEND and a RECEIVE |

# IBQ$_ABORT

Terminates execution of any outstanding device operation. This procedure finds the last command sent to the node and task specified and aborts it. If the operation does not complete successfully, an error is returned.

## Format

```
IBQ$_ABORT    (Device_pointer,
               Node_number,
               Task_number,
               [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte, defining the physical location of the node to be accessed, that is passed by value.

**Task_number** - A byte, defining which task on the node is to be accessed, that is passed by value.

**Status** - A variable, of type INTEGER, that indicates the completion of the device operations.

## Status Values

**IBQ$_SUCCESS** - The procedure was able to initiate device operations.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests. There are probably outstanding aborts in the system and your process must wait until they are completed. Synchronizing aborts through the user of MUTEXS, EVENTS OR SEMAPHORES will avoid this condition.

**IBQ$_INVALID_PARAMETER** - One of the input parameters was not specified correctly. No device operations were initiated.

**IBQ$_NO_EVENT_CREATED** - Routine could not create an event so that it could be informed upon device completion. Check system resources.

**IBQ$_NO_ENTRY_ABORTED** - Abort failed, probably because the device had already responded to the outstanding operation.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users to access it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has previously occurred which renders it unreachable. Check the device-status field for more information.

# IBQ$_BUSSEARCH

Performs a search of the BITBUS and makes a table in the IBQ01 listing all of the nodes available. Allow one minute for this command to complete. Use and IBQ$_ONLINE to node 0 to obtain this table. Please allow one minute for a full bus search to be completed.

## Format

```
IBQ$_BUSSEARCH  (Device_pointer,
                 [TRIGGER:=,]
                 [IOSB:=,]
                 [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes, the driver issues a SIGNAL with this parameter. If you do not include this argument the driver does not inform the user when this operation is complete and erases all information pertaining to this request.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call only. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the input parameters was not specified correctly. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users to access it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has previously occurred which renders it unreachable. Check the Device Status field for more information.

# IBQ$_DOWNLOAD

Sends information to be loaded into the internal memory (8044 external) of a slave node. The IBQ01 delivers data to the slave node up to 13 bytes, with the starting address in the first two message data bytes. For data lengths greater than 13 bytes, the IBQ01 loads a start address (16 bits) into the first two message bytes of each packet and fills the remaining bytes (11 for a full packet) with the data in its buffer. The slave task must read the first two bytes as starting offset and write the data to its RAM space. All subsequent packets sent within the same download request will have an updated start address in the first two bytes of the message.

This command is patterned after RAC DOWNLOAD command (you can use other commands as long as the receiving task recognizes the command and can respond appropriately). The IBQ01 checks the node command field and inserts a RAC DOWNLOAD command if a 0 is loaded.

## Format

```
IBQ$_DOWNLOAD (Device_pointer,
               Node_number,
               Task_number,
               Node_command,
               Node_address,
               [QBUS_ADDRESS:=,]
               [BUFFER_LENGTH:=,]
               [IOSB:=,]
               [TRIGGER:=,]
               [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte, defining the physical location of the node to be accessed, that is passed by value.

**Task_number** - A byte, defining which task on the node is to be accessed, that is passed by value.

**Node_command** - A byte, identifying the command which the IBQ01 is to send to the node, that is passed by value.

**Node_address** - A word (16 bits), identifying the location in external memory of the node (slave) from which the buffer is to be loaded, that is passed by value.

**Qbus_address** - A variable, of type INTEGER, that contains a 22-bit Q-bus address of the buffer. This address was obtained by adding an offset to the address obtained from the IBQ$_MAP procedure. This buffer must be on a word boundary, its value must be even.

**Buffer_length** - An integer, in the range 0 to 60000, that specifies the buffer length in bytes. The number of bytes must be an even number. If you specify a buffer length greater than 60000 bytes or an odd buffer length the procedure returns an error. The default value of this optional argument is zero, which implies no DMA, even if a buffer address is given.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request When the operation completes the driver issues a SIGNAL with this parameter. Failure to include this optional argument will cause the driver not to inform the user when this operation is complete and erases all the information pertaining to this request.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call only. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the parameters specified in this call was unacceptable. Most likely to be a bad Qbus_Address or Buffer_length. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device which does not allow other users to access it. This condition is temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has occurred which renders it unreachable. Check the Device Status field for more information.

# IBQ$_INIT

This procedure assigns the device and initializes it. The user supplies the device name and is returned the device pointer which he then uses in calls to other IBQ procedures. This procedure exits when the operation is complete.

This variable must be defined and loaded at the module level. For example:

```
Module FOO
VER Devname: STRING(4):= 'BQA0'
PROGRAM BAS
.
.
.
```

## Format

```
IBQ$_INIT       (Dev_Name,
                 Device_pointer,
                 [STATUS:=]);
```

## Arguments

**Dev_name** - A string of four characters specifying the name of the device to be initialized. This string must have its first two characters as BQ and its last character as 0 (zero). The third character identifies the device and is in the range of A through P. Legal strings are BQA0,BQB0,BQC0...BQP0.

**Device_pointer** - An identifier returned of type IBQ$_DEVPTR which the caller must use to further access this device.

**Status** - A variable, of type INTEGER, that indicates success or failure of the initialize operation.

## Status Values

**IBQ$_SUCCESS** - The procedure was able to initialize device.

**IBQ$_DEVICE_ALREADY_INITIALIZED** - Alternate success, indicating that the user is mapped to a device which has been initialized.

**IBQ$_NO_PROCESS_CREATED** - Driver was unable to create the driver process. Check system resources.

**IBQ$_NO_EVENT_CREATED** - Driver was unable to create an event necessary for the user process to communicate with the driver process. Check system resources.

**IBQ$_NO_SEMAPHORE_CREATED** - Driver was unable to create a semaphore necessary for the user process to communicate with the driver process. Check system resources.

**IBQ$_NO_AREA_CREATED** - Driver was unable to create an event necessary for the user process to share data with the driver process. Check system resources.

**IBQ$_INVALID_NAME** - User specified a device name which was not within the limits specified above.

# IBQ$_LOAD44

Downloads a task or data into the internal memory of a slave node. This procedure uses the IBQ$_ONLINE, IBQ$_MAP, IBQ$_TRADE, and IBQ$_UNMAP driver commands to send the data to the slave node. The task or data is supplied in the module IBQ$_DNLD_DAT.PAS, created by the utility IBQ$LD44E on VMS. IBQ$LD44E in turn, receives its inputs from hex files created by the 8044 cross assembler IBQ$ASM44 on VMS. The hexadecimal files are converted and grouped together in a PASCAL data structure in the module IBQ$_DNLD_DAT.PAS. The input to IBQ$ASM44 are 8044 assembly language programs created on VMS, which are either tasks or data to be loaded into BITBUS node memory. See the section titled "IBQ$LD44E Download Utility" for more details on the full downloading sequence.

Each call to IBQ$_LOAD44 chooses one of the files in the PASCAL data structure. It also chooses a BITBUS node which will receive the data into its memory. Any file can be sent to multiple nodes, through multiple calls to IBQ$_LOAD44. If the file is a task and has a start address (specified in the assembly language program), IBQ$_LOAD44 issues a create task command to the RAC task on the node, after loading the task data. If there is no start address, the data will only be loaded into node memory. The node response is returned, which in the case of a started task will be the assigned task-ID number. In the case of a data download, the node response should be zero. If there were problems at the node, such as no buffers available to start the task, the node response will contain an error code. See Chapter 3 for a list of node error codes.

## Format

