

Report No. 5000

# C/30 Native Mode Firmware System

## Programmer's Reference Manual

Revision 2  
Microcode  
Version  
M7U13

July 1984

Prepared for:  
Defense  
Communications  
Agency



**BBN Communications Corporation**

BBN Report No. 5000

C/30E Native Mode Firmware System  
Programmer's Reference Manual

Revision 2

Microcode Version m7u13

July 1984

Prepared for:

Defense Communications Agency

**\*\* NOTE \*\***

As before, change bars (\*) in the right margin indicate updates since the last revision. However, given the nature of this update (from 16-bit to 20-bit architecture), obvious changes to formats have not been so flagged since that would result in a constant flood of margin characters.

TABLE OF CONTENTS

CHAPTER 1: Introduction.....1-1

1.1 History.....1-2

1.2 C/30E Hardware.....1-4

1.3 Virtual Machine.....1-7

CHAPTER 2: Basic Instructions.....2-1

2.1 Formats.....2-8

2.2 Conversion.....2-13

2.3 Memory Reference.....2-19

2.4 Shift.....2-25

2.4.1 Logical Shift.....2-28

2.4.2 Arithmetic Shift.....2-31

2.4.3 Rotate.....2-34

2.5 Load/Store.....2-39

2.6 Arithmetic.....2-50

2.7 Logical.....2-60

2.8 Program Control.....2-68

2.9 Processor Control.....2-79

2.10 Queue Manipulation.....2-89

2.11 Stacks.....2-97

2.12 Byte Manipulation.....2-106

2.13 Interprocess Communication.....2-110

CHAPTER 3: Process System.....3-2

3.1 Operation.....3-3

3.1.1 Definitions.....3-4

3.1.2 Environment.....3-8

3.1.3 States.....3-10

3.2 Process Control Block.....3-18

3.3 Instructions.....3-23

3.3.1 Initialization and Termination.....3-25

3.3.2 Execution and Suspension.....3-29

3.3.3 Timing and Miscellaneous.....3-34

CHAPTER 4: I/O System.....4-2

4.1 Device Operation.....4-4

4.1.1 Firmware Drivers.....4-6

4.1.2 Configuration.....4-9

4.1.3 I/O Operation.....4-10

4.2 Process Control Block Device Header.....4-14

4.3 I/O Control Block.....4-16

4.5 Device Descriptions.....4-22

4.5.1 ARPANET 1822.....4-23

4.5.2	BCP/CRC-24 Synchronous.....	4-32
4.5.3	BOP/HDLC Synchronous.....	4-42
4.5.4	Asynchronous Control Port.....	4-55
CHAPTER 5: Firmware Facilities.....		5-1
5.1	Console Commands.....	5-2
5.2	Abnormal Conditions.....	5-7
5.3	Crash Area.....	5-10
5.4	Cassette Operation and New Power Condition.....	5-17
CHAPTER 6: Macrocode Development Support Tools.....		6-1
6.1	File Types and Naming Conventions.....	6-4
6.2	Binary File Conversion.....	6-6
6.3	Writing C/30E Cassettes.....	6-8
6.3.1	Configuration Generation and Tape Writing.....	6-9
6.3.2	MBBCASS.....	6-13
6.4	Sample Terminal Session.....	6-15
6.5	File Directories.....	6-17
CHAPTER 7: Assembler Manual.....		7-1
7.1	Introduction.....	7-2
7.2	Basic Code Generation.....	7-4
7.2.1	Source Line Format.....	7-5

7.2.2	Names and Variables.....	7-6
7.2.3	Labels.....	7-8
7.2.4	Numbers and Radix.....	7-9
7.2.5	Expressions.....	7-10
7.2.6	Instruction Formats.....	7-12
7.2.7	Addressing.....	7-15
7.2.8	Constants.....	7-16
7.2.9	Strings.....	7-19
7.2.10	Location Counter Control.....	7-21
7.2.11	Storage Reservation.....	7-24
7.2.12	Program Sections.....	7-26
7.2.13	Conditional Assembly.....	7-27
7.2.14	START.....	7-28
7.3	PL30 Constructs.....	7-29
7.3.1	Continuation Lines.....	7-30
7.3.2	Unary Operators.....	7-31
7.3.3	Bit Field Opcodes.....	7-32
7.3.4	Off-Page Referencing.....	7-34
7.3.5	Immediate Opcodes.....	7-35
7.3.6	Field Definitions.....	7-36
7.3.7	Structures.....	7-37
7.3.8	Assignment Statements.....	7-38

7.3.9 IF Statements.....7-40

7.3.10 Loops.....7-44

7.3.11 Procedures.....7-45

7.4 Programming Support.....7-46

7.4.1 Listing Format.....7-47

7.4.2 Comments.....7-48

7.4.3 Process Level Control.....7-50

7.4.4 Listing Control.....7-53

7.4.5 INCLUDE Files.....7-55

7.4.6 Symbol Files.....7-56

7.4.7 Symbol Table Manipulation.....7-57

7.4.8 Cross-Reference.....7-58

7.4.9 User Error Messages.....7-59

7.4.10 User Message Files.....7-60

7.5 Pseudo-Op Summary.....7-61

7.6 Assembler Variables Summary.....7-64

CHAPTER 8: References.....8-1

CHAPTER 9: Glossary.....9-1

CHAPTER 10: Index.....10-1

APPENDIX A: Instruction Tables.....A-1

A.1 Ordered by Mnemonic.....A-2

A.2 Ordered by Opcode.....A-6

A.3 Ordered by Instruction Name.....A-10

A.4 Ordered by Type, then Instruction Name.....A-14

APPENDIX B: Sample Configuration.....B-1

B.1 Two Site-Specific Configurations.....B-2

B.2 Macros used in Assembling Configuration Files.....B-3

B.3 Makefile Used in Generating C/30E Cassettes.....B-10

APPENDIX C: Sample Program and Makefile.....C-1

C.1 Sample Program: NMFS Profiling Package.....C-1

C.2 Makefile.....C-48

APPENDIX D: Firmware Crashes.....D-1

APPENDIX E: Firmware Programming.....E-1

E.1 Layout.....E-2

E.2 USYS.....E-4

E.3 INSTR - Instruction Emulation.....E-7

E.4 Extended Console Commands.....E-15

E.5 Crash Handler.....E-16

E.6 Process Manager (PM).....E-18

E.6.1. PCBs, Queues, and the State Word.....E-19

E.6.2 The Attention Mask.....E-21

E.6.3 The Goad Queue.....E-23

E.6.4 Scheduling.....E-28

E.6.5 PM's Prologue.....E-30

E.6.6 PM.CLEAR, NMFS Instruction.....E-31

E.6.7 INH and ENB.....E-33

E.6.8 APR (ADV), DPR (DDV), and DD Entries.....E-34

E.6.8 APR (ADV), DPR (DDV), and DD Entries.....E-34

E.6.9 XDV (ACT) and PDV (HPK).....E-35

E.6.10 GPR (SPK) and DUMMY.....E-36

E.6.11 TPR (TDV).....E-37

E.6.12 SPR (RFI).....E-38

E.6.13 IO.CLOCK.....E-40

E.7 I/O Library.....E-41

E.8 Microcode Device Drivers.....E-45

E.8.1 The Driver Entry Table and IO.DD.CALL.....E-47

E.8.2 Bidirectional Devices and Their Drivers.....E-49

E.8.3 Organizational Suggestions.....E-50

E.8.4 Source-Level Configuration.....E-56

E.9 The Microassembler.....E-58

## 1 Introduction

This document is a programmer's reference manual for the BBN \*  
C/30E computer running the Native Mode Firmware System (NMFS) and \*  
utilizing the full 20-bit word format of the Microprogrammable \*  
Building Block (MBB) processor. Previous versions of the NMFS \*  
microcode have been restricted to operating on data and addresses \*  
which used only the low order 16 bits of the 20-bit C/30 \*  
architecture. \*

This manual contains all the necessary information for  
designing, writing, and executing application software, on-line  
debugging, and generally operating the C/30E and its software  
support environment. The document also contains supplementary  
information required for performing any enhancements to the  
firmware system.

## 1.1 History

The C/30E NMFS computer is the latest in a series of engines \*  
used for executing the ARPANET IMP program and similar high-speed \*  
communications applications. The first member of this series was \*  
the Honeywell DDP-516 minicomputer, which was followed by the \*  
less expensive but program-compatible Honeywell 316 [1]. When \*  
the 516/316 line became obsolete, BBN created firmware for its \*  
Microprogrammable Building Block (MBB), and appropriate external \*  
device interfaces, to directly execute the existing 516/316-based \*  
software. This MBB-based engine was designated the C/30. Though \*  
it was based on the 20-bit MBB processor, it was restricted to \*  
operating as if it were a 16-bit machine. The C/30 faithfully \*  
emulated the instruction set and I/O structure of the 516/316 \*  
used by the existing application software. The subsequent C/30 \*  
based NMFS firmware retained most of the basic (non-I/O) \*  
instructions inherited from the 516/316, but provided a superior \*  
process and I/O structure optimized for real-time communications \*  
applications. Later versions of NMFS augmented the basic \*  
instruction set repertoire with operations to manipulate queues, \*  
a stack, unsigned integers, byte pointers, semaphores, locks, and \*  
"mailboxes". Finally, the present version uses the C/30 hardware \*

## History

with an extended 256k word memory, converts most instructions to \*  
operate on 20-bit entities, adds instructions for transformation \*  
between 16-bit and 20-bit entities, and replicates some \*  
instructions in 16-bit only variants. All I/O, it should be \*  
noted, is still performed in 16-bit words; the 20-bit \*  
architecture simply allows a wider addressing and calculating \*  
range. \*

Throughout this manual, the firmware is also referred to as  
microcode, and firmware entities in general are prefixed by  
"micro" (e.g. micromemory, microregisters, etc.). The  
application level software (the software of the emulated machine)  
is also referred to as macrocode, and software entities (where  
necessary to avoid ambiguity) are prefixed by "macro" (e.g.  
macromemory).

## 1.2 C/30E Hardware

The C/30E hardware consists of a basic MBB processor, special instruction decoder and address adaptors, application program memory, asynchronous device ports for a cassette drive and a control terminal, and I/O boards which support a variety of devices. The MBB is further described in [2].

The C/30E execution engine is composed of a processor board, \*  
a memory board, and up to five I/O boards. The processor board \*  
contains a 20-bit word length CPU with a .135 microsecond \*  
microcycle time, 8k of 32-bit word RAM microcode storage (plus \*  
some ROM for bootload), 1k of 12-bit word dispatch memory, and \*  
the instruction decoder and address adaptors. The processor \*  
board also contains a microregister file of 1k words, and two \*  
full-duplex asynchronous interfaces capable of speeds up to 19.2 \*  
kilobits/second. The memory board contains 256k words (all 20 \*  
bits used) of EDAC macromemory storage. Macromemory references \*  
(load or store) require three microcycles. With the exception of \*  
the microcode ROM, all memory is volatile to power failures. \*

A TU58 cassette drive is connected to one of the processor board asynchronous interfaces, and a full-duplex ASCII terminal

(hardcopy or CRT) to the other. When the C/30E is powered on or explicitly restarted, control is passed to the ROM microcode. The ROM bootstrap loader then reads from the cassette and reloads any segments of memory (micro and macro) indicated by the load formats on the cassette. Once the load is completed, control may be passed to the emulation microcode, which begins executing the macrocode.

Each I/O board can be one of three types: MII, MSYNC, or MTI. The MII contains four full-duplex ARPANET 1822 [3],[4] interfaces, six full-duplex BCP (byte-control protocol) synchronous interfaces, and a 20-light (LED) display visible from the front of the C/30E. The MSYNC contains sixteen full-duplex BOP (bit oriented protocol) synchronous interfaces and a 20-light display. The MTI contains one full-duplex ARPANET 1822 interface, one full-duplex BOP synchronous interface, and a 20-light display. The MTI also contains 32 full-duplex asynchronous/BCP synchronous interfaces not supported by the C/30E NMFS microcode. In addition, though not supported by the C/30E NMFS microcode, the MSYNC BOP interfaces can also be run in BCP mode, the MII BCP interfaces can be run in asynchronous mode, and the processor board asynchronous interfaces can be run in BCP

mode. The hardware device speed limitations are 200 kilobits/second for ARPANET 1822, 800 kilobits/second for BCP and 1 megabit/second for BOP. Firmware and software limitations are considerably lower.

### 1.3 Virtual Machine

The C/30E NMFS virtual machine is defined by the firmware system. This machine is the primary focus of this manual, and the target for any application software. This section describes the virtual hardware, and chapters 2 through 4 describe the operation of the virtual machine.

The virtual hardware consists of a 20-bit word, two's complement processor with an addressing range of 1 megaword. The memory cycle time is 3 microcycles (.405 microseconds), the minimum instruction execution time (NOP) is 6 microcycles (.81 microseconds), and a typical memory reference instruction execution time (ADD) is 10 microcycles (1.35 microseconds).

There are several programmable registers:

- A - a 20-bit accumulator and general-purpose register.
- B - a 20-bit register used as a secondary working register. It is concatenated with A during double-word (40-bit) operations.
- C - a 1-bit register representing the carry/borrow bit for arithmetic operations, and the last bit shifted in shift operations.
- O - a 1-bit register representing the carry/borrow bit for unsigned arithmetic operations, and a copy of the

last bit shifted out during shifting operations.

- X - a 20-bit index register and secondary working register. This register is always mapped as macromemory location 0.
- SP - a 20-bit stack pointer containing the macromemory address used by stack operations.
- PC - a 20-bit program counter containing the address of the next macroinstruction to be executed.
- LIT - a 20-bit write-only register which controls the LED display on the front of the C/30E.
- RTC - a 40-bit read-only register which is incremented by the firmware every 100 microseconds.

The virtual hardware has two operating modes, basic and NMFS. In the basic mode, the macrocode is a single process in sole possession of the virtual hardware, and has no I/O capability. In the NMFS mode, the macrocode consists of an arbitrary number of processes, each of which possesses an independent context which includes the virtual hardware. These processes can be arbitrarily assigned to 32 priority levels, where level 0 has the highest priority and 31 the lowest.

Processes may be optionally associated with I/O devices via device types and device handles. Full duplex I/O operation requires two processes, one for input and one for output. The device types are: asynchronous control port, ARPANET 1822, 24-

bit CRC BCP synchronous, and BOP synchronous. The device handles refer to specific interfaces and are a function of the physical hardware and firmware configuration.

Processes may be activated by software or device interrupts, or by per-process timeouts. The priority system normally enforces preemption of lower priority processes by higher ones, but this can be disabled under program control.

The firmware system interleaves the emulation of macroinstructions with the servicing of microinterrupts. I/O hardware interfaces and the system clock generate real microinterrupts and the process scheduling microcode generates process pre-emptions (macrointerrupts). The firmware always processes all microinterrupts before continuing with macroinstruction emulation. The firmware polls for microinterrupts after each emulated macroinstruction, and in some cases (e.g. BLT, see 2.5) even during macroinstruction emulation. Thus, the elapsed execution time for a macrocode sequence has as a lower bound the sum of execution times of the individual macroinstructions, but may in practice be greater, due to I/O activity.

The C/30E operation is controlled by the combination of ROM microcode and the contents of the cassette loaded into the TU58 drive. This combination not only defines the virtual machine, it also defines a user-accessible firmware environment, described in chapter 5. One of the functions of this user-accessible firmware environment is to provide support for the virtual machine in operational aspects (such as debugging) not directly related to the execution of the macrocode, process, or I/O system. Aspects of this support can be controlled by the application through parameters written on the cassette.

When a C/30E is reset by a power failure, control is passed to the ROM microcode. The ROM microcode then performs a bootstrap load from the cassette of the RAM microcode and other firmware memory. The firmware is not application dependent, except for a few parameters discussed below, but is hardware-configuration dependent. The cassette may also contain an arbitrary amount of application-dependent macromemory contents which are loaded during the bootstrap. If the C/30E is manually reset, or initiates a reset internally, it first checks a bootload inhibit flag in micromemory. If the flag is not inhibiting the bootload, the C/30E goes through the same process

as described above for a power failure. If the flag is inhibiting the bootload, control is passed to the user firmware environment, as described in chapter 5.

The firmware parameters available to the application are loaded into micromemory during the bootload. These parameters are:

**Bootload Inhibit (see section 5.4):**

The flag controlling whether to automatically reboot or pass control to the user firmware environment upon reset.

**Startup Address (see section 6.3):**

The microcode or macrocode address to which control is transferred after the bootload completes. If a macrocode address is specified, the microcode is started at the normal macroemulation address with the PC set to the specified macrocode address.

**Crash Area (see section 5.3):**

The macrocode address of the block of sequential locations used to save microcode variables after a crash (described in 5.3).

Once control is passed to the macrocode, the user firmware environment can regain control as described in chapter 5.

If, during macrocode execution, the firmware detects a data structure inconsistency, illegal instruction or instruction

parameter, or other unrecoverable error, the virtual machine "crashes." This exception condition is described in 5.2, and a full list of crash types is given in Appendix D. When the virtual machine crashes, all further macrocode execution is suspended and the microcode copies its state variables and other firmware information into the designated macromemory crash area. The microcode then initiates an internal reset. As discussed above, this will result either in a bootload from the cassette, or control passed to the user firmware environment. In either case, a dump of macromemory will contain the data necessary for explaining the cause for the crash.

## 2 Basic Instructions

This chapter describes the macrocode word formats, addressing modes, and instructions which are relevant to the virtual hardware. They are all available in basic (non-NMFS) mode and are independent of the process and I/O structure.

## Instruction Table

This table provides a summary, sorted by section and then by mnemonic, of the NMFS instructions contained in this chapter, and gives the page number on which complete information for each instruction can be found.

## Section 2.2 Conversion

Mnemonic.	OpCode	Instruction Name	Type	Cycles	Page
CAE	0001102	Clear A Extension Bits	CONV	6	2-15
CXE	0001103	Clear X Extension Bits	CONV	7	2-15
EAB	0001041	End Around Borrow	CONV	15	2-18
EAC	0001040	End Around Carry	CONV	15	2-18
SEA	0001100	Sign Extend A	CONV	8	2-14
SEI	0100023	Skip if Extension Insig.	CONV	11/13	2-16
SEN	0101022	Skip if Extension Nonzero	CONV	9	2-16
SES	0101023	Skip if Extension Significant	CONV	11/13	2-16
SEX	0001101	Sign Extend X	CONV	9	2-14
SEZ	0100022	Skip if Extension Zero	CONV	9	2-16
TAB	0001200	Truncate A and B	CONV	20	2-17
XAB	0001201	Extend A and B	CONV	15	2-17

## Section 2.4 Shift

Mnemonic.	OpCode	Instruction Name	Type	Cycles	Page
ALR16	0041700	A (Register) Left Rotate 16	SHIFT	16<141	2-38
ALR20	0041600	A (Register) Left Rotate 20	SHIFT	16<169	2-36
ALS	0041500	Arithmetic Left Shift	SHIFT	16<155	2-33
ARR16	0040700	A (Register) Right Rotate 16	SHIFT	16<140	2-37
ARR20	0040600	A (Register) Right Rotate 20	SHIFT	16<168	2-35
ARS	0040500	Arithmetic Right Shift	SHIFT	17<133	2-32
LGL	0041400	Logical Left Shift	SHIFT	16<134	2-29

LGR	0040400	Logical Right Shift	SHIFT	16<130	2-28
LLL	0041000	Long Left Logical Shift	SHIFT	17<234	2-30
LLR16	0041300	Long Left Rotate 16	SHIFT	17<254	2-38
LLR20	0041200	Long Left Rotate 20	SHIFT	17<310	2-36
LLS	0041100	Long Left Arithmetic Shift	SHIFT	28<287	2-33
LRL	0040000	Long Right Logical Shift	SHIFT	17<231	2-29
LRR16	0040300	Long Right Rotate 16	SHIFT	17<253	2-37
LRR20	0040200	Long Right Rotate 20	SHIFT	17<309	2-35
LRS	0040100	Long Right Arithmetic Shift	SHIFT	28<244	2-32

## Section 2.5 Load/Store

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
-----	-----	-----	-----	-----	-----
BLT	0000031	Block Transfer Memory	LD/ST	15 + 7n	2-48
CAL	0141050	Clear A Register Left Half	LD/ST	6	2-44
CAR	0141044	Clear A Register Right Half	LD/ST	6	2-44
CLR	0141144	Copy A Register Lt to Rt 1/2	LD/ST	7	2-46
COB	0140504	Complement Overflow Bit	LD/ST	6	2-47
CRA	0140040	Clear A Register	LD/ST	6	2-43
CRL	0141250	Copy Register Rt to Lt Byte	LD/ST	7	2-46
IAB	0000201	Interchange A and B Registers	LD/ST	9	2-43
ICA	0141340	Interchange A Register Halves	LD/ST	7	2-44
ICL	0141140	Interchange & Clr A (Lt 1/2)	LD/ST	8	2-45
ICR	0141240	Interchange & Clr A (Rt 1/2)	LD/ST	8	2-45
IMA	0026000	Interchange Memory & A Reg	LD/ST	11 + MR	2-41
LAI	0141700	Load A Indirectly thru Self	LD/ST	9	2-40
LDA	0004000	Load A Register	LD/ST	6 + MR	2-40
LDX	0072000	Load X Register	LD/ST	10 + MR	2-42
LXA	0141714	Load X Indirectly thru A	LD/ST	13	2-43
LXI	0141704	Load X Indirectly thru Self	LD/ST	13	2-42
RCB	0140200	Reset C Register (Bit)	LD/ST	7	2-46
ROB	0140604	Reset Overflow Bit	LD/ST	7	2-47
SCB	0140600	Set C Register (Bit)	LD/ST	7	2-46
SOB	0140204	Set Overflow Bit	LD/ST	7	2-47
STA	0010000	Store A Register	LD/ST	10 + EA	2-41
STX	0032000	Store X Register	LD/ST	10 + EA	2-41
SXA	0141614	Store X Indirectly thru A	LD/ST	13	2-42
TRB	0000041	Transfer Memory Backwards	LD/ST	11 + 7n	2-49

## Section 2.6 Arithmetic

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
ABX	0141000	Add B Register to X Register	ARITH	8	2-57
ACA	0141216	Add C Register to A Register	ARITH	11	2-53
ADD	0014000	Add Memory to A Register	ARITH	7 + MR	2-51
AOA	0141206	Add One to A Register	ARITH	7	2-54
AOX	0141014	Add One to X Register	ARITH	13	2-55
AVA	0141110	Add Overflow to A Register	ARITH	9	2-58
CHK	0000032	Checksum Block of Memory	ARITH	14 + 5n	2-56
ECK	0000202	End Around Checksum	ARITH	15 + 7n	2-57
SOA	0141306	Subtract One from A Register	ARITH	9	2-54
SOX	0141114	Subtract One from X Register	ARITH	13	2-55
SUB	0016000	Subtract Memory from A Reg	ARITH	8 + MR	2-52
SVA	0141310	Subtract Overflow from A Reg	ARITH	12	2-59
TCA	0140407	Two's Complement A Register	ARITH	7	2-52

## Section 2.7 Logical

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
ANA	0006000	AND Memory to A Register	LOGIC	6 + MR	2-61
CCRO	0001032	Convert & Clear Rightmost One	LOGIC	10 + f(A)	2-65
CHS16					
CHS20	0140024	Change (Complement) Sign of A	LOGIC	6	2-63
CMA	0140401	Ones Complement A Register	LOGIC	6	2-62
CSA16	0000053	Copy Sign & Set A's Sign to +	LOGIC	10	2-65
CSA20	0140320	Copy Sign & Set A's Sign to +	LOGIC	10	2-64
CXB	0140510	Complement Indexed Bit	LOGIC	9	2-67
ERA	0012000	Exclusive OR Memory to A Reg	LOGIC	6 + MR	2-61
FFO	0000033	Find First One	LOGIC	9 + 29	2-65
RXB	0140210	Reset Indexed Bit	LOGIC	9	2-66
SSM16	0000051	Set Sign of A Register Minus	LOGIC	6	2-63
SSM20	0140500	Set Sign of A Register Minus	LOGIC	6	2-63
SSP16	0000050	Set Sign of A Register Plus 16	LOGIC	6	2-62
SSP20	0140100	Set Sign of A Register Plus 20	LOGIC	6	2-62
SXB	0140610	Set Indexed Bit	LOGIC	9	2-66

## Section 2.8 Program Control

Mnemonic.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
CAS	0022000	Compare A Reg to Mem & Skip	CNTRL	11 + MR	2-70
IRS	0024000	Increment, Replace & Skip	CNTRL	10 + MR	2-71
JMP	0002000	Unconditional Jump	CNTRL	7 + JA	2-69
JST	0020000	Jump & Store Program Counter	CNTRL	10 + EA	2-70
NOP	0101000	No Operation	CNTRL	6	2-71
SGT16	0100202	Skip if A Register > 0 16	CNTRL	9/10	2-75
SGT20	0100401	Skip if A Register > 0 20	CNTRL	9/10	2-74
SKP	0100000	Unconditional Skip	CNTRL	6	2-71
SLE16	0101202	Skip if A Reg LT or EQ Zero 16	CNTRL	9/10	2-75
SLE20	0101401	Skip if A Reg LT or EQ Zero 20	CNTRL	9/10	2-75
SLN	0101100	Skip if Low Bit of A Reg Non0	CNTRL	8	2-76
SLZ	0100100	Skip if Low Bit of A Reg Zero	CNTRL	8	2-76
SMI16	0101201	Skip if A Reg Minus 16	CNTRL	8	2-74
SMI20	0101400	Skip if A Reg Minus 20	CNTRL	8	2-74
SNZ16	0101200	Skip if A Register Nonzero 16	CNTRL	8	2-73
SNZ20	0101040	Skip if A Register Nonzero 20	CNTRL	8	2-72
SOC	0100021	Skip if Overflow Clear	CNTRL	9	2-78
SOS	0101021	Skip if Overflow Set	CNTRL	9	2-77
SPL16	0100201	Skip if A Reg Plus (>=0) 16	CNTRL	8	2-73
SPL20	0100400	Skip if A Reg Plus (>=0) 20	CNTRL	8	2-73
SRC	0100001	Skip if Reset C Register	CNTRL	10	2-76
SSC	0101001	Skip if Set C Register	CNTRL	10	2-77
SZC	0100041	Skip if A Reg 0 & Reset C Reg	CNTRL	8/13	2-77
SZE16	0100200	Skip if A Register Zero 16	CNTRL	8	2-72
SZE20	0100040	Skip if A Register Zero 20	CNTRL	8	2-72
SZO	0100042	Skip if A Reg 0 & Ovrflo Reset	CNTRL	10	2-78

## Section 2.9 Processor Control

Mnemonic.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
ERB	0000200	Retrieve Error Bits	PCTRL	14	2-85
ERC	0000105	Error Light Clear	PCTRL	14	2-84
ERR	0000101	Interrogate Memory Errors	PCTRL	23	2-84
HLT	0000000	Halt the Processor	PCTRL	5	2-80
LITES	0000011	Write LIT (LITES) Register	PCTRL	21	2-81
MEMHI	0000012	Read Memory High Bound	PCTRL	9	2-81

RDCLOK	0000010	Read RTC Register	PCTRL	11	2-80
RSM	0000013	Read Special Memory	PCTRL	12+f(B)	2-87
VER	0000100	Return Version Number	PCTRL	9	2-83
WATCH	0000761	Watch On or Off	PCTRL	11/15	2-86
WSM	0000021	Write Special Memory	PCTRL	12+f(B)	2-88

## Section 2.10 Queue Manipulation

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
DEQ	0000022	Dequeue First Item from Queue	QUEUE	23/48	2-95
ENQ	0000002	Enqueue A New Item on Queue	QUEUE	46	2-94
RMQ	0000042	Rmv Specified Item from Queue	QUEUE	56	2-96

## Section 2.11 Stacks

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
CALL	0034000	Subroutine CALL Using Stack	STACK	11 + JA	2-100
CASP	0100011	Copy A Register to SP	STACK	7	2-102
CSPA	0100012	Copy SP to A Register	STACK	7	2-102
CSPX	0100013	Copy SP to X Register	STACK	10	2-103
ITS	0100010	Increment Top of Stack	STACK	13	2-102
LAT	0141510	Load A from Top of Stack	STACK	12	2-104
LXT	0141504	Load X from Top of Stack	STACK	13	2-105
POP	0036000	Pop Memory Contents off Stack	STACK	14 + EA	2-101
POPA	0101003	Pop A Reg Contents off Stack	STACK	9	2-104
PUSH	0030000	Push Memory Contents onto Stx	STACK	10 + MR	2-100
PUSHA	0101002	Push A Reg Contents onto Stk	STACK	11	2-103
RETN	0100002	Subroutine Return Using Stack	STACK	9	2-101
SAT	0141610	Store A in Top of Stack	STACK	12	2-103
SRETN	0100003	Sub Skip Return Using Stack	STACK	10	2-101
SXT	0141604	Store X in Top of Stack	STACK	12	2-104

## Section 2.12 Byte Manipulation

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
BLI	0001401	Byte Load and Increment	BYTE	22	2-108
BLO	0001400	Byte Load	BYTE	19	2-108
BSI	0001403	Byte Store and Increment	BYTE	29	2-109
BST	0001402	Byte Store	BYTE	25	2-108

## Section 2.13 Interprocess Communication

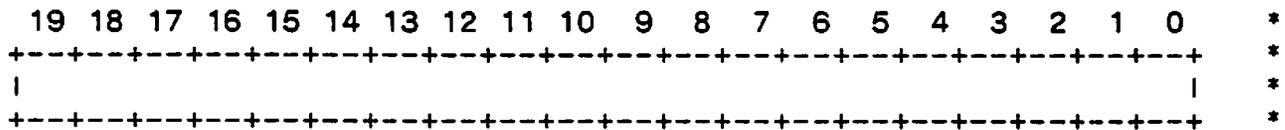
Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
LOK	0001011	Obtain Lock	IPCOM	14	2-112
P	0001013	Decrement Semaphore (Probeer)	IPCOM	14	2-115
RCV	0001012	Receive	IPCOM	14	2-115
SND	0001022	Send Trap	IPCOM	14	2-114
ULK	0001021	Release Lock	IPCOM	15	2-113
V	0001023	Increment Semaphore (Verlaat)	IPCOM	15	2-116

FORMATS

2.1 Formats

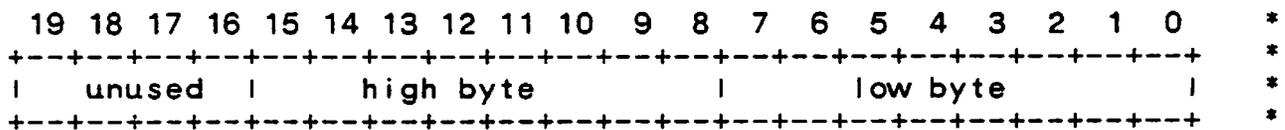
The various views of the 20-bit word format are:

1) general word format, unsigned integer format, address format.



There are 20 bits of data, with all bits having logical significance; i.e. all bits are meaningful. When regarded as an integer, the range of values is 0 ... 2<sup>20</sup>-1. When addressing, each different possible 20-bit value addresses a different word in a 20-bit physical address space, although most machines will probably have less than the full megaword of possible memory.

2) data format for byte storage:

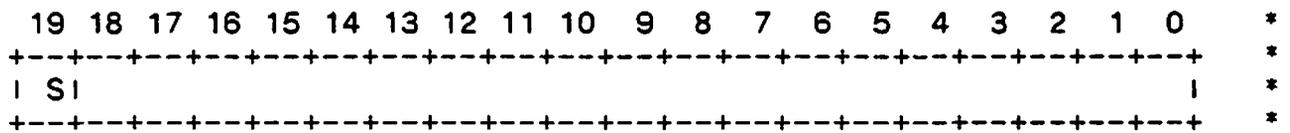


Bits 15-8 are referred to as the left or high order byte (HB), and bits 7-0 as the right or low order byte (LB). As data is transferred into or out of memory (from/to I/O devices), the

FORMATS

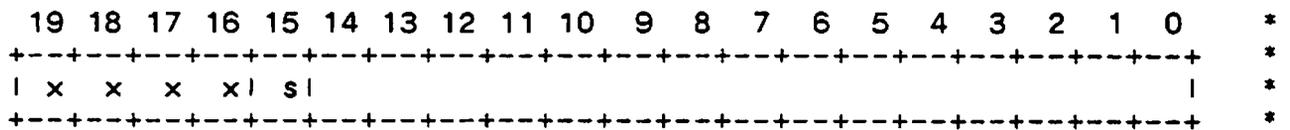
order of byte and bit transfer is through sequentially increasing \*  
 16-bit words, and first left then right halves for byte \*  
 operations, or bits 15 to 0 for bit operations. \*

3) data storage for signed integers: \*



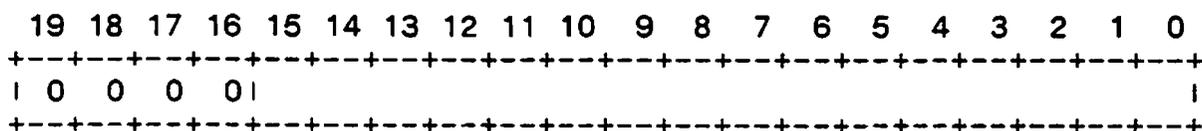
The word contains a 20-bit two's complement integer. The \*  
 range of values is  $-2^{19} \dots 2^{19}-1$ . \*

4) data storage for signed 16-bit integers: \*



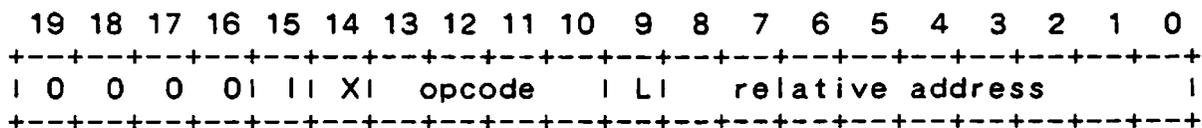
"s" is the sign bit. "x" is four copies thereof, i.e. there are \*  
 4 bits of sign extension. Bits 15 through 0 contain a 16-bit \*  
 two's complement integer, with the range of values  $-2^{15} \dots$  \*  
 $2^{15}-1$ . \*

5) unsigned 16-bit integer format



Bits 19 through 0 contain a 20-bit binary number with the range of value 0 ... 2<sup>16</sup>-1.

6) The instruction word format is organized around the requirements for the memory reference instructions, which have the format:

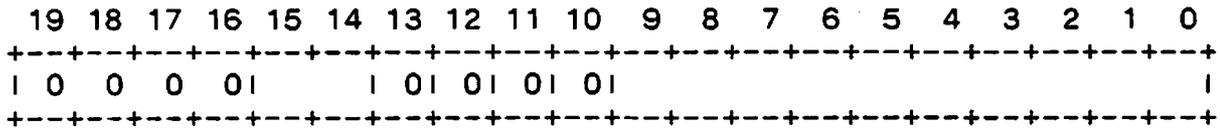


I specifies indirect reference, X specifies post-indexing, and L specifies local page base address. The addressing scheme is described in section 2.3.

FORMATS

7) instruction format, miscellaneous instructions: \*

All other instructions are the subset of the above specified  
by opcode = 0. \*



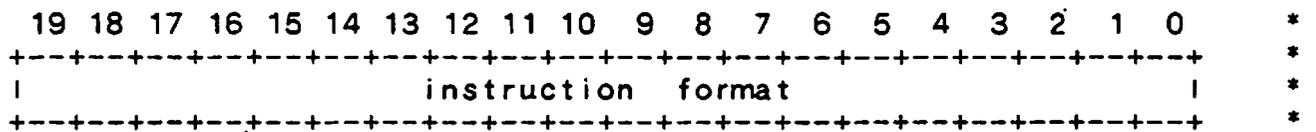
The high four bits of the instruction word are currently  
unused. The opcode consists of bits 15, 14, and 9 through 0. \*

The A, B, X registers of the machine are full 20-bit  
registers whose contents may be viewed in any of the above ways  
depending on usage. The SP and PC registers will always be  
viewed as containing a 20-bit address. The X register will  
normally be viewed as containing a 20-bit address. The O and C  
registers are 1 bit in size. Note that after arithmetic  
instructions and left shifts, the meaning of O and C is affected  
differently by the 20-bit version of the instruction than by the  
16-bit version. \*

FORMATS

The description of each macrocode instruction in this manual \*  
 contains the assembler mnemonic, the machine code (in octal \*  
 notation), the name of the instruction, the execution time (in \*  
 microcycles), the instruction format, a discussion of instruction \*  
 operation, and a list of possible machine error conditions caused \*  
 by instruction execution. This information is presented in the \*  
 following form: \*

ASSEMBLER MNEMONIC	NAME	EXECUTION TIME	*
MACHINE CODE			*



[Discussion of instruction operation.] \*

Crashes: \*

[List of possible machine error conditions caused \*  
 by instruction execution.] \*

Note: Execution times sometimes change with microcode releases. \*  
 The times given should be considered upper bounds. \*

## Instruction Table for Section 2.2

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page	
=====	=====	=====	=====	=====	=====	
CAE	0001102	Clear A Extension Bits	CONV	6	2-15	*
CXE	0001103	Clear X Extension Bits	CONV	7	2-15	*
EAB	0001041	End Around Borrow	CONV	15	2-18	*
EAC	0001040	End Around Carry	CONV	15	2-18	*
SEA	0001100	Sign Extend A	CONV	8	2-14	*
SEI	0100023	Skip if Extension Insig.	CONV	11/13	2-16	*
SEN	0101022	Skip if Extension Nonzero	CONV	9	2-16	*
SES	0101023	Skip if Extension Significant	CONV	11/13	2-16	*
SEX	0001101	Sign Extend X	CONV	9	2-14	*
SEZ	0100022	Skip if Extension Zero	CONV	9	2-16	*
TAB	0001200	Truncate A and B	CONV	20	2-17	*
XAB	0001201	Extend A and B	CONV	15	2-17	*

CONVERSION

2.2 Conversion

The 12 instructions in this new group of instructions are \*  
 for dealing with 16-bit data and conversion to the IMPs new 20- \*  
 bit internal word format. \*

SEA Sign Extend A 8 \*  
 0001100 - - - \*

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+---+																			
1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
+---+																			

Bits 19 through 16 of A are set to ones if bit 15 is 1. \*  
 Otherwise, they are set to 0. The result in A is a 20-bit \*  
 signed quantity which is algebraically equal to the 16-bit \*  
 signed quantity originally contained in bits 15 through 0. \*

SEX Sign Extend X 9 \*  
 0001101 - - - \*

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+---+																			
1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	1
+---+																			

Bits 19 through 16 of X are set to ones if bit 15 is 1. \*  
 Otherwise, they are set to 0. The result in X is a 20-bit \*  
 signed quantity which is algebraically equal to the 16-bit \*  
 signed quantity originally contained in bits 15 through 0. \*

CONVERSION

CAE Clear A Extension Bits 6 \*  
 0001102 - - - \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 | *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Bits 19 through 16 of A are set to zero. Bits 15 through 0  
 of A are left unchanged. \*

CXE Clear X Extension Bits 7 \*  
 0001103 - - - \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 | *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Bits 19 through 16 of X are set to zero. Bits 15 through 0  
 of X are left unchanged. \*

SES Skip if Extension Significant 11/13 \*  
 0101023 - - - \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1 | *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

If any of bits 19 through 16 in A has a value different  
 from that of bit 15, the next sequential instruction is  
 skipped. If this condition prevails then the 20-bit signed  
 quantity in A is too large to be expressed as a 16-bit  
 signed quantity. \*

SEI Skip if Extension Insignificant 11/13 \*  
 0100023 - - - \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 | *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

If all of bits 19 through 16 in A contain the same value the same as that of bit 15, the next sequential instruction is skipped. If this condition prevails then the 20-bit signed quantity in A may also be expressed as a 16-bit signed quantity. \*

SEZ Skip if Extension is Zero 9 \*  
 0100022 - - - \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 | *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

If each of bits 19 through 16 is zero, the next sequential instruction is skipped. \*

SEN Skip if Extension is Nonzero 9 \*  
 0101022 - - - \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 | *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

If any of bits 19 through 16 is nonzero, the next sequential instruction is skipped. \*

CONVERSION

TAB	Truncate A and B	20	*
0001200	- - -		*

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	*
+---																				*
1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	*
+---																				*

The contents of A and B, presumed to be a 40-bit number, with the high order part in A, is truncated to a 32-bit number. Bits 31 through 16 of this result are placed in bits 15 through 0 of A, and bits 15 through 0 remain undisturbed in B. Bits 19 through 16 of A and B are set to zero. If bits 39 through 32 of the input were significant then C is set to 1, otherwise, C is cleared to 0.

XAB	eXtend A and B	15	*
0001201	- - -		*

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	*
+---																				*
1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	*
+---																				*

The inverse transformation from TAB. A and B are expected to contain the high and low parts, respectively, of a 32 bit quantity, as two 16-bit words. These bits are rearranged into the low 32 bits of a 40-bit double precision number with the low 20 bits in B. Bits 39 through 32 are set to a copy of bit 31, i.e sign extended.



## MEMORY REFERENCE

## 2.3 Memory Reference

There are 16 memory reference instructions, of which 14 are indexable and 2 use the index bit as part of the operation code. These instructions can all reference relative or absolute locations.

Macrocode memory is logically divided into 512-word pages. The nine-bit relative address field of the memory reference instruction is sufficient to address all the words on a page. If the L bit is 0, the relative address is converted to an absolute address of one of the 512 words on page zero (octal locations 0 to 777). If the L bit is 1, the relative address is converted to an absolute address of one of the words on the local page, the page that the instruction itself is on. The address is relative to the beginning of the page; thus, if the instruction is at octal address 35421, and the relative address field of the instruction is octal 333, then the absolute address of the reference is octal 35333.

One level of indirect addressing is used to access locations outside of the local page and page zero. If the I bit of a memory reference instruction is 1, reference is made to a

## MEMORY REFERENCE

location which contains a 20-bit absolute address, which is in turn used for the memory reference. The use of indirect addressing is also required to move from the end of one page to the beginning of a subsequent page. It is no longer possible, in the 20-bit C/30E architecture, to "fall off the end of a page".

Indexing is used to access locations with a computed relative displacement with respect to another memory location. If the X bit of a memory reference instruction is 1, the displacement in the 20-bit index register is added to the 20-bit absolute address. The modulo  $2^{20}$  result becomes the new absolute address. The displacement in the index register may be thought of as a signed quantity because of the modulo  $2^{20}$  result.

These three aspects of the addressing scheme are combined into the following algorithm for determining the effective address (the location of the operand) for a memory reference instruction.

## MEMORY REFERENCE

1. Set EA (effective address) to the value of the relative address field of the instruction.
2. If  $L = 1$ , add to bits 19-9 of EA the value of bits 19-9 of the current instruction address.
3. If  $I = 1$ , set EA to the value in the location previously addressed by EA.
4. (Indexable instructions only) If  $X = 1$ , add the value in the index register to EA, modulo 20 bits.
5. EA is now the absolute address of the location containing the operand.

There are two conceptual variations in the use of indexing. In the first type, the contents of the index register are viewed as a positive or negative computed displacement within a vector whose elements are typically equivalent. Steps 1-3 above are used to determine an absolute reference to the vector. As an example of the first type, to access the  $n$ th element of an array located on a different macromemory page from the instruction; the index register would contain the value  $n$ , the  $X$ ,  $L$ , and  $I$  bits would be 1, and the relative address would refer to an on-page location containing the absolute address of the beginning of the array. In the second type, the  $L$  and  $I$  bits are always zero, and the contents of the index register are viewed as the absolute

## MEMORY REFERENCE

address of a data structure of typically non-equivalent elements. The relative address field in the instruction is then viewed as a positive assembled-in (i.e., not computed) displacement within the data structure, rather than an absolute address on page zero. As an example of the second type, to access a fixed field within a data structure located anywhere in macromemory, the index register would contain the address of the first location of the structure, the X bit would be 1, the L and I bits would be 0, and the relative address would refer to the word displacement of the field from the beginning of the data structure.

The index register may also be referenced as absolute location 0. Any operation on this location is reflected in the index register, and vice versa.

The program counter (PC) is incremented by one after each instruction is executed, with the exception that certain instructions (e.g., skips) conditionally increment the PC by an additional one or two locations, and others (e.g., jumps) unconditionally load the PC with the effective address, causing execution to continue at the new location.

## MEMORY REFERENCE

All memory reference type instructions contain either a memory reference component MR, or an effective address component EA. The latter applies if the old contents of the referenced location are not read out of memory by the instruction. The times for these components depend on the type of reference, and are as follows:

Type of reference ++	MR	EA	
direct	3	0	microcycles
indirect	6	4	microcycles

The JMP and CALL instructions involve a common component in \* which the address of the next instruction is computed and \* checked. The time for this computation depends on the particular \* type of memory address specified in the instruction. In the \* formulas for these instructions, the variation in the time due to \* this component is reflected in a variable called JA, the values \* of which are given in the following table: \*

---

++ there is no longer any difference in MR and EA caused by current page vs. page zero, or by indexed vs. unindexed.

Type of reference	JA	Microcycles	
page zero, direct	0		*
page zero, indirect	4		*
page zero, direct, indexed	1		*
page zero, indirect, indexed	4		*
local page, direct	-1		*
local page, indirect	4		*
local page, direct, indexed	1		*
local page, indirect, indexed	4		*

Individual memory reference instructions are described below under the appropriate instruction group types.

## Instruction Table for Section 2.4

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
ALR16	0041700	A (Register) Left Rotate 16	SHIFT	16<141	2-38
ALR20	0041600	A (Register) Left Rotate 20	SHIFT	16<169	2-36
ALS	0041500	Arithmetic Left Shift	SHIFT	16<155	2-33
ARR16	0040700	A (Register) Right Rotate 16	SHIFT	16<140	2-37
ARR20	0040600	A (Register) Right Rotate 20	SHIFT	16<168	2-35
ARS	0040500	Arithmetic Right Shift	SHIFT	17<133	2-32
LGL	0041400	Logical Left Shift	SHIFT	16<134	2-29
LGR	0040400	Logical Right Shift	SHIFT	16<130	2-28
LLL	0041000	Long Left Logical Shift	SHIFT	17<234	2-30
LLR16	0041300	Long Left Rotate 16	SHIFT	17<254	2-38
LLR20	0041200	Long Left Rotate 20	SHIFT	17<310	2-36
LLS	0041100	Long Left Arithmetic Shift	SHIFT	28<287	2-33
LRL	0040000	Long Right Logical Shift	SHIFT	17<231	2-29
LRR16	0040300	Long Right Rotate 16	SHIFT	17<253	2-37
LRR20	0040200	Long Right Rotate 20	SHIFT	17<309	2-35
LRS	0040100	Long Right Arithmetic Shift	SHIFT	28<244	2-32

## 2.4 Shift

The shift instructions use a shift count in bits 5-0 of the instruction word. If these bits are nonzero, this quantity is interpreted as the two's complement, modulo 64, of the number of places to be shifted. If bits 5-0 are zero, the X register is assumed to contain the two's complement, modulo  $2^{20}$ , of the number of places to be shifted.

If X contains zero, there is no shift, but the C and O registers are cleared to 0. Each shift instruction has a maximum number of places which may be shifted. For rotates, this number is one less than the number of bits participating in the operation, not counting C or O. For shifts, this maximum is the number of bits participating, not counting C or O. If the supplied shift count exceeds this limit, an instruction trap results.

The maximum lengths of the various shift instructions are:

LRL	40.	LLL	40.	*
LRS	40.	LLS	40.	*
LRR20	39.	LLR20	39.	*
LRR16	31.	LLR16	31.	*
LGR	20.	LGL	20.	*
ARS	20.	ALS	20.	*
ARR20	19.	ALR20	19.	*
ARR16	15.	ALR16	15.	*

If a shift instruction is executed with a length exceeding \*  
its specified maximum, trap 15 (octal), "overlong shift", will be \*  
issued. ++ \*

The instruction execution times for the shift instructions  
are given below in the either of the forms

$$C1(0), C2+C3(n) \quad \text{or} \quad C1(0), C2+C3(n-1)$$

where C1 is the number of microcycles if the shift count is zero,  
C2 is a fixed number of microcycles for a non-zero shift count,  
C3 is the additional number of microcycles per bit position  
shifted, and n is the number of bit positions shifted.

---

++ M7U13, Microrelease 1, and later versions, of the NMFS  
microcode conform to this specification.



LOGICAL SHIFT

LRL  
0040000

Long Right Logical Shift  
- - -

17(0), 41 + 5(n-1)  
max: n <= 40

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0		0	1	0	0	0	0	0	0	0		shift	count		

0	-->	A19			A0	-->	B19			B0	-->	C	-->	<
---	-----	-----	--	--	----	-----	-----	--	--	----	-----	---	-----	---

LGL  
0041400

LoGical Left Shift  
- - -

16(0), 39 + 5(n)  
max: n <= 20

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0		0	1	0	0	0	0	1	1	0	0		shift	count	

>	<--	C	<--	A19			A0	<--	0
---	-----	---	-----	-----	--	--	----	-----	---

LOGICAL SHIFT

LLL  
0041000

Long Left Logical Shift

17(0), 40 + 5(n)  
max: n <= 40

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	0	0	0	0	0	1	0	0	0	0	1	0	0	0		shift	count		
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

>	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<
	C		<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	0
	A	19		A	0		B	19		B	0									
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

## 2.4.2 Arithmetic Shift

The arithmetic shift group contains four instructions that shift the contents of the A register or the combined A and B registers keeping track of the sign bit. On right shifts, bits equal to the sign are supplied to the vacated bits, and the last bit shifted out is saved in the C register. On left shifts, zeros are supplied to the vacated bits, and the C register is 0 unless the value of the sign bit of A (A19) changed during the shift. \*

When these instructions are used to do arithmetic with single word quantities, sign extension ensures the same result with 16-bit signed data as with 20-bit signed data. \*

For left shifts C will reflect a change in position 19 or 39 (during the course of the shift); for right shifts, C reflects the contents of the last bit shifted out of position 0. \*

The O register receives a copy of the value shifted into the C bit.

ARITHMETIC SHIFT

ARS  
0040500

Arithmetic Right Shift  
- - -

17(0), 43 + 5(n-1)  
max: n <= 20

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	0	0	0	0	1	0	0	0	0	0	1	0	1						
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

LRS  
0040100

Long Right Arithmetic Shift  
- - -

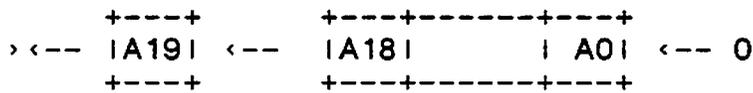
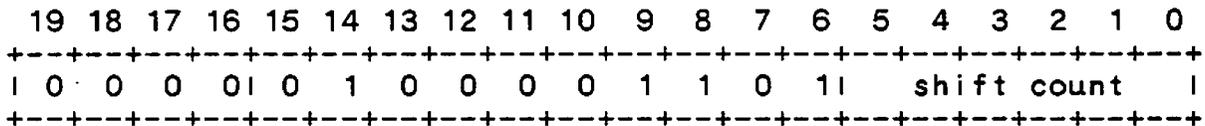
28(0), 54 + 5(n-1)  
max: n <= 40

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	0	0	0	0	1	0	0	0	0	0	0	0	0	1					
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+

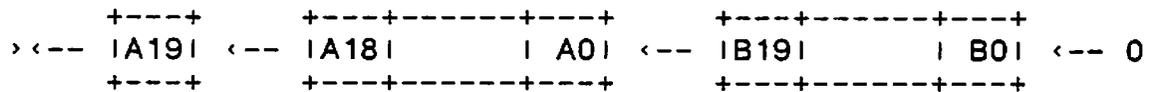
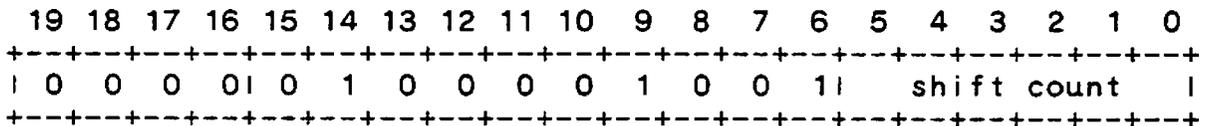
ARITHMETIC SHIFT

ALS                    Arithmetic Left Shift                    16(0), 41 + 6(n)  
 0041500               -                                       -                                       -                                       max: n <= 20



The C register is set to 1 if A19 changes during the shift, otherwise it is set to 0.

LLS                    Long Left Arithmetic Shift                    28(0), 53 + 6(n)  
 0041100               -                                       -                                       -                                       max: n <= 40



The C register is set to 1 if A19 changes during the shift, otherwise it is set to 0.

### 2.4.3 Rotate

The rotate group contains eight instructions, in two sets of \*  
four instructions each. The four instructions in each set \*  
reflect the distinctions left vs. right, and short vs. long. The \*  
two sets reflect the distinctions 20/40 vs. 16/32-bit operations. \*

The 16 and 32-bit versions operate on the low 16 bits of A \*  
and B entirely ignoring bits 19 through 16 of either register. \*

In both sets of rotates, the last bit shifted around the end \*  
is copied into the C register. \*

The O register receives a copy of the value shifted into the  
C bit.

ROTATE

ARR20                    A (Register) Right Rotate 20                    16(0), 35 + 7(n)  
 0040600                    -                    -                    -                    max: n <= 19

```

  19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 | 0 1 0 0 0 0 0 1 1 1 | shift count |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

```

+-----+
|        +---+---+---+---+        |        +---+
+--> |A19|                    | A0|        | C |---<
      +---+---+---+---+                    +---+

```

LRR20                    Long Right Rotate 20                    17(0), 36 + 7(n)  
 0040200                    -                    -                    -                    max: n <= 39

```

  19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 | 0 1 0 0 0 0 0 0 0 1 1 | shift count |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

```

+-----+
|        +---+---+---+---+        +---+---+---+        |        +---+
+--> |A19|                    | A0|        |B19|                    | B0|        | C |---<
      +---+---+---+---+                    +---+---+---+                    +---+

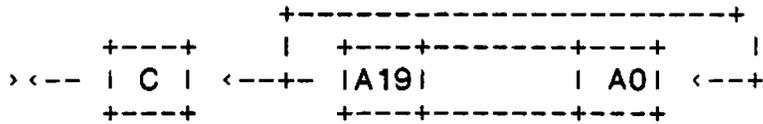
```

ROTATE

ALR20                    A (Register) Left Rotate 20                    16(0), 36 + 7(n)  
 0041600                    -                    -                    -                    max: n <= 19

```

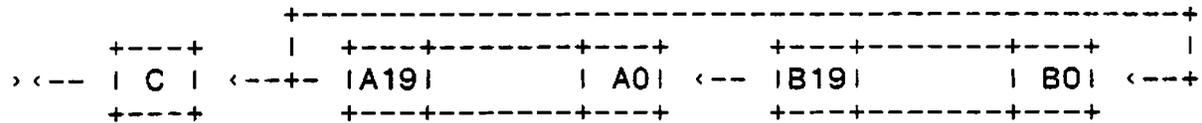
    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 | 0 1 0 0 0 0 1 1 1 1 | shift count |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```



LLR20                    Long Left Rotate 20                    17(0), 37 + 7(n)  
 0041200                    -                    -                    -                    max: n <= 39

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 | 0 1 0 0 0 0 1 0 1 1 | shift count |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```



ROTATE

ARR16                    A (Register) Right Rotate 16                    16(0), 35 + 7(n) \*  
 0040700                    -                    -                    -                    max: n <= 15        \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0        *
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 | 0 1 0 0 0 0 0 1 1 1 | shift count |        *
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

```

+-----+                    *
|        +---+---+---+---+        |        +---+        *
+--> |A15|                    | A0|        +--> | C |        <        *
+-----+                    +-----+                    *
    
```

LRR16                    Long Right Rotate 16                    17(0), 36 + 7(n) \*  
 0040300                    -                    -                    -                    max: n <= 31        \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0        *
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 | 0 1 0 0 0 0 0 0 0 1 1 | shift count |        *
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

```

+-----+                    *
|        +---+---+---+---+        +---+---+---+---+        |        +---+        *
+--> |A15|                    | A0|        +--> |B15|                    | B0|        +--> | C |        <        *
+-----+                    +-----+                    +-----+                    *
    
```



## LOAD/STORE

## 2.5 Load/Store

## Instruction Table for Section 2.5

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
BLT	0000031	Block Transfer Memory	LD/ST	15 + 7n	2-48
CAL	0141050	Clear A Register Left Half	LD/ST	6	2-44
CAR	0141044	Clear A Register Right Half	LD/ST	6	2-44
CLR	0141144	Copy A Register Lt to Rt 1/2	LD/ST	7	2-46
COB	0140504	Complement Overflow Bit	LD/ST	6	2-47
CRA	0140040	Clear A Register	LD/ST	6	2-43
CRL	0141250	Copy Register Rt to Lt Byte	LD/ST	7	2-46
IAB	0000201	Interchange A and B Registers	LD/ST	9	2-43
ICA	0141340	Interchange A Register Halves	LD/ST	7	2-44
ICL	0141140	Interchange & Clr A (Lt 1/2)	LD/ST	8	2-45
ICR	0141240	Interchange & Clr A (Rt 1/2)	LD/ST	8	2-45
IMA	0026000	Interchange Memory & A Reg	LD/ST	11 + MR	2-41
LAI	0141700	Load A Indirectly thru Self	LD/ST	9	2-40
LDA	0004000	Load A Register	LD/ST	6 + MR	2-40
LDX	0072000	Load X Register	LD/ST	10 + MR	2-42
LXA	0141714	Load X Indirectly thru A	LD/ST	13	2-43
LXI	0141704	Load X Indirectly thru Self	LD/ST	13	2-42
RCB	0140200	Reset C Register (Bit)	LD/ST	7	2-46
ROB	0140604	Reset Overflow Bit	LD/ST	7	2-47
SCB	0140600	Set C Register (Bit)	LD/ST	7	2-46
SOB	0140204	Set Overflow Bit	LD/ST	7	2-47
STA	0010000	Store A Register	LD/ST	10 + EA	2-41
STX	0032000	Store X Register	LD/ST	10 + EA	2-41
SXA	0141614	Store X Indirectly thru A	LD/ST	13	2-42
TRB	0000041	Transfer Memory Backwards	LD/ST	11 + 7n	2-49

2.5 Load/Store

The twenty-five instructions in this group are used to move or clear data.

LDA                      Load A Register                      6 + MR  
 0004000                      - - -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 0 1 1 X | 0 0 1 0 | L I                      relative address                      |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Load the contents of the location specified by the effective address into the A register.

LAI                      Load A Indirectly through Self                      9  
 0141700                      - - -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 0 1 1 0 | 0 0 0 0 | 1 1 1 1 | 0 0 0 0 | 0 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Load the contents of the location whose address is contained in the A register into the A register.

LOAD/STORE

STA                    STore A Register                    10 + EA  
 0010000                --        -

```

  19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  | 0 0 0 0 0 1 1 X 1 0 1 0 0 1 L |        relative address        |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  
```

Store the contents of the A register into the location specified by the effective address.

IMA                    Interchange Memory and A Register                    11 + MR  
 0026000                -                -                -

```

  19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  | 0 0 0 0 0 1 1 X 1 1 0 1 1 L |        relative address        |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  
```

Interchange the contents of the A register and the contents of the location specified by the effective address.

STX                    STore X Register                    10 + EA  
 0032000                --        -

```

  19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  | 0 0 0 0 0 1 1 0 1 1 0 1 L |        relative address        |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  
```

Store the contents of the X register into the location specified by the effective address. This instruction is not indexable.

SXA                      Store X Indirectly through A                      13  
 0141614                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Store the contents of the X register into the location whose address is contained in the A register.

LDX                      Load X Register                      10 + MR  
 0072000                      - - -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 1 1 0 1 | LI                      relative address                      |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Load the contents of the location specified by the effective address into the X register. This instruction is not indexable.

LXI                      Load X Indirectly through Self                      13  
 0141704                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Load the contents of the location whose address is contained in the X register into the X register.

LXA                      Load X Indirectly through A.                      13  
 0141714                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 | 1 1 0 0 0 0 1 1 1 1 0 0 1 1 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Load the contents of the location whose address is contained in the A register into the X register.

CRA                      Clear A Register                      6  
 0140040                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 | 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the A register are set to 0.

IAB                      Interchange A and B Registers                      9  
 0000201                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 | 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the A and B registers are interchanged.

CAL                      Clear A Register Left Byte                      6  
 0141050                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Bits 19 through 8 of the contents of the A register are set to 0. Bits 7 through 0 are not affected.

CAR                      Clear A Register Right Byte                      6  
 0141044                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Bits 7 through 0 of the contents of the A register are set to 0. Bits 19 through 8 are not affected.

ICA                      InterChange A Register Bytes                      7  
 0141340                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 1 0 1 1 1 0 0 0 0 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Bits 7 through 0 and bits 15 through 8 of the contents of the A register are interchanged. Bits 19 through 16 are not affected.



LOAD/STORE

CRL Copy A Register Right to Left Byte 7  
 0141250 - - -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 1 1 0 0 0 0 1 0 1 0 1 0 1 0 0 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Bits 7 through 0 of the contents of the A register are copied to bits 15 through 8. Bits 7 through 0 and bits 19 through 16 are not affected.

RCB Reset C Register (Bit) 7  
 0140200 - - -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the C Register are set to 0.

SCB Set C Register (Bit) 7  
 0140600 - - -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the C Register are set to 1.

SOB                      Set Overflow Bit                      7  
 0140204                  -        -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the O register are set to 1.

ROB                      Reset Overflow Bit                      7  
 0140604                  -        -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the O register are set to 0.

COB                      Complement Overflow Bit                      6  
 0140504                  -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the O bit are complemented and the result loaded back into the O register.

BLT                      BLock Transfer Memory                      15 + 7(n)  
 0000031                      --                      -                      n = number of locations

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1

A group of sequential memory locations, starting at the location specified by the contents of the X register, and with the number of locations specified by the contents of the B register, are loaded with the contents of equivalent memory locations starting at the location specified by the contents of the A register. When the transfer is completed, the contents of the A and X registers will have been incremented by the original contents of the B register, and the B register will be 0. The contents of the memory locations are moved one word at a time in increasing sequential order, and the address ranges of the memory blocks may overlap.

A BLT instruction referencing location 0 as part of either of the areas of memory specified by the A or X registers produces an undefined result.

The execution of this instruction is interruptable by a higher priority process as described in chapter 3.



## 2.6 Arithmetic

## Instruction Table for Section 2.6

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

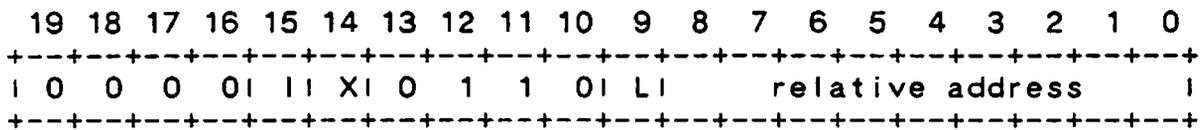
Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
ABX	0141000	Add B Register to X Register	ARITH	8	2-57
ACA	0141216	Add C Register to A Register	ARITH	11	2-53
ADD	0014000	Add Memory to A Register	ARITH	7 + MR	2-51
AOA	0141206	Add One to A Register	ARITH	7	2-54
AOX	0141014	Add One to X Register	ARITH	13	2-55
AVA	0141110	Add Overflow to A Register	ARITH	9	2-58
CHK	0000032	Checksum Block of Memory	ARITH	14 + 5n	2-56
ECK	0000202	End Around Checksum	ARITH	15 + 7n	2-57
SOA	0141306	Subtract One from A Register	ARITH	9	2-54
SOX	0141114	Subtract One from X Register	ARITH	13	2-55
SUB	0016000	Subtract Memory from A Reg	ARITH	8 + MR	2-52
SVA	0141310	Subtract Overflow from A Reg	ARITH	12	2-59
TCA	0140407	Two's Complement A Register	ARITH	7	2-52

ARITHMETIC

2.6 Arithmetic

The thirteen instructions in this group are used for two's complement arithmetic operations.

ADD                      ADD Memory to A Register                      7 + MR  
 0014000                      ---



Add the contents of the location specified by the effective address to the contents of the A register, and load the result into the A register.

Set the C register to 1 if overflow occurred, otherwise clear the C register to 0. Overflow occurs if the original contents of A and memory have the same sign, and the result has the opposite sign.

Set the O register to 1 if unsigned overflow occurs, otherwise, clear the O bit to 0. Unsigned overflow occurs if the original contents of A and memory have values such that a carry occurs out of bit 19 when they are added together, considering the two values and the final result as 20-bit unsigned numbers.

SUB                    SUBtract Memory from A Register                    8 + MR  
 0016000                    ---

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 1 1 X 0 1 1 1 1 |                    relative address                    |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Subtract the contents of the location specified by the effective address from the contents of the A register, and load the result into the A register.

Set the C register to 1 if overflow occurred, otherwise clear the C register to 0. Overflow occurs if the original contents of A and memory have opposite signs, and the result has a sign opposite to the original contents of A.

Set the O register to 1 if unsigned underflow occurs, otherwise, clear the O bit to 0. Unsigned underflow occurs if the original contents of memory are larger than the original contents of A, when the two values are considered as 20-bit unsigned numbers.

TCA                    Two's Complement A Register                    7  
 0140407                    -                    -                    -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 1 1 1 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The two's complement of the contents of the A register are loaded back into the A register.





ARITHMETIC

AOX Add One to X Register 13  
0141014 - - -

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0  
+---+  
| 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 1 1 0 0 |  
+---+

The contents of the X register are incremented by 1, and the result is loaded into the X register.

The C register is set to 1 if the previous contents of the X register were 1777777. Otherwise, the C register is set to 0. \*

The O register is set to 1 if the previous contents of the X register were octal 3777777. Otherwise the O register is set to 0. \*

SOX Subtract One from X Register 13  
0141114 - - -

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0  
+---+  
| 0 0 0 0 0 1 1 0 0 0 0 1 0 0 1 0 0 1 1 0 0 |  
+---+

The contents of the X register are decremented by 1, and the result is loaded into the X register.

The C register is set to 1 if the previous contents of the X register were 3000000. Otherwise, the C register is set to 0. \*

The O register is set to 1 if the previous contents of the X register were 0. Otherwise the O register is set to 0.

CHK                      CHecKsum Block of Memory                      14 + 5(n)  
 0000032                      --   -    n = number of locations

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+---+																			
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0
+---+																			

The contents of a group of sequential memory locations, starting at the location specified by the contents of the X register, and with the number of locations specified by the contents of the B register, taken as unsigned 20-bit values, are arithmetically added, one location at a time, to the contents of the A register, and the result is loaded into the A register. Overflows are ignored. When the summation is completed, the contents of the A register will be the sum of the original contents of the A register plus the contents of each memory location in the block, the X register will have been incremented by the original contents of the B register, and the B register will be 0. The execution of this instruction is interruptable by a higher priority process, as described in chapter 3.

ARITHMETIC

ECK                      End Around Checksum                      15 + 7(n)  
 0000202                      -                      -                      -                      n = number of locations

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of a group of sequential memory locations, taken as unsigned 16-bit values, starting at the location specified by the contents of the X register, and with the number of locations specified by contents of the B register, are added with "end around carry". The result is the logical sum of the set of locations, plus the number of times this addition resulted in overflow out of bit 15. When the summation is completed, and the contents of the A register are replaced by the sum of the original contents of the A register plus the result of the summation process just described, the X register will have been incremented by the original contents of the B register, the B register will be 0, and bits 19 through 16 will be set to zero. The execution of this instruction is interruptable by a higher priority process, as described in chapter 3.

ABX                      Add B Register to X Register                      8  
 0141000                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the B register are added to the contents of the X register and the result is loaded back into the X register.

ARITHMETIC

AVA Add Overflow to A Register  
0141110 - - -

9

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	0	0	0	0	1	0	0	1	0	0	1	0	0

The contents of the O register are added to the contents of the A register and the result loaded into the A register.

The O register is set to 1 if the previous contents of the A register were octal 377777 and the previous contents of the O register were 1. Otherwise the O register is set to 0.

\*  
\*  
\*  
\*

The C register is set to 1 if the previous contents of the A register were octal 177777 and the previous contents of the O register were 1. Otherwise the C register is set to 0.

\*  
\*  
\*  
\*

ARITHMETIC

SVA	Subtract Overflow from A Register	12
0141310	-                    -                    -	

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	
1	0	0	0	0	1	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0
+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+	+---+

The contents of the O register are subtracted from the contents of the A register, and the result loaded into the A register.

The C register is set to 1 if the previous contents of the A register were 3000000 and the previous contents of the O register were 1. Otherwise, the C register is set to 0.

\*  
\*  
\*

The O register is set to 1 if the previous contents of the A register were 0 and the previous contents of the O register were 1. Otherwise the O register is set to 0.

## 2.7 Logical

## Instruction Table for Section 2.7

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
ANA	0006000	AND Memory to A Register	LOGIC	6 + MR	2-61
CCRO	0001032	Convert & Clear Rightmost One	LOGIC	10 + f(A)	2-65
CHS16					
CHS20	0140024	Change (Complement) Sign of A	LOGIC	6	2-63
CMA	0140401	Ones Complement A Register	LOGIC	6	2-62
CSA16	0000053	Copy Sign & Set A's Sign to +	LOGIC	10	2-65
CSA20	0140320	Copy Sign & Set A's Sign to +	LOGIC	10	2-64
CXB	0140510	Complement Indexed Bit	LOGIC	9	2-67
ERA	0012000	Exclusive OR Memory to A Reg	LOGIC	6 + MR	2-61
FFO	0000033	Find First One	LOGIC	9 + 29	2-65
RXB	0140210	Reset Indexed Bit	LOGIC	9	2-66
SSM16	0000051	Set Sign of A Register Minus	LOGIC	6	2-63
SSM20	0140500	Set Sign of A Register Minus	LOGIC	6	2-63
SSP16	0000050	Set Sign of A Register Plus 16	LOGIC	6	2-62
SSP20	0140100	Set Sign of A Register Plus 20	LOGIC	6	2-62
SXB	0140610	Set Indexed Bit	LOGIC	9	2-66

LOGICAL

2.7 Logical

The twelve instructions in this group are used for bit-wise boolean operations.

ANA                                   AND Memory to A Register                                   6 + MR  
0006000                               --                                   -

```

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 0 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 | relative address |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
    
```

Logical AND (bit-wise) the contents of the location specified by the effective address with the contents of the A register, and load the result into the A register.

ERA                                   Exclusive OR Memory to A Register                                   6 + MR  
0012000                               -                                   -

```

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
| 0 0 0 0 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 | relative address |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
    
```

Exclusive OR (bit-wise) the contents of the location specified by the effective address with the contents of the A register, and load the result into the A register.

CMA                    Ones CoMplement A Register                    6  
 0140401                    - -                    -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The boolean inverse of the contents of the A register is loaded into the A register.

SSP20                  Set Sign of A Register Plus 20                  6  
 0140100                  - -                    -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Bit 19 of the contents of the A register is set to 0. The rest of the contents of the A register are not affected.

SSP16                  Set Sign of A Register Plus 16                  6 \*  
 0000050                  - -                    -                    \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Bit 15 of the contents of the A register is set to 0. The rest of the contents of the A register are not affected. \*  
 \*









CXB	Complement Indexed Bit	9	*
0140510	- - -		*

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	*
+---+																				*
1	0	0	0	0	1	1	0	0	0	0	1	0	1	0	0	1	0	0	0	*
+---+																				*

Complement the bit in the A register whose position is specified by the contents, modulo 16, of the X register. All other bits of the A register are not affected. \*

Note: This is a 16-bit instruction. \*

## 2.8 Program Control

## Instruction Table for Section 2.8

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
CAS	0022000	Compare A Reg to Mem & Skip	CNTRL	11 + MR	2-70
IRS	0024000	Increment, Replace & Skip	CNTRL	10 + MR	2-71
JMP	0002000	Unconditional Jump	CNTRL	7 + JA	2-69
JST	0020000	Jump & Store Program Counter	CNTRL	10 + EA	2-70
NOP	0101000	No Operation	CNTRL	6	2-71
SGT16	0100202	Skip if A Register > 0 16	CNTRL	9/10	2-75
SGT20	0100401	Skip if A Register > 0 20	CNTRL	9/10	2-74
SKP	0100000	Unconditional Skip	CNTRL	6	2-71
SLE16	0101202	Skip if A Reg LT or EQ Zero 16	CNTRL	9/10	2-75
SLE20	0101401	Skip if A Reg LT or EQ Zero 20	CNTRL	9/10	2-75
SLN	0101100	Skip if Low Bit of A Reg Non0	CNTRL	8	2-76
SLZ	0100100	Skip if Low Bit of A Reg Zero	CNTRL	8	2-76
SMI16	0101201	Skip if A Reg Minus 16	CNTRL	8	2-74
SMI20	0101400	Skip if A Reg Minus 20	CNTRL	8	2-74
SNZ16	0101200	Skip if A Register Nonzero 16	CNTRL	8	2-73
SNZ20	0101040	Skip if A Register Nonzero 20	CNTRL	8	2-72
SOC	0100021	Skip if Overflow Clear	CNTRL	9	2-78
SOS	0101021	Skip if Overflow Set	CNTRL	9	2-77
SPL16	0100201	Skip if A Reg Plus (>=0) 16	CNTRL	8	2-73
SPL20	0100400	Skip if A Reg Plus (>=0) 20	CNTRL	8	2-73
SRC	0100001	Skip if Reset C Register	CNTRL	10	2-76
SSC	0101001	Skip if Set C Register	CNTRL	10	2-77
SZC	0100041	Skip if A Reg 0 & Reset C Reg	CNTRL	8/13	2-77
SZE16	0100200	Skip if A Register Zero 16	CNTRL	8	2-72
SZE20	0100040	Skip if A Register Zero 20	CNTRL	8	2-72
SZO	0100042	Skip if A Reg 0 & Ovrflo Reset	CNTRL	10	2-78

## 2.8 Program Control

The twenty instructions in this group are used for controlling the sequence of program execution. References to tests on the contents of registers, e.g., "if the contents of the A register are zero" ... , are tests on the full 20 bits unless explicitly stated otherwise.

JMP                      Unconditional JuMP                      7 + JA  
0002000                      - - -

```

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 0 1 1 X 0 0 0 1 | L | relative address | 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

Set the Program Counter (PC) to the value of the effective address and continue execution at the new location.

Crashes:

- #6, JMP or JST to location 0. The virtual hardware traps this condition and treats it like an illegal instruction. See 5.3.

PROGRAM CONTROL

JST                      Jump and Store Program Counter                      10 + EA  
 0020000                      -                      --

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 | 1 1 X | 1 0 0 0 | L |                      relative address                      |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Store the address of the next sequential location into the location specified by the effective address. Set the PC to the value of the effective address plus one, and continue execution at the new location.

Crashes:

#6. JMP or JST to location 0. See JMP.

CAS                      Compare A Register to Memory and Skip                      11 + MR  
 0022000                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 | 1 1 X | 1 0 0 1 | L |                      relative address                      |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Compare, as signed integers, the contents of the A register with the contents of the location specified by the effective address. If A > memory, the next sequential instruction is executed. If A = memory, the next sequential instruction is skipped. If A < memory, the next two sequential instructions are skipped. Neither the contents of the A register nor memory is affected.









PROGRAM CONTROL

SGT16                   Skip if A Register Greater Than Zero 16                   10 \*  
 0100202               -   -   -   (9 if zero) \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0           *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 | 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 |       *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

If bits 15-0 of the A register contain a positive number which is greater than zero, the next sequential instruction is skipped. \*

SLE20                   Skip if A Less than or Equal to Zero 20                   10  
 0101401               -   -   -   (9 if zero)

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0           *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 | 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 |       *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

If the A register contains a negative number, or zero, the next sequential instruction is skipped.

SLE16                   Skip if A Less than or Equal to Zero 16                   10 \*  
 0101202               -   -   -   (9 if zero) \*

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0           *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 | 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 0 |       *
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

If bits 15-0 of the A register contain a negative number, or zero, the next sequential instruction is skipped. \*





SOC                                      Skip if Overflow Clear                                      9  
 0100021                                      -                                      -                                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

If the contents of the O register are 0, the next sequential instruction is skipped.

SZO                                      Skip if A register Zero and Overflow Reset                                      10  
 0100042                                      -                                      -                                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

If the contents of the A Register and the contents of the O register are both 0, the next sequential instruction is skipped.

## 2.9 Processor Control

## Instruction Table for Section 2.9

This table provides summaries, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
-----	-----	-----	-----	-----	-----
ERB	0000200	Retrieve Error Bits	PCTRL	14	2-85
ERC	0000105	Error Light Clear	PCTRL	14	2-84
ERR	0000101	Interrogate Memory Errors	PCTRL	23	2-84
HLT	0000000	Halt the Processor	PCTRL	5	2-80
LITES	0000011	Write LIT (LITES) Register	PCTRL	21	2-81
MEMHI	0000012	Read Memory High Bound	PCTRL	9	2-81
RDCLOK	0000010	Read RTC Register	PCTRL	11	2-80
RSM	0000013	Read Special Memory	PCTRL	12+f(B)	2-87
VER	0000100	Return Version Number	PCTRL	9	2-83
WATCH	0000761	Watch On or Off	PCTRL	11/15	2-86
WSM	0000021	Write Special Memory	PCTRL	12+f(B)	2-88



PROCESSOR CONTROL

LITES                      Write LIT (LITES) Register                      21 \*  
 0000011                      -----                      \*

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	*	
+---																					
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	*
+---																					

The contents of the A register is loaded into the 20-bit LED light register of the I/O board in the slot whose position in the I/O bus is given in the low three bits of the contents of the X register. I/O boards are numbered in the range 1 through 7. \*

MEMHI                      Read MEMory High Bound                      9  
 0000012                      ---                      --

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	*	
+---																					
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	*
+---																					

The A register is loaded with the highest allowed macromemory address value usable by the macrocode. This is computed by subtracting any portion used by the microcode or hardware from the physical macromemory limit.

The remainder of this section describes instructions provided for the purpose of maintenance and diagnosis. A number of specific instructions are provided for operational monitoring of the state of the machine. These constitute an interface between the microcode and the macrocode environments. The intent

is to provide such an instruction for each monitoring function which might be routinely performed by the application macro program. The information returned by instructions in this class is intended to be stable across microcode changes, although this cannot be absolutely guaranteed.

In addition, two special instructions are provided to read and write the contents of the MBB's various special memories. Although these instructions can often be used to achieve the same purpose as one of the other maintenance oriented instructions, knowledge of the current version of the microcode is required to do so. Thus these instructions are intended solely for the purpose of remote diagnosis by suitably informed personnel. The use of any particular location in one of the special memories is a function of the microcode and is subject to change without notice. The "I/O space" special memory is a partial exception to this rule, since I/O device register locations and device functions are determined by the configuration of the hardware. Equivalent special knowledge is still required, however, to reference these locations safely. The safety of a write to a device register (or for certain device registers, even a read of the register) depends on both the nature of the device and on the



ERR  
0000101

Interrogate Memory ERRors

23

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1

If no memory errors have occurred since the last execution of this instruction, the next sequential instruction is fetched. If correctable (single-bit errors) have occurred, the X register is set to the number of 1.6 microsecond clock intervals during which errors have occurred, and the A register is set to the value of the MBB's MISC3 register after the last error. MISC3 contains "syndrome" bits identifying the memory chip which failed most recently. (Decoding of syndrome bits to identify failing memory chips is described in section 4.2 of Reference 2.) The flags, count, and syndrome registers maintained by the microcode pertaining to correctable memory errors are then cleared, and the next sequential instruction is skipped. These variables are not cleared across a system microcode restart, so the previous execution of the ERR instruction may have been before the last system microcode restart. Uncorrectable memory errors cause a CRASH, and thus cannot be interrogated by the ERR instruction.