```
IBQ$_LOAD44     (Device_pointer,
                 File_number,
                 Node_number,
                 [NODE_RESPONSE:=,]
                 [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**File_number** - An integer specifying which data file in the data module IBQ$_DNLD_DAT to send to the node. The first file is number 1 and the value increases by 1 for each subsequent file. The order of files is as specified when IBQ$_DNLD_DAT was created using the utility IBQ$_LD44E.

**Node_number** - A byte defining, the physical location of the node to be accessed, that is passed by value.

**Node_response** - A variable, of type IBQ$_BYTE_TYPE, that will receive the response from the node receiving the download data. In the case of a downloaded and started task, this value will contain the task_ID number assigned by RAC when the task was created. Otherwise, the value will be 0 for no errors, or contain an error code.

**Status** - A variable, of type INTEGER, that indicates success or failure of the complete download operations. This is unlike most of the other driver calls, which only return the results of the procedure call in this field.

## Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to successfully download the data file.

**IBQ$_INVALID_PARAMETER** - One of the parameters specified in this call was unacceptable. Most likely to be a bad Qbus_Address or Buffer_length. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - User specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device which does not allow other users from accessing it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has occurred which renders it unreachable. Check the Device Status field for more information.

**IBQ$_FILE_NOT_FOUND** - The value specified in File_number was outside the ordinal range of data files in the module IBQ$_DNLD_DAT. The download could not proceed.

**IBQ$_BAD_COMMUNICATIONS** - Communications with the node specified in Node_number encountered problems, and the download did not complete. Node_response may provide more specifics.

## IBQ$_MAP

Maps user area DMA buffers to Q-bus space. This procedure must be done prior to invoking any IBQ DMA operations. The user supplies the pointer to the area and the size of that area. The procedure synchronously returns the Q-bus address of that region which the user inputs into the IBQ DMA procedures.

### Format

```
IBQ$_MAP    (Device_pointer,
             Area_pointer,
             Buffer_size,
             Qbus_address,
             [STATUS:=])
```

### Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Area_pointer** - A variable, of type ANYTYPE, representing an I/O buffer.

**Buffer_size** - An integer, in the range 0 to 65535, containing the buffer size in bytes.

**Qbus_address** - A variable, of type INTEGER, that returns a 22-bit Q-bus address of the buffer mapped.

**Status** - An INTEGER receiving completion status of this call.

### Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to initialize device.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

## IBQ$_NODEINFO

The user will provide this routine with a buffer and a node number. The routine will load this buffer with the entry index of all pending requests to that node. The buffer input must have allocation for an array of eight integers. When IBQ$_NODEINFO finds a request outstanding to that node, the entry_index of that package will get loaded into the array position corresponding to the task number indicated in the request package. Additional information about this request can be found by examining the IBQ Common Data Structure as described later in this chapter. If more than one request is outstanding to the same node and task, the one presently being waited for is reported.

### Format

```
IBQ$_NODEINFO(Device_pointer,
              Node_number,
              Node_Array,
              [STATUS:=]
```

### Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte identifying the node for which the information is desired.

**Node_array** - A buffer defined as:

ARRAY [0..7] of INTEGER

which holds the entry indices of all outstanding requests to the node.

**Status** - An INTEGER receiving completion status of this call.

### Status Values

**IBQ$_SUCCESS** - The procedure was able to initialize device.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

# IBQ$_NOTIFY

Requests the driver to signal the outstanding process when a network event occurs. The driver — > application signaling mechanism for this command should be similar to that of any other device operation command, with the exception that the calling process here will get signaled upon an event anywhere in the system and that no actual command is sent to the device to evoke this reply.

## Format

```
IBQ$_NOTIFY     (device_pointer,
                 trigger,
                 [IOSB:=,]
                 [ST^TUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes the driver will issue a SIGNAL with this parameter. This parameter is required for this routine.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call only. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the input parameters was not specified correctly. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - User specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users from accessing it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has previously occurred which renders it unreachable. Check the Device Status field for more information.

# IBQ$_OFFLINE

Selects one node or the IBQ01 BITBUS Controller to be inactive.

## Format

```
IBQ$_OFFLINE  (Device_pointer,
               Node_number,
               [TRIGGER:=,]
               [IOSB:=,]
               [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte, defining the physical location of the node to be accessed, that is passed by value.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes the driver will issue a SIGNAL with this parameter. If this optional argument is not included, the driver does not inform the user when this operation is complete and erases the information pertaining to this request.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call only. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the input parameters was not specified correctly. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users to access it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has occurred which renders it unreachable. Check the Device Status field for more information.

## IBQ$_ONLINE

ONLINE performs a single packet transmission to the designated node to determine if it is present on the BITBUS. The IBQ01 uses an exchange format with the exception that no DMA occurs from the host to the IBQ01. The IBQ01 formats a single packet and pack zeroes into the specified buffer length. The command field is definable and will default to '0F' hex if the field is left open. Return data is loaded into the buffer and DMA transferred to the host.

A special ONLINE case occurs when the node address is set to 0. The IBQ01 transfers the contents of its node table into the allocated buffer, and DMA transfers it back to the host. Data returned includes device code and revision number (first two bytes) and the chronological sequence of node addresses (byte integers) that are online. Unused bytes are packed with zeros.