ERC  
0000105

ERror Light Clear

14

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1	0	1

If the MBB's front panel ERROR light is on, it is turned off. The light is turned on by the MBB's system microcode after a correctable memory error has occurred.

ERB  
0000200

Retrieve ERror Bits  
-- -

14

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

If the error flag register maintained by the microcode is non-zero, the contents of the A register are replaced with the contents of the flag register, the flag register is cleared, and the next sequential instruction is skipped. This flag register represents the cause of the last system microcode restart. The bits are assigned the following meanings:

- 1 power up
- 2 small button pushed
- 4 microcode parity error
- 10 uncorrectable memory error
- 20 microcode jumped to zero
- 40 microcode breakpoint
- 100 microcode trap

Any or all of these bits may be on at the same time. The occurrence of any of the conditions represented turns on the corresponding bit. The occurrence of power up clears all of the bits except the power up bit. If the error flag is zero, the next sequential instruction is executed, and the A register remains unchanged.

WATCH	WATCH On or Off	15
0000761	-----	(11 if A=0)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+---+																			
1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	1
+---+																			

This instruction enables or disables the watching facility (also controlled by the NMFS console command W). The A register contains the address of an instruction to be watched; if A contains 0, watching is turned off, otherwise, watching is turned on. A 3 trap occurs if the location changes after watching is turned on. This is a debugging aid which may be used to turn watching on after a location has been initialized to its proper value, which is sometimes difficult to do using the console W command.

PROCESSOR CONTROL

RSM                      Read Special Memory                      12 + f(B)  
 0000013                      -                      -                      -                      f(B): depends on contents of B

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

The B register contains a number (0 - 5 inclusive) designating one of the MBB's special memories. The X register contains the address of a word within the designated memory, and the contents of that word are placed in the A register. Values in the B register correspond to the MBB's special memories as follows (value of f(B) given in parentheses):

- 0    Read macromemory, i.e. the main memory of the C/30, except that memory protection is bypassed. (3)
- 1    Read high half of URAM word. Words in microcode memory are 32 bits wide, but are read out in two 16-bit halves. (10)
- 2    Read low half of URAM word. (8)
- 3    Read dispatch memory. (3)
- 4    Read microcode register memory. (4)
- 5    Read I/O device register. The X register must contain the address of an I/O device register. The device register is read with 3 "I/O strobes". (8)

WSM                      Write Special Memory                      12 + f(B)  
 0000021                      -                      -                      -                      f(B): depends on contents of B

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    | 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 | 1 |
    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
    
```

The B register contains a number (0 - 5 inclusive) designating one of the MBB's special memories. The A register contains a value to be stored in the designated special memory at the location whose address is contained in the X register. (Values of f(B) given in parentheses.)

- 0    Write macromemory.    (3)
- 1    Write high half of URAM word.    (20)
- 2    Write low half of URAM word.    (12)
- 3    Write dispatch memory.    (4)
- 4    Write microcode register memory.    (3)
- 5    Write I/O device register.    (8)

## 2.10 Queue Manipulation

## Instruction Table for Section 2.10

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
DEQ	0000022	Dequeue First Item from Queue	QUEUE	23/48	2-95
ENQ	0000002	Enqueue A New Item on Queue	QUEUE	46	2-94
RMQ	0000042	Rmv Specified Item from Queue	QUEUE	56	2-96

## 2.10 Queue Manipulation

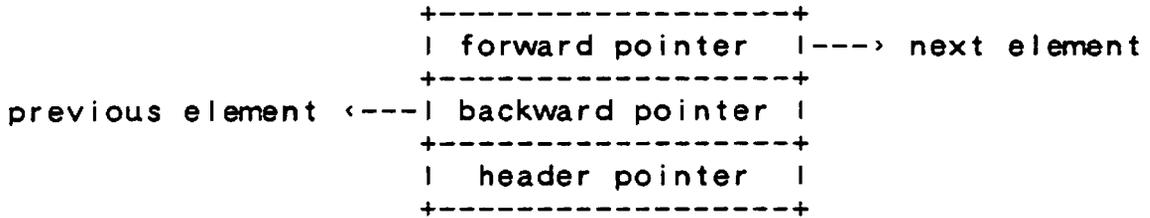
The three instructions in this group are used to perform atomic operations on queues. The virtual hardware defines a queue data structure format that must be observed by the macrocode in order to use the queue manipulation instructions. Since the instructions are atomic (i.e., not interruptable), they can be used with the process system described in chapter 3 to avoid synchronization problems between multiple processes (and firmware) sharing access to the same queue.

A queue data structure is composed of a queue header and zero or more queue items, arranged in a circular, doubly-linked list. Both header and items are queue elements. A queue header is a group of four consecutive locations referenced by the address of the first location. The contents of the first location is the forward pointer of a doubly-linked list. This is the address of the queue element that is on the "front" of the queue. If the queue is empty, this first location will contain the address of the queue header itself. The contents of the second location is the backward pointer of a doubly linked list. This is the address of the queue element that is at the "back" of



a queue header before operating on it with the queue manipulation instructions.

A queue item is an application-dependent structure which incorporates a group of three sequential locations referenced by the address of the first location. Because the addressing structure permits only positive relative addressing, it is often convenient to make these three queue-related locations the first ones of the application-dependent structure, and thus the address of the queue item also becomes the address of the structure. However, this organization is not a requirement for the queue manipulations instructions, which are not concerned with the relationship of the queue item to the application-dependent structure. The three words of the queue item correspond to the first three words of the queue header: forward pointer (to the next element on the list), backward pointer (to the previous element on the list), and queue header pointer. The next element, previous element, or both may be the queue header. The queue header pointer serves to identify which queue a particular item is on. A diagram of a queue item is:



Execution of the queue manipulation instructions using an inconsistent queue data structure will result in one of the following crashes:

- #20, illegal forward pointer
- #21, illegal backward pointer
- #22, illegal header pointer
- #23, illegal item count

Since the firmware uses the same basic queueing primitives for internal operation, these crashes are not necessarily a result of macroprogram errors, they may also be caused by faulty microcode. If a crash occurs, the firmware will initiate a virtual reset as described in 5.3.

QUEUE MANIPULATION

ENQ            ENQueue A New Item on Queue  
 0000002       ---

46

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The queue item addressed by the value in the X register is inserted into the queue before the element addressed by the value in the A register. The item count in the queue header is incremented, and the next sequential instruction is skipped if the queue was previously non-empty. The contents of the A and X registers are not affected.

If the element addressed by the value in the A register is the queue header, then this instruction is the traditional enqueue operation, placing the new item at the back of the queue. In general, this instruction inserts the item X relative to the element A in the sequence ... B A ... yielding the new sequence ... B X A ... If the instruction does not skip, then the item has been added to a previously empty queue.





## STACKS

## 2.11 Stacks

## Instruction Table for Section 2.11

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
CALL	0034000	Subroutine CALL Using Stack	STACK	11 + JA	2-100
CASP	0100011	Copy A Register to SP	STACK	7	2-102
CSPA	0100012	Copy SP to A Register	STACK	7	2-102
CSPX	0100013	Copy SP to X Register	STACK	10	2-103
ITS	0100010	Increment Top of Stack	STACK	13	2-102
LAT	0141510	Load A from Top of Stack	STACK	12	2-104
LXT	0141504	Load X from Top of Stack	STACK	13	2-105
POP	0036000	Pop Memory Contents off Stack	STACK	14 + EA	2-101
POPA	0101003	Pop A Reg Contents off Stack	STACK	9	2-104
PUSH	0030000	Push Memory Contents onto Stx	STACK	10 + MR	2-100
PUSHA	0101002	Push A Reg Contents onto Stk	STACK	11	2-103
RETN	0100002	Subroutine Return Using Stack	STACK	9	2-101
SAT	0141610	Store A in Top of Stack	STACK	12	2-103
SRETN	0100003	Sub Skip Return Using Stack	STACK	10	2-101
SXT	0141604	Store X in Top of Stack	STACK	12	2-104

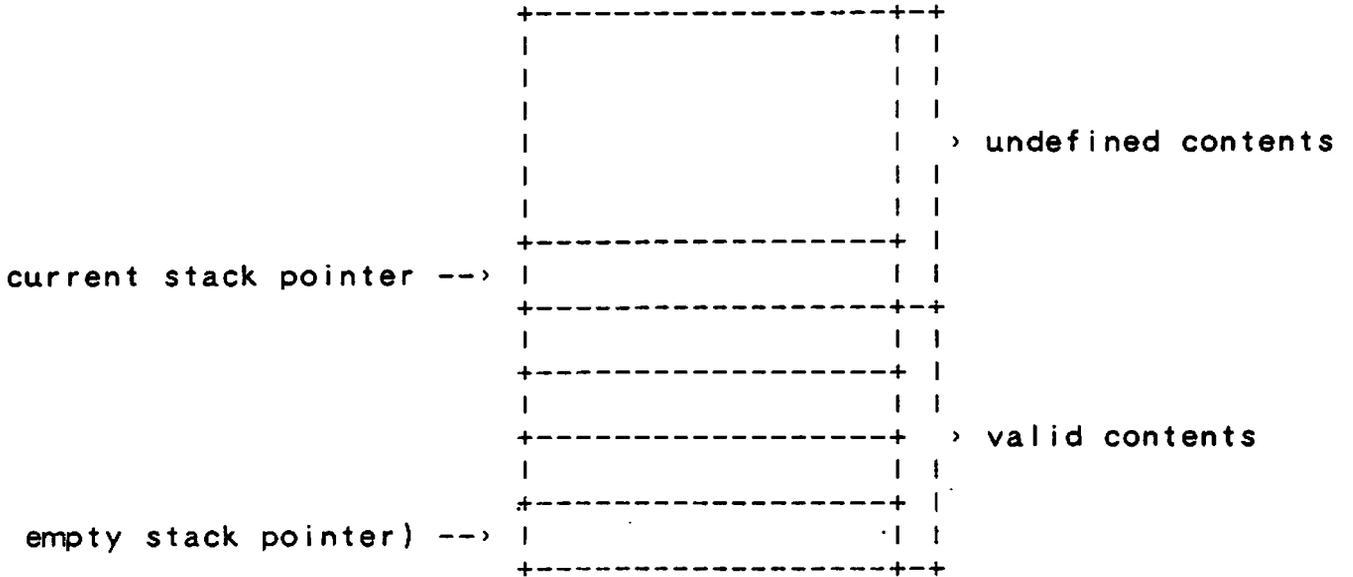
## 2.11 Stacks

The fifteen instructions in this group are used for performing stack-related operations. The virtual hardware contains a stack pointer register (SP) for maintaining the status of a macromemory stack area designated by the application. Since SP is part of the context of a process, the application macrocode may designate private stack areas for each process that uses stacking operations. It is the responsibility of the application to initialize SP. The virtual hardware is not aware of the stack boundaries and thus cannot detect stack overflow or underflow.

The stack can be used to store and retrieve subroutine return addresses and local variables. The stack grows (i.e., is PUSHed) in the direction of decreasing memory addresses, so that stacked items can be referenced as positive displacements relative to the current value of the stack pointer. The stack can be visualized as a data structure whose boundary of valid contents is a dynamically alterable variable:

STACKS

(full stack pointer)->



When an item is PUSHed onto a stack, the item value is stored in the location specified by SP, and then SP is decremented. When an item is POPed from a stack, SP is first incremented and then the item value is loaded from the location specified by SP.

Execution of stack-related instructions that use SP to access locations in the stack area will check the contents of SP and will result in crash #5 (illegal stack pointer) if SP is 0.



STACKS

POP                      POP Memory Contents off Stack                      14 + EA  
 0036000                      ---

```

    19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 | relative address |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
    
```

Increment the contents of SP by 1 and store the contents of the location addressed by the contents of SP in the location specified by the effective address.

RETN                      Subroutine RETurN Using Stack                      9  
 0100002                      --- -

```

    19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
| 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
    
```

Increment the contents of SP by 1, set the PC to the contents of the location addressed by the contents of SP, and continue execution at the new location.

SRETN                      Subroutine Skip RETurN Using Stack                      10  
 0100003                      - --- -

```

    19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
| 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---
    
```

Increment the contents of SP by 1, set the PC to 1 plus the contents of the location addressed by the contents of SP, and continue execution at the new location.



STACKS

CSPX                    Copy SP to X Register                    10  
 0100013                -    --    -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Copy the contents of SP into the X register.

PUSHA                    PUSH A Register Contents onto Stack                    11  
 0101002                -----

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Store the contents of the A register in the location addressed by the contents of SP and decrement the contents of SP by 1.

SAT                      Store A in Top of Stack                      12  
 0141610                -    -    -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 1 1 0 0 0 0 1 1 1 0 0 0 1 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Store the contents of the A register into the location whose address is one plus the contents of the SP register.

STACKS

SXT                      Store X in Top of Stack                      12  
 0141604                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 1 1 0 0 0 0 1 1 1 0 0 0 0 1 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Store the contents of the X register into the location whose address is one plus the contents of the SP register.

POPA                      POP A Register Contents off Stack                      9  
 0101003                      --- -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 1 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Increment the contents of SP by 1 and load the contents of the location addressed by the contents of SP into the A register.

LAT                      Load A from Top of Stack                      12  
 0141510                      -                      -                      -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 0 1 1 0 0 0 0 1 1 0 1 0 0 1 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

Load the contents of the location whose address is one plus the contents of the SP register into the A register.

LXT  
0141504

Load X from Top of Stack  
- - -

13

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	0	0	0	1	1	0	0	0	0	1	1	0	1	0	0	0	1	0	0	1

Load the contents of the location whose address is one plus the contents of the SP register into the X register.

## 2.12 Byte Manipulation

## Instruction Table for Section 2.12

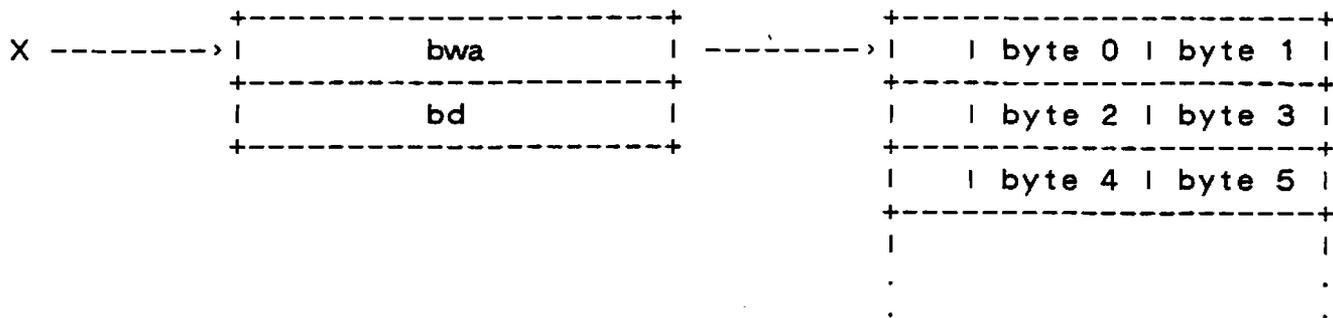
This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
BLI	0001401	Byte Load and Increment	BYTE	22	2-108
BLO	0001400	Byte Load	BYTE	19	2-108
BSI	0001403	Byte Store and Increment	BYTE	29	2-109
BST	0001402	Byte Store	BYTE	25	2-108

2.12 Byte Manipulation

The four instructions in this group are used for byte manipulation.

The X register always contains a pointer to a two word descriptor area: the first word of this descriptor contains a base word address (bwa) and the second contains a byte displacement (bd) from that address. The address  $(bwa + (bd/2))$  thus denotes a word in memory; the byte selected within this word is the high byte (bits 15-8) of the word if bd is even, or the low byte (bits 7-0) of the word if bd is odd. The high byte of the word at the base address is considered byte 0 and the low byte of the word at the base address is considered byte 1, etc.





BSI           Byte Store and Increment  
0001403 -       -       -

29

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1

Bits 7-0 of the A register are stored in the selected byte. The other byte of the word being stored into is not affected. The byte displacement (bd) is incremented by 1 and stored back into memory.

## 2.13 Interprocess Communication

## Instruction Table for Section 2.13

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this section including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
LOK	0001011	Obtain Lock	IPCOM	14	2-112
P	0001013	Decrement Semaphore (Probeer)	IPCOM	14	2-115
RCV	0001012	Receive	IPCOM	14	2-115
SND	0001022	Send Trap	IPCOM	14	2-114
ULK	0001021	Release Lock	IPCOM	15	2-113
V	0001023	Increment Semaphore (Verlaat)	IPCOM	15	2-116

### 2.13 Interprocess Communication

The six instructions in this group are used for interprocess signaling. They may be used to implement various styles of simple process interaction, without the use of INH/ENB, when queues are deemed too expensive. These instructions could also be used when cleaning up old code which was not being totally redesigned but was being modified, to make it clearer, where the interaction is amenable to direct replacement by a semaphore, mailbox, or lock.

Except for initialization, a semaphore, mailbox, or lock is intended to be manipulated solely with the instructions described herein. A semaphore, mailbox, or lock is conventionally initialized to zero. A lock found in this state in memory has never been held.

P and V manipulate a semaphore. LOK and ULK obtain and free a lock. SND and RCV send and receive a command value from or to a mailbox. A semaphore, lock, or mailbox consists of a single location in memory. These instructions supplement ENQ and DEQ, the primary means of process interaction in NMFS.

INTERPROCESS COMMUNICATION

LOK  
0001011

LOcK  
-- -

14 \*  
\*

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	1

Fetch the contents of the location specified by the address contained in the X register. If the value is even, the lock is considered to have been free. In this case, the address of the PCB of the current process is inclusive ored with the constant 1, stored back into the original location, and the next sequential instruction is skipped. If the value is odd, the next sequential instruction is executed.

If LOK falls through, another process is holding the lock and a TPR and SPR may be in order. If LOK skips, the current process now holds the lock, and the high bits of the PCB address provide the identity of this process.



INTERPROCESS COMMUNICATION

SND  
0001022

SeND  
- - -

14 \*  
\*

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0	1

If the value in A is zero, a trap is issued. Otherwise, the location in memory at the address contained in X is fetched. If the value is nonzero, the next sequential instruction is executed; otherwise, the value in A is stored into the location, and the next sequential instruction is skipped.

If SND skips, the message in A has been sent, and a GPR may be in order. If SND falls through, the previous message has not yet been taken.

Crashes:

#14, NULL COMMAND. The SND instruction executed with A containing value 0.



INTERPROCESS COMMUNICATION

V Increment Semaphore (Verlaat) 15 \*  
 0001023 - \*

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	1	1

The location in memory at the address contained in X is fetched, and the value obtained is incremented. If the value is now zero, a trap is issued. Otherwise the new value is stored back into memory at the same location. If the new value is 1, the next sequential instruction is skipped. Otherwise, the next sequential instruction is executed. \*

If V skips, a GPR may be in order. If a trap is issued, the sender has gotten 2^20 ahead of the receiver. This is probably a bug. In any case there is no correct action which V can take except to trap. \*

Crashes: \*

#12, SEMAPHORE OVERFLOW. The V instruction executed with X containing address of semaphore containing value 3777777. \*

## 3 Process System

## Instruction Table

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this chapter, and gives the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
APR	0000003	Activate Process	PRSYS	144	3-28
DPR	0000023	Deactivate Process	PRSYS	148	3-28
ENB	0000401	Enable Other Processes	PRSYS	10/25	3-32
GPR	0000043	Goad Process	PRSYS	39	3-30
INH	0001001	Inhibit Other Processes	PRSYS	9	3-32
MMD	0001002	Measurement Mode Disable	PRSYS	6	3-37
MME	0000402	Measurement Mode Enable	PRSYS	6	3-36
NMFS	0000030	NMFS Mode Control	PRSYS	22/281	3-26
PCB	0000020	Load PCB Into X Register	PRSYS	11	3-37
SPR	0000103	Suspend Process	PRSYS	163<++	3-31
TPR	0000203	Timeout Process	PRSYS	22	3-35
UGS	0000113	Ungoad Self	PRSYS	13	3-33

### 3 Process System

In the basic operating mode, the virtual hardware supports a single program sequence. By invoking the NMFS mode, the application activates a firmware system that is capable of supporting multiple program sequences, or processes. This firmware system provides:

- the ability to define, activate, and deactivate an arbitrary number of processes,
- inter-process signalling,
- automatic context switching between processes,
- the ability to optionally associate I/O devices with processes,
- multi-level preemptive priority scheduling of processes,
- the ability to time out individual processes,
- co-routine like process dismissal/wakeup, and
- per process CPU utilization measurement.

Note that the group of instructions described in the last section of Chapter 2, "Interprocess Communication", provide facilities for writing macro programs which interact as separate processes, but that those interactions are not part of, and do not directly affect the Process System.

### 3.1 Operation

In the NMFS mode, each process executes as if it were in sole possession of the virtual hardware. Processes share the same macromemory space, however, so they can communicate with each other via shared memory locations and by signalling each other. The application macrocode provides process-related data structures in macromemory and observes conventions for the use of these structures. The firmware system makes use of these and additional micromemory structures to schedule the execution of the processes. The macrocode uses the process instructions (described in 3.3) to access the scheduling and other facilities provided by the firmware system and to enter and exit NMFS mode itself.

## DEFINITIONS

## 3.1.1 Definitions

A process is an incarnation of the virtual hardware associated with a PCB (Process Control Block). The PCB is a macrocode-designated macromemory data structure that completely characterizes the state of the virtual hardware and process parameters at any point during the process' lifetime. The PCB address is the process "handle" used by the firmware system for all manipulation and interaction with the process. The PCB format is given in section 3.2.

Any process can be activated and deactivated by any other process. A process is activated by making its PCB known to the firmware system. Once activated, a process is scheduled for execution based on its own and other processes' requirements. When activated, the portion of the PCB structure used by the firmware system is not accessed by the macrocode. A process is deactivated by removing the PCB from the collection being manipulated by the firmware system. When a process is not activated, the PCB has no special meaning to the firmware and may be accessed in any manner by the macrocode.

## DEFINITIONS

The process has a state which determines its disposition by the scheduling algorithm, as described in section 3.1.3. The process also has an application-determined priority level which is used by the scheduling algorithm when deciding which process should be executed. Processes of higher priority (lower numerical values) are scheduled for execution pre-emptively over lower priority ones, and processes at the same priority level are scheduled for execution serially in a round-robin order. A process context, reflecting the state of the virtual hardware, is stored in the PCB whenever the process is not executing.

The process context consists of the current value of the PC (location of the next instruction to be executed), and the contents of the A register, the B register, the X register, the C register, macromemory location 1, and macromemory location 2.

A process is stopped from continuing execution, and its context saved, when it is interrupted by the execution of a higher priority process. The process execution is resumed, with the restored context, when there is no higher priority level process to execute. A process may prevent or allow interruption by inhibiting or enabling the process scheduling system. As a general practice, a process should execute in inhibited mode only

## DEFINITIONS

for a duration sufficient to manipulate a data structure (or call a subroutine) whose local variables are shared with a higher-priority process.

A process may voluntarily suspend execution by dismissing. Execution will be resumed if the process reaches a timeout set by itself, or if the process is "goaded" by itself (before dismissing), another process, or an I/O device. In all cases, the context is saved on dismissal and restored when execution is resumed, with the exception that if resumption is due to a goad (instead of a timeout), the program counter is incremented by an additional location before execution resumes.

A process is timed out by specifying a time at which the process should be resumed. If the process is goaded before the time has elapsed, the timeout is cancelled and the process is immediately scheduled for execution.

Under control of the macrocode, a measurement mode can be activated which accumulates actual execution time of a process in the PCB. This accumulated value is updated each time the context is saved. When measurement mode is on, there is some additional execution time associated with each context switch.

## DEFINITIONS

The program sequence executed by a process is determined by its context, which includes the program counter. In addition, the PCB can be arbitrarily expanded to include any application-specific variables and parameters which are part of the application process context. Thus, any number of processes can use the same program module or modules, since the program sequence is qualified by the application process context. Of course, macromemory outside of the PCBs is not part of the context and thus is potentially sharable with other processes. In particular, the subroutine return address stored by the JST instruction is not automatically part of the process context and must be explicitly protected if there is the possibility of simultaneous access to the subroutine by several processes.

### 3.1.2 Environment

When the macrocode enters NMFS mode, it provides the firmware system with a pointer to an application-designated 140-word macromemory area containing 35 consecutive queue headers. These queue headers are used by the firmware to manage the process scheduling, correspond to process states described in section 3.1.3, and contain PCBs as queue items. While in NMFS mode, the macrocode should not write to locations in this area except to add or remove PCBs from the first of these queues, the idle queue. If the macrocode exits NMFS mode, or if macrocode execution causes a crash to occur, the queue header area ceases to have any special meaning for the firmware system and may be accessed in any manner by the macrocode.

Entering the NMFS mode also requires that the designated idle queue contain at least one PCB. The process represented by the first PCB on the idle queue will be activated automatically and put into execution as the first process in NMFS mode. This process is then responsible for activating and requesting execution of all remaining processes.

The firmware also maintains some internal state variables that, together with the macromemory queue header area and the PCBs, completely characterize the status of the process system. These variables are: the inhibit bit, which is set when interrupts are inhibited; the measurement mode bit, which is set when measurement mode is enabled; and the timeout clock, which is incremented every 1.6 milliseconds and used as the absolute time base for determining if a process has timed out.

## STATES

## 3.1.3 States

A process is always in one of the following states: IDL \*  
(idle), RDY (ready), TIM (timing), PEN (pending), RET (retimed), \*  
or REP (repoked). \*

When an event happens to a process, the process transitions \*  
to a new state, and may also cause certain actions. \*

The events which can happen to a process include the actions \*  
of the following instructions: \*

- APR - Activate Process \*
- DPR - Deactivate Process \*
- TPR - Timeout Process \*
- GPR - Goad Process \*
- SPR - Suspend Process \*
- UGS - Ungoad Self \*

and actions which may result from the running of the TIMEOUT \*  
process. \*

The following diagram shows the new current state a process \*  
transitions to, depending on A) its initial state before the \*  
event happens to it, and B) which event happens to it. \*

STATES

INITIAL PROCESS STATE

OCCURRING EVENT	IDL	RDY	TIM	PEN	RET	REP
APR	RDY	*ill*	*ill*	*ill*	*ill*	*ill*
DPR	IDL	IDL	IDL	IDL	IDL	IDL
TPR	*imp*	*imp*	*imp*	RET setim	RET setim	REP
Timeout	*imp*	*imp*	PEN	*imp*	*imp*	*imp*
GPR	*ill*	PEN incpc	PEN incpc	REP	REP	REP
SPR	*imp*	*imp*	*imp*	RDY	TIM sched	PEN incpc
UGS	*imp*	*imp*	*imp*	PEN	RET	PEN

PROCESS STATE TRANSITION DIAGRAM

STATES:

IDL = idle  
 RDY = ready  
 TIM = timing  
 PEN = pending  
 RET = retimed  
 REP = repoked

ACTIONS:

sched = search & insert on timing queue  
 setim = set time in PCB  
 incpc = increment pc  
 \*imp\* = impossible, cannot happen  
 \*ill\* = illegal; the process crashes

## STATES

In the IDLE state, the PCB is on the idle queue and the process is not manipulated by the firmware system. The macrocode may add or remove PCBs to/from the idle queue using the queuing primitives (to eliminate synchronization problems with the firmware). A process cannot be activated unless its PCB is on the idle queue and properly initialized. When a process is deactivated, its PCB is placed back on the idle queue. Thus, the IDLE state characterizes a process that has been defined but is not a candidate for execution. Processes are activated with the APR instruction and deactivated with the DPR instruction.

When a process is activated, it enters the READY state and its PCB is on the ready queue. The READY state characterizes a process that is a candidate for execution (i.e., is runnable), but one that is not scheduled for execution.

When a process is timing out (scheduled for execution at a particular time), it is in the TIMING state and its PCB is on the timing queue. Whereas the idle and ready queues are treated like "heaps" in the sense that the order of the queue items is not significant, the timing queue is kept in order of increasing absolute time (modulo the timer size). A process entering the TIMING state by executing a TPR instruction will be placed in the

## STATES

appropriate position on the timing queue. As will be shown later, the only time the TPR will not be honored is when the process is in the REPOKED state. A process enters the TIMING state directly from the RETIMED state when the process dismisses after execution. When the firmware system's timeout clock reaches the timeout value for a process, the PCB is removed from the timeout queue and the process enters the PENDING state, as described below. If, while in the TIMING state, a process is goaded into execution by a GPR instruction (executed by another process) or by a device interrupt (for processes that are associated with I/O devices, described in chapter 4), the PCB is removed from the timeout queue and the process enters the PENDING state also, except that in this case the program counter in the saved context is incremented by one so that when the process begins execution it can tell whether it was as a result of a timeout or a goad.

When a process is scheduled for execution, and while it is being executed, its PCB is on one of the 32 pending queues, according to the process priority. While its PCB is on a pending queue, the process can be in one of three states: PENDING, RETIMED, or REPOKED. The PENDING state signifies that immediate

## STATES

execution is requested, either as a result of a timeout or a goad. If a process reaches the end of execution and dismisses while still in the PENDING state, the PCB is removed from the pending queue and the process reverts back to the READY state. The RETIMED state signifies that while the PCB was still on the pending queue and the process was in the PENDING or RETIMED states, a timeout was requested (via a TPR instruction). In this case, the timeout value is preserved and the process enters the TIMING state when it dismisses from its current execution. The REPOKED state signifies that while the PCB was still on the pending queue and the process was in the PENDING or RETIMED states, the process was goaded (via a GPR instruction from any process, including itself, or by a device interrupt). In this case, the process reverts to the PENDING state and its PCB is put back on the pending queue when it dismisses from its current execution, with the value of the context's program counter incremented.

The pending queues are always treated as strict FIFO lists for purposes of scheduling the execution of processes. Thus, all pending processes at the same priority level are given equal treatment. There is no distinction between a process whose PCB

## STATES

has just reached the front of a pending queue and one whose execution has been suspended in order to execute a higher priority process. In both cases, resumption of execution will cause the context saved into the PCB to be restored to the virtual hardware. At the time a PCB is placed on a pending queue, the program counter in the saved context is incremented by one if the process was goaded, and not changed if the process timed out.

The firmware system polls for process rescheduling at appropriate points during the macrocode emulation. With the exception of some instructions with long execution times, e.g., CHK and BLT, these polls only take place in between execution of individual instructions. In the case of CHK and BLT, there may be a long time for the instruction to complete, and so it is desirable to permit interrupts by higher priority processes. These instructions use a subset of the context as working storage, so that the execution of the instructions is transparent to an arbitrary number of interruptions by other processes. A potential process rescheduling is indicated by either a tick of the 1.6 millisecond timeout clock, by a software (GPR) or hardware (completion microinterrupt) goad, by a process

## STATES

suspension (SPR), or by interrupt enabling (ENB). At the clock tick, one or more processes may time out and their PCBs will then be placed on the appropriate pending queues. A goad may result in a PCB not previously on a pending queue being placed on one. A process suspension requires a new process to be selected. After an interrupt enable, higher priority processes may be ready for execution. For efficiency, the firmware system maintains an "attention" bit string of 32 bits, each bit indicating, when on, that at least one PCB is on the corresponding pending queue at that priority level. By setting and clearing these bits as the pending queues become non-empty and empty, the attention string can be relied on to give a quick summary of execution requests for all priority levels, without checking the queue headers. After servicing any timeout/goad requests, the firmware system compares the most significant (highest priority) bit in the attention string with the priority of the process being executed. If the most significant bit of the attention string is of no higher priority than that of the current process, the current process execution is continued. Otherwise, the context of the current process is saved, and, if measurement mode is active, the accumulated processing time is computed and added to the value saved in the current PCB; then the accumulated time is reset for

a new process. Finally, the PCB at the front of the pending queue at the highest level indicated by the attention string is made current, its context restored, and process execution begun. A similar scheduling scan occurs after a process dismisses (via an SPR instruction), but in that case the attention string is examined without comparison for selection of the next process to execute. If at any time the scheduling scan finds no process pending for execution, the firmware will crash (see SPR, crash #47).

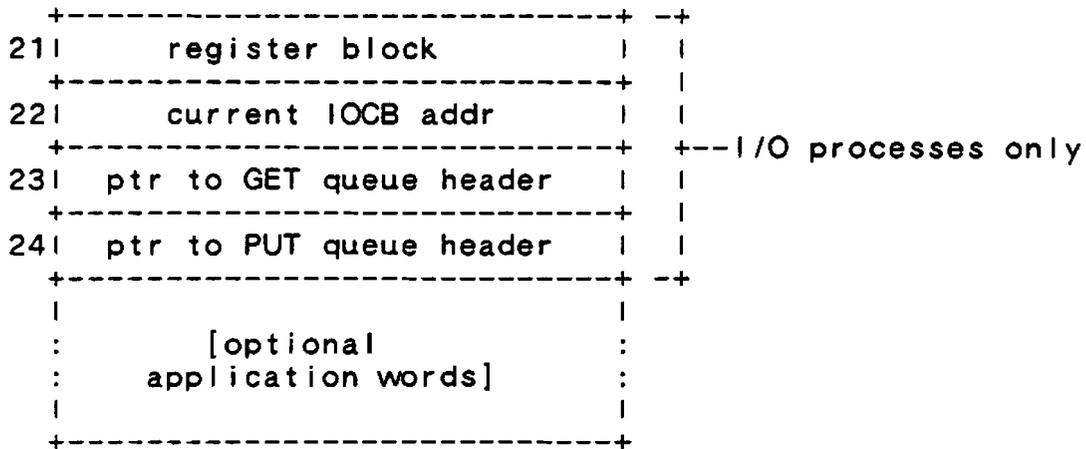
### 3.2 Process Control Block

A PCB is a group of 17 consecutive macromemory locations used by the firmware system and referenced by the address of the first location. As will be discussed in chapter 4, an additional 4 locations are required for I/O-associated processes. It may also be convenient for the application to associate some additional number of sequential locations with the PCB address for use by the application process, without any effect on the firmware system. The firmware system requires certain portions of the PCB to be properly initialized while the PCB is in the IDLE state. Except where noted in the PCB format description below, the firmware area is not accessed by the macrocode once the process has been activated. The PCB format is shown below.

PROCESS CONTROL BLOCK

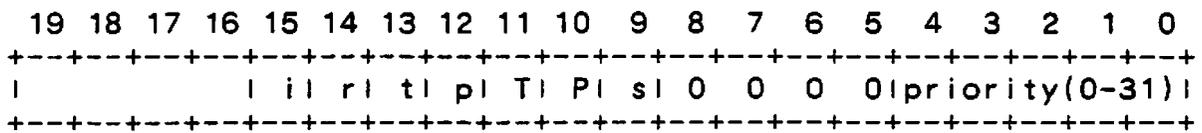
0	forward pointer			
1	backward pointer			standard queue item
2	header pointer			
3	state, priority			
4	goad queue forward pointer			
5	process type			
6	saved PC			
7	saved A register			
10	saved X register			
11	MBB ALU status			context area
12	saved B register			
13	saved SP register			
14	saved macromemory loc 1			
15	saved macromemory loc 2			
16	timeout time			
17	measurement time hi bits			
20	measurement time lo bits			40-bit measurement area

PROCESS CONTROL BLOCK



The standard queue item area is used to place the PCB on the various process system queues.

The state/priority word has the format:



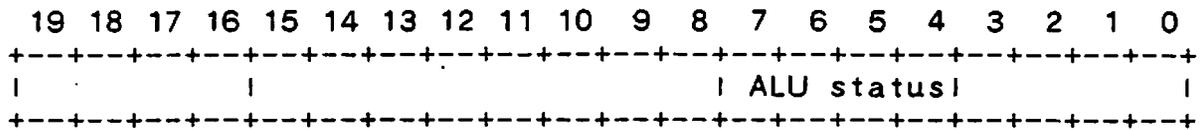
where

- i = 1 if the process is in the IDLE state,
- r = 1 if the process is in the READY state,
- t = 1 if the process is in the TIMING state,
- p = 1 if the process is in the PENDING state,
- T = 1 if the process is in the RETIMED state,
- P = 1 if the process is in the REPOKED state, and
- s = 1 if the process wants to skip after SPR.

The s bit and the goad queue forward pointer are used internally by the process system to decouple microinterrupt processing from process scheduling, as described in Appendix E.

The process type is 0 for non-I/O processes. Chapter 4 has a list of the I/O process types.

All of the values in the context area are 20 bits except for the saved flags word



where

ALU status = MBB (micromachine) bits used to emulate the virtual machine C (carry) and O (overflow) bits.

The timeout time is the 20-bit absolute value in 1.6 \*  
 millisecond ticks of the timeout clock when the process will time \*  
 out if it is in the TIMING state. \*

The 40-bit measurement area holds the high and low order \*  
 20-bit accumulated process time, in 100 microsecond increments, \*  
 when measurement mode is on. \*

Before a process is activated, the PCB must be initialized as follows:

- The PCB must be somewhere on the IDLE queue.
- The state/priority word must have the i bit set to 1, the priority set to the value desired by the macrocode, and all other bits set to 0.
- The process type must be set to 0 or some valid I/O type.
- The starting location for process execution must be in the PC word of the context, and all other context area values must be initialized according to the application.
- If measurement is contemplated, the measurement time words must be initialized to a known value.

When measurement mode is not on, the macrocode may access the measurement area in order to read the accumulated value or to reset it.

### 3.3 Instructions

There are eleven process-related instructions. Three are used for system initialization and process activation, four for guiding the scheduling of processes, and four for timing and miscellaneous functions.

Best and worst case scheduling overhead is included in the execution times given for instruction that may cause process scheduling.

The firmware performs consistency checks for malformed or illegal data structures, or improper instruction or instruction parameter use. Failure of any consistency check will result in a crash, some of the general ones being:

- #30, PCB address was 0
- #46, illegal PCB process type
- #67, attempted execution of process instruction in basic (non-NMFS) mode
- #35,36,37,40,41,42,  
various forms of inconsistent or illegal PCB contents
- #20,21,22,23,  
various forms of inconsistent or illegal queue structures

Additional crashes are discussed under the description of the instructions which may cause them.

## INITIALIZATION and TERMINATION

## 3.3.1 Initialization and Termination

The firmware system makes transitions between the basic and NMFS modes through execution of the NMFS instruction. The transition to or from NMFS performs a master clear of all I/O devices, putting them all into a known initial state.

INITIALIZATION and TERMINATION

NMFS                      NMFS Mode Control                      if A = 0, 22  
 0000030                      ----                      if A not 0, 281

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The I/O system is master cleared and all devices are set to a known state. The process system is cleared of all knowledge of any processes, and the 1.6 millisecond timeout clock is set to 0.

If the contents of the A register are 0, the next sequential instruction is executed. The contents of the A register are not affected.

If the contents of the A register are not 0, the value is taken as the starting address of a block of consecutive macromemory locations containing queue headers for the idle queue, ready queue, timing queue, and 32 pending queues (in order priority 0 -> priority 31). All queue headers except for the idle queue must be initialized to be empty. The idle queue must have at least one PCB. The first PCB on the idle queue is automatically activated and goaded, so that control is passed to the process represented by that PCB, with the initial PC and other context specified by the contents of the PCB context area. Unlike a normal goad, however, the PC is not automatically incremented before macroprogram execution begins.

Crashes:

- #52, empty idle queue at startup. The idle queue, addressed by the contents of the A register, had no PCB on it.

Processes are activated and deactivated by use of the APR (Activate Process) and DPR (Deactivate Process) instructions. A





## EXECUTION and SUSPENSION

## 3.3.2 Execution and Suspension

Processes are queued for immediate execution, subject to priority constraints, by using the GPR (Goad PRocess) instruction. A process may be goaded by any other process, including itself. A process suspends execution (i.e., dismisses) by using the SPR (Suspend PRocess) instruction. While it is executing, a process may lock out potential execution of all higher-priority processes by using the INH (INHibit) instruction, and then permit again the potential execution of all higher-priority processes by using the ENB (ENaBle) instruction.

EXECUTION and SUSPENSION

GPR  
0000043

Goad PProcess  
- --

39

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1

The contents of the X register are used as a PCB address to goad the corresponding process. If the process is in the READY or TIMING states, its PCB is placed at the end of the pending queue for the process's priority level and the process is put in the PENDING state. The contents of the PC word of the saved context are incremented by one and loaded back into the PCB. A scheduling scan is performed to determine if the process should be executed. If the process is in the PENDING, RETIMED, or REPOKED states, the PCB is left in place on the pending queue and the process is put in the REPOKED state. The contents of the X register are not affected.

Crashes:

#50. attempt to goad IDLE process.

EXECUTION and SUSPENSION

SPR	Suspend Process	180 if REPOKED
0000103	- --	193 ++ if RETIMED
		163 otherwise

```

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 | 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 | 1 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The current process is suspended from further execution and its context is loaded into the current PCB. If the process is in the PENDING state, the current PCB is placed on the end of the ready queue and the process is put in the READY state. If the process is in the RETIMED state, the current PCB is placed on the timing queue in a relative position determined by the value of the timeout word in the PCB, and the process is put in the TIMING state. If the process is in the REPOKED state, the current PCB is placed on the end of the appropriate pending queue, the contents of the PC word of the saved context are incremented by one and loaded back into the PCB, and the process is put in the PENDING state. A scheduling scan is performed to select the next process to execute.

Crashes:

- #65, attempt to SPR while INHibited. The process executed an INH instruction and then attempted to suspend itself without first executing the matching ENB instruction.
- #47, nothing to run. No processes remain that are scheduled for immediate execution.

EXECUTION and SUSPENSION

INH INHibit Other Processes  
0001001 ---

9

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-
1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-

The process system enters (or remains in) the inhibited state. No other process besides the current one may execute.

ENB ENaBle Other Processes 10  
0000401 -- - (10 if already enabled)  
(25 if higher priority waiting)

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-

The process system leaves (or remains out of) the inhibited state. A scheduling scan is performed if any higher-priority process has been scheduled during the inhibited state.

EXECUTION and SUSPENTION

UGS  
0000113

UnGoad Self  
- - -

13

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1

If the process executing this instruction is in the REPOKED state, it is returned to the PENDING state.

The UGS instruction may be executed by a process to prevent extra wakeups.

### 3.3.3 Timing and Miscellaneous

Processes may schedule themselves for deferred execution at a specified wakeup time relative to the current time using the TPR (Timeout Process) instruction. Timing is done on the basis of the firmware system's timeout clock, which is incremented every 1.6 milliseconds and has a range of almost 28 minutes using a 20-bit unsigned value. Since the timeout value is established while a process is executing, it is possible that by the time the process suspends itself, the absolute wakeup time may have passed. It is desirable that this case result in an immediate wakeup and not a full-period delay. Thus the convention is established that the TPR instruction specifies a signed displacement from the current time. This displacement is added to the unsigned 20-bit clock value and the result is saved in the PCB timeout word. The program-specified timeout range can then only be half the system timing range, or 838.86 seconds. When the PCB is about to be inserted on the timeout queue after the SPR has been executed, a check is made to see if the timeout value in the PCB is in the range between the current value of the timeout clock and that clock value plus 838.86 seconds. If it is, the PCB is inserted on the timing queue. Otherwise, the timeout

TIMING and MISCELLANEOUS

value is considered to be in the "past" window of 838.86 seconds, \*  
 and the process is considered to have "missed" the wakeup time. \*  
 In this case, the process is "timed-out" immediately and put back \*  
 on the pending queue.

TPR                      Timeout PProcess                      22  
 0000203                      -                      --

```

    19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 11
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the A register are added to the current value of the timeout clock and the result is loaded into the timeout word of the current PCB. The contents of the A register are not affected.

Cumulative execution times for all active processes in the system are collected when the firmware system is in measurement mode. Measurement mode is turned on by executing the MME (Measurement Mode Enable) instruction and turned off by executing the MMD (Measurement Mode Disable) instruction. When measurement \*  
 mode is on, the firmware accumulates actual execution time in 100 \*  
 microsecond increments in the 40-bit concatenation of two 20-bit \*  
 words (a range of approximately 3.5 years) for each PCB. When a \*

TIMING and MISCELLANEOUS

process context is loaded in order to begin execution of the \*  
 process, the current value of the firmware's 40-bit, 100 \*  
 microsecond clock is saved in temporary firmware registers. When \*  
 a process context is saved in order to switch execution to  
 another process, the saved clock time is subtracted from the  
 current clock time and the result is added to the accumulated  
 measurement time in the PCB; the result is then loaded back into  
 the PCB. When measurement mode is off, there is no measurement  
 overhead associated with context switching, and the macrocode can  
 read and/or initialize the accumulated values in each PCB.

MME Measurement Mode Enable 6  
 0000402 - - -

```

  19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 0 0 0 0 | 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

The process system enters (or remains in) measurement mode.

TIMING and MISCELLANEOUS

MMD Measurement Mode Disable  
0001002 - - -

6

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0

The process system leaves (or remains out of) measurement mode.

A process can always find the address of its PCB by executing the PCB instruction.

PCB Load PCB Into X Register  
0000020 ---

11

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

The address of the current PCB is loaded into the X register.

4 I/O System

Instruction Table

This table provides a summary, sorted by mnemonic, of the NMFS instructions contained in this chapter, including the page number on which complete information for each instruction can be found.

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
PDV	0001003	Poke Device Driver	I/O	47 + I/O	4-20
XDV	0000403	Execute Device Function	I/O	48 + I/O	4-21

#### 4 I/O System

In NMFS mode, the firmware system supports a number of different I/O device types, and permits the macrocode to interact with an arbitrary number of physical devices of each type. The limit on the aggregate number of physical interfaces is determined by the hardware fanout of the MBB and by the capacity of the microregister space to hold the state information for the interfaces. This I/O system presents a uniform interface to the macrocode and minimizes the number of device-dependent aspects of I/O operation. The chief characteristics of the I/O system are:

- all devices are full duplex,
- a process, and thus any priority level, is associated with each side (input and output) of an active device,
- data transfer between the device interfaces and the macrocode can be buffered to a depth controlled by the macrocode,
- input data buffers for several devices may be allocated from a common pool,
- switching the transfer of data to/from successive buffers is accomplished within the I/O firmware without need for macrocode intervention, and
- I/O completion can be used in a flexible way to goad the associated process into execution.

Each device type is supported by firmware I/O drivers that interface to the generalized NMFS I/O functions and perform device-specific operations.

#### 4.1 Device Operation

Each class of I/O interface is assigned two types, one each for input and output sides. Each physical interface is assigned two microregister blocks (one per side) which hold the state information for that interface and which serve as the "handle" for references to the interface by the macrocode. When a process is associated with an interface, it carries both the device type and handle in its PCB. The device type is used by the firmware system to select the correct firmware driver for I/O operations, and the handle is used by the driver to select the interface from among all the interfaces of that type served by the same driver.

Processes associated with I/O devices carry an extension to the PCB which governs the buffered interface for transferring data between the device and the macrocode. This buffered interface is the same for all device types and makes use of I/O Control Blocks (IOCBs) to specify input and output data areas, the sequential relationships between them, and the process-gating policy. The PCB I/O extension and IOCBs are described in 4.2 and 4.3 respectively.

The macrocode can interact directly with the device driver for a specific interface by using the I/O instructions PDV (Poke DeVice) and XDV (eXecute DeVice function), as well as APR and DPR. In each case, the interface is referenced via its associated process's PCB. PDV and XDV are described in 4.4. Device-specific aspects of the APR, DPR, PDV, and XDV instructions, IOCBs, and I/O operation are described in 4.5.

## 4.1.1 Firmware Drivers

There are a pair of device types assigned to every possible combination of generic device type and I/O board type. The generic device types are ARPANET 1822, BCP/CRC-24 synchronous, BOP/HDLC synchronous, and asynchronous control port. The I/O board types are MBP (processor), MII (IMP), MTI (terminal), and MSYNC (BOP/HDLC). The permissible combinations yielding device types are:

Type Code	Board	Interface	Generic Type	Side
1	MI I	2651	BCP/CRC-24 synchronous	Input
2	MI I	2651	BCP/CRC-24 synchronous	Output
3	MI I	1822	ARPANET 1822	Input
4	MI I	1822	ARPANET 1822	Output
5	MBP	2651	Asynchronous control port	Input
6	MBP	2651	Asynchronous control port	Output
7	MTI	2652	BOP/HDLC synchronous	Input
8	MTI	2652	BOP/HDLC synchronous	Output
9	MTI	1822	ARPANET 1822	Input
10	MTI	1822	ARPANET 1822	Output
11	MSYNC	2652	BOP/HDLC synchronous	Input
12	MSYNC	2652	BOP/HDLC synchronous	Output
13	MI I	2651	BCP/CRC-24 synchronous satellite	Input
14	MI I	2651	BCP/CRC-24 synchronous satellite	Output

The device drivers respond to macrocode I/O instructions and microinterrupts. When called as a result of a macrocode instruction, the device driver performs the requested function

## FIRMWARE DRIVERS

immediately and returns control to the macroprogram. In some cases, the function may involve starting or stopping buffered I/O, which proceeds in parallel to macroprogram execution. When a device driver receives a microinterrupt from the hardware interface, it also processes it immediately and clears the interrupt. Microinterrupts occur when characters of data arrive or are sent, or when a normal or abnormal exception condition is detected. A microinterrupt may result in activity of the buffered I/O (e.g., filling one IOCB and moving on to the next), and also may result in a goad of the associated process (e.g., requested I/O completion).

The ultimate latency constraint in the C/30's ability to handle high-speed I/O is in the delay to respond to microinterrupts. Since each character of input or output gives rise to a microinterrupt, the firmware system is structured so that microcode execution on behalf of macrocode emulation or even process system activity is limited to a maximum number of microcycles between polls for microinterrupts. Once one or more microinterrupts occur, they are all serviced before continuing with the rest of the firmware system execution. As a point of reference, there are 1,058 microcycles during one character time

on a 56 kilobits/second synchronous line, and 258 microcycles during one character time on a 230 kilobits/second line.

Each device driver presents a standard microcode interface to the general I/O firmware, consisting of entry points to be called for each of the APR, DPR, XDV, and PDV instructions. The device driver carries out any device-specific functions implied or defined by the instructions, on the specific interface determined by the register block in the PCB. Thus, the device driver entry points are serially reentrant on behalf of all the interfaces of that type.

#### 4.1.2 Configuration

An interface handle is the address of the microregister block associated with that interface. The allocation of device register blocks in microregister space is a function of the C/30 I/O board configuration, the logical population of devices used on each board, and the size of the register blocks used by each driver. Section 6.3.1 describes the facility for producing the microcode for specific board and device configurations and Appendix B has some sample configurations.

Since there is no mechanism for the macrocode or microcode to read or otherwise "discover" the I/O board configuration on which it is running, the configuration and/or register block assignments must be constructed by the application. In situations where the macroprogram is capable of adapting to several different configurations, this information is typically loaded from a cassette during the bootload phase.

### 4.1.3 I/O Operation

When an I/O-associated process is activated, the corresponding interface is initialized and the register block is bound to the process via a back pointer to its PCB. By convention, the input side of an interface is always activated first. When an I/O-associated process is deactivated, its corresponding interface is cleared and the register block is disconnected from the PCB.

The PCB extension for I/O-associated processes specifies GET and PUT queues (see 4.2) of IOCBs for that process. The microcode-buffered I/O system dequeues IOCBs from the GET queue, uses them to perform input or output of data, and then enqueues the IOCBs on the PUT queue when the I/O is complete.

To use this buffered system, the macrocode enqueues one or more IOCBs on the GET queue. For input interfaces, each IOCB specifies the location and extent of the data area to be filled, and the completion conditions for goading the associated process. For output interfaces, each IOCB specifies the location and extent of the data area to be output, any special conditions (such as end of frame), and the completion criteria for goading

## I/O OPERATION

the associated process.

Once the macrocode has enqueued IOCBs on the GET queue, it guarantees the attention of the interface by issuing a PDV instruction. In the general case, the interface may not always need a PDV to start processing IOCBs on the GET queue, so IOCBs should always be ready for processing before they are placed on the GET queue. The interface then processes the IOCBs on the queue, in order, and one at a time. IOCB processing continues automatically until the GET queue is exhausted. If the GET queue becomes exhausted, it is necessary to PDV again when new IOCBs are added to the GET queue in order to guarantee that the attention of the interface is recaptured.

The device driver normally dequeues an IOCB as soon as the PDV is executed. For input interfaces, the IOCB is held in anticipation of input. When input arrives, the data is stored in the data area specified by the IOCB. When the input terminates, or the IOCB data area is filled, or there is some other termination condition, the IOCB is enqueued on the PUT queue and the next IOCB is dequeued from the GET queue. For output devices, all the data stored in the specified data area is output. When output of this data terminates, or there is some

other termination condition, the IOCB is enqueued on the PUT queue and the next IOCB is dequeued from the GET queue.

When an IOCB is enqueued on the PUT queue, the driver marks the IOCB with the appropriate completion status. It then examines the IOCB to determine if the current completion warrants a goad of the associated process, and if so, does the firmware-simulated GPR. The effects of this goad are identical to those of the GPR instruction. Section 4.3 describes the device-independent aspects of the IOCB completion status and goading conditions, and section 4.5 describes any additional device-dependent ones.

IOCBs are dequeued from the PUT queue by the macrocode, typically after being goaded by IOCB completion. For input interfaces, the IOCBs contain the location and extent of the input data, completion status, and any special conditions (such as end of frame). For output interfaces, the IOCBs contain the location and extent of output data, completion status, and any special conditions originally set by the macrocode.

In addition to directing the interface to process buffered I/O via the PDV instruction, the macrocode may also request that

## I/O OPERATION

the interfaces perform some immediate, device-specific operation by means of the XDV instruction. These operations are always device-specific, but typically include reset, status, and looping commands.

All I/O is in multiples of 16-bit words; user (Host) programs must perform any and all packing and unpacking. The firmware performs consistency checks for malformed data structures or improper instruction or instruction parameter use. Failure of any consistency check will result in a crash, some of the general ones being:

- #56, illegal IOCB size (0 or not byte multiple).
- #60, illegal IOCB address (e.g., 0).
- #67, attempted execution of an I/O instruction in basic (non-NMFS) mode.
- #20,21,22,23,  
various forms of inconsistent or illegal IOCB queue structures.

Additional crashes are discussed under the description of the instructions that may cause them.

#### 4.2 Process Control Block Device Header

The PCBs of I/O-associated processes have a 4-word I/O device area extension immediately following the standard 17-word NMFS area.

```

+-----+
21|      register block      |
+-----+
22|      IOCB address       |
+-----+
23| GET queue header pointer |
+-----+
24| PUT queue header pointer |
+-----+

```

The register block is the "handle" for the specific interface associated with the process, and is a micro-register address as specified by the configuration procedure.

The IOCB ADDRESS word is used by the firmware driver to store the current IOCB pointer while input or output is in progress. During this time, the IOCB is on neither the GET queue nor the PUT queue.

The GET and PUT queue header pointers are the addresses of the queue headers designated to serve as GET and PUT queues. Since the PCB contains pointers to the queue headers, rather than

## PROCESS CONTROL BLOCK DEVICE HEADER

the queue headers themselves, it is possible for different PCBs to share the same queues. For example, a group of similar input processes could share a common GET queue consisting of a "free list" of IOCBs. This technique allows the application to take advantage of statistical averaging and to allocate fewer IOCBs to the group of input processes than it would if each process had to be separately buffered.

### 4.3 I/O Control Block

An IOCB is a group of ten consecutive macromemory locations used by both the firmware and the macrocode (but not simultaneously, see below), and referenced by the address of the first location. It may also be convenient for the application to associate additional sequential locations with the IOCB address for use by application processes, without any effect on the firmware I/O system.

Before the IOCB is enqueued on a GET queue, and after it has been enqueued on a PUT queue, the macrocode may reference and/or modify any portion of it, since during that time it is not within the domain of the firmware I/O system. Whenever an IOCB is on a GET queue or serving in-progress I/O, it is subject to reference/modification by the firmware and must not be referenced by the macrocode. The only exception to this rule is that the macrocode may execute DEQ instructions to remove IOCBs from the GET queue before the firmware uses them.

The IOCB format is:

```

+-----+ ---+
0| forward pointer | |
+-----+ |
1| backward pointer | | - standard queue item
+-----+ |
2| header pointer | |
+-----+ ---+
3| data size (bits) |
+-----+
4| data address |
+-----+
5| status and control |
+-----+ ---+
6| hi-order GETq tstamp | |
+-----+ |
7| lo-order GETq tstamp | | filled in only if IOCB
+-----+ | - timestamping is turned
10| hi-order PUTq tstamp | | on (t bit is turned on
+-----+ | in iocb.flags word)
11| lo-order PUTq tstamp | |
+-----+ ---+

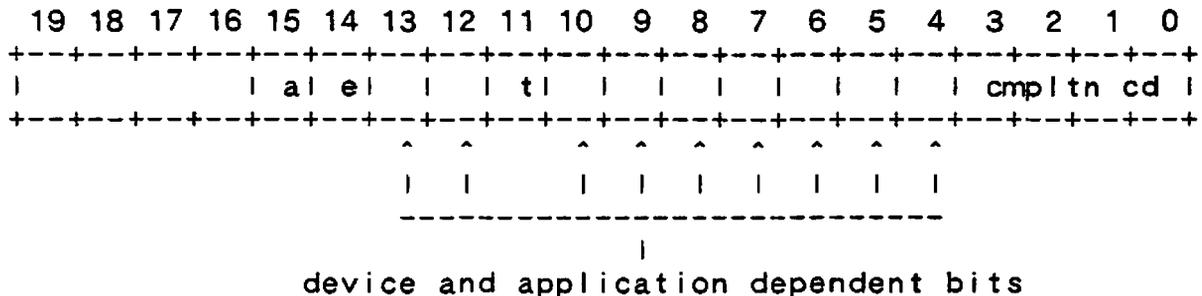
```

The standard queue item area is used to place the IOCB on GET and PUT queues.

The data size is the maximum allowed transfer prior to input, the actual transfer after input, and always the actual transfer before and after output. The size is expressed in bits; however, the I/O system only supports transfers in multiples of 8-bit bytes or 16-bit words, so the low order three bits should always be 0. The size is modified only by input drivers to

reflect the actual amount of data transferred into the available input area. The data address is the starting location of the data area and is not modified by the firmware.

The status and control word has the format:



where

- a = 1 if the process is always to be goaded when the IOCB is enqueued on the PUT queue.
  - e = 1 if the process is to be goaded when the IOCB is enqueued on the PUT queue AND the completion code is nonzero (error).
  - t = 1 if this IOCB is to use timestamping.
- completion code = the completion status for the IOCB. A value of 0 indicates completion was normal, and a nonzero value usually indicates an error. The significance of nonzero completion codes is device dependent.

The a, e, and t flags are always under the control of the macrocode and are never modified by the firmware. The firmware will goad the process if any of the goading criteria (including

device-dependent ones) are true. The completion code is always set by the firmware, which ignores any previous contents.

Bits 13, 12 and 10-4 are reserved for device-dependent use, described in section 4.5. As general rule, the macrocode may assume that all bits are preserved by the firmware except those specifically designated as firmware driver settable (e.g., the completion code above, and some device-dependent bits).

IOCB timestamping is a facility for recording in an IOCB the times at which the NMFS microcode gets that IOCB off the GET queue and puts it on the PUT queue. This can be used for such applications as measuring line speeds.

A timestamp is a two-word quantity. It records a value of the NMFS clock, in 100 us units. The first word of the timestamp holds the high-order clock bits, and the second word holds the low-order clock bits.

IOCB timestamping is in effect if bit 11. (0004000) is set in the IOCB.FLAGS word of the IOCB. If this bit is off, IOCB timestamping is not performed. This bit is set/cleared by the macrocode.

4.4 Instructions

There are two I/O-related instructions. Three process-related instructions (NMFS, APR and DPR) also have effects on the I/O system.

PDV                      Poke DeVice Driver                      47 + I/O PDV  
 0001003                      -       -       -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the X register are used as a PCB address, and the contents of the device type and the register block words of that PCB are used to initiate I/O on the corresponding interface. If I/O is already active, or if the device type is 0 (not an I/O process), the instruction is a no-operation. Otherwise, the device driver dequeues an IOCB from the GET queue specified by the PCB and initiates I/O. If the driver is unable to get an IOCB because the GET queue is empty, I/O is not initiated and the instruction is a no-operation. In all cases, the contents of the X register are not affected.

Crashes:

- #53. PCB in register block and X do not match. The back pointer in the register block is not the same as the PCB parameter of the instruction used to locate the register block.

XDV                      EXecute DeVice Function                      48 + I/O XDV  
 0000403                      -                      - -

```

    19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    | 0 0 0 0 | 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 |
    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    
```

The contents of the X register are used as a PCB address, and the contents of the device type and the register block words of that PCB are used to request immediate execution of a device-dependent function by the device driver. The function is specified by the contents of the A register and the contents of the B register are used as a function parameter. The effects of the function on the interface and on the resulting contents of the A and B registers are device-dependent and described in section 4.5. If the device type is 0 (not an I/O device), the instruction is a no-operation. In all cases, the contents of the X register are not affected.

Crashes:

- #32. XDV of idle process.
- #53. PCB in register block and X don't match (see PDV).
- #54,55. bad XDV arguments; the function code in A or the parameter in B is illegal for that device.

#### 4.5 Device Descriptions

The following sections contain descriptions of all the device-dependent aspects of the supported generic device types. The description for each generic type includes:

DEVICE TYPE	the input and output device types covered
INPUT IOCB	additional flags, completion codes
OUTPUT IOCB	additional flags, completion codes
APR	the effects of input and output APR instructions
DPR	the effects of input and output DPR instructions
PDV	the effects of input and output PDV instructions
XDV	the codes, parameters, and effects of input and output XDV instructions
INPUT PROCESSING	(narrative explanation)
OUTPUT PROCESSING	(narrative explanation)

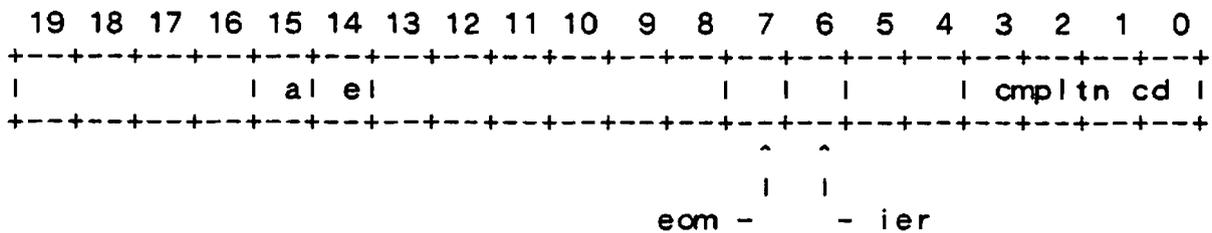
## 4.5.1 ARPANET 1822

## DEVICE TYPE

This generic device covers input device types 3 and 9 and output device types 4 and 10 (see Table on page 4-6). It serves the ARPANET asynchronous serial host/IMP interface with a 4-way handshake as described in BBN Report 1822 [Reference 3]. Data is exchanged in units of messages, which are delimited by the end-of-message signal. Data transfer of each bit occurs only when both sender and receiver are ready for the transfer. There is no inherent limit on the message size, and it may be any bit multiple, although the message is always treated within the IOCB framework as containing an integral multiple of 16-bit words. The hardware interface controller is a custom design which implements the requirements of the IMP side of the 1822 interface.

INPUT IOCB

The IOCB status and control word has the format:



where

eom = 1 if the IOCB contains the last data of the message. The eom flag is always set or cleared by the firmware, which ignores any previous contents.

ier = 1 if eom = 1 and the host's ready line flapped during input of the data in the IOCB, or any of the immediately preceding IOCBs with eom = 0. The macro program should discard the data in the IOCBs when this occurs.

completion code = 1 if the IOCB was aborted by the macrocode.

The IOCB data size is always a multiple of 16-bit words. When the firmware detects an end of message signal, it adds padding to complete the current 16-bit word. The padding consists of at least a single bit with value 1 and from 0 to 15 bits of value 0. Note that if the last data bit occurs on a 16-bit word boundary, the firmware will add an entire additional

word with bit 15 set to 1 and bits 14-0 set to 0.

#### OUTPUT IOCB

The IOCB status and control word has the same format and meaning as the input IOCB, except that the eom flag is always under the control of the macrocode and is never modified by the firmware. The completion codes are the same as for input.

The IOCB data size is always a multiple of 16-bit words. When the eom flag is on, the firmware asserts the end of message signal on the last bit of the last word of the data. There is no padding on output.

#### APR

The APR entry points of the input and output drivers store the PCB pointer in the register block.

The input side of the device must be APRed first.

## DPR

The DPR entry points of the input and output drivers clear the 1822 device hardware. They do not delete the PCB pointer or clean up any IOCBs that might be in progress.

## PDV

The PDV entry points of the input and output drivers start up I/O on that interface if it is not already processing an IOCB. The drivers attempt to get an IOCB from the GET queue and quit if there are none. Otherwise, the input driver holds onto the IOCB in anticipation of input, and the output driver begins to transmit the data in the IOCB.

## XDV

The XDV entry points of the input and output drivers are identical. Some functions affect both input and output sides, and some affect only the side whose PCB is in the X register; individual XDV entries specify which is the case. Except where noted, the contents of the A and B registers are not affected. The functions are:

- 0 - ABORT. If there is a current IOCB, set its completion code to 1 and enqueue it on the PUT queue. Then continue normal operation.
- 1 - RAISE. Set the IMP Master Ready line to up.
- 2 - LOWER. Set the IMP Master Ready line to down.
- 3 - STATUS. Read the Control and Status Register (CSR) for the full duplex device into the A register. The status bits have the format:

```

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                                     | |                                     | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                                     ^                                     ^                                     ^
                                     |                                     |                                     |
                                     hrd                                 lpd                                 ird

```

where

ird = 1 if IMP ready line is up,

lpd = 1 if interface is looped,

hrd = 1 if host ready line is down,

and all other bit positions are significant only to the firmware.

- 4 - LOOP. Crosspatch the 1822 hardware interface and abort whichever side is represented by the PCB. The other side is not aborted.
- 5 - UNLOOP/RESET. Reset the hardware interface (IMP ready line up, interface uncrosspatched) and abort whichever side is represented by the PCB. The other side is not aborted.

## INPUT PROCESSING

Before input processing can start, the macrocode must enqueue one or more "empty" IOCBs on the GET queue. The macrocode initializes each IOCB's data size, data address, and control/status words. The data address and size specify where the IOCB's data input should go, and how much input the IOCB can absorb. The goading condition flags are set based on when the macrocode input driver wishes to be executed.

To start input processing, the macrocode executes a PDV using the PCB of the input driver process. The firmware then dequeues the first IOCB from the GET queue and holds it in anticipation of input. If input from the host has been waiting, it starts filling the IOCB. A copy of the IOCB pointer is stored in the "current IOCB" word of the PCB.

As input data arrives, the firmware stores consecutive words into the data area specified by the IOCB and decrements the size of space remaining. Processing of the IOCB terminates when the end of message is reached, the IOCB data area is exhausted, or the input is aborted:

- The firmware considers the end of message condition to be true only when the full padding has been placed in macromemory. Thus, if the last data bit exactly fills the last word of the IOCB, the end of message does not occur until the next IOCB. When the end of message occurs, the actual data size is copied into the IOCB data size word. The microcode sets the eom flag in the control/status word of the IOCB. It also sets the completion code to 0. If the IOCB contains the last word of a message, the hardware's host error flop is read and cleared. If the flop contained 1, then the ier bit in the IOCB is set to 1. The error flop will contain a 1 if the host's ready line was down any time since the last time the flop was cleared.
- If the remaining data size for the IOCB goes to zero, the IOCB has been exhausted and input must be switched to a new IOCB. The data size word of the IOCB is left at its original value. The eom flag and completion code are set to 0.
- If the input is aborted by the macrocode, the current data size is copied into the IOCB, the eom flag is set to 0, and the completion code is set to 1.

In all cases, the current IOCB word of the PCB is cleared, the IOCB is enqueued on the PUT queue, and the macrocode input driver is goaded if any of the enabling conditions (a and e flags) are true. Then the firmware driver attempts to dequeue a new IOCB from the GET queue, and if successful, continues input as described above.

If there are no IOCBs left on the GET queue, input processing lapses until a new PDV is executed. Since the nature

of the 1822 interface permits either side to wait indefinitely for the other to supply or receive data, there will be no data loss while waiting for a new PDV.

#### OUTPUT PROCESSING

Before output processing can start, the macrocode must enqueue one or more IOCBs on the GET queue. The data size and data address specify the location and extent of data to be output on behalf of that IOCB. In addition, the eom flag is set in any IOCB that contains the last data bit of a message. The goading condition flags are set based on when the macrocode output driver wishes to be executed.

To start output processing, the macrocode executes a PDV using the PCB of the output driver process. The firmware then dequeues the first IOCB from the GET queue and starts to transmit the data. Actual transmission is dependent on the host also accepting the data. A copy of the IOCB pointer is stored in the current IOCB word of the PCB.

The firmware transmits consecutive data words from the specified data area and decrements the count of remaining words.

Processing of the IOCB terminates when the IOCB data is exhausted, or the output is aborted:

- If the data is exhausted, and the eom flag is on, the firmware causes the 1822 device hardware to assert the end of message signal. The completion code is set to 0 (eom or not).
- If the output is aborted by the macrocode, the completion code of the IOCB is set to 1.

In all cases, the current IOCB word of the PCB is cleared, the IOCB is enqueued on the PUT queue, and the macrocode output driver process is goaded if any of the enabling conditions (a and e flags) are true. Then the firmware driver attempts to dequeue a new IOCB from the GET queue, and if successful, continues output as described above.

If there are no IOCBs left on the GET queue, output processing lapses until a new PDV is executed. As with input, there will be no data loss (to the host) while waiting for a new PDV to start output, even in the middle of a message.

## 4.5.2 BCP/CRC-24 Synchronous

## DEVICE TYPE

This generic device covers input device type 1 and output device type 2. It serves a synchronous serial line which uses the bisync byte control protocol (byte stuffing) for framing and data transparency, and a special 24-bit cyclic redundancy checksum (CRC) for error detection. There is no inherent limit to the frame size but it must be an integral multiple of 16-bit words, exclusive of the framing and CRC. The hardware interface uses a 2651 Programmable Communications Interface (PCI) chip operating in synchronous mode. The CRC is performed by the driver microcode using a table lookup technique.

## INPUT IOCB

The IOCB status and control word uses no additional device-dependent bits. The nonzero completion codes are:

- 1 - some input error occurred: macrocode abort, CRC error, or IOCB overrun.

## OUTPUT IOCB

The IOCB status and control word uses no additional device-dependent bits. The nonzero completion codes are:

- 1 - the IOCB was aborted by the macrocode.

## APR

The APR entry point of the input and output drivers cause a hardware reset of the 2651 and the initialization of the register block, including storing of the PCB in the register block.

The input side of the device must be APRed first.

## DPR

The DPR entry points of the input and output drivers cause a hardware reset of the 2651 and clearing of the register block, including deletion of the PCB pointer. Any IOCB in progress is aborted and placed on the PUT queue, and further I/O processing is stopped.

## PDV

The PDV entry points of the input and output drivers ignore the PDV if an IOCB is already being processed. Otherwise, the input driver attempts to get an IOCB from the GET queue, and, if successful, holds onto it in anticipation of input. Similarly, the output driver attempts to get an IOCB from the GET queue, and if successful, begins to transmit a new frame. In either case, if the attempt is unsuccessful, the PDV is ignored.

## XDV

The XDV entry points of the input and output drivers are identical. Some functions affect both input and output sides, and some affect only the side whose PCB is in the X register. The contents of the A and B registers are not affected. The functions are:

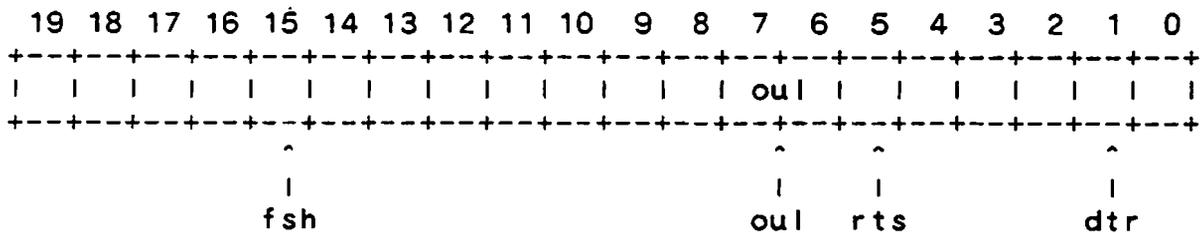
0 - NOP. This is a no-operation.

1 - ENABLE. Reset the 2651 PCI and abort any IOCB on the side specified by the PCB. Restore the signals rts, dtr, and xtr (see SET CONTROL, below) to their state before the ENABLE. Attempt to restart I/O operation on that side. \*  
\*  
\*

2 - DISABLE. Reset the 2651 PCI and abort any IOCB on the

side specified by the PCB. Clear rts, dtr, and xtr. Do not restart I/O operation on that side. \*

- 3 - LOOP INTERFACE. Reset the 2651 PCI and abort any IOCB on the side specified by the PCB. Put the 2651 in internal loopback mode and attempt to restart I/O operation on that side.
- 4 - LOOP MODEM. Abort any IOCB on the side specified by the PCB, assert the RTS signal to loop the modem, and attempt to restart I/O operation on that side.
- 5 - UNLOOP. Clear any loopback conditions by doing an ENABLE.
- 6 - unused.
- 7 - unused.
- 10 - SET CONTROL. Write the Control Register (CR) for the full duplex device using the contents of the B register. The contents of the B register are not affected. This XDV may be performed any time when the device is APR'd. The contents of the B register have the format: \*



where

- fsh = 1 if the current IOCB, if any, is to be placed on the PUT Queue with an error indication at the end of the XDV operation, and = 0 otherwise.
- oul = 3 if the full duplex device is to be outward looped. When the device is outward

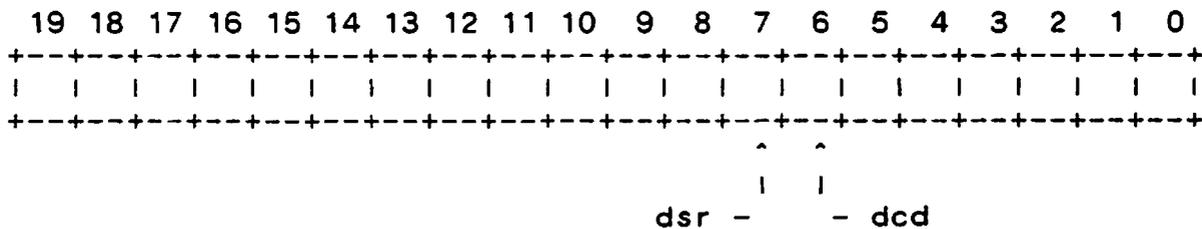
looped, the hardware will copy each data byte arriving on the input side of the interface to the output side of the interface. While in this state IOCBs are not serviced by either side of the device.

oul = 0 if the full duplex device is to have outward looping terminated, or to remain in the state of having outward looping turned off. This state is also set by the ENABLE and UNLOOP XDV's.

rts = 1 if the modem control signal Request to Send (RTS) is to be asserted, and = 0 if RTS is not to be asserted. RTS is also set by the operation of the LOOP MODEM XDV, and cleared by the ENABLE and UNLOOP XDV's.

dtr = 1 if the modem control signal Data Terminal Ready (DTR) be asserted, and = 0 if DTR is not to be asserted. DTR is cleared by the ENABLE and UNLOOP XDV's.

11 - READ STATUS. Read the Status Register (SR) of the full duplex device into the B register. The value placed into the B register will have the format:

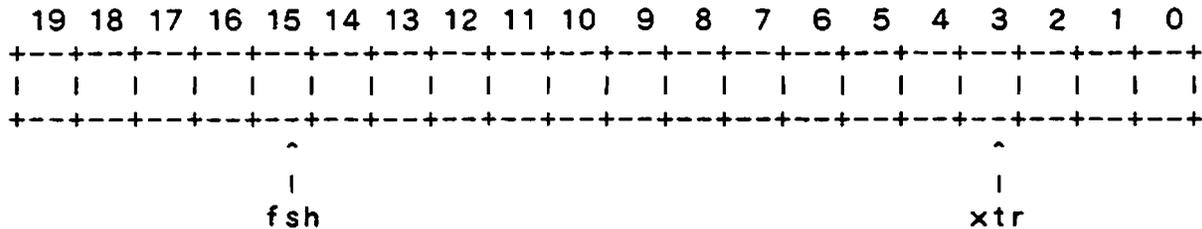


where

dsr = 1 if the modem is presently asserting the Data Set Ready (DSR) status signal.

dcd = 1 if the modem is presently asserting the Data Carrier Detect (DCD) status signal.

12 - SET EXTRA CONTROL LINE. Set the state of the extra control line associated with the full duplex device. The contents of the B register have the format



where

fsh = 1 if the current IOCB, if any, is to be placed on the PUT queue with an error indication at the end of the XDV operation.

xtr = 1 if the extra control line provided by "bit 3" of the Control and Status register of the MII is to be asserted, and = 0 if the control signal is not to be asserted. This control line is normally not asserted.

INPUT PROCESSING

Before input processing can start, the macrocode must queue one or more "empty" IOCBs on the GET queue. The macrocode initializes each IOCB's data size, data address, and control/status words. The data address and size specify where that IOCB's data input should go, and how much input the IOCB can

absorb. Since the driver uses one IOCB per frame on the line, the data area size must be large enough to accommodate the maximum-sized expected frame in order to guarantee that no data will be lost. The goading condition flags are set based on when the macrocode input driver wishes to be executed.

To start input processing, the macrocode executes a PDV using the PCB of the input driver process. The firmware then dequeues the first IOCB from the GET queue and holds it in anticipation of input. A copy of the IOCB pointer is stored in the "current IOCB" word of the PCB.

When a frame begins to arrive, the firmware stores consecutive data words (but not framing characters) into the data area specified by the IOCB and decrements the size of space remaining. Processing of the IOCB terminates when the end of frame is reached, the IOCB data area is exhausted, or the input is aborted:

- The firmware detects the end of frame sequence and does not store these characters in the IOCB. It also compares the accumulated CRC with the transmitted CRC and determines if there was a transmission error. The actual data size is copied into the IOCB data size word. The completion code is set to 0 if the CRC check was OK, and to 1 if not.

- If the remaining data size for the IOCB goes to zero without the end of frame being detected, there is an input overrun condition. The data size word of the IOCB is left at its original value, and the completion code is set to 1. The rest of the input frame is discarded before continuing with a new IOCB.
- If the input is aborted by the macrocode, the current data size is copied into the IOCB and the completion code is set to 1. The rest of the input frame is discarded before continuing with a new IOCB.

In all cases, the current IOCB word of the PCB is cleared, the IOCB is enqueued on the PUT queue, and the macrocode input driver process is goaded if any of the enabling conditions (a or e flags) are true. Then the firmware driver attempts to dequeue a new IOCB from the GET queue. If successful, input continues as described above.