### Format

```
IBQ$_ONLINE     (Device_pointer,
                 Node_number,
                 Task_number,
                 [NODE_COMMAND:=,]
                 [QBUS_ADDRESS:=,]
                 [BUFFER_LENGTH:=,]
                 [TRIGGER:=,]
                 [IOSB:=,]
                 [STATUS:=]);
```

**NOTE**

**When a remote node loses power and recycles back up, issue the ONLINE command twice. Since the controller and the node will not be synchronized, the first ONLINE will return an error. Ignore the first error. The first ONLINE will resynchronize the nodes and the second ONLINE should then proceed normally.**

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte, defining the physical location of the node to be accessed, that is passed by value.

**Task_number** - A byte, defining which task on the node is to be accessed, that is passed by value.

**Node_command** - A byte, identifying the command which the IBQ01 is to send to the node, that is passed by value.

**Qbus_address** - A variable, of type INTEGER, that contains a 22-bit Q-bus address of the buffer. This was obtained by adding an offset to the address obtained from the IBQ$_MAP procedure. This buffer must be on a word boundary, its value must be even. Failure to include this optional argument means that no DMA will be performed.

**Buffer_length** - An integer, in the range 0 to 60000 if the node is 0 or 0 to 12 for all other nodes, supplying the buffer length in bytes. The number of bytes must be an even number. If you specify a buffer length greater than 60000 bytes or an odd buffer length the procedure returns an error. The default value is zero, which implies no DMA, even if a buffer address is given.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes the driver will issue a SIGNAL with this parameter. If this optional argument is not included the driver will not to inform the user when this operation is complete and it erases all information pertaining to this request.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of the procedure call. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the parameters specified in this call was unacceptable. Most likely to be a bad Qbus_Address or Buffer_length. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device which does not allow other users from accessing it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has occurred which renders it unreachable. Check the Device Status field for more information.

# IBQ$_RECEIVE

Commands a slave node to return a message with data. No data is sent on transmit to the slave node, only the packet header and user-defined command field. This function assumes a user-defined command that instructs the slave task to return data. The length of the data returned is dependent on the responding slave node and is captured as follows.

The buffer_length defines the maximum data length receivable during a command. The IBQ01 calculates the number (n) of 13-byte packets needed to retrieve the buffer length and delivers a command packet (no data) n times to the slave. The slave response can return up to 13 bytes per command packet (up to the DMA buffer size). The IBQ01 collect and buffers the slave-node responses and delivers them to the host after the final command packet is finished.

## Format

```
IBQ$_RECEIVE  (Device_pointer,
               Node_number,
               Task_number,
               Node_command,
               [QBUS_ADDRESS:=,]
               [BUFFER_LENGTH:=,]
               [TRIGGER:=,]
               [IOSB:=,]
               [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte, defining the physical location of the node to be accessed, that is passed by value.

**Task_number** - A byte, defining which task on the node is to be accessed, that is passed by value.

**Node_command** - A byte, identifying the command which the IBQ01 is to send to the node, that is passed by value.

**Qbus_address** - A variable, of type INTEGER, that contains a 22-bit Q-bus address of the buffer. This address was obtained by adding an offset to the address obtained from the IBQ$_MAP procedure. This buffer must be on a word boundary, its value must be even. If this argument is not included, no DMA will be performed.

**Buffer_length** - An integer, in the range 0 to 60000, containing the buffer length in bytes. The number of bytes must be an even number. If you specify a buffer length greater than 60000 bytes or an odd buffer length the procedure returns an error. The default value for this optional argument is zero, which implies no DMA even if a buffer address is given.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes the driver issues a SIGNAL with this parameter. If this optional argument is not included, the driver does not inform the user when this operation is complete and it erases information pertaining to this request.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call only. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the parameters specified in this call was not acceptable. Most likely to be a bad Qbus_Address or Buffer_length. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - User specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users from accessing it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has occurred which renders it unreachable. Check the Device Status field for more information.

# IBQ$_RESET

This procedure resets slave node or the IBQ01 BITBUS controller (node 0). no data is handled on this command. The IBQ01 delivers an eight-byte command packet to the node. If reset is to the Controller (node 0) the routine will wait for the device to recover before returning to the caller. This will cause an IBQ$_DRIVER_ON_HOLD status for any other processes attempting to access the driver while this is happening.

## Format

```
IBQ$_RESET      (Device_pointer,
                 Node_number,
                 [TRIGGER:=,]
                 [IOSB:=,]
                 [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte, defining the physical location of the node to be accessed, that is passed by number.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes the driver issues a SIGNAL with this parameter. If this optional argument is not included, the driver does not inform the user when this operation is complete and it erases information pertaining to this request.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call only. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the input parameters was not specified correctly. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - User specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users to access it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has previously occurred which renders it unreachable. Check the Device Status field for more information.

## IBQ$_SEND

Sends a message to a node. Data is partitioned on 13-byte boundaries such that a 50-byte data buffer requires four message packets; the first three have 13 data bytes and the last one 11 data bytes. No Data replies are expected for SEND. The IBQ01 indicates completion to the host upon receipt of the fourth packet-acknowledgment from the slave node.

### Format

```
IBQ$_SEND       (Device_pointer,
                 Node_number,
                 Task_number,
                 Node_command,
                 [QBUS_ADDRESS:=,]
                 [BUFFER_LENGTH:=,]
                 [TRIGGER:=,]
                 [IOSB:=,]
                 [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte, defining the physical location of the node to be accessed, that is passed by value.

**Task_number** - A byte, defining which task on the node is to be accessed, that is passed by value.

**Node_command** - A byte, identifying the command which the IBQ01 is to send to the node, that is passed by value.

**Qbus_address** - A variable, of type INTEGER, that contains a 22-bit Q-bus address of the buffer. This was obtained by adding an offset to the address obtained from the IBQ$_MAP procedure. This buffer must be on a word boundary, its value must be even. Failure to include this optional argument means that no DMA will be performed.

**Buffer_length** - An integer, in the range 0 to 60000, supplying the buffer length in bytes. The number of bytes must be an even number. If you specify a buffer length greater than 60000 bytes or an odd buffer length the procedure returns an error. The default value is zero, which implies no DMA, even if a buffer address is given.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes the driver issues a SIGNAL with this parameter. If this argument is not included, the driver will not inform the user when this operation is complete and it erases all information pertaining to this request.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the parameters specified in this call was unacceptable. Most likely to be a bad Qbus_Address or Buffer_length. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users from accessing it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has occurred which renders it unreachable. Check the Device Status field for more information.

# IBQ$_SHUTDOWN

Deassigns device and aborts any pending driver calls. This operation completes after the operation is complete.

## Format

```
IBQ$_SHUTDOWN   (Device_pointer,
                 [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, obtained from the IBQ$_INIT procedure.

**Status** - A variable, of type INTEGER, that indicates success or failure of the initialize operation.

## Status Values

**IBQ$_SUCCESS** - Procedure was able to shutdown device.

**IBQ$_NO_DELETE** - Could not delete one of the following: Device Process, Device Interface Area, Device Init Semaphore, Device. Check system resources.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there were too many outstanding requests.

**IBQ$_INVALID_PARAMETER** - One of the input parameters was not specified correctly. No device operations were initiated.

**IBQ$_NO_EVENT_CREATED** - Routine could not create an event so that it could be informed upon device completion. Check system resources.

**IBQ$_NO_ENTRY_ABORTED** - Abort failed probably because the device already responded to the outstanding operation.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users to access it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has previously occurred which renders it unreachable. Check the Device Status field for more information.

# IBQ$_SYSINFO

This routine returns information about all pending requests to the device. The caller supplies the area address. The routine writes a series of condensed request packets into this area until there is either no more information to write or the area is full.

## Format

```
IBQ$_SYSINFO  (Device_pointer,
               Info_Area,
               [ENTRY_COUNT:=],
               [STATUS:=])
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Info_area** - Area defined as IBQ$_SYS_ARRAY. IBQ$_SYS_ARRAY is a record containing the following fields:

1. command : Byte
2. node : Byte
3. Task : Byte
4. Index : Integer

The size of the array should be at least equal to the number of outstanding requests for which the caller desires information. These fields form a unique identifier for any outstanding request.

**Entry_Count** - A variable, of the type INTEGER, that returns the number of pending request packets written to the info_area.

**Status** - A variable, of type INTEGER, that indicates the success or failure of the operation.

## Status Values

**IBQ$_SUCCESS** - Indicates the procedure was able to perform the operation.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

## IBQ$_TESTIBQ

Tests the IBQ01 board by performing a series of tests:

1. testcpu - tests the 80186 CPU
2. testram - tests the controller ram
3. testclock - tests the on-board clock
4. testmaster - tests the 8044 CPU

Please allow one minute for this operation to complete

### Format

```
IBQ$_TESTIBQ   (Device_pointer,
                [TRIGGER:=,]
                [IOSB:=,]
                [STATUS:=]);
```

### Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes the driver will issue a SIGNAL with this parameter. If this optional argument is not included, the driver will not inform the user when this operation is complete and it erases all information pertaining to this request.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call only. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the parameters specified in this call was unacceptable. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users to access it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has occurred which renders it unreachable. Check the Device Status field for more information.

# IBQ$_TRADE

Similar to the XCHANGE procedure except that the data buffer is formatted as a BITBUS message packet. The format will be the same as that used for the VMS driver.

## Format

```
IBQ$_TRADE      (Device_pointer,
                 Node_number,
                 Task_number,
                 Qbus_address,
                 Buffer_length,
                 [TRIGGER:=,]
                 [IOSB:=,]
                 [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte, defining the physical location of the node to be accessed, that is passed by value.

**Task_number** - A byte, defining which task on the node is to be accessed, that is passed by value.

**Qbus_address** - A variable, of type INTEGER, that contains a 22-bit Q-bus address of the buffer. This was obtained by adding an offset to the address obtained from the IBQ$_MAP procedure. This buffer must be on a word boundary, its value must be even. This parameter is required for the trade command.

**Buffer_length** - An integer, in the range 4 to 60000, containing the buffer length in bytes. The number of bytes must be an even number. If you specify a buffer length greater than 60000 bytes or an odd buffer length or a length of less than 4 bytes, the procedure returns an error. This parameter is required for the trade command.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes the driver will issue a SIGNAL with this parameter. If you do not include this optional argument, the driver does not inform the user when this operation is complete and erases information pertaining to this request.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call only. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the parameters specified in this call was unacceptable. Most likely to be a bad Qbus_Address or Buffer_length. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users from accessing it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has occurred which renders it unreachable. Check the Device Status field for more information.

## IBQ$_UNMAP

Unmaps user area DMA buffers from Q-bus space. The user inputs all information described below.

### Format

```
IBQ$_UNMAP    (Device_pointer,
               Area_pointer,
               Buffer_size,
               Qbus_address,
               Status)
```

### Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Area_pointer** - A variable, of type ANYTYPE, representing an I/O buffer.

**Buffer_size** - An integer, in the range from 0 to 65535, containing the buffer size in bytes.

**Qbus_address** - A variable, of type INTEGER, that contains a 22-bit Q-bus address of the buffer mapped.

**Status** - An INTEGER defining the success or failure of the operation.

### Status Values

**IBQ$_SUCCESS** - Indicates the procedure was able to unmap registers.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

# IBQ$_UPLOAD

Sends information from the internal memory (8044 external) of a slave node, using the slave node address. The IBQ01 collects data starting at this address up to the buffer length specified, up to 13 bytes. For data lengths greater than 13 bytes, the IBQ01 loads a start address (16 bits) into the first two message bytes of each packet and fills the remaining bytes (11 for a full packet) with zeros. The slave-node task must read the first two bytes as starting offset and write the data to its RAM space. All subsequent packets send within the same upload request will have an updated start address in the first two bytes of the message.

This procedure is patterned after RAC UPLOAD: other commands can be issued as long as the receiving task recognizes the command and can respond appropriately. The IBQ01 checks the node command field and inserts a RAC UPLOAD command if a 0 is loaded.