If there are no IOCBs left on the GET queue, input processing lapses until a new PDV is executed -- all arriving input is discarded. Whenever input processing has lapsed and a frame is in progress on the line and the macrocode executes a PDV of the input driver; a new IOCB is then dequeued from the GET queue but is not used until the remainder of the current frame is discarded and a new frame arrives.

## OUTPUT PROCESSING

Before output processing can start, the macrocode must enqueue one or more IOCBs on the GET queue. The data size and data address specify the location and extent of the data frame to be output on behalf of that IOCB. The goading condition flags are set based on when the macrocode output driver wishes to be executed.

To start output processing, the macrocode executes a PDV using the PCB of the output driver process. The firmware then dequeues the first IOCB from the GET queue and immediately begins to transmit the frame on the line. A copy of the IOCB pointer is stored in the current IOCB word of the PCB.

The firmware transmits consecutive data words from the specified data area onto the line and decrements the count of remaining words. Processing of the IOCB terminates when the IOCB data is exhausted or the output is aborted:

- If the data is exhausted, the microcode terminates the frame on the line and sends the accumulated CRC. The completion code is set to 0.
- If the output is aborted by the macrocode, the firmware merely stops sending data and reverts back to interframe fill. The completion code is set to 1.

In all cases, the current IOCB word of the PCB is cleared, the IOCB is enqueued on the PUT queue, and the macrocode output driver process is goaded if any of the enabling conditions (a and e flags) are true. Then, the firmware driver attempts to dequeue a new IOCB from the GET queue. If successful, output continues as described above. If a new IOCB is not available, then output lapses until a new PDV is executed.

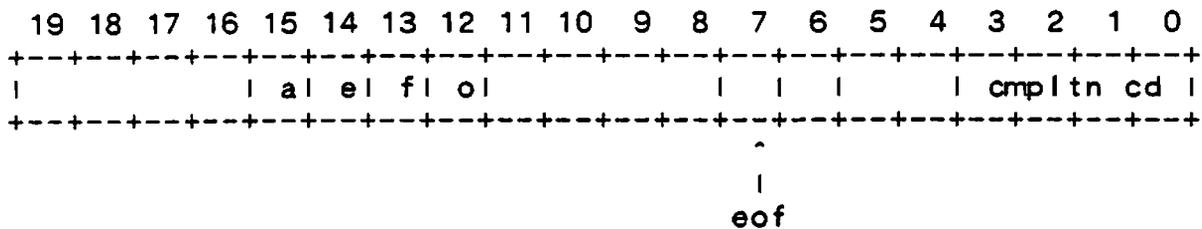
### 4.5.3 BOP/HDLC Synchronous

#### DEVICE TYPE

This generic device covers input device types 7 and 11 and output device types 8 and 12 (see Table p. 4-6). It serves a synchronous serial line which uses a bit-oriented protocol (bit-stuffing) for framing and data transparency, and the CCITT-16 cyclic redundancy checksum (CRC) for error detection. There is no inherent limit to the frame size but it must be an integral multiple of 8-bit bytes. The hardware interface uses a 2652 Multi-Protocol Communications Controller (MPCC) chip operating in BOP mode.

INPUT IOCB

The IOCB status and control word has the format



where

- f = 1 if the process is to be goaded when the IOCB is enqueued on the PUT queue AND it contains the last data bit of the frame.
- o = 1 if the designated data area starts at the odd byte (bit 7) of the first word, rather than at the even byte (bit 15), and
- eof = 1 if the IOCB contains the last data bit of the frame.

The f and o flags are always under the control of the macrocode and are never modified by the firmware. The eof flag is always set or cleared by the firmware, which ignores any previous contents.

The valid nonzero completion codes are:

- 1 - the IOCB was aborted by the macrocode.
- 2 - a data overrun occurred right after the current IOCB because no more IOCBs were available to receive the rest of the frame.

- 3 - the frame had a CRC error.
- 4 - the frame had an abort sequence.
- 5 - the frame ended in a byte of less than 8 bits.
- 6 - a data overrun occurred while filling this IOCB because the microcode failed to service one or more character-available interrupts soon enough.

#### OUTPUT IOCB

The IOCB status and control word has the same format and meaning as the input IOCB, except that the eof flag is always set by the macrocode and is never modified by the firmware. The valid nonzero completion codes are:

- 1 - the IOCB was aborted by the macrocode.
- 2 - a data underrun occurred right after the current IOCB because no more IOCBs were available to complete the rest of the frame.
- 6 - a data underrun occurred while emptying this IOCB because the microcode failed to service one or more transmitter-ready interrupts soon enough.

#### APR

The APR entry points of the input and output drivers cause a hardware reset of the 2652 and the initialization of the input or output register block, including storing the PCB pointer in the

register block.

The output side of the device must be APRed first.

#### DPR

The DPR entry points of the input and output drivers cause a hardware reset of the 2652, cleanly abort processing of the current IOCB (and enqueue it on the PUT queue), and clear the register block, including deleting the PCB pointer. If an IOCB is aborted, it will have a completion code of 1.

#### PDV

The PDV entry points of the input and output drivers ignore the PDV if an IOCB is already being processed. Otherwise, the input driver attempts to get an IOCB from the GET queue, and, if successful, holds onto it in anticipation of input. Similarly, the output driver attempts to get an IOCB from the GET queue, and if successful, begins to transmit a new frame. In either case, if the attempt is unsuccessful, the PDV is ignored.

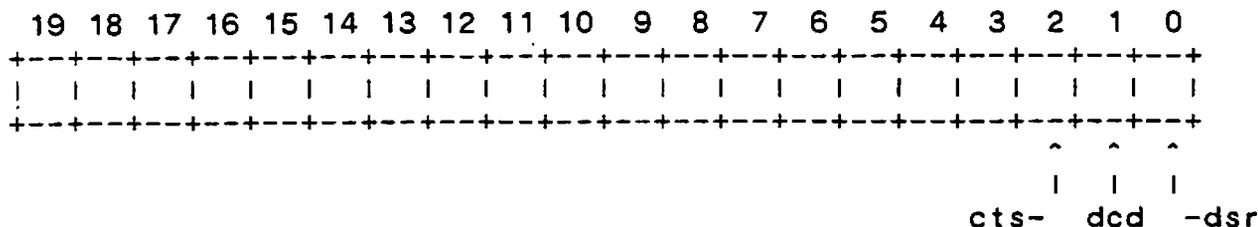
## XDV

The XDV entry points of the input and output drivers are identical. Some functions affect both input and output sides, and some affect only the side whose PCB is in the X register. Except where noted, the contents of the A and B register are not affected. The functions are:

- 0 - NOP. This is a no-operation.
- 1 - ABORT. If there is a current output IOCB, terminate the output data stream with an ABORT++ sequence. Set the completion code of the current IOCB to 1, place the IOCB on the PUT queue, and continue normal operation. If there is no current output IOCB, do nothing. ++ \*
- 2 - DISABLE. Disable the 2652 MPCC and ABORT both input and output sides, except do not restart I/O operation. Clear the state of the modem control signals (see SETSTAT, below). \*
- 3 - ENABLE. Disable, then restore the modem control signals to their state before the ENABLE, then enable the 2652 MPCC and abort the current side, except always restart I/O operation if a GET queue is not empty. \*
- 4 - GETSTAT. Read the Control and Status Register (CSR) for the full-duplex device into the B register. The status bits in the B register have different meanings for the 7/8 device and the 11/12 device. This XDV may be performed any time when the device is APR'd. \*

++ ONLY the "ABORT" operation will cause an abort sequence to be sent. Other uses of the word ABORT refer to all the other actions taken by the ABORT xdv.

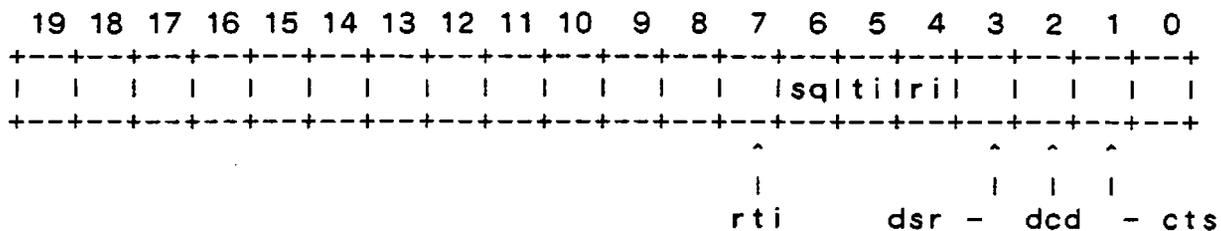
For 7/8, the format of the B register is:



where

- dsr = 1 if DATA SET READY signal is on,
- dcd = 1 if DATA CARRIER DETECT signal is on,
- cts = 1 if CLEAR TO SEND signal is on,
- and all other bit positions are undefined.

For 11/12, the format of the B register is:



where

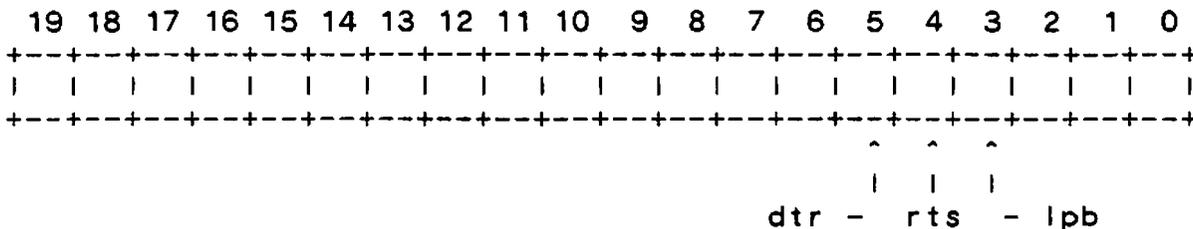
- cts = 1 if CLEAR TO SEND signal is on,
- dcd = 1 if DATA CARRIER DETECT signal is on,
- dsr = 1 if DATA SET READY signal is on,

ri = 1 if RING INDICATOR signal is on,  
 ti = 1 if TEST INDICATOR signal is on,  
 sq = 1 if SIGNAL QUALITY signal is on,  
 rti = 1 if RATE INDICATOR signal is on.  
 and all other bit positions are undefined.

5 - SETSTAT. Write the Control and Status Register (CSR) for the full-duplex device using the contents of the B register. The contents of the B register are not affected. The status bits in the B register have different meanings for the 7/8 device and the 11/12 device. This XDV may be performed any time when the device is APR'd.

\*  
\*

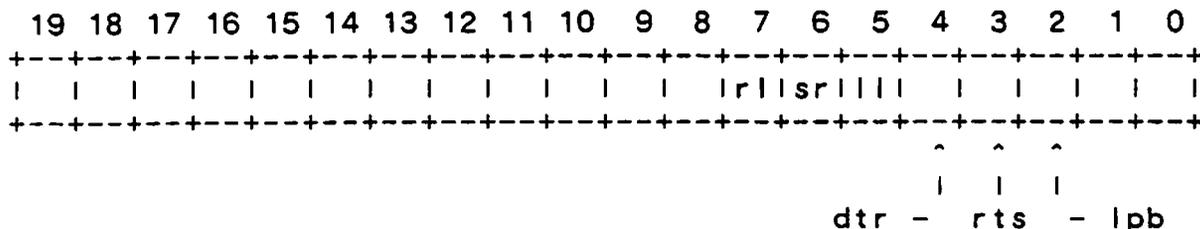
For 7/8, the format of the B register is:



where

lpb = 1 if the interface is to be looped,  
 rts = 1 if REQUEST TO SEND signal to be turned on,  
 dtr = 1 if DATA TERMINAL READY to be turned on,  
 and all other bit positions are undefined.

For 11/12, the format of the B register is:



where

- lpb = 1 if the interface is to be looped.
- rts = 1 if REQUEST TO SEND signal is to be turned on.
- dtr = 1 if DATA TERMINAL READY signal is to be turned on.
- ll = 1 if LOCAL LOOPBACK signal is to be turned on.
- sr = 1 if we are driving the Signalling Rate Selector signal.
- rl = 1 if we are driving the Remote Loopback signal.

and all other bit positions are undefined.

### INPUT PROCESSING

Before input processing can start, the macrocode must enqueue one or more "empty" IOCBs on the GET queue. The macrocode initializes each IOCB's data size, data address, and

control/status words. The data address and size, and the odd byte flag, specify where that IOCB's data input should go, and how much input the IOCB can absorb. The goading condition flags are set based on when the macrocode input driver wishes to be executed. For example, if input processing is on a frame basis, each IOCB on the GET queue might have a=0, e=1, and f=1.

To start input processing, the macrocode executes a PDV using the PCB of the input driver process. The firmware then dequeues the first IOCB from the GET queue and holds it in anticipation of input. A copy of the IOCB pointer is stored in the "current IOCB" word of the PCB.

When a frame begins to arrive, the firmware stores consecutive characters into the data area specified by the IOCB and decrements the size of space remaining. Processing of the IOCB terminates when the end of frame is reached, the IOCB data area is exhausted, or the input is aborted:

- The 2652 MPCC notifies the microcode when an end of frame is reached, and makes the last data character available. The two characters of CRC are not passed to the microcode, but an indication is given whether the CRC was correct or not. The actual data size is copied into the IOCB data size word. The microcode sets the eof flag in the control/status word of the IOCB. It also sets the completion code to 0 if the CRC is correct, and 3 if not.

## BOP/HDLC SYNCHRONOUS

- If the remaining data size for the IOCB goes to zero, the IOCB has been exhausted and input must be switched to a new IOCB. The data size word of the IOCB is left at its original value. The eof flag and completion code are set to 0.
- If the input is aborted by the macrocode, the current data size is copied into the IOCB, the eof flag is set to 0, and the completion code is set to 1.

In all cases, the current IOCB word of the PCB is cleared, a pointer to the IOCB's control/status word is saved in a temporary microregister, the IOCB is enqueued on the PUT queue, and the macrocode input driver process is goaded if any of the enabling conditions (a, e, and f flags) are true. Then the firmware driver attempts to dequeue new IOCB from the GET queue. If successful, input continues as described above.

If there are no IOCBs left on the GET queue, input processing lapses until a new PDV is executed -- all arriving input is discarded. If the previous IOCB did not contain an end of frame (eof flag was 0), the saved pointer to the control/status word is used to set the completion code to 2 (overrun) before quitting the firmware driver.

Whenever input processing has lapsed and a frame is in progress on the line and the macrocode executes a PDV of the

input driver, a new IOCB is then dequeued from the GET queue but is not used for the current frame. Instead, the driver continues to discard the input data until an end of frame is reached, and then finally continues in normal input mode as described above.

#### OUTPUT PROCESSING

Before output processing can start, the macrocode must enqueue one or more IOCBs on the GET queue. The data size, data address, and odd byte flag specify the location and extent of data to be output on behalf of that IOCB. In addition, the eof flag is set in any IOCB that contains the last data bit of a frame. The goading condition flags are set based on when the macrocode output driver wishes to be executed.

To start output processing, the macrocode executes a PDV using the PCB of the output driver process. The firmware then dequeues the first IOCB from the GET queue and immediately begins to transmit a new frame on the line. A copy of the IOCB pointer is stored in the current IOCB word of the PCB.

The firmware transmits consecutive data characters from the specified data area onto the line and decrements the count of

remaining characters. Processing of the IOCB terminates when the IOCB data is exhausted or the output is aborted:

- If the data is exhausted and the eof flag is on, the 2652 MPCC is instructed to send the CRC and terminate the frame with a flag character. The completion code of the IOCB is set to 0 (eof or not).
- If the output is aborted by the macrocode, the 2652 MPCC is instructed to send an abort sequence, and the completion code of the IOCB is set to 1.

In all cases, the current IOCB word of the PCB is cleared, a pointer to the IOCB's control/status word is saved in a temporary microregister, the IOCB is enqueued on the PUT queue, and the macrocode output driver process is goaded if any of the enabling conditions (a, e, and f flags) are true. The firmware driver then attempts to dequeue a new IOCB from the GET queue.

If a new IOCB is available, and the previous IOCB did not contain an end of frame (eof flag was 0), then output continues as described above without any discontinuity between the last character transmitted from the previous IOCB and the first character transmitted from the new IOCB. If the previous IOCB did contain an end of frame, output continues as described above, starting a new frame.

If a new IOCB is not available, output lapses until a new PDV is executed. If the previous IOCB did not contain an end of frame, the saved pointer to the control/status word is used to set the completion code to 2 (underrun) before quitting the firmware driver.

#### 4.5.4 Asynchronous Control Port

##### DEVICE TYPE

The generic device covers input device type 5 and output device type 6 (see Table p. 4-6). It serves the asynchronous serial ports on the MBB processor board, and permits the exchange of characters that have one start bit, 8 data bits (including parity which is ignored), and one stop bit. The hardware interface uses a 2651 Programmable Communications Interface (PCI) chip operating in asynchronous mode.

##### INPUT IOCB

The IOCB status and control word uses no additional device-dependent bits. The nonzero completion codes are:

- 1 - the IOCB was aborted by the macrocode.
- 2 - a data overrun occurred because a new character arrived before the previous one was read out of the receiver holding register.
- 3 - a break sequence was detected.

## OUTPUT IOCB

The IOCB status and control word uses no additional device-dependent bits. The nonzero completion code is:

- 1 - the IOCB was aborted by the macrocode.

## APR

The APR entry points of the input and output drivers cause the initialization of the register block, including storing of the PCB in the register block.

The output side of the device must be APRed first.

## DPR

The DPR entry points of the input and output drivers cause a deletion of the PCB pointer. Any IOCB in progress is aborted and placed on the PUT queue, and further I/O processing is stopped.

## PDV

The PDV entry points of the input and output drivers ignore the PDV if an IOCB is already being processed. Otherwise, the

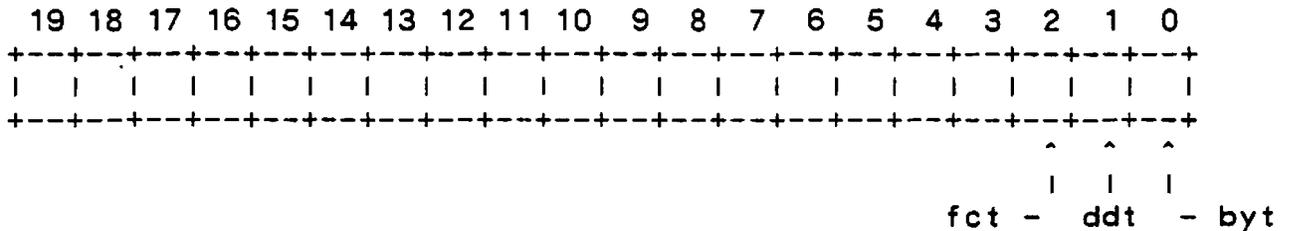
input driver attempts to get an IOCB from the GET queue, and, if successful, holds onto it in anticipation of input. Similarly, the output driver attempts to get an IOCB from the GET queue, and if successful, begins to transmit characters. In either case, if the attempt is unsuccessful, the PDV is ignored.

#### XDV

The XDV entry points of the input and output drivers are identical. Some functions affect both input and output sides, and some affect only the side whose PCB is in the X register. Except where noted, the contents of the A and B registers are not affected. The functions are:

- 0 - NOP. This is a no-operation.
- 1 - DISABLE. Reset 2651 hardware, relinquish control of the port and abort any IOCBs pending on either input or output sides. Do not restart I/O operation on either side. Affects both sides.
- 2 - ENABLE. Reset 2651 hardware, regain control of the port and abort any IOCBs pending on either input or output sides. Attempt to restart I/O operation on both sides. The contents of the B register contain flags which control the byte sequence, DDT mode, and flow control parameters. Affects both sides.

The format of the B register is:



where

- byt = 1    if bytes are to be transmitted in normal sequence (bits 15-8 first, then bits 7-0), and = 0 if bytes are to be transmitted in reverse sequence (bits 7-0 first, then bits 15-8),
- ddt = 1    if in DDT mode, and = 0 if in transparent mode (in DDT mode, ctl-N transfers control to user-accessible firmware operating environment, and E transfers control back to the application),
- fct = 1    if XON/XOFF flow control is enabled (ctl-S to stop transmission to device, ctl-Q to resume transmission to device), and = 0 if transparent to flow control characters,

and all other bit positions are undefined.

- 3 - BREAK. Issue a "break" sequence at the end of the current transmitted character, and abort any IOCB that may be pending on the output side. The break condition persists until a MARK is issued. Output side PDVs are ignored during the break condition. This XDV is a no-operation for the input side PCB.
- 4 - MARK. Cancel the "break" condition, and resume output processing. This XDV is a no-operation for the input side PCB.

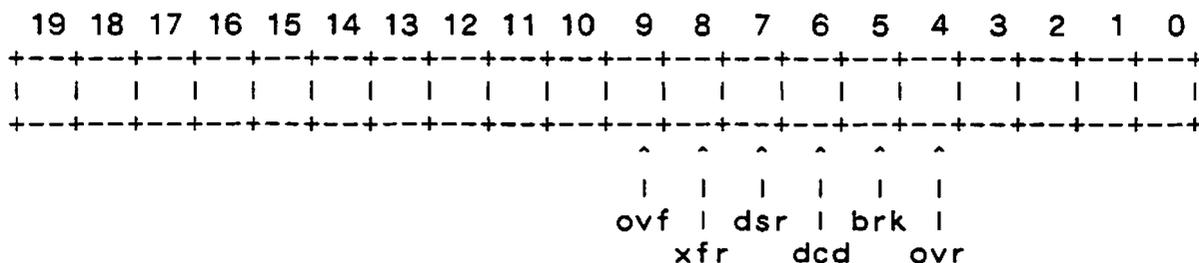
## ASYNCHRONOUS CONTROL PORT

- 5 - BAUD. Set the baud rate for the 2651 (both sides). The contents of the B register are an index between 0 and 7 that selects one of the following line speeds:

B Register	Baud Rate
0	110
1	300
2	1200
3	1800
4	2400
5	4800
6	9600
7	19200

- 6 - XFER. Do a single-character transfer to or from the B Register. If the X register specifies the input PCB, the next input character is loaded into the B register if the transfer is successful. The contents of the A register are set to 0 if the transfer is successful, unchanged if no character is available or an IOCB is in progress, and -1 if there is an input overrun condition, the port is DISABLED, or there is some other input error condition. If the X register specifies the output PCB, the character in the B register is output if a successful transfer response is returned. If an IOCB is in progress, the XFER character is output after the current transmitted character from the IOCB, and then output is resumed from the IOCB. The contents of the A register are set to 0 if the transfer is successful, unchanged if the previous XFER character is still being output, and -1 if the port is DISABLED or if there is some other output error condition.
- 7 - ABORT. If there is a current IOCB, set its completion code to 1 and enqueue it on the PUT queue. Then continue normal operation. Affects current side only.
- 8 - STATUS. Read the device status for the full duplex device into the B register. STATUS affects neither side, but may be issued for either.

The format for the B register is:



where

- ovr = 1 if 2651 input overrun bit on,
- brk = 1 if 2651 break detect bit on,
- dcd = 1 if DATA CARRIER DETECT signal on,
- dsr = 1 if DATA SET READY signal on,
- xfr = 1 if on input PCB, a character has been received and no IOCB is available, or on output PCB, a character transmission is in progress,
- ovf = 1 (on input PCB only) if a character arrived while xfr was still = 1.

and all other bit positions are undefined.

- 9 - CLEARERR. Clear the 2651 overrun and break flags, and the input side ovf and xfr flags in the firmware status register.

## INPUT PROCESSING

Input can be processed in two modes, standard and character-at-a-time (CAAT).

In CAAT mode, characters are transferred one at a time from the device to the macrocode by means of the XDV 6 (XFER) instruction. If a character is not yet available, the instruction will return with the A register unchanged, indicating the macrocode must try again later. If a character is available, it is returned in the B register and the A register is set to 0. If the port is disabled, or if there is an overrun or break condition, the instruction returns with the A register set to -1, and the macrocode must issue an XDV 9 (CLEARERR) before any more characters can be transferred. An unchanged A register value is also returned if input is already in progress in standard mode.

In standard mode, input processing can be started after the macrocode enqueues one or more "empty" IOCBs on the GET queue. The macrocode initializes each IOCB's data size, data address, and control/status words. The data address and size specify where that IOCB's data input should go, and how much input the IOCB can absorb. The goading condition flags are set based on

when the macrocode input driver wishes to be executed.

To start input processing, the macrocode executes a PDV using the PCB of the input driver process. The firmware then dequeues the first IOCB from the get queue and holds it in anticipation of input. A copy of the IOCB pointer is stored in the "current IOCB" word of the PCB.

As input arrives, the firmware stores consecutive data characters into the data area specified by the IOCB and decrements the size of space remaining. Processing of the IOCB terminates when the IOCB data area is exhausted, the input is aborted, or a break is detected:

- If the data area is exhausted, the data size word is left at its original value and the completion code is set to 0.
- If the input is aborted, the current data size is copied into the IOCB and the completion code is set to 1.
- If a break is detected, the current data size is copied into the IOCB and the completion code is set to 3.

In all cases, if a break or overrun condition preceded the use of the current IOCB, and no XDV 9 (CLEARERR) was issued to clear these conditions before the IOCB terminated, the completion code is set to 3 or 2 accordingly. This completion code will be

set in all IOCBs until the condition is cleared. The current IOCB word of the PCB is then cleared, the IOCB is enqueued on the PUT queue, and the macrocode input driver process is goaded if any of the enabling conditions (a and e flags) are true. Then the firmware driver attempts to dequeue a new IOCB from the GET queue. If successful, input continues as described above for standard mode.

If there are no more IOCBs left on the GET queue, then standard mode input processing lapses until a new PDV is executed.

If an input arrives and there is no IOCB waiting, the first character is buffered by the firmware. If the character is read in CAAT mode, or if standard input is started by PDV, this character is not lost and is passed to the macrocode. If a second character arrives before the first is disposed of in either input mode, the overrun condition is set. The break condition is set anytime a break occurs, whether in CAAT or standard mode.

## OUTPUT PROCESSING

Output can be generated simultaneously in CAAT mode and standard mode.

In CAAT mode, characters are transferred one at a time from the macrocode to the device by means of the XDV 6 (XFER) instruction. If a previous CAAT character has not yet been sent, the instruction will return with the A register unchanged, indicating that output is being attempted. If standard output is not in progress, and no previous CAAT character is pending, the character in the B register is sent and the A register is set to 0. Similarly, if standard output is in progress, and no previous CAAT character is pending, the character in the B register will be sent as soon as the current standard mode character is sent, and the A register is set to 0. This has the effect of inserting the CAAT character into the standard mode character stream. Finally, if the device port has been disabled, the instruction returns with the A register set to -1.

In standard mode, output processing can be started after the macrocode enqueues one or more IOCBs on the GET queue. The data size and data address specify the location and extent of the data

characters to be output on behalf of that IOCB. The goading condition flags are set based on when the macrocode output driver wishes to be executed.

To start output processing, the macrocode executes a PDV using the PCB of the output driver process. The firmware then dequeues the first IOCB from the GET queue and immediately begins to transmit the characters. A copy of the IOCB pointer is stored in the current IOCB word of the PCB.

The firmware transmits consecutive characters from the specified data area to the line and decrements the count of remaining characters. Processing of the IOCB terminates when the IOCB data is exhausted or the output is aborted:

- If the data is exhausted, the completion code is set to 0.
- If the output is aborted, the completion code is set to 1.

In all cases, the current IOCB word of the PCB is cleared, the IOCB is enqueued on the PUT queue, and the macrocode output driver process is goaded if any of the enabling conditions (a and e flags) are true. Then, the firmware driver attempts to dequeue a new IOCB from the GET queue. If successful, output continues

as described above for standard mode. Otherwise, standard output lapses until a new PDV is executed.

## 5 Firmware Facilities

This section describes the user-accessible firmware operating environment for a C/30E MBB running the NMFS microcode, i.e., the console commands for local DDT.

The NMFS microcode includes MBB standard software known as "USYS". USYS is responsible for maintaining general machine operation: operating the console terminal and loader, refreshing the MBB's dynamic macromemory, etc. Much of what is documented here pertains to USYS (rather than NMFS), although this fact is not necessarily important to the user's operation of the machine.

## CONSOLE COMMANDS

## 5.1 Console Commands

Unless otherwise usurped by the application, the C/30E's console terminal is serviced entirely by the firmware. Specifically, characters entered are processed by the microcode enabling the user to control the execution of the physical machine and the values of its various memory spaces. The C/30E console firmware indicates its readiness to accept commands by printing a "!" in the left-hand column upon receipt of a carriage-return command.

Console commands are always entered as upper-case, ASCII characters. Numbers, letters, carriage-return (CR) and line-feed (LF) make up the legal character set. All numbers are octal. All other characters as well as illegal commands print a "?" on the console. RUBOUT (DELETE, ASCII octal code 377) is the recommended way to abort an erroneous command as it is being entered. The firmware will print an "X" followed by two spaces indicating the cancelling of the command.

There are two classes of console command: action commands and examine/modify commands. Action commands are of the form "nC" where "n" is an optional numeric argument and "C" is an

## CONSOLE COMMANDS

action command letter. These commands perform specific actions (as the name implies). The second class of command, known as "examine/modify" allow the user to "open" (select) a given address, read and optionally alter its value. The basic scenario for all examine/modify commands is the same:

```
aC vvvvvvv nnnnnnnCR (or)
           nnnnnnnLF (or)
           nnnnnnn^ (or)
           (another command)
```

Where "a" is the desired address, "C" is the examine/modify command letter, "vvvvvvv" (typed back by the machine) is the current value at that address and "nnnnnnn" is the optional, user-supplied new value, terminated by carriage return, line feed, or caret ("^"). Terminating the new value with CR will cause the value to be deposited and the location (address "a") to be closed. Terminated with LF is like CR except that after closing address "a", address "a+1" is opened as if the user had done so explicitly. Caret ("^"), like CR and LF, closes the current location but opens the previous one ("a-1") when done. If the new value is omitted, CR, LF and caret act as before except no value is deposited into address "a". If "a" is not specified, the last value of "a" is assumed. If another command

## CONSOLE COMMANDS

is entered after "vvvvvv" is printed, this new command is processed immediately, closing address "a" without modifying its value in any way.

The following list shows available commands:

Cmd	Type	Example (note)	Description
A	E/M	A (1)	Open the emulated A register (203R)
B	E/M	B (1)	Open the emulated B register (204R)
nD	E/M	1076D (3)	Open dispatch address n
E	A	E (2)	Give the console to the application
nG	A	2000G	Reset NMFS and jump to macrocode address n
H	A	H	Halt application
nI	E/M	2I (3)	Open I/O address n
nJ	A	26J	Jump to microcode address n
nM	E/M	105M	Open macromemory address n
N	E/M	N (1)	Open the NMFS process register (206R)
P	E/M	P (1)	Open the emulated PC register (200R)
nR	E/M	200R (4)	Open microregister address n
S	E/M	S (1)	Open the emulated SP register (211R)
nU	E/M	20667U (4)	Open microcode URAM address n
nW	A	1234W (5)	Trap if macromemory location n is altered
W	A	W (5)	Cancel previous memory watch
X	E/M	X (1)	Open the emulated X register (205R)
@	n.a	@ (6)	Has the value of last typed number
^	n.a	^	Open previous address
<lf>	n.a	<lf>	Open next address
<cr>	n.a	<cr>	Close any open address
n<cr>	n.a	7<cr>	Deposit n in any open address, then close it
n+	n.a	4+	Open current address plus n
n-	n.a	4-	Open current address minus n
ctrl-N	A	ctrl-N (2)	Give the console to the MBB
ctrl-E	A	ctrl-E (7)	Toggle the console automatic echo flag

## CONSOLE COMMANDS

## Notes:

- (1) A, B, P, N, S, and X open specific registers in the C/30E's register space which are used in the emulation of the virtual machine. Typing "A" is exactly equivalent to typing "203R" (open register 203), but is far more mnemonic. It is probably unwise to use LF and caret in conjunction with these commands because such use would require an intimate knowledge of emulation register layout.
- (2) These commands are used to pass the console terminal back and forth between the firmware and the macrocode application. It is possible for the application to permanently usurp the use of the console through its NMFS device driver, however.
- (3) The "D" and "I" commands have the same syntax as the other E/M commands but instead of opening a memory location, they open an MBB dispatch or I/O address. Consult the "MBB Microprogrammer's Handbook" for a description of the dispatch and I/O address space.
- (4) The microregister address space contains 20-bit values and the micromemory address space contains 32-bit values (MBB microinstructions).
- (5) The "W" command allows the user to "watch" for changes in the value of some macromemory location. When the "W" command is issued, the microcode notes the current value of the specified location. Until cancelled with a null-argument W command, the microcode will pause between each macrocode instruction being emulated and compare the location's current value with the one stored when the W was issued. If a mismatch occurs, the machine will crash with a code #3. It is possible to instruct the "W" command to watch for changes in only part of a given address by placing the desired bit mask value in microregister 223. The default value for 223R is all ones, meaning watch for any bit change. If one sets 223R to 2000003, for example, only changes in the sign or the low two bits would be considered a watch hit. Location 223 is not reset to ones unless the microcode is reloaded

## CONSOLE COMMANDS

-- this is left to the user. Macrocode emulation speed is degraded while a watch is active.

- (6) The character "@" has the value of any currently-opened address. If one opened location 1234 with "1234M" and the printed value was "321", typing "@M" would open location "321M". One could have said "@R" to open "321R", etc.
- (7) The console microcode can be in either "silent" or "echo" mode, and the control-E command switches back and forth between them. In echo mode the microcode accepts responsibility for echoing characters, while in silent mode it does not.

## ABNORMAL CONDITIONS

## 5.2 Abnormal Conditions

The microcode is generally on guard for abnormal hardware or software conditions and makes an attempt to report these conditions in various ways. These abnormal conditions are called "crashes" as they invariably abort application execution. Each crash has a corresponding character code which is printed on the console as it is detected. A list of these codes is presented below. When a software/data inconsistency is detected, the microcode saves a large portion of the machine context in pre-determined macromemory locations (see section 5.3) and returns machine control to the USYS module. When severe hardware failures are detected, this context save is not attempted, since it may precipitate further failures; control is given directly to the USYS module. When USYS (hopefully) obtains control of the machine, it will attempt to re-read the cassette to load a fresh copy of the firmware (see section 5.4). The following list gives the character codes and explanations for each crash type.

- b Little button pushed (detected by hardware). The C/30E "button microinterrupt" was seen by USYS. This interrupt can be generated by depressing the small vertical button just on the inside of the larger new power button of the MBP board. Control is passed to USYS but no tape reading occurs.

## ABNORMAL CONDITIONS

- d Uninitiated dispatch cell (detected by firmware). The firmware attempted to access a non-existent address in the dispatch memory of the MBB. Faulty firmware loaded; control is immediately passed to USYS to re-read the tape.
- h The macrocode executed a HLT instruction. The microcode state is saved as for "t" below.
- j Jump to microcode address zero (detected by hardware). The firmware attempted to jump to microcode address zero. Although this can be caused by a firmware bug, it is generally the result of the user's "OJ" command (see Section 5.4). Control is passed to USYS to re-read the tape.
- m Uncorrectable macromemory error (detected by Error Detection and Correction hardware). The firmware referenced an address in macromemory which had bad parity. Consult Reference 2 for further information about macromemory EDAC. Control is passed directly to USYS to re-read the tape.
- n New power (detected by hardware). The C/30E received a new-power interrupt. This can also be caused by pushing the "big" button on the back of the MBP board of the machine or by issuing a "26J" command to the console. Control is passed immediately to USYS. See Section 5.4 for a discussion of the new power condition.
- p Microcode parity error (detected by hardware). The microcode instruction space (URAM/UPROM) had a word with bad parity. Control is passed immediately to USYS to re-read the tape.
- t Program trap (detected by firmware). The microcode detected a data inconsistency in itself or in the macrocode application's NMFS data structures which prevents further operation. Additionally, the "W" command causes a program trap when the location in question is modified. Upon a program trap, the firmware saves a large portion of the machine context in the "crash area" of macromemory (see Section 5.3) and then

## ABNORMAL CONDITIONS

returns to USYS to read the tape.

- w Wild microcode jump (detected by firmware). The firmware attempted to jump to a non-existent microcode (URAM/UPROM) address. Control is passed directly to USYS to re-read the tape.
- \* Breakpoint (software). The microcode jumped to USYS location "PDT.BRKPNT" in order to effect a breakpoint. Microcode breakpoints are generally used during program development only and thus may never occur in an operational environment. USYS is given immediate control but no tape motion is attempted, as reading the cassette will overwrite some of the machines state, and the purpose of the breakpoint will be defeated.

## CRASH AREA

## 5.3 Crash Area

The saved machine context takes the form of seven sets of microregister blocks. The first six are fixed blocks of general interest: RB.STATE (registers 20-37) contains the physical registers of the micromachine when the program trap occurred, RB.MAIN (200-217) contains emulation data, RB.CLOCK (40-57) contains the system clock and configuration parameters, RB.IOOS (300-317) contains NMFS context information, RB.SOFTINT (220-227) contains "watch" command information, and RB.TID (230-237) contains timing queue information. Readers should consult the microcode source file for further information regarding the exact format of these register blocks, but their rough formats are documented below.

The seventh saved register block is determined dynamically from the value of BASE as seen in the RB.STATE block at the time of the trap. This is done in the hopes of catching significantly more of the machine's context, especially in the case where the trap occurred in a microcode device driver module.

Note that the "crash code / TEMP" field of the RB.STATE contains a number from the list in Appendix D. This number

identifies the reason for the program trap.

There are seven application-dependent symbols defined in the microcode source file which determine the macromemory locations used by the firmware to save the five microregister blocks. These symbols (in `crash.mic`, see Appendix E) are:

```
crash.system = 3400
crash.state  = 3420
crash.main   = 3440
crash.clock  = 3460
crash.ioos   = 3500
crash.xxx    = 3520
crash.soft  = 3540
```

The formats of the first four blocks are as follows:

## RB.STATE (20):

20	BASE	used to find fifth block **
21		
22		
23	MAR	
24	MBR	
25	crash code / TEMP	"t" crash number
26	INTS	next microinterrupt address
27	ALUST	ALU status register
30	SN	MBP serial, same as 3401
31	MISC	MBB "miscellaneous register"
32	MISC2	MBB "miscellaneous register"
33	MISC3	MBB "miscellaneous register"
34	MIR	
35		
36		
37	reload disable password	333 if tape reload inhibited

\*\* The format of the fifth block depends on the release of the microcode. See the listing of the current microcode release. This block is often associated with the I/O device to which the trap belongs.