## Format

```
IBQ$_UPLOAD   (Device_pointer,
               Node_number,
               Task_number,
               Node_command,
               Node_address,
               [QBUS_ADDRESS:=,]
               [BUFFER_LENGTH:=,]
               [TRIGGER:=,]
               [IOSB:=,]
               [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte, defining the physical location of the node to be accessed, that is passed by value.

**Task_number** - A byte, defining which task on the node is to be accessed, that is passed by value.

**Node_command** - A byte, identifying the command which the IBQ01 is to send to the node, that is passed by value.

**Node_address** - A word (16 bits) identifying the location in external memory of the node (slave) in which the buffer is to be loaded from.

**Qbus_address** - A variable, of type INTEGER, that contains a 22-bit Q-bus address of the buffer. This was obtained by adding an offset to the address obtained from the IBQ$_MAP procedure. This buffer must be on a word boundary, its value must be even. Failure to include this optional argument means that no DMA will be performed.

**Buffer_length** - An integer, in the range 0 to 60000, continuing the buffer length in bytes. The number of bytes must be an even number. If you specify a buffer length greater than 60000 bytes or an odd buffer length the procedure returns an error. The default value is zero, which implies no DMA, even if a buffer address is given.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes the driver issues a SIGNAL with this parameter. If this argument is not included, the driver will not inform the user when this operation is complete and erases all information pertaining to this request.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call only. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the parameters specified in this call was unacceptable. Most likely to be a bad Qbus_Address or Buffer_length. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users to access it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has occurred which renders it unreachable. Check the Device Status field for more information.

# IBQ$_XCHANGE

Combines the functionality of the SEND and RECEIVE commands. XCHANGE assumes a bilateral data transfer of commands from the master node to a slave node and back to the master node. The commands include slave-node addressing directives within the message data field. The IBQ01 assumes the same message size is returned in the slave-nodes response. For data lengths greater than a single packet size, the IBQ01 divides the entire buffer into data packets 13 bytes long. This partition may cause the need for byte stuffing in order to obtain data alignment from packet-to-packet in a multi-packet exchange.

The major difference between the TRADE and XCHANGE command is the format of the data buffers. IBQ$_XCHANGE is potentially easier to use as the header for each BITBUS packet is constructed by the IBQ01 controller from information in the input parameters. IBQ$_TRADE is constructed from information in the DMA input buffer.

## Format

```
IBQ$_XCHANGE (Device_pointer,
              Node_number,
              Task_number,
              Node_command,
              [QBUS_ADDRESS:=,]
              [BUFFER_LENGTH:=,]
              [TRIGGER:=,]
              [IOSB:=,]
              [STATUS:=]);
```

## Arguments

**Device_pointer** - An identifier, of type IBQ$_DEVPTR, returned by the IBQ$_INIT procedure.

**Node_number** - A byte, defining the physical location of the node to be accessed, that is passed by value.

**Task_number** - A byte, defining which task on the node is to be accessed, that is passed by value.

**Node_command** - A byte, identifying the command which the IBQ01 is to send to the node, that is passed by value.

**Qbus_address** - A variable, of type INTEGER, that contains a 22-bit Q-bus address of the buffer. This was obtained by adding an offset to the address obtained from the IBQ$_MAP procedure. This buffer must be on a word boundary, its value must be even. Failure to include this optional argument means that no DMA will be performed.

**Buffer_length** - An integer, in the range 0 to 60000, supplying the buffer length in bytes. The number of bytes must be an even number. If you specify a buffer length greater than 60000 bytes or an odd buffer length the procedure returns an error. The default value is zero, which implies no DMA, even if a buffer address is given.

**Trigger** - A variable, of type EVENT, which identifies the return mechanism of the request. When the operation completes the driver will issue a SIGNAL with this parameter. If this optional argument is not included, the driver will not inform the user when this operation is complete and it erases information pertaining to this request.

**IOSB** - A data structure that optionally returns a status code, number of bytes transferred, and device- and function-dependent information. The variable assigned to IOSB:= must be of type IBQ$_IOSB. Refer to the Device Data Structures section at the end of this chapter for the fields of this structure.

**Status** - A variable, of type INTEGER, that indicates success or failure of procedure call only. Success or failure of device operations may be examined in the IOSB after the TRIGGER event is satisfied.

## Status Values

**IBQ$_SUCCESS** - Indicates procedure was able to initiate device operations.

**IBQ$_INVALID_PARAMETER** - One of the parameters specified in this call was unacceptable. Most likely to be a bad Qbus_Address or Buffer_length. No device operations were initiated.

**IBQ$_INSUFFIENT_MEMORY** - Could not begin operations because there are too many outstanding requests.

**IBQ$_INVALID_DEVICE** - The user specified an invalid device pointer.

**IBQ$_DRIVER_ON_HOLD** - Another operation is being performed on the device at this time which does not allow other users to access it. This condition is only temporary.

**IBQ$_DRIVER_NOT_AVAILABLE** - The driver process has signaled that an abnormality has previously occurred which renders it unreachable. Check the Device Status field for more information.

# DEVICE DATA STRUCTURES

Although mainly intended for use by the driver, these structures contain information that may be useful for applications. We therefore provide definition of some of these fields. These fields are intended to be READ ONLY, any writing to these fields will produce unpredictable results and will render any service and support agreements void. For all the following assume that the field "device_pointer" refers to the value returned from IBQ$_INIT. Users may substitute their own pointers if they are defined as type IBQ$_DEVPTR.

## Device Wide Parameters

**Device_Pointer.DevNam : VARYING_STRING(4)** - This string contains a four-character name of the device which this structure refers to. The first three characters will be in the range of 'BQA' to 'BQP', the last one must be '0'.

Note: The field IBQ$_QUEUE_ENTRY consists of the following:

```
F_LINK: INTEGER
B_LINK: INTEGER
```

It is similar to the standard QUEUE_ENTRY field except that the elements are of type INTEGER and represent the index of the first and the last queue element rather than their addresses. This way they can be accessed across process boundaries by the driver.

**Device_Pointer.IOSR_Q_Hd : IBQ$_QUEUE_ENTRY** - This represents the header of the queue of requests which have not been sent to the device.

**Device_Pointer.OP_Q_Hd : IBQ$_QUEUE_ENTRY** - This represents the header of the queue of requests that have been sent to the device but have not received responses from the device.

**Device_Pointer.Idle_Q_hd : IBQ$_QUEUE_ENTRY** - This represents the header of the queue of unused request entries. The driver subroutines will remove an entry from this queue when the application requests a device operation.

**Device_pointer.Abort_Q_hd : IBQ$_QUEUE_ENTRY** - This represents the header of a special queue to be used only for IBQ$_ABORT requests. This is kept separate from the Idle_Q to allow aborts to continue when no other requests are outstanding. This eliminates a condition where memory cannot be freed because there is no memory to service the request.

**Driver_Status** - Indicates the current health of the driver. If an asynchronous exception forced the driver into a state which renders it unable to function, the driver process will write an error into this field.

**Q_entry** - Defined as:

Q_entry: ARRAY [0..IBQ$_Max_headroom+IBQ$_Reserve_Headroom] OF IBQ$_IO_Q_ENTRY

Each of these fields defines the full context of a request. The number of these fields that are initiated in the Idle_Q is IBQ$_Max_headroom. The additional reserve placed in the Abort_Q is IBQ$_Reserve_Headroom. These values are set to 64 and 2, respectively, for now, but can be changed by DIGITAL at any time. IBQ$_IO_Q_ENTRY is a record whose fields are defined in the next section.

## Request Fields

This section defines the fields of user interest in any outstanding call to an IBQ$ function. Each request is assigned a block of memory called a Q_entry. Each Q_entry is defined by an entry_index which can have a value of 0 to IBQ$_Max_headroom-1 for most requests and IBQ$_Max_headroom to IBQ$_Reserve_Headroom-1 for the IBQ$_ABORT request.

**Device_pointer.Q_entry[entry_index].Q_Link : IBQ$_QUEUE_ENTRY** - This consists of the forward and backward link of the entry in relation to the queue in which it currently resides. Please see the note above for a description of IBQ$_QUEUE_ENTRY.

**Device_pointer.Q_entry[entry_index].DMA_Address : INTEGER** - The Q-bus address (if any) of the DMA area in which the operation is to occur.

**Device_pointer.Q_entry[entry_index].Dma_Length: 0..255** - The length of the DMA area in which the operation is to occur.

**Device_pointer.Q_entry[entry_index].Node_number: [byte] 0..255** - The physical location of the node in which the operation is to occur.

**Device_pointer.Q_entry[entry_index].Task_number: [byte] 0..255** - The task on the node in which the operation is to occur.

**Device_pointer.Q_entry[entry_index].node_command: [byte] 0..255** - The command to be sent to the task on the slave node. For the RAC task the following codes are used:

1 IBQ$_RAC_RESET - 00

2 IBQ$_RAC_CREATE - 01

3 IBQ$_RAC_DELETE - 02

4 IBQ$_RAC_GETID - 03

5 IBQ$_RAC_PROTECT -04

6 IBQ$_RAC_READIO - 05

7 IBQ$_RAC_WRITEIO - 06

8 IBQ$_RAC_UPDATE - 07

9 IBQ$_RAC_UPLOAD - 08

10 IBQ$_RAC_DOWNLOAD - 09

11 IBQ$_RAC_OR - 0A

12 IBQ$_RAC_AND - 0B

13 IBQ$_RAC_XOR - 0C

14 IBQ$_RAC_READ - 0D

14 IBQ$_RAC_RDSTATUS - 0E

16 IBQ$_RAC_WRITE - 0F

17 IBQ$_RAC_WRSTATUS - 00

**Device_pointer.Q_entry[entry_index].Download_address: [word] 0..65535** - Relevant for IBQ$_DOWNLOAD and IBQ$_UPLOAD functions only. It is the address of memory on the node in which these operations are to occur.

**Device_pointer.Q_entry[entry_index].ibq_command: [byte] 0..255** - This field is loaded with a unique field corresponding to the subroutine called and the command to send to the device. The various commands and their corresponding values are as follows (in hex):

1. $SEND - 02
2. $RECEIVE - 01
3. $XCHANGE - 03
4. $TRADE - 07
5. $DOWNLOAD - 06
6. $UPLOAD - 05
7. $RESET - 08
8. $ONLINE - 0D
9. $OFFLINE - 8D
10. $TESTIBQ - CF
11. $ABORT - 80
12. $NOTIFY - 00
13. $SEARCH - C7

All other commands do not communicate with the device process and thus do not require this field to be set.

**Device_pointer.Q_entry[entry_index].Async_requested: BOOLEAN** - Indicates whether this call includes a TRIGGER := parameter. If it did not, then the user requires no notification of operation completion and no completion information is stored in the entry. Upon completion, this entry goes straight from the pending queue to the abort or idle queue without going into the response queue.

Note: The following fields are valid only if

Device_pointer.Q_entry[entry_index].Async_requested = TRUE.

**Device_pointer.Q_entry[entry_index].Async : EVENT** - User defined event accessed through the TRIGGER:= parameter in the subroutine call. The driver process will signal this event upon operation completion.

**IBQ$_IOSB** - The IOSB structure incorporates all the responses from the components of the DECscan system. In particular, these fields are:

```
Driver_reply : INTEGER;
IBQ_reply : [byte] 0..255;
Node_reply : [byte] 0..255;
Node_number: [byte 0..255;
```

The Node_number field is valid only for responses to the IBQ$_NOTIFY command. All commands return entries in the Driver_reply, IBQ_reply and Node_reply fields. For information on the Driver_reply codes, see the Error Values section of this chapter. For information on the IBQ_reply and Node_reply codes, see the Troubleshooting section of the BITBUS Controller chapter.

**Device_pointer.Q_entry[entry_index].Node_Reply** - Reply of the device to the current request. Please refer to the BITBUS Controller Overview chapter under BITBUS Errors for a complete list of these responses.

# IBQ$LD44E DOWNLOAD UTILITY

IBQ$LD44E is a VMS based utility which facilitates the downloading to nodes under VAXELN of tasks created with the ASM44 cross-assembler. You invoke IBQ$LD44E as an image and run it when you want to send data to a node. The process for downloading a task to a BITBUS node under ELN is described in the following sections.

The IBQ01 ELN driver package contains utilities and subroutines for the creation of tasks and data that may be downloaded to BITBUS nodes using the Intel 8044 processor. The tasks or data are generated in 8044 assembly language under VMS.

The steps necessary to successfully perform a download to a node from an ELN application follow.

1. Create assembly language programs containing the tasks or data intended for download. Use a VMS editor.

   If the task is to be started automatically by the driver, the starting address (address of the Initial Task Descriptor (ITD)) must be specified in the final END statement of the program. If the start address is left out, the file will be downloaded only.

2. Assemble these programs with the 8044 cross assembly utility IBQ$ASM44 under VMS. The output of the assembler is an ASCII hex file in Intel hex format for each task.

   To invoke the assembler, first issue the following command:

   ```
   ASM44 :== "$SYS$SYSTEM:IBQ$ASM44"
   ```

   The assembler may now be invoked as described in Chapter 3 of this manual, such as:

   ```
   ASM44 TASK_source,TASK_obj,TASK_list
   ```

   The hex file contents are described in the 8044 cross assembler manual. If a starting address is specified, it will appear in the final EOF record (the last line). Otherwise, the address field there is zero. You may leave out the EOF record completely if you desire to edit the hexadecimal file manually. If it is zero or absent, the file is downloaded, but not started as a task. If you wish to download several tasks to one node and start them all at once, leave out the start address in the assembler programs for all but the last task.

Make sure the ITDs link together properly (see Intel documentation for RMX51). The last task download issues the create task command to RAC, which will follow the ITD chain and start them all. Otherwise, include the start address for each task, and they will be started at download time individually.

3. Convert and package the hex files into a PASCAL data structure using the utility IBQ$LD44E under VMS. The user specifies each hex file to include for the ELN application, and their ordering. IBQ$LD44E creates a PASCAL module IBQ$_DNLD_DAT.PAS which contains the hex files converted to PASCAL data statements, in the order specified. Separator records are included so that the proper task can be located later during download.

   To invoke, issue the following command:

   RUN SYS$SYSTEM:IBQ$LD44E

   IBQ$LD44E requests the path name of each hex file in order. It processes them after the input of each name. If a checksum or other error is found, it will be reported to the user. A CTRL-Z terminates the input. If an invalid file name is entered, the user will be reprompted for the name. The aggregate total of hex-data bytes for all files cannot exceed 65535. This is due to the limitation of initialization size of a single PASCAL data structure.

   Two files are created: IBQ$_DNLD_DAT.PAS and IBQ$_DNLD_DAT.INC. The latter is simply an include file for the former containing the total number of bytes of data. The format of the PASCAL data structure is in records. Each record starts with a descriptor value. The values are: 0 = data record, 1 = start address record and comes after all data records for a file, 2 = EOT records after the data and start address records, 3 = EOT at the end of the structure. Each data record contains the number of bytes, 2 bytes containing the load address, and then the data bytes. A start address contains the number of bytes (2), and 2 bytes containing the start address (ITD address).

4. Compile the module from step 3, using the statement:

   EPASCAL IBQ$_DNLD_DAT

   The output of this is an object module IBQ$_DNLD_DAT.OBJ, which should be included in the LINK of the ELN application.

   Be sure the file IBQ$_DNLD_DAT.INC is also present when compiling, as it is an include file for IBQ$_DNLD_DAT.PAS.

5. Write an application program using the IBQ01 ELN driver subroutine IBQ$_LOAD44. Each call to this routine specifies the file number of the task to download. The ordering is the same as when using IBQ$LD44E (the first task number is 1). Also the node to receive the download data is specified. The same file may be sent to multiple nodes by multiple calls to IBQ$_LOAD44. The task ID assigned to each started task is returned.

   The *IBQELN User's Manual* contains a detailed description of the call to IBQ$_LOAD44. The returned task ID may be used in the application program to specify the task number if communications to the task/node pair is necessary.