## RB.MAIN (200):

```

+-----+
200|   macrocode program cntr   |
+-----+
201|   ucode level #1 return   |
+-----+
202|   ucode level #2 return   |
+-----+
203|           A-register      |
+-----+
204|           B-register      |
+-----+
205|           X-register      |
+-----+
206|   current PCB address     | 0 if NMFS turned off
+-----+
207|   MBB ALU status          | saved machine status
+-----+
210| i l                        | MI bit 19=INHibit, lsb=Meas. mode
+-----+
211|           SP-register     |
+-----+
212|                           |
+-----+
213|                           |
+-----+
214|                           |
+-----+
215|                           |
+-----+
216|                           |
+-----+
217|                           |
+-----+

```

RB.CLOCK (40):

```
+-----+
40|                                     |
+-----+
41|                                     |
+-----+
42|                                     |
+-----+
43|      low order 100us time          |
+-----+
44|      high order 100us time         |
+-----+
45|                                     |
+-----+
46|                                     |
+-----+
47|                                     |
+-----+
50|                                     |
+-----+
51| value of MISC3 at last EDAC        |
+-----+
52| number of macromemory errs         |
+-----+
53|      macromemory size               |
+-----+
54|                                     |
+-----+
55|      4-LED control word             |
+-----+
56|                                     |
+-----+
57|                                     |
+-----+
```

RB.100S (300):

```

+-----+
300|  address of PENDING queue  |
+-----+
301|  microcode level #1 return  |
+-----+
302|  microcode level #2 return  |
+-----+
303|  microcode level #3 return  |
+-----+
304|  address of READY queue    |
+-----+
305|  address of TIMING queue    |
+-----+
306|  PCB address pointer        | for all PCB operations
+-----+
307|  microdevice driver return  |
+-----+
310|  high order attention word  | bit 15=prio. 0, lsb=prio. 15.
+-----+
311|  low order attention word   | bit 15=prio. 16., lsb=prio. 31.
+-----+
312|                               |
+-----+
313|                               |
+-----+
314|                               |
+-----+
315|  NMFS timer 1.6ms ticks    |
+-----+
316|  microdevice saved BASE    |
+-----+
317|  address of IDLE queue     |
+-----+

```

## RB.SOFTINT (220)

```
+-----+
220|                                     |
+-----+
221|                                     |
+-----+
222|                                     |
+-----+
223|          watch mask                 |
+-----+
224|          watch address               |
+-----+
225| instruction left off PC             |
+-----+
226|          watch value                 |
+-----+
227|          pmscan flag                 |
+-----+
```

## RB.TID (230)

```
+-----+
230| next timing list location           |
+-----+
231|                                     |
+-----+
232|                                     |
+-----+
233|                                     |
+-----+
234| time of PCB being inserted         |
+-----+
235| address of PCB being inserted      |
+-----+
236|                                     |
+-----+
237|                                     |
+-----+
```

## CRASH AREA

## 5.4 Cassette Operation and New Power Condition

In addition to the causes mentioned above, tape reading may be commanded by the user at the C/30E console in a direct way.

Typing "0J" (jump to firmware address zero) will cause USYS to print the appropriate "j" crash code and then begin reading the tape. No application-dependent data is altered by this tape operation except, of course, for information specifically overwritten by the tape's contents. Typing "0J" is the preferred way for reading or re-reading from the tape when preserving the current machine state is important.

Normally tape input begins at block 10 (octal), the first block after the tape-resident bootstrap code. The tape is read until an end-of-file marker is seen. Generally, there is one tape "file" for each MBBCASS command given during its creation (see section 6.3). It is sometimes desirable to read a file other than the one located at block 10. This is done by placing the desired octal block address in microregister 177 before issuing the "0J". When so instructed, USYS will position the tape at that block and begin reading one file. Upon completion, 177R will be reset to zero so that the next tape input will begin

at the usual block 10.

When debugging new firmware, it is often useful to disable USYS's re-boot attempts upon abnormal conditions. To do this, set register 37R to "333". When so set, USYS will not attempt to read from the tape under any circumstances except new power or "26J" (see below) until 37R is no longer "333."

When new power is applied to the C/30E, the hardware generates a microinterrupt to address 26 (octal) in USYS's PROM portion. At that time, USYS clears all writable address spaces, resets internal hardware and begins to read a bootstrap fragment from the tape starting at block 0. After reading the bootstrap, the first file is read as usual. It is important to note the tremendous difference between typing "0J" (which only does tape read-in) and typing "26J" which does a massive machine reset thus erasing all previous context.

## 6. Macrocode Development Support Tools

A number of UNIX[\*]-based support tools exist that aid in the development of NMFS programs and system. Some of these, such as M4, PEN, and MAKE, are standard UNIX program development tools, while others, which will be described in this chapter and the next, are designed specifically for C/30E macrocode and microcode development. These include:

### ASM20,

a basic assembler that takes C/30E assembly instructions and produces both binary output and a listing, as well as several optional informational output files. ASM20 was based upon the TENEX-based DAP assembler, but it does not support macros and has some differences from DAP, mostly dealing with addressing specifications in instructions. ASM20 is described in chapter 7.

### PL30,

a high-level assembler based upon C30ASM. To make up for the fact that C30ASM does not support macros, PL30 was developed. It adds such high-level language features as run-time expression evaluation, structures, byte operations, constructs such as if-then-else, repeat loops, and procedures, and automatic off-page references. Where C30ASM generally has a one-to-one correspondence between source lines and assembled instructions, PL30 can generate many instructions from a single line of code. PL30 is also described in chapter 7. ++

---

[\*]UNIX is a trademark of Bell Laboratories

MA,

the microcode assembler. See Appendix E.9 of this document for more information.

MBBCASS,

a program used to write both macrocode and microcode onto a C/30E boot tape using a TU-58 tape drive directly connected to the UNIX system. MBBCASS is described in section 6.3.

CASWRITE and CASREAD,

programs used to write and read both macrocode and microcode to and from boot tapes on remote C/30Es on a network such as the ARPANET. These programs exist as commands on the UNIX system, running the NU Network Monitoring and Control System.

BINCOPY,

another NU command that is used to send macrocode binary files to a remote C/30E on a network.

C316BN and CMBNBIN,

programs used to convert between microcode and macrocode binary files. These are described in section 6.2.

Further information concerning BINCOPY, CASWRITE and CASREAD, as well as other NU commands of possible interest to a C/30E programmer, can be found in the NU User's Manual, BBN

---

++ At this time (July 1984) PL30 is not yet available for the \*  
20-bit architecture. \*

Report No. 4823, and in the NU Reference Manual (available on many systems in the form of UNIX-style MAN pages, via the commands NUMAN, NUHELP, etc.).

Further information concerning the DAP assembler can be found in the DAP Assembler Manual, BBN Report No. 3124.

## 6.1 File Types and Naming Conventions

The above programs all operate upon and produce a number of different types of files. To keep all of these different file types organized, a number of file naming conventions have been adopted. These conventions usually involve adding a standard suffix to a base file name to identify the type of the contents of the file. In the UNIX environment, all files are simply strings of bytes, and any structure in the files has to be imposed by the programs that read and write the files. The standard suffixes allow a programmer to immediately ascertain the structure of the contents of the file without requiring a program to read the file and print out its type. These suffixes include:

- .c30 This is the input file to ASM20 and it includes assembly language instructions. It is usually generated directly by a text editor. (If macro expansion is desired, process first with the UNIX M4 macroprocessor.)
- .pl30 This is the input file to PL30. (If macro expansion is desired, process first with the UNIX M4 macroprocessor.) Since a ASM20 program generally will not correctly assemble under PL30, or vice-versa, two different file types are used to signify which assembler the program was written for.
- .tmp This is the macro-expanded output from M4, ready for assembly with ASM20 or PL30.
- .bin This is the binary file output from either ASM20 or PL30.

The format of a .bin file is the same regardless of which assembler was used to produce it; it is also compatible with the binary file format used by the TENEX-based DAP assembler.

- .b A synonym for .bin (used primarily by the NU system).
- .lst This is the listing output for ASM20 and PL30.
- .err The error listing for an assembly.
- .sym The symbol file used and produced by ASM20 and PL30. Both programs use identical symbol file formats, and a symbol file produced by one can be used by the other. This file is NOT compatible with the symbol files used by the DAP assembler.
- .s A synonym for .sym (used primarily by the NU system).
- .mic This is the input for the MA microassembler. The microassembler is discussed in Appendix E.
- .mbn The MA microassembler places its output into a .mbn file which is formatted such that it can contain data for any of the C/30E's four memory spaces (macromemory, micromemory, micro registers, and dispatch memory). A .bin file, on the other hand, can hold only macromemory data. A .bin file can be easily converted to a .mbn file, and a .mbn file can be converted to a .bin file containing the macromemory portions of the .mbn file as described in the following section. C/30E boot cassettes are generally written from the contents of .mbn files.
- .cnf Since different C/30Es have different configurations, and there is no hardware method for NMFS to determine which I/O addresses correspond with which type of I/O interface, the .cnf (configuration) file has been developed to describe a particular C/30E's configuration. This file type is described in detail in section 6.3.
- .pkg NU convention for naming binary files for "packages".

## 6.2 Binary File Conversion

The assemblers produce .bin files, the microassembler produces .mbn files, and very often one needs to convert binary files between these formats. C316BN and CMBNBIN are used to perform the conversions.

C316BN is used to produce an .mbn file from a .bin file. It uses the following command line format:

```
c316bn inputfile [-o outputfile]
```

The "-o outputfile" is optional. If the -o option is not used, c316bn writes the .mbn data to the standard output. The following examples both convert abc.bin to abc.mbn.

```
c316bn abc.bin >abc.mbn  
c316bn abc.bin -o abc.mbn
```

CMBNBIN is used to produce a .bin file from an .mbn file. It uses the following command line format:

```
cmbnbin inputfile [-o outputfile]
```

Again, the "-o outputfile" is optional. If the -o option is not used, cmbnbin writes the .bin data to the standard output. The

following examples both convert abc.mbn to abc.bin.

```
cmbnbin abc.mbn >abc.bin  
cmbnbin abc.mbn -o abc.bin
```

Generally, one can convert a .bin file to an .mbn file and then back to a .bin file with no loss of information. However, .mbn files that contain any data not in the macromemory space (such as that contained in the micromemory space) will lose that data if they are converted to .bin files.

### 6.3 Writing C/30E Cassettes

TU-58 tapes consist of 512 blocks of 512 data bytes each, for a total capacity of 262,144 bytes of data (not including formatting overhead, which is invisible outside of the TU-58 drive). A C/30E boot tape has a boot block at block 0 on the tape, and the actual micro and/or macrocode, which usually starts at block 10 on the tape. The code is contained in contiguous blocks on the tape, and is ended by a "transfer block" which instructs the C/30E bootstrap loader to begin C/30E execution at a specified macro or microaddress.

A typical small application, the NMFS microcode and the NMFS ARPANET macrocode loader/dumper, use blocks 10-70 on the tape. A \*  
larger application would, of course, use more blocks on the tape, \*  
but there is more than enough room on the tape to hold the \*  
largest possible NMFS application program (using all 256K words \*  
of macromemory). \*

### 6.3.1 Configuration Generation and Tape Writing

The MBBCASS command, described in section 6.3.2, is used to write .mbn files onto a C/30E boot tape. Normally, however, one need not directly call MBBCASS to write a NMFS application tape. An automatic Makefile procedure has been developed to read a configuration file that describes the devices and interfaces that will be used by a certain C/30E that will run the application. The Makefile dynamically puts together a list of .mbn files that contain the NMFS microcode, the application macrocode, and configuration information required for the C/30E that will run the application, and then calls MBBCASS to write a tape that contains all of the necessary micro and macrocode and tables.

This procedure is best described with an example, as shown below. The input format for the configuration, or .cnf, files shown here is somewhat specific to the ARPANET IMP application code, but it provides a good example of the way any application can format the configuration specification. Also, the example has been somewhat simplified, and information that is completely IMP-specific (i.e., unrelated to any other application) has been removed. In this example we first show a sample configuration file, and then describe each line in the file. The line numbers

are for illustration only, and are not in the actual configuration file.

1. IMP(63., 34324.)
2. HARDWARE(256., mii) \*
3. CM1TRUNK(0., 1, 0)
4. CM2TRUNK(1., 3, 5)
5. CM2TRUNK(2., 3, 7)
6. CM1TRUNK(3., 2, 2)
7. C18HOST(0., 1, 0)
8. C18HOST(1., 2, 1)
9. C18HOST(3., 2, 3)
10. CM1VDH(2., 1, 1)
11. CM2HDH(4., 3, 0)

For the purposes of this example, assume that the above configuration file describes a C/30E ARPANET IMP that has MII boards in I/O slots 1 and 2 and an MSYNC board in I/O slot 3. Further, the IMP has four trunks to neighboring IMPs, three 1822 hosts, a VDH host, and an HDH host. In this particular configuration, two MII boards are not really necessary, but both are present for future expansion.

Line 1 identifies the ARPANET IMP that this configuration refers to, and gives some general information about it. In this example, the IMP's number is 63 and the C/30E's serial number is 34324. This information is all put into the

proper place in macromemory.

Line 2 tells how much memory the C/30E has, and the type of board in the first I/O slot. The former causes the CRC tables needed by the CM1 device driver to be put into the proper place in macromemory, and the latter is used by the LITES instruction (different board types require different actions by the instruction).

Lines 3-6 define the trunks to other ARPANET IMPs. Two of the trunks, on lines 3 and 6, are BCP/CRC-24 lines on MII boards. The other two trunks are BOP/HDLC lines on an MSYNC board. In each case, the three arguments in parentheses list the trunk number (used by the IMP macrocode), the board number, and the interface number on the board. Thus, trunk 0 is a BCP trunk using modem interface 0 on the MII board in I/O slot 1, trunk 1 is a BOP trunk using modem interface 5 on the MSYNC board in I/O slot 3, and so on.

Lines 7-9 define the 1822 hosts. Host 0 uses ARPANET 1822 interface 0 on the MII board in slot 1, and so on. Line 10 defines a BCP VDH host, which uses modem interface 1 on MII board 1. Line 11 defines a BOP HDH host that uses interface 0 on the MSYNC board in slot 3.

Note that in the real ARPANET .cnf files, each of these input lines contains additional IMP-specific information concerning such information as line speeds, host access permission bits, and so on, which gets placed into pre-defined host and trunk blocks in the IMP's macrocode.

The above .cnf file is processed by M4 to produce a file that is assembled using the MA microassembler, and the whole operation is automated by the Makefile facility. Assuming that

the above .cnf file is called imp63.cnf, one can process the .cnf file by typing:

```
make imp63.files
```

and write the C/30E boot tape by typing:

```
make imp63.tape.
```

For further information about I/O configurations, one should read section E.8.3 in Appendix E. Section E.8.4 gives a simple example of some general M4 macros that any application could use. Section 6.4 contains a sample terminal session that uses the above Makefile facility for writing a tape, and section 6.5 has a sample Makefile and M4 macro file for the .cnf file format described above. Appendix B gives a sample configuration file.

## 6.3.2 MBBCASS

MBBCASS is the actual program that is used to write both macrocode and microcode onto a C/30E boot cassette. MBBCASS assumes that a TU-58 cassette deck is connected, via an RS-232 or current loop connector, to the /dev/tty0 interface on the C/70 running MBBCASS.

MBBCASS takes a list of .mbn files as its argument and places the binary contents of these files onto the tape in a contiguous fashion. When it reaches the end of the last file, it completes by adding the transfer block to the tape. This transfer block defaults to the starting address for micro-ddt, but can be specified by the user to be any macro or microaddress.

MBBCASS also takes a number of optional switches that precede the list of .mbn files to be written on the tape. It has the following syntax:

```
mbbcass [switches] filename1 [ ... [filenameN] ]
```

The optional switches are:

**-a address** : The tape block address to start writing the specified files. This address defaults to 10.

- b baudrate : The speed to set the terminal port that the TU-58 is connected to. This defaults to 9600 baud.
- g address : After the boot load is complete, the macrocode should auto-Go to the specified address, and begin running using normal instruction emulation.
- h : Print helpful information.
- i : Initialize the tape with the default boot block file at block 0.
- i bootfile : Initialize the tape with the specified boot block file at block 0. This allows the default to be overridden.
- j address : After the boot load is complete, the microcode should auto-Jump to the specified micro-address, and begin running. If neither the -g nor -j options are used, control will return to the user-accessible firmware environment following the load.
- p : When MBBCASS is finished writing the files on the tape, it should print the number of the last block written on the tape.
- t ttyname : The C/70 terminal interface to which the TU-58 drive is connected. This defaults to /dev/tty0.

#### 6.4 Sample Terminal Session

The following sample shows a terminal session that can produce a NMFS application and write it onto a boot tape. In this example, terminal interactions will be indented. It should be noted that Makefiles can be used in place of direct calls to the support programs to further simplify the program development process. One should read the UNIX manuals for information concerning the MAKE command.

First, one should write the application and enter it using an editor:

```
pen applic.pl30
```

Once the application program has been edited, it should be assembled:

```
pl30 applic.pl30 -ob applic.bin -l applic.lst -e applic.err
```

Once the application is fully written and assembles without errors, it can be written onto a tape and tested. First, the object file has to be converted to .mbn format:

```
c316bn applic.bin -o applic.mbn
```

Next, a configuration file for the test machine has to be entered: This example assumes that one has a "config" subdirectory to the main application directory, and that the config subdirectory contains the standard configuration Makefile described in section 6.5.

```
cd config
pen test.cnf
```

The configuration file is then processed to make the correct microcode configuration:

```
make test.files
```

Finally, the tape can be written. The Makefile should be edited, if necessary, to contain the name of the application program and the location where it should start running. One should then place a cassette tape in the tape writer attached to the C/70 and type:

```
make test.tape
```

When the make completes, it has created a tape that can be loaded and run in a C/30E.



CMBMBIN, and MBBCASS. The /microcode subdirectory itself has subdirectories, one for each release of the NMFS microcode. Likewise, the /macrocode subdirectory has a subdirectory for each release of the NMFS IMP. Finally, each imp directory has a /config subdirectory, which holds the different configurations for the ARPANET IMPs running that release, and from which the C/30E boot tapes are written. Incidentally, on BBNCCS the PL30 and ASM20 assemblers are kept in the /usr/otherbin directory, which is one of the standard UNIX system directories.

There is no rigid requirement for such a file structure. However, splitting up an application into such directories allows the use of the UNIX Make facility, which is a great time-saver and is highly recommended. (For information on the Make facility, see "Make -- A Program for Maintaining Computer Programs" in Volume 2a: SUPPLEMENTARY DOCUMENTS (A) of the UNIX Programmer's Manual. Another resource is the discussion of Make in BBNCC's NU Lecture Series materials.)

Each directory has its own Makefile, which allows complete reassemblies of either the macrocode or microcode by simply typing "make", without having to remember any arcane command arguments, lists of files, or the like. In each IMP release

subdirectory there is a Makefile which lists the files that make up the IMP assembly and also automates calling the assembler with the proper arguments. For an example of such a Makefile, see Appendix C.

Another type of Makefile that should be mentioned is the one described in section 6.3.1 for using .cnf files to generate C/30E tapes. This Makefile is in the /config subdirectory of each IMP. For an example of this Makefile, see Appendix B.

Finally, the M4 macros used in the M4 step in the above Makefile correspond to the .cnf file format described in section 6.3.1, and are found in the file macros\_.mic (or macros.mic) in the /config subdirectory. Most recently, this file resides in the /CfgMac subdirectory of the latest revision of the latest release of the microcode, e.g.

/microcode/m7u13/Release\_1/CfgMac.

For an example of the macros\_.mic file, see appendix B.

## 7 Assembler Manual

There are two assemblers for the C/30E, ASM20 and PL30 ++. The latter, essentially a superset of the former, permits high-level constructs to compensate for the lack of macroprocessing capability. The reason the two assemblers have not been combined is that the basic instruction syntax is different, requiring manual conversion of existing ASM20 applications (mainly the IMP program). This manual describes either assembler, except that section 7.3 and its subsections apply only to PL30, and the instruction syntax differences are discussed in section 7.2.6.

---

++ The PL30 assembler is presently (July 1984) not available for the 20-bit architecture.

## Introduction

## 7.1 Introduction

The assembler operates as a classic two-pass processor, assigning symbol values on the first pass and generating code and other output files on the second. It runs under UNIX on the C/70 and produces binary files for execution on the C/30E. It can generate object code for all the C/30E NMFS instruction set and provides assembler pseudo-ops for the generation of additional object code and other output files.

The format of the UNIX command to run the assembler is:

```
<asm name> <file name> [<flags>]
```

where <asm name> = ASM20 or PL30,

<file name> = source file name, and

<flags> = 0 or more of the following:

-c = suppress CTL-L for CRT

-ol = list object code (PL30 only -  
default for C3OASM)

-l <file name> = listing output to file name

-pl <number> = page length of output listing,  
<number> lines per page

-e <file name> = error output to file name

-o <file name> = object output to file name  
(C3OASM only)

-ob <file name> = object output to file name  
(PL30 only)

-f <file name> [...-f<filenameN>] = user message output to  
file name ... filenameN

The error and user message output facilities are described in sections 7.4.9 and 7.4.10. The default listing output file is

Introduction

STDOUT, and the default error output file is STDERR. There is no default object output file.

A sample assembler command line is:

```
C30AS imp.C30 -l imp.l -o imp.o -f spares -f traps -f locs -f crash
```

## 7.2 Basic Code Generation

The assembler provides basic facilities for assembly-time expression computation, object instruction and data generation, and location counter manipulation.

### 7.2.1 Source Line Format

The assembler accepts free-format source lines of the general form:

```
label:          operation          ;comment
```

The three fields must be separated from each other by at least one blank or tab. One or more of the fields may be absent from a line, but those that remain must be in the same order. The operation field may be an expression, an instruction, an equate statement, or a pseudo-operation, and determines the object code (if any) that will be generated.

### 7.2.2 Names and Variables

A name is a symbolic element that is manipulated by the assembler. A name may be up to 12 characters long, may contain the characters a-z, A-Z, 0-9, "\_" (underline), "." (period), "\$", and "%", and may not start with a number. Upper and lower case letters are equivalent. A name may not be identical to an instruction mnemonic, assembler pseudo-op, or assembler variable. Mnemonics are listed in Appendix A, assembler pseudo-ops in section 7.5, and assembler variables in section 7.6.

A name can be assigned a value either explicitly as an assembly variable or implicitly as a label.

Assembly variables are analogous to variables in a normal program. They provide the ability to define, change and use interesting values during a compilation. Probably the most important use of assembly variables is symbolic constants. A value can be defined once and referenced symbolically thereafter. If it ever changes, only one line of the program must be modified. Assembly variables are declared and modified by equate statements. The format is:

```
<name> = <expression>
```

An example of the definition and use of an assembly variable is:

```
BUFLEN = 100
```

```
LDÄ [BUFLEN]
```

## Labels

## 7.2.3 Labels

Any C/30E memory location may be labeled so that it can be referenced symbolically. A label is a name followed by a colon (':'), and must precede anything else on the line. Labels may be placed on any line, and multiple labels per line are permitted. The value of the label is the value of the logical location counter (see section 7.2.10) at the time the label was encountered. In addition, a capability is available to verify the location counter. An expression (see section 7.2.5) followed by a colon is evaluated and compared to the location counter. An error occurs if they do not match.

## Numbers and Radix

## 7.2.4 Numbers and Radix

The various forms of numbers with specified radix are:

```
nnnnn.           ;decimal number
nnnnnn.O         ;octal number
nnnnn.X          ;hexadecimal number
nnnnnnnnnnnnnnnn.B ;binary number
```

Note that all numbers must start with a decimal digit (0-9).

In addition, the assembler has a default radix used to interpret numbers entered without radix specification. The pseudo-ops used to change the default radix are:

```
.OCTAL           ;change radix to octal
.DECIMAL         ;change radix to decimal
.HEX             ;change radix to hexadecimal
```

The default default radix is octal. There is some danger associated with using multiple default radices in the same program. For instance, the number 100 is a valid octal, decimal or hex number, but has different values depending on the default radix. The assembler will check for illegal digits in numbers but the programmer must be aware of things that the assembler cannot check.

The current radix is kept in the assembler variable %RADIX.

## Expressions

## 7.2.5 Expressions

The assembler allows arbitrary expressions in many places. Expressions are composed of names, numbers and operators. The arithmetic, logical and conditional operators allowed in expressions are:

*	multiply
/	divide
\	modulo
>>	right shift
<<	left shift
+	addition
-	subtraction
-	unary minus (arithmetic negation)
&	bitwise and
!	bitwise or
?	bitwise xor
'	unary ones complement
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
=	equal to
≠	not equal
&&	conditional and
!!	conditional or
!	unary conditional not (logical negation)
.DEF.	defined expression (1 if completely defined, 0 otherwise)

## Expressions

The comparison and conditional operators may also be written in keyword form:

```
.EQ.  
.NEQ.  
.GT.  
.LT.  
.GE.  
.LE.  
.AND.  
.OR.  
.NOT.
```

The precedence of operators is:

```
(unary operators)  
.DEF. ! , - , +  
(binary operators)  
* , / , \ , >> , <<  
+ , -  
&  
! , ?  
> , < , >= , <= , = , =,  
&&  
!!
```

Parentheses may be used to modify the order of evaluation. Arithmetic is 20-bit signed. Comparison and compound conditional operators produce 1 when 'true' and 0 when 'false'.

### 7.2.6 Instruction Formats

There are three different source code instruction formats processed by the assembler; these formats correspond to C/30E parameterless, shift, and memory reference instructions.

The source format of a parameterless instruction is:

⟨opcode mnemonic⟩

The opcode may be any C/30E instruction that requires no parameters, for example:

AOA ; add 1 to A register

The source format of a shift instruction is:

⟨opcode mnemonic⟩[sp.]⟨shift count⟩

For example:

LGR 3 ; logical right shift 3 places

The opcode may be any C/30E shift instruction (see section 2.4). The shift count may be any expression that evaluates to less than 64 (decimal). The assembler performs the two's-complement, modulo 64 operation before inserting the shift count

## Instruction Formats

in the object code, so that the shift count as specified by the programmer is the actual number of bits that will be shifted.++

The source format of a memory reference instruction is different for ASM20 and PL30.

For ASM20, the format is:

```
<opcode mnemonic> [sp.] <address> [sp.] <modifier>
```

The opcode may be any C/30E memory reference instruction, and the address and modifier fields are optional. The address field may be any expression, but it must evaluate to an address on page zero or the current page. The assembler will generate the 9-bit relative address and L bit values accordingly. The modifier field may be one of "I", "X", or "IX" (the LDX and STX instructions may only have "I" specified); the assembler will set the I and X bits of the object instruction accordingly, for example:

```
LDA   FOO           ;load location FOO into A
STA   FOO+3        ;store A into location FOO + 3
IRS   FOO I        ;increment location pointed to by FOO
IMA   TAB-TABLEN X ;interchange A and indexed location
ADD   TABI IX      ;add indirect indexed location to A
```

---

++ There is a flag "-novarshift" in the assembler to provide backward compatability with the prior method

## Instruction Formats

For PL30, the source format of a memory reference instruction is:

```
<opcode mnemonic> [sp.] <address> (<base> [sp.] <modifier>)
```

The opcode may be any C/30E memory reference instruction, one of the bit field opcodes described in section 7.3.3, or one of the immediate opcodes described in section 7.3.5. The address, base, and modifier fields are optional. The address field may not be an arbitrary expression, but must consist of a single name or constant (see section 7.2.9). The base field may be an arbitrary expression; the address and base are added together and must evaluate to a valid address (may be off-page, see section 7.3.4). The assembler will generate the 9-bit relative address, L bit, and indirect pointer values accordingly. The modifier field may be one of "I", "X", or "IX" (the LDX and STX instructions may only have "I" specified); the assembler will set the I and X bits of the object instruction accordingly.

Examples: (identical function as those above for ASM20)

```
LDA      FOO
STA      FOO(3)
IRS      FOO(1)
IMA      TAB(-TABLEN X)
ADD      TABI(IX)
```

### 7.2.7 Addressing

As described in section 2.3, the C/30E supports only on-page or page 0 relative direct memory references. In order to extend the addressing range, it is necessary to use indirect addressing (the "I" bit), which requires the allocation of an on-page (or page zero) memory location and the generation of an address pointer value for that location. The constants facility described in section 7.2.8 relieves the programmer of having to manually set up the indirect pointer.

The use of indirect addressing is also required to move from \* the end of one page to the beginning of a subsequent page. It is \* no longer possible, in the 20-bit C/30E architecture, to "fall \* off the end of a page".

## Constants

## 7.2.8 Constants

A constant directs the assembler to allocate a memory location on the current page, initialize it to a given value, and use the address of the constant to access the value needed. Constants can be used to simulate both immediate operands and off-page referencing. A constant is indicated by enclosing an expression inside square brackets ("[" and "]""). The following example shows how a constant simulates an immediate operand.

```
LDA [5]           :load 5
```

This is equivalent to

```
LDA FIVE
FIVE:  5
```

The next example shows how an off-page reference would be done.

```
JMP [FOO] I      :off page jump
```

This is equivalent to

```
JMP ADR I
ADR:  FOO
```

## Constants

Constants must be on the page that uses them, and the programmer must tell the assembler where to put the constants into memory. The .CONSTANTS pseudo-op causes the assembler to dump all accumulated constants at the current location.

Because the assembler allocates the storage for constants on pass 1, before it is aware of the value of the constants, it is possible on pass 2 that it will discover that two constants have the same value. When this happens, it combines the constant references and reduces the number of locations needed for constant storage. However, on pass 1 the .CONSTANTS pseudo-op causes the location counter to be incremented the maximal amount. In order to provide the programmer with the means to recover unused constant storage, the assembler sets the variable UNCON to be the last location actually used by the latest .CONSTANTS pseudo-op.

Since it is often the case that certain constants are used on several different pages, the assembler provides a facility for remembering constants that are created on page 0, using the .PZCON pseudo-op: (on page zero)

```
.PZCON -1
```

## Constants

The `.PZCON` pseudo-op allocates the constant on page zero (just like a `.WORD` pseudo-op, see section 7.2.11) and adds it to the list of page zero constants. Whenever this constant is referenced, the assembler generates the page 0 address and does not create a new on-page constant. This feature can be disabled and re-enabled using `.PZOFF` and `.PZON`; when disabled, all constants are created on-page.

## Strings

## 7.2.9 Strings

Ascii character strings may be placed in memory (two characters per location) with the .ASCII pseudo-op. Examples:

String 1: .ASCII This is a text string.

String 2: .ASCII / This is a text string. /

This pseudo-op illustrates the use of "delimited" strings. When interpreting the operand, if the first non-blank character in the string is not alphanumeric, it is assumed to be the string delimiter, and the actual character string consists of all the characters between delimiters. For non-delimited strings, the character string starts with the first non-blank character of the string and ends with the last non-blank character of the string. Any string that begins with an alphanumeric character and has no leading or trailing blanks may be entered without delimiters. Delimited strings may use the convention of doubling the delimiter to insert one into the string.

Special characters may be inserted in a string with the following escapes:

```
\n = linefeed  
\r = carriage return
```

```
\t = tab  
\f = formfeed  
\0 = null  
\ddd = character specified in octal
```

### 7.2.10 Location Counter Control

The location counter may be set by direct equate to the symbol '...'.  
.

```
. = <location>
```

If we wanted to set the location counter to 100 hex, we would say:

```
. = 100.X
```

It is not safe, however, to do arbitrary changes in location. In particular, changing between pages will not work because of the table of constants that the assembler keeps for the current page. At the minimum, a .CONSTANTS must be issued before moving the location counter to a different page. When it is necessary to switch back and forth between pages, the .SECTION (see section 7.2.12) mechanism should be used.

A facility is also available for assembling code in one location that is meant for execution in a different location. Since C/30E addressing is absolute, code is not generally relocatable, and the assembler must adjust the addresses in memory reference instructions. The .OFFSET pseudo-op specifies

the difference between the physical and logical locations for a code section. The rule for determining the offset is:

$$\text{offset} = \text{logical location} - \text{physical location}$$

As an example, if some code is physically located at 2000 and is supposed to look like it is at 1000, then the offset would be  $1000 - 2000 = -1000$ . The format of the .OFFSET pseudo-op is:

```
.OFFSET <expression>
```

The symbol '%' always gives the physical location, while '.' is always the logical location. The symbol %OFFSET gives the offset value.

Note that the location counter is adjusted by changing the logical location '.'. The physical location is adjusted by the compiler only, based on the logical location and the offset. Some confusion is possible since the logical and physical locations are the same when the offset is zero. If we say:

```
. = 2000  
.offset -1000
```

we are thinking: "at location 2000 put some code that is compiled for location 1000". But what this will do is make the logical

location 2000 and the physical location  $2000 - (-1000) = 3000$ .

Since '.' refers to the logical location, we should really say:

```
.offset -1000
```

```
. = 1000
```

### 7.2.11 Storage Reservation

Blocks of memory locations may be reserved by using the .BLOCK pseudo-op. Two forms are available. To reserve space without assigning values, the format is:

```
.BLOCK <count>
```

To reserve a block and assign initial values, the format is:

```
.BLOCK <count> , <value>
```

To reserve 100 (decimal) words of storage, we would say:

```
.BLOCK 100.
```

To reserve 100 words initialized to 0, say:

```
.BLOCK 100.,0
```

The .WORD pseudo-op reserves memory locations one word at a time. The format is:

```
.WORD [<value>[,<value>....]]
```

If no value is present, the location counter is advanced by 1 without generating a code word, otherwise, successive memory

locations are initialized to the values given.

In addition, a line that contains an expression, or a list of expressions, generates memory locations having the value of the expressions.

<value>[,<value>....]

## Program Sections

## 7.2.12 Program Sections

Sections allow a program to be organized according to logical function rather than physical location. Basically, sections allow jumping between various physical areas of memory as the assembly progresses rather than being constrained to a physically linear order for the code. Sections encapsulate the pc, offset and constants area. When changing sections, the pc, offset and constants area of the old section are saved, and the pc, offset and constants area of the new section are restored. Multiple constants areas per section are allowed; sections may cross page boundaries; and there may be multiple sections per page. The usual restrictions regarding constants still apply: 1) a section of code using a single constants area may not cross a page boundary, and 2) moving the pc by the ".=" construct may not leave the bounds of the current constants area. The format of the .SECTION pseudo-op is:

```
.SECTION <section number>
```

### 7.2.13 Conditional Assembly

In the same way that "if" statements provide conditional execution of code in higher level languages, "if" statements can provide conditional assembly. A condition is evaluated, and the truth or falsity of the condition determines which groups of statements are assembled. The formats for simple and compound conditions are:

```
.IF <condition>  
    <true body>  
.ENDIF
```

```
.IF <condition>  
    <true body>  
.ELSE  
    <false body>  
.ENDIF
```

The condition may be any expression. The condition is true if the value of the expression is non-zero, and false if the value of the expression is zero.

START

7.2.14 START

The START pseudo-op is used to specify the starting address entered into the binary file jump block. The format is:

START <address>

START terminates the assembly. Any statements after the START are ignored.

### 7.3 PL30 Constructs

In addition to the basic facilities offered by ASM20, PL30 has high level constructs for specifying control flow, run-time computation, and data structures. While there is some attempt at optimization of the generated code, use of these constructs will in general not yield as efficient (in time or space) code as could be generated manually. In order to list the basic assembly code generated by PL30 constructs it is necessary to use the `-ol` flag on the assembler command line.

## Continuation Lines

## 7.3.1 Continuation Lines

Since PL30 constructs permit long statements (source lines), a source line may be continued onto a new physical line by placing a pound sign ("#") at the end of the current physical line (but before any comment).

## Unary Operators

## 7.3.2 Unary Operators

PL30 has some additional unary operators: .MASK., .SHFT. and .LEN. may be applied only to single variable names and they return the bit-mask, shift count and bit-field length for that variable. The .BYTVAL. operator returns a value that is constructed out of bit-field specifications (see section 7.3.6). The format is:

```
.BYTVAL. (<field1>=<value1>[,<field2>=<value2>....])
```

As an example, if a word were divided into three bit-fields named F1, F2, and F3, a value for that word might be constructed by:

```
.BYTVAL.(F1=2, F2=5, F3=77)
```

## Bit Field Opcodes

## 7.3.3 Bit Field Opcodes

Bit field opcodes are assembled into sequences of basic machine instructions to perform various manipulations on groups of bits within a machine word.

The source format of bit field operations is identical to basic memory reference instructions (PL30 version, see section 7.2.6) with the addition that the base field may be the expression "%AC" (see below). The address field must be a defined bit field (see section 7.3.6).

The bit field opcodes are:

- LDB      Load bit field into AC, right-justified, left zero filled. For base = %AC, the value in the AC is right-justified and left zero filled.
- EXTB     Load bit field into AC, NO justification, left and right-zero filled. For base = %AC, the value in the AC is left and right zero filled.
- DPB      Store right-justified bit field from AC into "address". For base = %AC, the value in the AC is shifted so the bit field is in its home position.
- DPBZ     Store right-justified bit field from AC into "address", assume destination was zero. For base = %AC, the value in the AC is shifted so the bit field is in its home position.
- CLB      Clear bit field at "address" to zeros. For base = %AC, the bit field in the AC is cleared.
- CMB      Complement bit field at "address". For base = %AC, the

## Bit Field Opcodes

bit field of the AC is complemented.

- STB Set bit field at "address" to ones. For base = %AC, the bit field in the AC is set to ones.
- XRBM Exclusive-or right-justified bit field in AC with "address". For base = %AC, the value in the AC is shifted so the bit field is in its home position (like DPB).
- SZB Skip if bit field at "address" is all zeros. For base = %AC, skip if the bit field in the AC is all zeros.
- SNB Skip if bit field at "address" is not all zeros. For base = %AC, skip if the bit field in the AC is not all zeros.

#### 7.3.4 Off-Page Referencing

PL30 can recognize off-page references and automatically set up the constant and specify indirection. For example,

```
JMP [FOO] (1)
```

can be stated as

```
JMP FOO
```

and the assembler will automatically generate the constant and the indirect reference.

This capability can be turned on and off with the `.OFFPAGE` and `.ONPAGE` pseudo-ops. The default is to automatically generate off-page references.

### 7.3.5 Immediate Opcodes

Each memory reference opcode mnemonic has an analogous immediate opcode mnemonic formed by appending "I" to the original. LDA becomes LDAI. The operand of an immediate opcode is an expression which the assembler makes into a constant. No square brackets are necessary, and would generate the wrong constant if used. For example,

```
LDA [5]
```

becomes,

```
LDAI 5
```

## Field Definitions

## 7.3.6 Field Definitions

Bit fields are sub-divisions of memory locations. PL30 supports bit fields with a number of bit field operations (see section 7.3.3) and with the "assignment" statement (see section 7.3.8).

Bit fields are declared with the `.FIELD` pseudo-op. The formats are:

```
.FIELD <field name>,<field width>  
.FIELD <field width>
```

Bit fields are assigned in a fashion similar to that for normal memory locations. Fields are assigned sequentially from high order (bit 15) to low order (bit 0) with the following restrictions: 1) an individual field may not cross a word boundary, and 2) word allocations force alignment to the next word boundary. Note that there is a difference between a field "name" and a label. It is the field name that is associated with a bit offset and field width as well as a regular memory address, while a label has only a memory address. A field must be named in order to be used with the bit field operations or with assignment statements.

### 7.3.7 Structures

Structures are descriptions of data areas. They are used to define symbols whose value is an offset rather than an absolute address. These symbols may be used to reference data items that contain an address which is not known until run time, but one which has known offsets. The format of a structure is:

```
STRUCTURE <name>[, <base address>]
    ...
    declarations
    ...
ENDSTRUCTURE
```

This is shorthand for:

```
save = .
. = <base address>
...
declarations
...
name = .-<base address>
. = save
```

The declarations allowed within structures are .BLOCK, .FIELD, and .WORD without a value. Nothing that would cause a word to be initialized is legal. Note that the name of the structure is assigned a value equal to the length of the structure. The name may be used in subsequent .BLOCKs to allocate space for the structure.

### 7.3.8 Assignment Statements

Assignment statements have the form:

`<destination> := <expression>`

Valid destinations are the accumulator (%AC), the index register (%XR), or a memory address in the form acceptable for memory reference or bit field instructions. Terms in expressions are also these forms plus constants. Note that [1] is the constant 1 while plain 1 refers to address 1. The following operators are available:

+	addition
-	subtraction or unary minus
&	bitwise and
?	bitwise xor
	unary ones complement
shift	shift operations
gen	generic operations

Shift operators are binary operators using the shift instruction mnemonic as the operator, and taking an expression as the shift count. The count expression must be parenthesized if it has more than one term.

Generic operators like CAL (Clear A Left-half) are unary operations specified by the instruction mnemonic.

Note that the `ac` (`%AC`) is only valid as a source operand before any operation changes it, i.e., `%AC := %AC + [2]` is legal.

If an expression is sufficiently complex, the assembler will have to use one or more temporary variables for saving intermediate results. These temporary variables must be allocated by the programmer by using the `.TEMPS` pseudo-op. Like `.CONSTANTS`, the `.TEMPS` pseudo-op must occur on the same page where one or more complex run-time expressions have occurred. Such expressions may occur in assignment statements, `IF` statements (see section 7.3.9), and loops (see section 7.3.10). There may be multiple `.TEMPS` per page.

## 7.3.9 IF Statements

IF statements have the form

```
[SHORT] IF <boolean expression>
    ... text ...
ELSEIF <boolean expression>
    ... text ...
...
...
...
ELSE
    ... text ...
ENDIF
```

The `SHORT` modifier is used to indicate that the body of the IF is only one instruction long and that a skip should be generated instead of a skip-jump. This may not be used if the body is longer than one word, or an `ELSE` or `ELSEIF` clause is present.

`ELSE` and `ELSEIF` clauses are optional. IF statements may be nested.

The boolean expression may be a simple condition or a compound condition. A simple condition may be a relational condition of the form:

```
<arithmetic expression> <relop> <arithmetic expression>
```

where relop is one of:

=	equals
≠	not equals
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

These tests generate a CAS followed by the appropriate SKPs, JMPs or NOPs. Note that for range checks (like `foo > [1] AND foo < [10]`) it is more efficient to use the range test boolean, which can generate tightly nested CASs.

Examples of simple relational conditions are:

```
foo > [1]
foo + [1] <= fum
foo + [1] = fum - [5]
```

A simple condition may also be one of the skip conditions:

```
ZERO
NOTZERO
PLUS
MINUS
EVEN
ODD
CRESET
CSET
SKIPS
NOSKIP
```

An example of an IF statement with a skip condition as a simple condition is:

```

if zero                ; if ac is zero
    lda foo            ; load foo
endif

```

The skip condition tests generate a conditional skip instruction followed by a JMP.

A simple condition may also have the form:

```

<arithmetic expression> = <skip condition>

```

An example might be:

```

if foo - 1 = minus    ; if foo-1 negative
    foo := 0          ; clear it
endif

```

These tests also generate a skip-jmp, but load the ac with the arithmetic expression first.

A simple condition may also be a range check of the form:

```

.RANGE. ( <address> <op> <arithmetic expression> <op>
<address> )

```

## IF Statements

<address> may be a memory address or a constant. It may not be an arithmetic expression. <op> may be '<' or '<='.

An example of a range check is:

```
if .range. ([47] < foo < [75])
    jmp ok
endif
```

Range checks generate two CAS instructions followed by the appropriate sequence of SKPs, JMPs and NOPs. The four cases are different in that <,< generates CAS,CAS,JMP,JMP,JMP, <=,< and <=,<= generate CAS,NOP,CAS,JMP,JMP,JMP, and <,<= needs CAS,JMP,JMP,JMP,CAS,JMP,JMP,JMP.

Compound conditions are formed by combining simple conditions with the boolean operators .OR., .AND. and .NOT.. Code is generated in such a way that execution of the condition stops as soon as the true or false value is determined.

## Loops

## 7.3.10 Loops

The basic form of a loop is:

```
REPEAT
    ... body ...
TEST
    ... termination test ...
ENDREPEAT
```

The keywords REPEAT, ENDREPEAT, and TEST mark the beginning, end, and termination test respectively. Execution is controlled by the keywords CONTINUE, BREAK and NEXT. These keywords generate jumps to the REPEAT, ENDREPEAT and TEST. The TEST is optional, in which case, NEXT is equivalent to CONTINUE. BREAK has the capability of exiting multiple levels of loops. When given an operand, as in BREAK 2, the operand is the number of extra levels to exit. The UNTIL construct may also be used to exit a loop. The form UNTIL <condition> is equivalent to:

```
SHORT IF <condition>
    BREAK 0
ENDIF
```

### 7.3.11 Procedures

Procedures are subroutines. The PROCEDURE and END keywords bracket the body of the procedure. The PROCEDURE keyword allocates space for saving the return address and may optionally declare the name of the procedure. The format of a procedure is:

```
PROCEDURE [<routine name>]
  ...
  procedure body
  ...
END
```

Procedures are exited by the RETURN statement, whose format is:

```
RETURN [<skip count>]
```

The <skip count> causes the generation of IRS instructions that increment the return address before the return is done.

Procedures may cross a page boundary. Returns from off-page will generate the proper constants and indirect references.

#### 7.4 Programming Support

The assembler provides numerous programming support facilities. These include source file generation and control, listing file control, symbol cross-reference and table manipulation, and supplementary user output message file generation.

#### 7.4.1 Listing Format

The listing format is designed for narrow (i.e., 80-column) paper. A typical line of the listing consists of seven octal digits of location counter, seven octal digits of object code, two single-digit columns specifying the process levels (see section 7.4.3), and the source line.

## Comments

## 7.4.2 Comments

A comment begins with a semicolon (";") and ends with a carriage-return. This means that anything on a line after a ";" is ignored. A sample line with a comment is:

```
LDA FOO                ;this is the comment
```

A block comment facility is also available and has the form:

```
.COMMENT  
    <comment text>  
.ENDCOMMENT
```

The keyword form is used rather than a delimited string form to avoid the problems caused when the delimiter character is inadvertently contained in the comment text. Block comments may be nested.

In the output listing, the position of comments relative to the source code is normally the same as in the original input text. This means that comments that begin in column one of the input text are indented in the output listing to match the rest of the source code. This indentation can be turned off in order to provide more room for comments. This is done individually for each block comment by saying:

## Comments

```
.COMMENT WIDE  
    'comment text'  
.ENDCOMMENT
```

"Wide" comments begin at column one of the output listing. In addition, source lines that begin with a comment (i.e., a ";" in the first column) can also be made "wide" on the listing by using the .WIDE pseudo-op, which sets the listing mode for comment-only source lines. This mode can be cancelled using the .NARROW pseudo-op.

### 7.4.3 Process Level Control

The .LEV, .LCK, .INH, .ENB, and .RET pseudo-ops are used to manipulate the self-documenting system that protects shared data structures from corruption by process level interrupts. Only the .INH and .ENB pseudo-ops generate instructions, the others are solely for the purpose of program documentation.

The highest priority process that accesses a variable may do so with impunity since it can only be interrupted by yet higher priority processes which do not access the variable. All lower priority processes which manipulate it, though, must take care that they are not interrupted while modifying the variable. The INH instruction and the .INH pseudo-op, which issues an INH instruction and also affects the program self-documentation described here, effect this protection by simply preventing all process level interrupts. The remaining process level control pseudo-ops, with the exception of .ENB, do not insert any value in the object field. They only document which levels access the data structure(s) undergoing modification.

For this purpose of program self-documentation, two priority levels, %XLEV and %YLEV, are printed in the program listing following the two seven digit fields of location counter and object code with the exception that if they are equal the printing of the second column is suppressed.

The .LEV statement is used to document every line of code with a %XLEV priority level equal to the priority of the process under which it runs. It also sets the value of the non-printed %YLEV equal to the value of %XLEV. A .LCK or .INH is used to change the second priority, %YLEV, to the priority of the highest priority level which touches the variable(s) undergoing modification. A .LCK, which does not issue an inhibit instruction, is used to document a lower priority process that runs under an inhibit, but the actual inhibit is done elsewhere, i.e., in the calling or called routine, and local documentation of that is all that is required. ++ The .ENB and .RET pseudo-ops are used, in analogous fashion, to cancel .INH and .LCK respectively, and reset %YLEV to %XLEV.

---

++ For examples of the actions and interactions of these pseudo-ops, see the the Back Host 5 code in the background process of the IMP and Appendix C, "Sample Program", of this document.

Incorrect usage of one of these pseudo-ops, e.g. a request for a redundant lock or inhibit, will result in an error condition.

The arguments and actions of these pseudo-ops are as follows:

.LEV - accepts as argument a list of symbols each of which has been assigned a priority level, and sets both %XLEV and %YLEV to the highest priority entity on the list.

.INH - accepts as argument a list of symbols each of which has been assigned a priority level, sets %YLEV to the highest priority entity on the list, and generates an INH instruction.

.ENB - accepts a priority level entity as argument, sets %YLEV to %XLEV, and generates an ENB instruction.

.LCK - accepts as argument a list of symbols each of which has been assigned a priority level, and sets %YLEV to the highest priority entity on the list.

.RET - accepts a process name as argument, and sets %YLEV to %XLEV.

#### 7.4.4 Listing Control

A number of pseudo-ops are available to control aspects of the compiled listing.

.TITLE and .STITL specify the titles and subtitles to be printed in the page headers of the listing. The title and subtitle texts are specified with delimited strings.

.LIST and .XLIST control listing output. .LIST turns the listing on while .XLIST turns it off. These pseudo-ops work in a nested fashion which means that the listing only occurs when enough .LISTs have been seen to neutralize any preceding .XLISTs.

.OLIST and .SLIST control listing of the source and object parts of the listing. .OLIST causes listing of the object columns only, and only for instructions that emit object code. .SLIST re-enables the full source listing. These pseudo-ops may be nested. .OLIST and .SLIST are active only while in .LIST mode, they will not override .XLIST.

.LISTOBJ and .XLISTOBJ turn the "disassembler" feature

## Listing Control

on and off. When on, the disassembler will list, following each source line, the disassembled version of all the object code generated by the source line in the form <opcode mnemonic> <absolute address>. This is particularly useful for seeing the actual instruction generated by PL30 runtime expressions or other constructs, but is also of general use in determining the ultimate resolution of symbolic references.

.PAGE causes a page break in the listing. A control-L will also do so.

#### 7.4.5 INCLUDE Files

The .INCLUDE pseudo-op allows one file to be inserted into another. The format is:

```
.INCLUDE <filename>
```

Nesting is permitted up to 10 levels.

#### 7.4.6 Symbol Files

Symbol files are read and written with the `.INSYM` and `.OUTSYM` pseudo-ops. The formats are:

```
.INSYM <filename>  
.OUTSYM <filename>
```

`.INSYM` works only on pass 1 and `.OUTSYM` work only on pass 2.

Only assembly symbols are included.

## Symbol Table Manipulation

## 7.4.7 Symbol Table Manipulation

Various pseudo-ops manipulate the symbol table. The .EXPUNGE pseudo-op removes symbols from the symbol table. The format is:

```
.EXPUNGE <symbol> [, <symbol>....]
```

.EQUALS and .OPSYN create new symbols having the same characteristics as existing symbols. .EQUALS works on both pass 1 and pass 2, while .OPSYN works only on pass 1. The format is:

```
.EQUALS <new symbol> , <old symbol>
```

```
.OPSYN <new symbol> , <old symbol>
```

## Cross-Reference

## 7.4.8 Cross-Reference

The cross-reference (concordance) function is built into the assembler. As the source program is assembled, references to names are saved in a file, recording the page number, process levels, and type of reference (use or definition). At the end of pass 2, this file is sorted by symbol name and formatted into the cross-reference listing. In addition to the automatic recording of references, the programmer may force a cross-reference notation in two ways. The .XREF pseudo-op takes a list of names that are to be included in the cross-reference as if they were used at that point. The old DAP convention of using "O&" followed by a name list is also supported if it is located after an instruction, i.e., it is not a syllable as previously allowed.

#### 7.4.9 User Error Messages

User generated error messages are provided by the .ERROR pseudo-op. When encountered, a normal compiler error is generated with the text provided.

```
.ERROR <text>
```

#### 7.4.10 User Message Files

The assembler provides alternate listing files for user generated messages. The file names are specified in the assembler command line with the -f flag. Multiple files may be used. During the compilation, the files are referenced by a file number, where file 0 is the first file declared in the command line and file 1 is the second, etc. Two pseudo-ops, .PRINT and .PNTNUM are used to output data to the message files. .PRINT outputs strings and .PNTNUM outputs numbers. The formats are:

```
.PRINT <file number> , <text>
```

```
.PNTNUM <file number> , <expression>
```

The format of <text> is the same as for .ASCII. Note that neither .PRINT or .PNTNUM automatically append <CR><LF>; new lines must be explicitly created.

## Pseudo-Op Summary

## 7.5 Pseudo-Op Summary

Pseudo-op	Reference	PL30 only	Purpose
.ASCII	7.2.9		generate character string
.BLOCK	7.2.11		reserve block of storage
.COMMENT	7.4.2		begin comment block
.CONST	7.2.8		(same as .CONSTANTS)
.CONSTANTS	7.2.8		generate constants
.DECIMAL	7.2.4		default radix to decimal
.ELSE	7.2.13		continue conditional assembly
.ENB	7.2.4		generate ENB and unlock levels
.ENDCOMMENT	7.4.2		end comment block
.ENDIF	7.2.13		end conditional assembly
.EQUALS	7.4.7		equate symbols on pass 1 and pass 2
.ERROR	7.4.9		print error message
.EXPUNGE	7.4.7		remove symbol from table
.FIELD	7.3.6	*	define bit field
.HEX	7.2.4		default radix to hexadecimal
.IF	7.2.13		begin conditional assembly
.INCLUDE	7.4.5		insert file
.INH	7.2.4		generate INH and lock out levels
.INSYM	7.4.6		read symbol table
.LCK	7.4.3		specify common process level(s)
.LEV	7.4.3		specify process level
.LIST	7.4.4		turn on listing
.LSTOBJ	7.4.4		list disassembled object code
.NARROW	7.4.4		leave wide-comment mode
.NMFSOPS	--		use NMFS opcode mnemonics
.NOP			
.OCTAL	7.2.4		default radix to octal
.OFFPAGE	7.3.4	*	generate automatic offpage references
.OFFSET	7.2.10		adjust logical location counter
.OLIST	7.4.4		list object only
.ONPAGE	7.3.4	*	don't generate automatic offpage references
.OPSYN	7.4.7		equate symbols on pass 1 only
.OUTSYM	7.4.6		write symbol table
.PAGE	7.4.4		start new listing page
.PNTNUM	7.4.10		print number on user message file
.PRINT	7.4.10		print string on user message file
.PZCON	7.2.8		designate page 0 constants

## Pseudo-Op Summary

.PZOFF	7.2.8		don't use page 0 constants
.PZON	7.2.8		use page 0 constants
.SECTION	7.2.12		begin or continue program section
.SLIST	7.4.4		list source and object
.STITL	7.4.4		(same as .STITLE)
.STITLE	7.4.4		specify listing page subtitle
.TEMPS	7.3.8	*	generate temporary variables
.TITLE	7.4.4		specify listing page title
.WIDE	7.4.2		enter wide-comment mode
.WORD	7.2.11		reserve word of storage
.X16OPS	--		use H316 opcode Mnemonics
.XLIST	7.4.4		turn off listing
.XLSTOBJ	7.4.4		don't list disassembled object code
.XREF	7.4.8		include symbols in cross-reference
ADDI	7.3.5	*	immediate opcode
ANAI	7.3.5	*	immediate opcode
BREAK	7.3.10	*	quit loop statement
CALLI	7.3.5	*	immediate opcode
CASI	7.3.5	*	immediate opcode
CLB	7.3.3	*	clear bit field to zeros
CMB	7.3.3	*	complement bit field
CONTINUE	7.3.10	*	goto body of loop statement
DPB	7.3.3	*	store bit field, justified
DPBZ	7.3.3	*	store into cleared bit field, justified
ELSE	7.3.9	*	last IF statement clause
ELSEIF	7.3.9	*	next IF statement clause
END	7.3.10	*	end of subroutine body
ENDIF	7.3.9	*	end IF statement
ENDREPEAT	7.3.10	*	end loop statement
ENDSTRUCTURE	7.3.7	*	end structure definition
ERAI	7.3.5	*	immediate opcode
EXTB	7.3.3	*	load bit field, not justified
IF	7.3.9	*	first IF statement clause
IMAI	7.3.5	*	immediate opcode
IRSI	7.3.5	*	immediate opcode
JMPI	7.3.5	*	immediate opcode
JSTI	7.3.5	*	immediate opcode
LDAI	7.3.5	*	immediate opcode
LDB	7.3.3	*	load bit field, justified
LDXI	7.3.5	*	immediate opcode
NEXT	7.3.10	*	goto termination test for loop statement

## Pseudo-Op Summary

POPI	7.3.5	*	immediate opcode
PROCEDURE	7.3.10	*	begin subroutine body
PUSHI	7.3.5	*	immediate opcode
REPEAT	7.3.10	*	begin loop statement
RETURN	7.3.10	*	return from subroutine
SHORT	7.3.9	*	IF clause modifier
SNB	7.3.3	*	skip if bit field nonzero
STAI	7.3.5	*	immediate opcode
START	7.2.14		end of source program
STB	7.3.3	*	set bit field to ones
STRUCTURE	7.3.7	*	begin structure definition
STXI	7.3.5	*	immediate opcode
SUBI	7.3.5	*	immediate opcode
SZB	7.3.3	*	skip if bit field zero
TEST	7.3.10	*	termination test for loop statement
UNTIL	7.3.10	*	termination test for loop statement
XRBM	7.3.3	*	exclusive-OR bit field, justified

## Assembler Variables Summary

## 7.6 Assembler Variables Summary

Variable	Reference	Purpose
%OFFSET	7.2.10	difference between logical and physical PCs
%OLIST	7.4.4	control over source listing
%PASS	—	assembly pass (1 or 2)
%RADIX	7.2.4	current radix
%XLEV	7.4.3	native process level
%XLIST	7.4.4	control over listing
%YLEV	7.4.3	common process level
.	7.2.10	logical location counter
UNCON	7.2.8	pass 2 end of constants

8 References

1. "Models 316 and 516 Programmer's Reference Manual", Honeywell Inc., November 1970, Document #70130072156E.
2. "MBB Microprogrammer's Handbook", BBN Report #4268.
3. "Specifications for the Interconnection of a Host and an IMP", BBN Report #1822.
4. "The ARPANET 1822L Host Access Protocol", BBN Report No. 5506 \*

9 Glossary

[to be provided]

BBN Report No. 5000

Index

10 Index

[to be provided]

## APPENDIX A Instruction Tables

The instruction tables in this appendix provide sorted summaries of the NMFS instructions contained in this manual, including the page number on which complete information for each instruction can be found.

The tables in this appendix are sorted as follows:

Appendix A.1 - By Mnemonic

Appendix A.2 - By Opcode

Appendix A.3 - By Instruction Name

Appendix A.4 - By Type and then Instruction Name

Symbols used in the "Cycles" column (execution times) are as follows:

< times fall within the range indicated

/ times are one of the two values given

MR, EA, JA depends on type of memory reference (see section 2.3)

n number of locations to be manipulated

f(B) depends on contents of the B register

++ depends on the value of the timeout word in the PCB

I/O depends on I/O time required

## A.1 Ordered by Mnemonic

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
ABX	0141000	Add B Register to X Register	ARITH	8	2-57
ACA	0141216	Add C Register to A Register	ARITH	11	2-53
ADD	0014000	Add Memory to A Register	ARITH	7 + MR	2-51
ALR16	0041700	A (Register) Left Rotate 16	SHIFT	16 + 141	2-38
ALR20	0041600	A (Register) Left Rotate 20	SHIFT	16 + 169	2-36
ALS	0041500	Arithmetic Left Shift	SHIFT	16 + 155	2-33
ANA	0006000	AND Memory to A Register	LOGIC	6 + MR	2-61
AOA	0141206	Add One to A Register	ARITH	7	2-54
AOX	0141014	Add One to X Register	ARITH	13	2-55
APR	0000003	Activate Process	PRSYS	144	3-28
ARR16	0040700	A (Register) Right Rotate 16	SHIFT	16 + 140	2-37
ARR20	0040600	A (Register) Right Rotate 20	SHIFT	16 + 168	2-35
ARS	0040500	Arithmetic Right Shift	SHIFT	17 + 133	2-32
AVA	0141110	Add Overflow to A Register	ARITH	9	2-58
BLI	0001401	Byte Load and Increment	BYTE	22	2-108
BLO	0001400	Byte Load	BYTE	19	2-108
BLT	0000031	Block Transfer Memory	LD/ST	15 + 7n	2-48
BSI	0001403	Byte Store and Increment	BYTE	29	2-109
BST	0001402	Byte Store	BYTE	25	2-108
CAE	0001102	Clear A Extension Bits	CONV	6	2-15
CAL	0141050	Clear A Register Left Half	LD/ST	6	2-44
CALL	0034000	Subroutine CALL Using Stack	STACK	11 + JA	2-100
CAR	0141044	Clear A Register Right Half	LD/ST	6	2-44
CAS	0022000	Compare A Reg to Mem & Skip	CNTRL	11 + MR	2-70
CASP	0100011	Copy A Register to SP	STACK	7	2-102
CCRO	0001032	Convert & Clear Rightmost One	LOGIC	10 + f(A)	2-65
CHK	0000032	Checksum Block of Memory	ARITH	14 + 5n	2-56
CHS16	0000052	Change (Complement) Sign of A	LOGIC	6	2-64
CHS20	0140024	Change (Complement) Sign of A	LOGIC	6	2-63
CLR	0141144	Copy A Register Lt to Rt 1/2	LD/ST	7	2-46
CMA	0140401	Ones Complement A Register	LOGIC	6	2-62
COB	0140504	Complement Overflow Bit	LD/ST	6	2-47
CRA	0140040	Clear A Register	LD/ST	6	2-43
CRL	0141250	Copy Register Rt to Lt Byte	LD/ST	7	2-46
CSA16	0000053	Copy Sign & Set A's Sign to +	LOGIC	10	2-65
CSA20	0140320	Copy Sign & Set A's Sign to +	LOGIC	10	2-64
CSPA	0100012	Copy SP to A Register	STACK	7	2-102
CSPX	0100013	Copy SP to X Register	STACK	10	2-103

CXB	0140510	Complement Indexed Bit	LOGIC	9	2-67
CXE	0001103	Clear X Extension Bits	CONV	7	2-15
DEQ	0000022	Dequeue First Item from Queue	QUEUE	23/48	2-95
DPR	0000023	Deactivate Process	PRSYS	148	3-28
EAB	0001041	End Around Borrow	CONV	15	2-18
EAC	0001040	End Around Carry	CONV	15	2-18
ECK	0000202	End Around Checksum	ARITH	15 + 7n	2-57
ENB	0000401	Enable Other Processes	PRSYS	10/25	3-32
ENQ	0000002	Enqueue A New Item on Queue	QUEUE	46	2-94
ERA	0012000	Exclusive OR Memory to A Reg	LOGIC	6 + MR	2-61
ERB	0000200	Retrieve Error Bits	PCTRL	14	2-85
ERC	0000105	Error Light Clear	PCTRL	14	2-84
ERR	0000101	Interrogate Memory Errors	PCTRL	23	2-84
FFO	0000033	Find First One	LOGIC	9·29	2-65
GPR	0000043	Goad Process	PRSYS	39	3-30
HLT	0000000	Halt the Processor	PCTRL	5	2-80
IAB	0000201	Interchange A and B Registers	LD/ST	9	2-43
ICA	0141340	Interchange A Register Halves	LD/ST	7	2-44
ICL	0141140	Interchange & Clr A (Lt 1/2)	LD/ST	8	2-45
ICR	0141240	Interchange & Clr A (Rt 1/2)	LD/ST	8	2-45
IMA	0026000	Interchange Memory & A Reg	LD/ST	11 + MR	2-41
INH	0001001	Inhibit Other Processes	PRSYS	9	3-32
IRS	0024000	Increment, Replace & Skip	CNTRL	10 + MR	2-71
ITS	0100010	Increment Top of Stack	STACK	13	2-102
JMP	0002000	Unconditional Jump	CNTRL	7 + JA	2-69
JST	0020000	Jump & Store Program Counter	CNTRL	10 + EA	2-70
LAI	0141700	Load A Indirectly thru Self	LD/ST	9	2-40
LAT	0141510	Load A from Top of Stack	STACK	12	2-104
LDA	0004000	Load A Register	LD/ST	6 + MR	2-40
LDX	0072000	Load X Register	LD/ST	10 + MR	2-42
LGL	0041400	Logical Left Shift	SHIFT	16·134	2-29
LGR	0040400	Logical Right Shift	SHIFT	16·130	2-28
LITES	0000011	Write LIT (LITES) Register	PCTRL	21	2-81
LLL	0041000	Long Left Logical Shift	SHIFT	17·234	2-30
LLR16	0041300	Long Left Rotate 16	SHIFT	17·254	2-38
LLR20	0041200	Long Left Rotate 20	SHIFT	17·310	2-36
LLS	0041100	Long Left Arithmetic Shift	SHIFT	28·287	2-33
LOK	0001011	Obtain Lock	IPCOM	14	2-112
LRL	0040000	Long Right Logical Shift	SHIFT	17·231	2-29
LRR16	0040300	Long Right Rotate 16	SHIFT	17·253	2-37
LRR20	0040200	Long Right Rotate 20	SHIFT	17·309	2-35
LRS	0040100	Long Right Arithmetic Shift	SHIFT	28·244	2-32
LXA	0141714	Load X Indirectly thru A	LD/ST	13	2-43

LXI	0141704	Load X Indirectly thru Self	LD/ST	13	2-42
LXT	0141504	Load X from Top of Stack	STACK	13	2-105
MEMHI	0000012	Read Memory High Bound	PCTRL	9	2-81
MMD	0001002	Measurement Mode Disable	PRSYS	6	3-37
MME	0000402	Measurement Mode Enable	PRSYS	6	3-36
NMFS	0000030	NMFS Mode Control	PRSYS	22/281	3-26
NOP	0101000	No Operation	CNTRL	6	2-71
P	0001013	Decrement Semaphore (Probeer)	IPCOM	14	2-115
PCB	0000020	Load PCB Into X Register	PRSYS	11	3-37
PDV	0001003	Poke Device Driver	I/O	47 + I/O	4-20
POP	0036000	Pop Memory Contents off Stack	STACK	14 + EA	2-101
POPA	0101003	Pop A Reg Contents off Stack	STACK	9	2-104
PUSH	0030000	Push Memory Contents onto Stx	STACK	10 + MR	2-100
PUSHA	0101002	Push A Reg Contents onto Stk	STACK	11	2-103
RCB	0140200	Reset C Register (Bit)	LD/ST	7	2-46
RCV	0001012	Receive	IPCOM	14	2-115
RDCLOK	0000010	Read RTC Register	PCTRL	11	2-80
RETN	0100002	Subroutine Return Using Stack	STACK	9	2-101
RMQ	0000042	Rmv Specified Item from Queue	QUEUE	56	2-96
ROB	0140604	Reset Overflow Bit	LD/ST	7	2-47
RSM	0000013	Read Special Memory	PCTRL	12+f(B)	2-87
RXB	0140210	Reset Indexed Bit	LOGIC	9	2-66
SAT	0141610	Store A in Top of Stack	STACK	12	2-103
SCB	0140600	Set C Register (Bit)	LD/ST	7	2-46
SEA	0001100	Sign Extend A	CONV	8	2-14
SEI	0100023	Skip if Extension Insig.	CONV	11/13	2-16
SEN	0101022	Skip if Extension Nonzero	CONV	9	2-16
SES	0101023	Skip if Extension Significant	CONV	11/13	2-16
SEX	0001101	Sign Extend X	CONV	9	2-14
SEZ	0100022	Skip if Extension Zero	CONV	9	2-16
SGT16	0100202	Skip if A Register > 0 16	CNTRL	9/10	2-75
SGT20	0100401	Skip if A Register > 0 20	CNTRL	9/10	2-74
SKP	0100000	Unconditional Skip	CNTRL	6	2-71
SLE16	0101202	Skip if A Reg LT or EQ Zero 16	CNTRL	9/10	2-75
SLE20	0101401	Skip if A Reg LT or EQ Zero 20	CNTRL	9/10	2-75
SLN	0101100	Skip if Low Bit of A Reg Non0	CNTRL	8	2-76
SLZ	0100100	Skip if Low Bit of A Reg Zero	CNTRL	8	2-76
SMI16	0101201	Skip if A Reg Minus 16	CNTRL	8	2-74
SMI20	0101400	Skip if A Reg Minus 20	CNTRL	8	2-74
SND	0001022	Send Trap	IPCOM	14	2-114
SNZ16	0101200	Skip if A Register Nonzero 16	CNTRL	8	2-73
SNZ20	0101040	Skip if A Register Nonzero 20	CNTRL	8	2-72
SOA	0141306	Subtract One from A Register	ARITH	9	2-54

SOB	0140204	Set Overflow Bit	LD/ST	7	2-47
SOC	0100021	Skip if Overflow Clear	CNTRL	9	2-78
SOS	0101021	Skip if Overflow Set	CNTRL	9	2-77
SOX	0141114	Subtract One from X Register	ARITH	13	2-55
SPL16	0100201	Skip if A Reg Plus ( $\geq 0$ ) 16	CNTRL	8	2-73
SPL20	0100400	Skip if A Reg Plus ( $\geq 0$ ) 20	CNTRL	8	2-73
SPR	0000103	Suspend Process	PRSYS	163 $\llcorner$ ++	3-31
SRC	0100001	Skip if Reset C Register	CNTRL	10	2-76
SRETN	0100003	Sub Skip Return Using Stack	STACK	10	2-101
SSC	0101001	Skip if Set C Register	CNTRL	10	2-77
SSM16	0000051	Set Sign of A Register Minus	LOGIC	6	2-63
SSM20	0140500	Set Sign of A Register Minus	LOGIC	6	2-63
SSP16	0000050	Set Sign of A Register Plus 16	LOGIC	6	2-62
SSP20	0140100	Set Sign of A Register Plus 20	LOGIC	6	2-62
STA	0010000	Store A Register	LD/ST	10 + EA	2-41
STX	0032000	Store X Register	LD/ST	10 + EA	2-41
SUB	0016000	Subtract Memory from A Reg	ARITH	8 + MR	2-52
SVA	0141310	Subtract Overflow from A Reg	ARITH	12	2-59
SXA	0141614	Store X Indirectly thru A	LD/ST	13	2-42
SXB	0140610	Set Indexed Bit	LOGIC	9	2-66
SXT	0141604	Store X in Top of Stack	STACK	12	2-104
SZC	0100041	Skip if A Reg 0 & Reset C Reg	CNTRL	8/13	2-77
SZE16	0100200	Skip if A Register Zero 16	CNTRL	8	2-72
SZE20	0100040	Skip if A Register Zero 20	CNTRL	8	2-72
SZO	0100042	Skip if A Reg 0 & Ovrflo Reset	CNTRL	10	2-78
TAB	0001200	Truncate A and B	CONV	20	2-17
TCA	0140407	Two's Complement A Register	ARITH	7	2-52
TPR	0000203	Timeout Process	PRSYS	22	3-35
TRB	0000041	Transfer Memory Backwards	LD/ST	11 + 7n	2-49
UGS	0000113	Ungoad Self	PRSYS	13	3-33
ULK	0001021	Release Lock	IPCOM	15	2-113
V	0001023	Increment Semaphore (Verlaat)	IPCOM	15	2-116
VER	0000100	Return Version Number	PCTRL	9	2-83
WATCH	0000761	Watch On or Off	PCTRL	11/15	2-86
WSM	0000021	Write Special Memory	PCTRL	12+f(B)	2-88
XAB	0001201	Extend A and B	CONV	15	2-17
XDV	0000403	Execute Device Function	I/O	48 + I/O	4-21

## A.2 Ordered by Opcode

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
HLT	0000000	Halt the Processor	PCTRL	5	2-80
ENQ	0000002	Enqueue A New Item on Queue	QUEUE	46	2-94
APR	0000003	Activate Process	PRSYS	144	3-28
RDCLOK	0000010	Read RTC Register	PCTRL	11	2-80
LITES	0000011	Write LIT (LITES) Register	PCTRL	21	2-81
MEMHI	0000012	Read Memory High Bound	PCTRL	9	2-81
RSM	0000013	Read Special Memory	PCTRL	12+f(B)	2-87
PCB	0000020	Load PCB Into X Register	PRSYS	11	3-37
WSM	0000021	Write Special Memory	PCTRL	12+f(B)	2-88
DEQ	0000022	Dequeue First Item from Queue	QUEUE	23/48	2-95
DPR	0000023	Deactivate Process	PRSYS	148	3-28
NMFS	0000030	NMFS Mode Control	PRSYS	22/281	3-26
BLT	0000031	Block Transfer Memory	LD/ST	15 + 7n	2-48
CHK	0000032	Checksum Block of Memory	ARITH	14 + 5n	2-56
FFO	0000033	Find First One	LOGIC	9<29	2-65
TRB	0000041	Transfer Memory Backwards	LD/ST	11 + 7n	2-49
RMQ	0000042	Rmv Specified Item from Queue	QUEUE	56	2-96
GPR	0000043	Goad Process	PRSYS	39	3-30
SSP16	0000050	Set Sign of A Register Plus 16	LOGIC	6	2-62
SSM16	0000051	Set Sign of A Register Minus	LOGIC	6	2-63
CHS16	0000052	Change (Complement) Sign of A	LOGIC	6	2-64
CSA16	0000053	Copy Sign & Set A's Sign to +	LOGIC	10	2-65
VER	0000100	Return Version Number	PCTRL	9	2-83
ERR	0000101	Interrogate Memory Errors	PCTRL	23	2-84
SPR	0000103	Suspend Process	PRSYS	163<++	3-31
ERC	0000105	Error Light Clear	PCTRL	14	2-84
UGS	0000113	Ungoad Self	PRSYS	13	3-33
ERB	0000200	Retrieve Error Bits	PCTRL	14	2-85
IAB	0000201	Interchange A and B Registers	LD/ST	9	2-43
ECK	0000202	End Around Checksum	ARITH	15 + 7n	2-57
TPR	0000203	Timeout Process	PRSYS	22	3-35
ENB	0000401	Enable Other Processes	PRSYS	10/25	3-32
MME	0000402	Measurement Mode Enable	PRSYS	6	3-36
XDV	0000403	Execute Device Function	I/O	48 + I/O	4-21
WATCH	0000761	Watch On or Off	PCTRL	11/15	2-86
INH	0001001	Inhibit Other Processes	PRSYS	9	3-32
MMD	0001002	Measurement Mode Disable	PRSYS	6	3-37
PDV	0001003	Poke Device Driver	I/O	47 + I/O	4-20

LOK	0001011	Obtain Lock	IPCOM	14	2-112
RCV	0001012	Receive	IPCOM	14	2-115
P	0001013	Decrement Semaphore (Probeer)	IPCOM	14	2-115
ULK	0001021	Release Lock	IPCOM	15	2-113
SND	0001022	Send Trap	IPCOM	14	2-114
V	0001023	Increment Semaphore (Verlaat)	IPCOM	15	2-116
CCRO	0001032	Convert & Clear Rightmost One	LOGIC	10 + f(A)	2-65
EAC	0001040	End Around Carry	CONV	15	2-18
EAB	0001041	End Around Borrow	CONV	15	2-18
SEA	0001100	Sign Extend A	CONV	8	2-14
SEX	0001101	Sign Extend X	CONV	9	2-14
CAE	0001102	Clear A Extension Bits	CONV	6	2-15
CXE	0001103	Clear X Extension Bits	CONV	7	2-15
TAB	0001200	Truncate A and B	CONV	20	2-17
XAB	0001201	Extend A and B	CONV	15	2-17
BLO	0001400	Byte Load	BYTE	19	2-108
BLI	0001401	Byte Load and Increment	BYTE	22	2-108
BST	0001402	Byte Store	BYTE	25	2-108
BSI	0001403	Byte Store and Increment	BYTE	29	2-109
JMP	0002000	Unconditional Jump	CNTRL	7 + JA	2-69
LDA	0004000	Load A Register	LD/ST	6 + MR	2-40
ANA	0006000	AND Memory to A Register	LOGIC	6 + MR	2-61
STA	0010000	Store A Register	LD/ST	10 + EA	2-41
ERA	0012000	Exclusive OR Memory to A Reg	LOGIC	6 + MR	2-61
ADD	0014000	Add Memory to A Register	ARITH	7 + MR	2-51
SUB	0016000	Subtract Memory from A Reg	ARITH	8 + MR	2-52
JST	0020000	Jump & Store Program Counter	CNTRL	10 + EA	2-70
CAS	0022000	Compare A Reg to Mem & Skip	CNTRL	11 + MR	2-70
IRS	0024000	Increment, Replace & Skip	CNTRL	10 + MR	2-71
IMA	0026000	Interchange Memory & A Reg	LD/ST	11 + MR	2-41
PUSH	0030000	Push Memory Contents onto Stx	STACK	10 + MR	2-100
STX	0032000	Store X Register	LD/ST	10 + EA	2-41
CALL	0034000	Subroutine CALL Using Stack	STACK	11 + JA	2-100
POP	0036000	Pop Memory Contents off Stack	STACK	14 + EA	2-101
LRL	0040000	Long Right Logical Shift	SHIFT	17<231	2-29
LRS	0040100	Long Right Arithmetic Shift	SHIFT	28<244	2-32
LRR20	0040200	Long Right Rotate 20	SHIFT	17<309	2-35
LRR16	0040300	Long Right Rotate 16	SHIFT	17<253	2-37
LGR	0040400	Logical Right Shift	SHIFT	16<130	2-28
ARS	0040500	Arithmetic Right Shift	SHIFT	17<133	2-32
ARR20	0040600	A (Register) Right Rotate 20	SHIFT	16<168	2-35
ARR16	0040700	A (Register) Right Rotate 16	SHIFT	16<140	2-37
LLL	0041000	Long Left Logical Shift	SHIFT	17<234	2-30

LLS	0041100	Long Left Arithmetic Shift	SHIFT	28<287	2-33
LLR20	0041200	Long Left Rotate 20	SHIFT	17<310	2-36
LLR16	0041300	Long Left Rotate 16	SHIFT	17<254	2-38
LGL	0041400	Logical Left Shift	SHIFT	16<134	2-29
ALS	0041500	Arithmetic Left Shift	SHIFT	16<155	2-33
ALR20	0041600	A (Register) Left Rotate 20	SHIFT	16<169	2-36
ALR16	0041700	A (Register) Left Rotate 16	SHIFT	16<141	2-38
LDX	0072000	Load X Register	LD/ST	10 + MR	2-42
SKP	0100000	Unconditional Skip	CNTRL	6	2-71
SRC	0100001	Skip if Reset C Register	CNTRL	10	2-76
RETN	0100002	Subroutine Return Using Stack	STACK	9	2-101
SRETN	0100003	Sub Skip Return Using Stack	STACK	10	2-101
ITS	0100010	Increment Top of Stack	STACK	13	2-102
CASP	0100011	Copy A Register to SP	STACK	7	2-102
CSPA	0100012	Copy SP to A Register	STACK	7	2-102
CSPX	0100013	Copy SP to X Register	STACK	10	2-103
SOC	0100021	Skip if Overflow Clear	CNTRL	9	2-78
SEZ	0100022	Skip if Extension Zero	CONV	9	2-16
SEI	0100023	Skip if Extension Insig.	CONV	11/13	2-16
SZE20	0100040	Skip if A Register Zero 20	CNTRL	8	2-72
SZC	0100041	Skip if A Reg 0 & Reset C Reg	CNTRL	8/13	2-77
SZO	0100042	Skip if A Reg 0 & Ovrflo Reset	CNTRL	10	2-78
SLZ	0100100	Skip if Low Bit of A Reg Zero	CNTRL	8	2-76
SZE16	0100200	Skip if A Register Zero 16	CNTRL	8	2-72
SPL16	0100201	Skip if A Reg Plus (>=0) 16	CNTRL	8	2-73
SGT16	0100202	Skip if A Register > 0 16	CNTRL	9/10	2-75
SPL20	0100400	Skip if A Reg Plus (>=0) 20	CNTRL	8	2-73
SGT20	0100401	Skip if A Register > 0 20	CNTRL	9/10	2-74
NOP	0101000	No Operation	CNTRL	6	2-71
SSC	0101001	Skip if Set C Register	CNTRL	10	2-77
PUSHA	0101002	Push A Reg Contents onto Stk	STACK	11	2-103
POPA	0101003	Pop A Reg Contents off Stack	STACK	9	2-104
SOS	0101021	Skip if Overflow Set	CNTRL	9	2-77
SEN	0101022	Skip if Extension Nonzero	CONV	9	2-16
SES	0101023	Skip if Extension Significant	CONV	11/13	2-16
SNZ20	0101040	Skip if A Register Nonzero 20	CNTRL	8	2-72
SLN	0101100	Skip if Low Bit of A Reg Non0	CNTRL	8	2-76
SNZ16	0101200	Skip if A Register Nonzero 16	CNTRL	8	2-73
SMI16	0101201	Skip if A Reg Minus 16	CNTRL	8	2-74
SLE16	0101202	Skip if A Reg LT or EQ Zero 16	CNTRL	9/10	2-75
SMI20	0101400	Skip if A Reg Minus 20	CNTRL	8	2-74
SLE20	0101401	Skip if A Reg LT or EQ Zero 20	CNTRL	9/10	2-75
CHS20	0140024	Change (Complement) Sign of A	LOGIC	6	2-63