   It is necessary to first initialize the driver using IBQ$_INIT prior to calling IBQ$_LOAD44, but no other driver calls are required to perform the download.

6. Link the application with the driver and the module IBQ$_DNLD_DAT.OBJ.

   This will bind the data from the hex files to the ELN application and satisfy an external reference in IBQ$_LOAD44 to the PASCAL data structure in the module IBQ$_DNLD_DAT. IBQ$_LOAD44 traverses the data structure for each call to find the requested file (by ordinal number, as created using IBQ$LD44E), builds TRADE command buffers containing the hex data, sends these down to the node, and then issues an RAC CREATE_TASK command if there is a start address record.

7. Download the ELN application to the target VAX and let it download the 8044 BITBUS tasks and data files to the nodes.

   Status returns provide the application with information as to whether downloads were successful, and also task ID numbers for further communications using other driver commands.

# TROUBLESHOOTING

BITBUS errors and IBQ01 BITBUS controller errors are listed at the end of Chapter 2.

## Common Errors Made

1. TRADE does not have a DMA length of at least 2.
2. Using the same buffer for multiple TRADES and XCHANGES.
3. Failure to create or clear events used in TRIGGER:=parameter.
4. Failure to initialize driver (through IBQ$_INIT) before performing operations.
5. Devices initialized must be defined at build time.

# ERROR VALUES

*Success*

1 - IBQ$_SUCCESS (normal return)

5 - IBQ$_DEVICE_ALREADY_INITIALIZED (alternate success for IBQ$_INIT)

*Error Values*

6 - IBQ$_INVALID_NAME

8 - IBQ$_INSUFFIENT_MEMORY

10 - IBQ$_INVALID_PARAMETER

12 - IBQ$_DRIVER_NOT_AVAILABLE

14 - IBQ$_INVALID_DEVICE

16 - IBQ$_NO_ENTRY_ABORTED

18 - IBQ$_DEVICE_ON_HOLD

20 - IBQ$_NO_DELETE

22 - IBQ$_NO_PROCESS_CREATED

24 - IBQ$_NO_AREA_CREATED

26 - IBQ$_NO_EVENT_CREATED

30 - IBQ$_NO_DEVICE_CREATED

32 - IBQ$_NO_DEVICE_ACCESSED

34 - IBQ$_NO_SEMAPHORE_CREATED

36 - IBQ$_FILE_NOT_FOUND

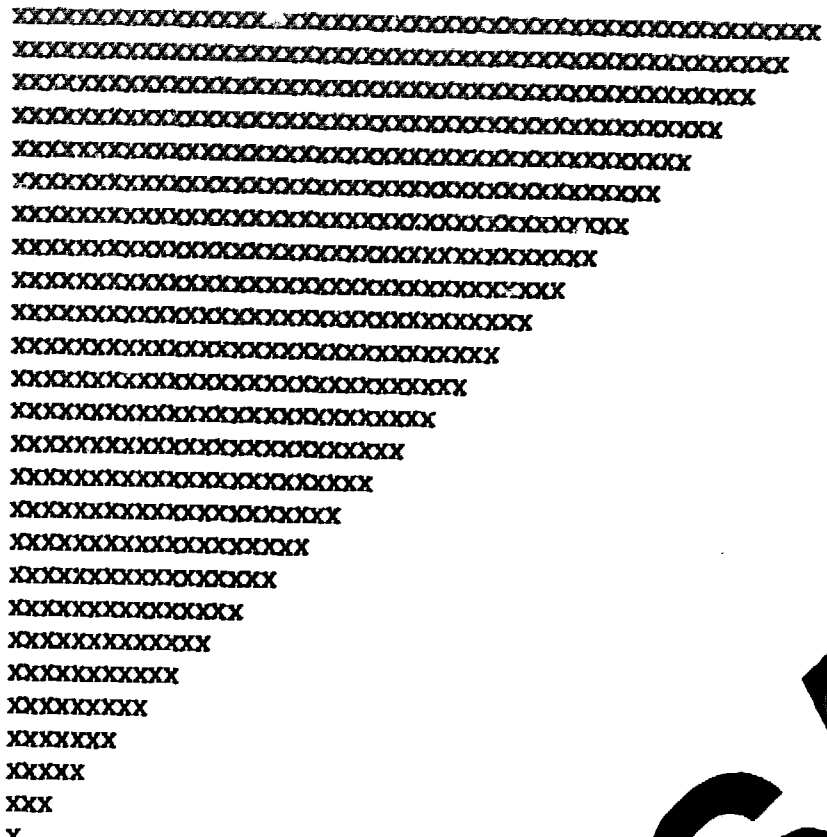
38 - IBQ$_BAD_COMMUNICATIONS

*Returned in the DRIVER_REPLY field of the IOSB*

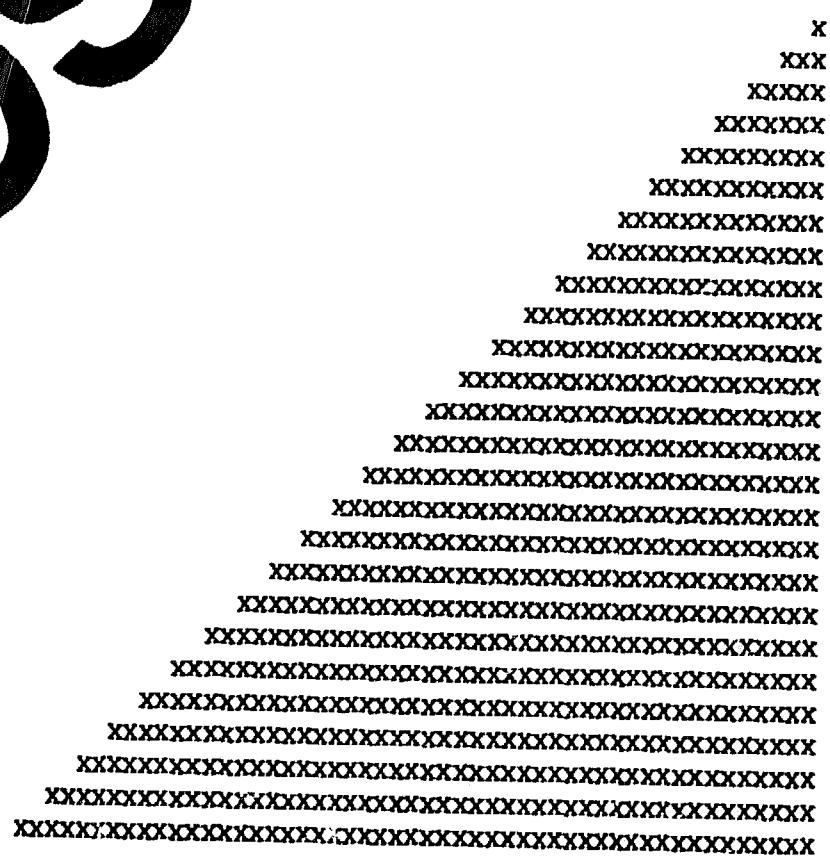
1 - IBQ$_SUCCESS

1002 - IBQ$_ABORTED

PAGE 168 INTENTIONALLY LEFT BLANK



# GLOSSARY



# GLOSSARY

**ASM44** — DATEM IDDCM844 absolute assembler.

**Asynchronous System Trap (AST)** — A software-simulated interrupt to a user-defined service routine.

**BCS** — BITBUS Control Service

**BCS commands** — BITBUS Control Service set of commands supported by the BCS utility.

**BITBUS** — A serial control bus

**BITBUS Control Service** — The interactive interface for the IBQ01.

**BITBUS Master Controller** — An 8-bit microprocessor with built in communications control.

**bit-cell-time** — The time interval required to transfer one bit of data on a serial line.

**command** — An instruction that instructs some part of the system to perform some will defined activity.

**Command Status Register (CSR)** — A portion of memory used to give commands to and obtain information from an I/O device.

**Direct Memory Access (DMA)** — The accessing of memory without CPU intervention.

**DOWNLOAD Utilities** — IBQ$LD44V for VAX/VMS, and IBQ$LD44E for VAXELN.

**DRAM** — Dynamic Random Access Memory.

**EPROM** — Electrically Programmable Read Only Memory.

**interactive interface** — An environment in which an operator and a program communicate mutually. See BITBUS Control Service

**interface** — A shared boundary between two elements within a system.

**master controller** — That part of the IBQ01 which controls the operations between the IBQ01 and the slave nodes.

**master node** — Controls all BITBUS interconnect operations. See node.

**message passing** — The transfer and control of structured data between two tasks.

**node** — A node is a member of a BITBUS interconnect system. Each node requires RS-485 transceivers and an SDLC controller to support the data-link-protocol requirements defined in the BITBUS specifications. There are two types of nodes:

| | |
|---|---|
| **master node** | This node initiates all communications interactions. The IBQ01 acts as a master node and can support up to 250 slave nodes. |
| **slave node** | These nodes respond to commands or requests from the master node. A slave node cannot initiate a transaction without a master node. |

**operation** — The process whereby information is transferred between two elements across an interface.

**Process Input Output (PIO)** — The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software context.

**protocol** — The rules by which information is exchanged across an interface.

**RAC commands** — The set of commands supported by the RAC utility to access and control slave nodes.

**Remote Access and Control (RAC)** — A utility which interfaces between the BITBUS master controller and the BITBUS slave nodes.

**RMX51** — Intel real-time operating system.

**segment** — A segment is a length of wire with BITBUS nodes attached. Any segment can have up to 28 nodes, and depending on speed selection, multiple segments can be attached using repeaters. See Chapter 1 Section 1.5.1 for the relationship of speed to distance and the number of segments.

**slave node** — A replier in the BITBUS system. See node.

**repeater** — A repeater is an RS-485 transceiver pair which rebroadcasts BITBUS messages from one segment of a BITBUS to another. A repeater represents the load of one node but is not intelligent.

**task** — An entity which competes for system resources in order to execute a program.

PAGE 172 INTENTIONALLY LEFT BLANK


INDEX

# INDEX