CRA	0140040	Clear A Register	LD/ST	6	2-43
SSP20	0140100	Set Sign of A Register Plus 20	LOGIC	6	2-62
RCB	0140200	Reset C Register (Bit)	LD/ST	7	2-46
SOB	0140204	Set Overflow Bit	LD/ST	7	2-47
RXB	0140210	Reset Indexed Bit	LOGIC	9	2-66
CSA20	0140320	Copy Sign & Set A's Sign to +	LOGIC	10	2-64
CMA	0140401	Ones Complement A Register	LOGIC	6	2-62
TCA	0140407	Two's Complement A Register	ARITH	7	2-52
SSM20	0140500	Set Sign of A Register Minus	LOGIC	6	2-63
COB	0140504	Complement Overflow Bit	LD/ST	6	2-47
CXB	0140510	Complement Indexed Bit	LOGIC	9	2-67
SCB	0140600	Set C Register (Bit)	LD/ST	7	2-46
ROB	0140604	Reset Overflow Bit	LD/ST	7	2-47
SXB	0140610	Set Indexed Bit	LOGIC	9	2-66
ABX	0141000	Add B Register to X Register	ARITH	8	2-57
AOX	0141014	Add One to X Register	ARITH	13	2-55
CAR	0141044	Clear A Register Right Half	LD/ST	6	2-44
CAL	0141050	Clear A Register Left Half	LD/ST	6	2-44
AVA	0141110	Add Overflow to A Register	ARITH	9	2-58
SOX	0141114	Subtract One from X Register	ARITH	13	2-55
ICL	0141140	Interchange & Clr A (Lt 1/2)	LD/ST	8	2-45
CLR	0141144	Copy A Register Lt to Rt 1/2	LD/ST	7	2-46
AOA	0141206	Add One to A Register	ARITH	7	2-54
ACA	0141216	Add C Register to A Register	ARITH	11	2-53
ICR	0141240	Interchange & Clr A (Rt 1/2)	LD/ST	8	2-45
CRL	0141250	Copy Register Rt to Lt Byte	LD/ST	7	2-46
SOA	0141306	Subtract One from A Register	ARITH	9	2-54
SVA	0141310	Subtract Overflow from A Reg	ARITH	12	2-59
ICA	0141340	Interchange A Register Halves	LD/ST	7	2-44
LXT	0141504	Load X from Top of Stack	STACK	13	2-105
LAT	0141510	Load A from Top of Stack	STACK	12	2-104
SXT	0141604	Store X in Top of Stack	STACK	12	2-104
SAT	0141610	Store A in Top of Stack	STACK	12	2-103
SXA	0141614	Store X Indirectly thru A	LD/ST	13	2-42
LAI	0141700	Load A Indirectly thru Self	LD/ST	9	2-40
LXI	0141704	Load X Indirectly thru Self	LD/ST	13	2-42
LXA	0141714	Load X Indirectly thru A	LD/ST	13	2-43

## Ordered by Instruction Name

## A.3 Ordered by Instruction Name

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
ALR16	0041700	A (Register) Left Rotate 16	SHIFT	16 + 141	2-38
ALR20	0041600	A (Register) Left Rotate 20	SHIFT	16 + 169	2-36
ARR16	0040700	A (Register) Right Rotate 16	SHIFT	16 + 140	2-37
ARR20	0040600	A (Register) Right Rotate 20	SHIFT	16 + 168	2-35
ANA	0006000	AND Memory to A Register	LOGIC	6 + MR	2-61
APR	0000003	Activate Process	PRSYS	144	3-28
ABX	0141000	Add B Register to X Register	ARITH	8	2-57
ACA	0141216	Add C Register to A Register	ARITH	11	2-53
ADD	0014000	Add Memory to A Register	ARITH	7 + MR	2-51
AOA	0141206	Add One to A Register	ARITH	7	2-54
AOX	0141014	Add One to X Register	ARITH	13	2-55
AVA	0141110	Add Overflow to A Register	ARITH	9	2-58
ALS	0041500	Arithmetic Left Shift	SHIFT	16 + 155	2-33
ARS	0040500	Arithmetic Right Shift	SHIFT	17 + 133	2-32
BLT	0000031	Block Transfer Memory	LD/ST	15 + 7n	2-48
BLO	0001400	Byte Load	BYTE	19	2-108
BLI	0001401	Byte Load and Increment	BYTE	22	2-108
BST	0001402	Byte Store	BYTE	25	2-108
BSI	0001403	Byte Store and Increment	BYTE	29	2-109
CHS16	0000052	Change (Complement) Sign of A	LOGIC	6	2-64
CHS20	0140024	Change (Complement) Sign of A	LOGIC	6	2-63
CHK	0000032	Checksum Block of Memory	ARITH	14 + 5n	2-56
CAE	0001102	Clear A Extension Bits	CONV	6	2-15
CRA	0140040	Clear A Register	LD/ST	6	2-43
CAL	0141050	Clear A Register Left Half	LD/ST	6	2-44
CAR	0141044	Clear A Register Right Half	LD/ST	6	2-44
CXE	0001103	Clear X Extension Bits	CONV	7	2-15
CAS	0022000	Compare A Reg to Mem & Skip	CNTRL	11 + MR	2-70
CXB	0140510	Complement Indexed Bit	LOGIC	9	2-67
COB	0140504	Complement Overflow Bit	LD/ST	6	2-47
CCRO	0001032	Convert & Clear Rightmost One	LOGIC	10 + f(A)	2-65
CLR	0141144	Copy A Register Lt to Rt 1/2	LD/ST	7	2-46
CASP	0100011	Copy A Register to SP	STACK	7	2-102
CRL	0141250	Copy Register Rt to Lt Byte	LD/ST	7	2-46
CSPA	0100012	Copy SP to A Register	STACK	7	2-102
CSPX	0100013	Copy SP to X Register	STACK	10	2-103
CSA16	0000053	Copy Sign & Set A's Sign to +	LOGIC	10	2-65
CSA20	0140320	Copy Sign & Set A's Sign to +	LOGIC	10	2-64

Ordered by Instruction Name

DPR	0000023	Deactivate Process	PRSYS	148	3-28
P	0001013	Decrement Semaphore (Probeer)	IPCOM	14	2-115
DEQ	0000022	Dequeue First Item from Queue	QUEUE	23/48	2-95
ENB	0000401	Enable Other Processes	PRSYS	10/25	3-32
EAB	0001041	End Around Borrow	CONV	15	2-18
EAC	0001040	End Around Carry	CONV	15	2-18
ECK	0000202	End Around Checksum	ARITH	15 + 7n	2-57
ENQ	0000002	Enqueue A New Item on Queue	QUEUE	46	2-94
ERC	0000105	Error Light Clear	PCTRL	14	2-84
ERA	0012000	Exclusive OR Memory to A Reg	LOGIC	6 + MR	2-61
XDV	0000403	Execute Device Function	I/O	48 + I/O	4-21
XAB	0001201	Extend A and B	CONV	15	2-17
FFO	0000033	Find First One	LOGIC	9<29	2-65
GPR	0000043	Goad Process	PRSYS	39	3-30
HLT	0000000	Halt the Processor	PCTRL	5	2-80
V	0001023	Increment Semaphore (Verlaat)	IPCOM	15	2-116
ITS	0100010	Increment Top of Stack	STACK	13	2-102
IRS	0024000	Increment, Replace & Skip	CNTRL	10 + MR	2-71
INH	0001001	Inhibit Other Processes	PRSYS	9	3-32
ICL	0141140	Interchange & Clr A (Lt 1/2)	LD/ST	8	2-45
ICR	0141240	Interchange & Clr A (Rt 1/2)	LD/ST	8	2-45
ICA	0141340	Interchange A Register Halves	LD/ST	7	2-44
IAB	0000201	Interchange A and B Registers	LD/ST	9	2-43
IMA	0026000	Interchange Memory & A Reg	LD/ST	11 + MR	2-41
ERR	0000101	Interrogate Memory Errors	PCTRL	23	2-84
JST	0020000	Jump & Store Program Counter	CNTRL	10 + EA	2-70
LAI	0141700	Load A Indirectly thru Self	LD/ST	9	2-40
LDA	0004000	Load A Register	LD/ST	6 + MR	2-40
LAT	0141510	Load A from Top of Stack	STACK	12	2-104
PCB	0000020	Load PCB Into X Register	PRSYS	11	3-37
LXA	0141714	Load X Indirectly thru A	LD/ST	13	2-43
LXI	0141704	Load X Indirectly thru Self	LD/ST	13	2-42
LDX	0072000	Load X Register	LD/ST	10 + MR	2-42
LXT	0141504	Load X from Top of Stack	STACK	13	2-105
LGL	0041400	Logical Left Shift	SHIFT	16<134	2-29
LGR	0040400	Logical Right Shift	SHIFT	16<130	2-28
LLS	0041100	Long Left Arithmetic Shift	SHIFT	28<287	2-33
LLL	0041000	Long Left Logical Shift	SHIFT	17<234	2-30
LLR16	0041300	Long Left Rotate 16	SHIFT	17<254	2-38
LLR20	0041200	Long Left Rotate 20	SHIFT	17<310	2-36
LRS	0040100	Long Right Arithmetic Shift	SHIFT	28<244	2-32
LRL	0040000	Long Right Logical Shift	SHIFT	17<231	2-29
LRR16	0040300	Long Right Rotate 16	SHIFT	17<253	2-37

## Ordered by Instruction Name

LRR20	0040200	Long Right Rotate 20	SHIFT	17+309	2-35
MMD	0001002	Measurement Mode Disable	PRSYS	6	3-37
MME	0000402	Measurement Mode Enable	PRSYS	6	3-36
NMFS	0000030	NMFS Mode Control	PRSYS	22/281	3-26
NOP	0101000	No Operation	CNTRL	6	2-71
LOK	0001011	Obtain Lock	IPCOM	14	2-112
CMA	0140401	Ones Complement A Register	LOGIC	6	2-62
PDV	0001003	Poke Device Driver	I/O	47 + I/O	4-20
POPA	0101003	Pop A Reg Contents off Stack	STACK	9	2-104
POP	0036000	Pop Memory Contents off Stack	STACK	14 + EA	2-101
PUSHA	0101002	Push A Reg Contents onto Stk	STACK	11	2-103
PUSH	0030000	Push Memory Contents onto Stx	STACK	10 + MR	2-100
MEMHI	0000012	Read Memory High Bound	PCTRL	9	2-81
RDCLOK	0000010	Read RTC Register	PCTRL	11	2-80
RSM	0000013	Read Special Memory	PCTRL	12+f(B)	2-87
RCV	0001012	Receive	IPCOM	14	2-115
ULK	0001021	Release Lock	IPCOM	15	2-113
RCB	0140200	Reset C Register (Bit)	LD/ST	7	2-46
RXB	0140210	Reset Indexed Bit	LOGIC	9	2-66
ROB	0140604	Reset Overflow Bit	LD/ST	7	2-47
ERB	0000200	Retrieve Error Bits	PCTRL	14	2-85
VER	0000100	Return Version Number	PCTRL	9	2-83
RMQ	0000042	Rmv Specified Item from Queue	QUEUE	56	2-96
SND	0001022	Send Trap	IPCOM	14	2-114
SCB	0140600	Set C Register (Bit)	LD/ST	7	2-46
SXB	0140610	Set Indexed Bit	LOGIC	9	2-66
SOB	0140204	Set Overflow Bit	LD/ST	7	2-47
SSM16	0000051	Set Sign of A Register Minus	LOGIC	6	2-63
SSM20	0140500	Set Sign of A Register Minus	LOGIC	6	2-63
SSP16	0000050	Set Sign of A Register Plus 16	LOGIC	6	2-62
SSP20	0140100	Set Sign of A Register Plus 20	LOGIC	6	2-62
SEA	0001100	Sign Extend A	CONV	8	2-14
SEX	0001101	Sign Extend X	CONV	9	2-14
SZO	0100042	Skip if A Reg 0 & Ovrflo Reset	CNTRL	10	2-78
SZC	0100041	Skip if A Reg 0 & Reset C Reg	CNTRL	8/13	2-77
SLE16	0101202	Skip if A Reg LT or EQ Zero 16	CNTRL	9/10	2-75
SLE20	0101401	Skip if A Reg LT or EQ Zero 20	CNTRL	9/10	2-75
SMI16	0101201	Skip if A Reg Minus 16	CNTRL	8	2-74
SMI20	0101400	Skip if A Reg Minus 20	CNTRL	8	2-74
SPL16	0100201	Skip if A Reg Plus (>=0) 16	CNTRL	8	2-73
SPL20	0100400	Skip if A Reg Plus (>=0) 20	CNTRL	8	2-73
SGT16	0100202	Skip if A Register > 0 16	CNTRL	9/10	2-75
SGT20	0100401	Skip if A Register > 0 20	CNTRL	9/10	2-74

## Ordered by Instruction Name

SNZ16	0101200	Skip if A Register Nonzero 16	CNTRL	8	2-73
SNZ20	0101040	Skip if A Register Nonzero 20	CNTRL	8	2-72
SZE16	0100200	Skip if A Register Zero 16	CNTRL	8	2-72
SZE20	0100040	Skip if A Register Zero 20	CNTRL	8	2-72
SEI	0100023	Skip if Extension Insig.	CONV	11/13	2-16
SEN	0101022	Skip if Extension Nonzero	CONV	9	2-16
SES	0101023	Skip if Extension Significant	CONV	11/13	2-16
SEZ	0100022	Skip if Extension Zero	CONV	9	2-16
SLN	0101100	Skip if Low Bit of A Reg Non0	CNTRL	8	2-76
SLZ	0100100	Skip if Low Bit of A Reg Zero	CNTRL	8	2-76
SOC	0100021	Skip if Overflow Clear	CNTRL	9	2-78
SOS	0101021	Skip if Overflow Set	CNTRL	9	2-77
SRC	0100001	Skip if Reset C Register	CNTRL	10	2-76
SSC	0101001	Skip if Set C Register	CNTRL	10	2-77
STA	0010000	Store A Register	LD/ST	10 + EA	2-41
SAT	0141610	Store A in Top of Stack	STACK	12	2-103
SXA	0141614	Store X Indirectly thru A	LD/ST	13	2-42
STX	0032000	Store X Register	LD/ST	10 + EA	2-41
SXT	0141604	Store X in Top of Stack	STACK	12	2-104
SRETN	0100003	Sub Skip Return Using Stack	STACK	10	2-101
CALL	0034000	Subroutine CALL Using Stack	STACK	11 + JA	2-100
RETN	0100002	Subroutine Return Using Stack	STACK	9	2-101
SUB	0016000	Subtract Memory from A Reg	ARITH	8 + MR	2-52
SOA	0141306	Subtract One from A Register	ARITH	9	2-54
SOX	0141114	Subtract One from X Register	ARITH	13	2-55
SVA	0141310	Subtract Overflow from A Reg	ARITH	12	2-59
SPR	0000103	Suspend Process	PRSYS	163<++	3-31
TPR	0000203	Timeout Process	PRSYS	22	3-35
TRB	0000041	Transfer Memory Backwards	LD/ST	11 + 7n	2-49
TAB	0001200	Truncate A and B	CONV	20	2-17
TCA	0140407	Two's Complement A Register	ARITH	7	2-52
JMP	0002000	Unconditional Jump	CNTRL	7 + JA	2-69
SKP	0100000	Unconditional Skip	CNTRL	6	2-71
UGS	0000113	Ungoad Self	PRSYS	13	3-33
WATCH	0000761	Watch On or Off	PCTRL	11/15	2-86
LITES	0000011	Write LIT (LITES) Register	PCTRL	21	2-81
WSM	0000021	Write Special Memory	PCTRL	12+f(B)	2-88

Ordered by Type, then Instruction Name

## A.4 Ordered by Type, then Instruction Name

Mnemon.	OpCode	Instruction Name	Type	Cycles	Page
=====	=====	=====	=====	=====	=====
ABX	0141000	Add B Register to X Register	ARITH	8	2-57
ACA	0141216	Add C Register to A Register	ARITH	11	2-53
ADD	0014000	Add Memory to A Register	ARITH	7 + MR	2-51
AOA	0141206	Add One to A Register	ARITH	7	2-54
AOX	0141014	Add One to X Register	ARITH	13	2-55
AVA	0141110	Add Overflow to A Register	ARITH	9	2-58
CHK	0000032	Checksum Block of Memory	ARITH	14 + 5n	2-56
ECK	0000202	End Around Checksum	ARITH	15 + 7n	2-57
SUB	0016000	Subtract Memory from A Reg	ARITH	8 + MR	2-52
SOA	0141306	Subtract One from A Register	ARITH	9	2-54
SOX	0141114	Subtract One from X Register	ARITH	13	2-55
SVA	0141310	Subtract Overflow from A Reg	ARITH	12	2-59
TCA	0140407	Two's Complement A Register	ARITH	7	2-52
BLO	0001400	Byte Load	BYTE	19	2-108
BLI	0001401	Byte Load and Increment	BYTE	22	2-108
BST	0001402	Byte Store	BYTE	25	2-108
BSI	0001403	Byte Store and Increment	BYTE	29	2-109
CAS	0022000	Compare A Reg to Mem & Skip	CNTRL	11 + MR	2-70
IRS	0024000	Increment, Replace & Skip	CNTRL	10 + MR	2-71
JST	0020000	Jump & Store Program Counter	CNTRL	10 + EA	2-70
NOP	0101000	No Operation	CNTRL	6	2-71
SZO	0100042	Skip if A Reg 0 & Ovrflo Reset	CNTRL	10	2-78
SZC	0100041	Skip if A Reg 0 & Reset C Reg	CNTRL	8/13	2-77
SLE16	0101202	Skip if A Reg LT or EQ Zero 16	CNTRL	9/10	2-75
SLE20	0101401	Skip if A Reg LT or EQ Zero 20	CNTRL	9/10	2-75
SMI16	0101201	Skip if A Reg Minus 16	CNTRL	8	2-74
SMI20	0101400	Skip if A Reg Minus 20	CNTRL	8	2-74
SPL16	0100201	Skip if A Reg Plus (>=0) 16	CNTRL	8	2-73
SPL20	0100400	Skip if A Reg Plus (>=0) 20	CNTRL	8	2-73
SGT16	0100202	Skip if A Register > 0 16	CNTRL	9/10	2-75
SGT20	0100401	Skip if A Register > 0 20	CNTRL	9/10	2-74
SNZ16	0101200	Skip if A Register Nonzero 16	CNTRL	8	2-73
SNZ20	0101040	Skip if A Register Nonzero 20	CNTRL	8	2-72
SZE16	0100200	Skip if A Register Zero 16	CNTRL	8	2-72
SZE20	0100040	Skip if A Register Zero 20	CNTRL	8	2-72
SLN	0101100	Skip if Low Bit of A Reg Non0	CNTRL	8	2-76
SLZ	0100100	Skip if Low Bit of A Reg Zero	CNTRL	8	2-76
SOC	0100021	Skip if Overflow Clear	CNTRL	9	2-78

Ordered by Type, then Instruction Name

SOS	0101021	Skip if Overflow Set	CNTRL	9	2-77
SRC	0100001	Skip if Reset C Register	CNTRL	10	2-76
SSC	0101001	Skip if Set C Register	CNTRL	10	2-77
JMP	0002000	Unconditional Jump	CNTRL	7 + JA	2-69
SKP	0100000	Unconditional Skip	CNTRL	6	2-71
CAE	0001102	Clear A Extension Bits	CONV	6	2-15
CXE	0001103	Clear X Extension Bits	CONV	7	2-15
EAB	0001041	End Around Borrow	CONV	15	2-18
EAC	0001040	End Around Carry	CONV	15	2-18
XAB	0001201	Extend A and B	CONV	15	2-17
SEA	0001100	Sign Extend A	CONV	8	2-14
SEX	0001101	Sign Extend X	CONV	9	2-14
SEI	0100023	Skip if Extension Insig.	CONV	11/13	2-16
SEN	0101022	Skip if Extension Nonzero	CONV	9	2-16
SES	0101023	Skip if Extension Significant	CONV	11/13	2-16
SEZ	0100022	Skip if Extension Zero	CONV	9	2-16
TAB	0001200	Truncate A and B	CONV	20	2-17
XDV	0000403	Execute Device Function	I/O	48 + I/O	4-21
PDV	0001003	Poke Device Driver	I/O	47 + I/O	4-20
P	0001013	Decrement Semaphore (Probeer)	IPCOM	14	2-115
V	0001023	Increment Semaphore (Verlaat)	IPCOM	15	2-116
LOK	0001011	Obtain Lock	IPCOM	14	2-112
RCV	0001012	Receive	IPCOM	14	2-115
ULK	0001021	Release Lock	IPCOM	15	2-113
SND	0001022	Send Trap	IPCOM	14	2-114
BLT	0000031	Block Transfer Memory	LD/ST	15 + 7n	2-48
CRA	0140040	Clear A Register	LD/ST	6	2-43
CAL	0141050	Clear A Register Left Half	LD/ST	6	2-44
CAR	0141044	Clear A Register Right Half	LD/ST	6	2-44
COB	0140504	Complement Overflow Bit	LD/ST	6	2-47
CLR	0141144	Copy A Register Lt to Rt 1/2	LD/ST	7	2-46
CRL	0141250	Copy Register Rt to Lt Byte	LD/ST	7	2-46
ICL	0141140	Interchange & Clr A (Lt 1/2)	LD/ST	8	2-45
ICR	0141240	Interchange & Clr A (Rt 1/2)	LD/ST	8	2-45
ICA	0141340	Interchange A Register Halves	LD/ST	7	2-44
IAB	0000201	Interchange A and B Registers	LD/ST	9	2-43
IMA	0026000	Interchange Memory & A Reg	LD/ST	11 + MR	2-41
LAI	0141700	Load A Indirectly thru Self	LD/ST	9	2-40
LDA	0004000	Load A Register	LD/ST	6 + MR	2-40
LXA	0141714	Load X Indirectly thru A	LD/ST	13	2-43
LXI	0141704	Load X Indirectly thru Self	LD/ST	13	2-42
LDX	0072000	Load X Register	LD/ST	10 + MR	2-42
RCB	0140200	Reset C Register (Bit)	LD/ST	7	2-46

Ordered by Type, then Instruction Name

ROB	0140604	Reset Overflow Bit	LD/ST	7	2-47
SCB	0140600	Set C Register (Bit)	LD/ST	7	2-46
SOB	0140204	Set Overflow Bit	LD/ST	7	2-47
STA	0010000	Store A Register	LD/ST	10 + EA	2-41
SXA	0141614	Store X Indirectly thru A	LD/ST	13	2-42
STX	0032000	Store X Register	LD/ST	10 + EA	2-41
TRB	0000041	Transfer Memory Backwards	LD/ST	11 + 7n	2-49
ANA	0006000	AND Memory to A Register	LOGIC	6 + MR	2-61
CHS16	0000052	Change (Complement) Sign of A	LOGIC	6	2-64
CHS20	0140024	Change (Complement) Sign of A	LOGIC	6	2-63
CXB	0140510	Complement Indexed Bit	LOGIC	9	2-67
CCRO	0001032	Convert & Clear Rightmost One	LOGIC	10 + f(A)	2-65
CSA16	0000053	Copy Sign & Set A's Sign to +	LOGIC	10	2-65
CSA20	0140320	Copy Sign & Set A's Sign to +	LOGIC	10	2-64
ERA	0012000	Exclusive OR Memory to A Reg	LOGIC	6 + MR	2-61
FFO	0000033	Find First One	LOGIC	9-29	2-65
CMA	0140401	Ones Complement A Register	LOGIC	6	2-62
RXB	0140210	Reset Indexed Bit	LOGIC	9	2-66
SXB	0140610	Set Indexed Bit	LOGIC	9	2-66
SSM16	0000051	Set Sign of A Register Minus	LOGIC	6	2-63
SSM20	0140500	Set Sign of A Register Minus	LOGIC	6	2-63
SSP16	0000050	Set Sign of A Register Plus 16	LOGIC	6	2-62
SSP20	0140100	Set Sign of A Register Plus 20	LOGIC	6	2-62
ERC	0000105	Error Light Clear	PCTRL	14	2-84
HLT	0000000	Halt the Processor	PCTRL	5	2-80
ERR	0000101	Interrogate Memory Errors	PCTRL	23	2-84
MEMHI	0000012	Read Memory High Bound	PCTRL	9	2-81
RDCLOK	0000010	Read RTC Register	PCTRL	11	2-80
RSM	0000013	Read Special Memory	PCTRL	12+f(B)	2-87
ERB	0000200	Retrieve Error Bits	PCTRL	14	2-85
VER	0000100	Return Version Number	PCTRL	9	2-83
WATCH	0000761	Watch On or Off	PCTRL	11/15	2-86
LITES	0000011	Write LIT (LITES) Register	PCTRL	21	2-81
WSM	0000021	Write Special Memory	PCTRL	12+f(B)	2-88
APR	0000003	Activate Process	PRSYS	144	3-28
DPR	0000023	Deactivate Process	PRSYS	148	3-28
ENB	0000401	Enable Other Processes	PRSYS	10/25	3-32
GPR	0000043	Goad Process	PRSYS	3C	3-30
INH	0001001	Inhibit Other Processes	PRSYS	9	3-32
PCB	0000020	Load PCB Into X Register	PRSYS	11	3-37
MMD	0001002	Measurement Mode Disable	PRSYS	6	3-37
MME	0000402	Measurement Mode Enable	PRSYS	6	3-36
NMFS	0000030	NMFS Mode Control	PRSYS	22/281	3-26

SPR	0000103	Suspend Process	PRSYS	163<+++	3-31
TPR	0000203	Timeout Process	PRSYS	22	3-35
UGS	0000113	Unload Self	PRSYS	13	3-33
DEQ	0000022	Dequeue First Item from Queue	QUEUE	23/48	2-95
ENQ	0000002	Enqueue A New Item on Queue	QUEUE	46	2-94
RMQ	0000042	Rmv Specified Item from Queue	QUEUE	56	2-96
ALR16	0041700	A (Register) Left Rotate 16	SHIFT	16<141	2-38
ALR20	0041600	A (Register) Left Rotate 20	SHIFT	16<169	2-36
ARR16	0040700	A (Register) Right Rotate 16	SHIFT	16<140	2-37
ARR20	0040600	A (Register) Right Rotate 20	SHIFT	16<168	2-35
ALS	0041500	Arithmetic Left Shift	SHIFT	16<155	2-33
ARS	0040500	Arithmetic Right Shift	SHIFT	17<133	2-32
LGL	0041400	Logical Left Shift	SHIFT	16<134	2-29
LGR	0040400	Logical Right Shift	SHIFT	16<130	2-28
LLS	0041100	Long Left Arithmetic Shift	SHIFT	28<287	2-33
LLL	0041000	Long Left Logical Shift	SHIFT	17<234	2-30
LLR16	0041300	Long Left Rotate 16	SHIFT	17<254	2-38
LLR20	0041200	Long Left Rotate 20	SHIFT	17<310	2-36
LRS	0040100	Long Right Arithmetic Shift	SHIFT	28<244	2-32
LRL	0040000	Long Right Logical Shift	SHIFT	17<231	2-29
LRR16	0040300	Long Right Rotate 16	SHIFT	17<253	2-37
LRR20	0040200	Long Right Rotate 20	SHIFT	17<309	2-35
CASP	0100011	Copy A Register to SP	STACK	7	2-102
CSPA	0100012	Copy SP to A Register	STACK	7	2-102
CSPX	0100013	Copy SP to X Register	STACK	10	2-103
ITS	0100010	Increment Top of Stack	STACK	13	2-102
LAT	0141510	Load A from Top of Stack	STACK	12	2-104
LXT	0141504	Load X from Top of Stack	STACK	13	2-105
POPA	0101003	Pop A Reg Contents off Stack	STACK	9	2-104
POP	0036000	Pop Memory Contents off Stack	STACK	14 + EA	2-101
PUSHA	0101002	Push A Reg Contents onto Stk	STACK	11	2-103
PUSH	0030000	Push Memory Contents onto Stx	STACK	10 + MR	2-100
SAT	0141610	Store A in Top of Stack	STACK	12	2-103
SXT	0141604	Store X in Top of Stack	STACK	12	2-104
SRETN	0100003	Sub Skip Return Using Stack	STACK	10	2-101
CALL	0034000	Subroutine CALL Using Stack	STACK	11 + JA	2-100
RETN	0100002	Subroutine Return Using Stack	STACK	9	2-101

APPENDIX B Sample Configuration

In this section are included:

1. Two site-specific ASCII configuration (".cnf") files
2. Macros used in assembling these files
3. Makefile used in generating C/30 cassettes

## B.1 Two Site-Specific Configurations (".cnf" files)

```
IMP(86., "backroom 86", 1234., 0, 72., 0., 174000, 12)
HARDWARE(256.,mii, fm.mbn)
M52TRUNK( 1., 2, 1, 8., 2., 0, msync)
C18MII( 0., 1, 0, 0, 4)
X25HOST( 1., 2, 0, 0, 4, 3, msync)
X25HOST( 15., 2, 7, 0, 4, 3, msync)
```

+++++

```
IMP(86., "backroom 86", 1234., 0, 72., 0., 174000, 10.)
HARDWARE(256.,mii, fm.mbn)
CM1TRUNK( 0., 1, 0, 8., 2., 0)
CM1TRUNK( 1., 1, 1, 8., 2., 0)
CM1TRUNK( 2., 1, 2, 8., 2., 0)
CM1TRUNK( 3., 1, 3, 8., 2., 0)
CM1TRUNK( 4., 1, 4, 8., 2., 0)
CM1TRUNK( 5., 1, 5, 8., 2., 0)
C18MII( 2., 1, 0, 0, 4)
HDHHOST( 0., 2, 0, 0, 4, 3, 10, mti)
```

## Macros

## B.2 Macros used in Assembling Configuration Files ("macros\_.mic" file)

```

_divert(-1)
_define(expand,$1)

_define('DEVNO','0')
_define('_newdev','_define('DEVNO',_eval(DEVNO + 1)))')
_define('_label',' '$1'DEVNO'$2:')')
_define('_insert_', 'include($1)')
_define('goget','_insert_(RELEASE/CfgMac/$1)')
_define('ifcall','_ifdef('$1',,'goget($1.mic)')
$1')
_define('IF16','_ifdef('TWENTY','$2','$1)')')
_define('IF20','_ifdef('TWENTY','$1','$2)')')

:This file contains all the macros necessary to configure a NMFS IMP.
:The Makefile automatically appends it to your file

:IMP( imp_number, imp_name, serial_number, flags, noc_imp, noc_host,
:      noc_link, net-id)

:IMP( 76.      , "TESTIP",      443.      , 0 , 72.      , 2      ,
:      174000, 10.)

_define(IMP,';imp number=$1, name=$2, serial=$3, flags=$4, noc=$5,$6,$7
;net number=$8

mainm 3000
fill 177, 0
mainm 3400
exp $1
exp $3
mainm 3404
exp $4
exp $5
exp $6
exp $7
exp $8
mainm 4000
fill 777, 0
)

```

## Macros

```

;SIMP( simp_number, simp_name, serial_number, rate, noc_id1,noc_id2,
;      noc_pro, net_id)
;SIMP( 4. , "Raisting", 443. , 1 , 5002 , 82. ,
;      69. , 4. )
_define(SIMP, ' ;simp number=$1, name=$2, serial=$3, rate=$4, noc=$5,$6,
        $7, net=$8.

        mainm 3000
        fill 377, 0
        mainm 3400
        exp $1
        exp $3
        mainm 3404
        exp $4
        exp $5
        exp $6
        exp $7
        exp $8
')

;HARDWARE(memory amount K words, list-of-board-types, microcode release)
;HARDWARE( 128. , M11 , m20.mbn )

_define(HARDWARE, ' ;Memory = $1 K Words
        loadsyms "RELEASE/$3"
        uram memhi.patch
        _define(mem_size,$1)
        ($1_12) -> 13
        uram .+1
        13 - (2_12) -> 13 ; take 2k from top
        cm1crcaddr = ($1-1)_12 ; put crcs in top 1k
        _define(CRCFLAG)
        BOARDS($2)
')

_define('BOARDS', '
        mii.litad == 360
        mti.litad == 43_6
        msync.litad == 1400
        _board($1,1)
        _board($2,2)
        _board($3,3)
        _board($4,4)

```

```

_board($5,5)
_board($6,6)
_board($7,7)
)

_define('_board', '_dnl
_ifelse($1,,,'
    _define(bd$2_typ, '$1')
    uram board.lit + $2          ; address lights on $1
    g0 + $1.litad -> mar(wio)   ; in io slot $2
')_dnl
)

;CM1TRUNK( n, slot, interface, channels, prop, speed )
;CM1TRUNK( 5, 1, 3, 16, 2, 0 )

_define(CM1TRUNK, '
    _ifdef('CRCFLAG', '
        goget('cm1_crc.mic')
        _undefine('CRCFLAG')
    )
    board = $2
    device = $3
    ifcall('cm1_mii')
    mainm 3000 + 10*$1
    exp 1
    exp $4
    exp $5
    exp 0
    exp ($2_10)!$3
    exp $6
    exp inrb_6 + 1
    exp outrb_6 + 2

    regs inrb ;trunk #$1 device $2/$3, channels=$4, prop=$5, speed=$6
    regs outrb
)

;SM1TRUNK( n, slot, interface, channels, prop, speed )
;SM1TRUNK( 5, 1, 3, 16, 2, 0 )

_define(SM1TRUNK, '
    _ifdef('CRCFLAG', '

```

```

        goget('sm1_crc.mic')
        _undefine('CRCFLAG')
    )
board = $2
device = $3
goget('sm1a_mii.mic')
mainm 3000 + 10*$1
exp 1
exp $4
exp $5
exp 0
exp ($2_10)!$3
exp $6
exp inrb_6 + 1
exp outrb_6 + 2

regs inrb :trunk #$1 device $2/$3, channels=$4, prop=$5, speed=$6
regs outrb
)
;SM1SATDEV(n, slot, interface)
;SM1SATDEV(0, 1, 0)

_define(SM1SATDEV,
    _ifdef('CRCFLAG',
        goget('sm1_crc.mic')
        _undefine('CRCFLAG')
    )
    board = $2
    device = $3
    goget('sm1s_mii.mic')
    mainm 3000 + 10*$1
    exp 2
    exp 0
    exp 0
    exp 0
    exp ($2_10)!$3
    exp 0
    exp inrb_6 + 13.
    exp outrb_6 + 14.

    regs inrb
    regs outrb
)

```

## Macros

```

_define('dev_mti_m52', '7')
_define('dev_msync_m52', '13')

:M52TRUNK( n, slot, interface, channels, prop, speed, boardtype)
:M52TRUNK( 6, 2, 15, 16, 2, 0, msync)

_define(M52TRUNK, '
    board = $2
    device = $3
    ifcall('m52_$7')
    mainm 3000 + 10*$1
    exp 1
    exp $4
    exp $5
    exp 0
    exp ($2_10)!$3
    exp $6
    exp inrb_6 + expand(dev_$7_m52)
    exp outrb_6 + expand(dev_$7_m52) + 1

    regs inrb ;trunk #$1 device $2/$3, channels=$4, prop=$5,
    regs outrb ;          speed=$6, board type=$7
)

_define('C18MII', 'C18HOST($1,$2,$3,$4,$5,mii)')

: one c18 per mti, no device param
_define('C18MTI', 'C18HOST('$1','$2',0,'$3','$4',mti)')

:C18HOST( n, slot, device, haccom, hacmem, boardtype )
:C18HOST( 8, 2, 3, 0, 4, mii )

_define('dev_mii_c18', '3')
_define('dev_mti_c18', '11')

_define(C18HOST, '

    board = $2
    device = $3
    ifcall('c18_$6')

    mainm 4000 + 10*$1

```

## Macros

```

exp 1
exp $4
exp $5
exp 0
exp ($2_10)!$3
exp 0
exp inrb_6 + expand(dev_$6_c18)
exp outrb_6 + expand(dev_$6_c18) + 1

regs inrb ;1822 host #$1, device = $2/$3, haccom=$4, hacmem=$5
regs outrb ; , board type=$6
')

;CM1VDH( n , slot, device, haccom, hacmem, speed )
;CM1VDH( 8., 3 , 2 , 0 , 4 , 3 )

_define(CM1VDH, '
    _ifdef('CRCFLAG', '
        goget('cm1_crc.mic')
        _undefine('CRCFLAG')
    ')

    board = $2
    device = $3
    ifcall('cm1_mii')

    mainm 4000 + 10*$1
    exp 2
    exp $4
    exp $5
    exp 0
    exp ($2_10)!$3
    exp $6
    exp inrb_6 + 1
    exp outrb_6 + 2

    regs inrb ;vdh host #$1, device=$2/$3, haccom=$4, hacmem=$5, speed=$6
    regs outrb
')

;HDHHOST(n, slot, device, haccom, hacmem, spd, frame size indx, boardtype)
;HDHHOST(8., 2 , 3 , 0 , 4 , 3 , 10 , msync )

```

## Macros

```

_define(HDHOST, '
    board = $2
    device = $3
    ifcall('m52_$8')

    mainm 4000 + 10*$1
    exp 3
    exp $4
    exp $5
    exp 0
    exp ($2_10)!$3
    exp ($7_10)!$6
    exp inrb_6 + expand(dev_$8_m52)
    exp outrb_6 + expand(dev_$8_m52) + 1

    regs inrb ;HDH host #$1, device = $2/$3, haccom=$4, hacmem=$5
    regs outrb ;           , speed=$6, frame size indx=$7, board type=$8
)

:X25HOST( n , slot, device, haccom, hacmem , speed , boardtype )
:X25HOST( 8., 2 , 2 , 0 , 4 , 3 , msync )

_define(X25HOST, '
    board = $2
    device = $3
    ifcall('m52_$7')

    mainm 4000 + 10*$1
    exp 4
    exp $4
    exp $5
    exp 0
    exp ($2_10)!$3
    exp $6
    exp inrb_6 + expand(dev_$7_m52)
    exp outrb_6 + expand(dev_$7_m52) + 1

    regs inrb ;X25 host #$1, device = $2/$3, haccom=$4, hacmem=$5
    regs outrb ;           , speed=$6, board type=$7
)
_divert(0)

```

## Makefile

## B.3 Makefile Used in Generating C/30E Cassettes

```

C30=/usr/c30sys
BIN=$(C30)/bin
MIC=$(C30)/microcode/m7u13/Release_0
CFG=$(MIC)/CfgMac

MACROS=$(CFG)/macros_.mic $(CFG)/m52_mti.mic $(CFG)/m52_msync.mic \
        $(CFG)/cm1_mii.mic $(CFG)/cm1_crc.mic $(CFG)/c18_mii.mic \
        $(CFG)/c18_mti.mic $(CFG)/mii_lite.mic $(CFG)/msync_lite.mic \
        $(CFG)/mti_lite.mic

MA=$(BIN)/ma
MBBCASS=$(BIN)/mbbcass

MICROCODE=$(MIC)/m20.mbn
XUCODE=$(MIC)/x.mbn
FULLUCODE=$(MIC)/fm.mbn
SATUCODE=$(MIC)/sm.mbn

MAC=$(C30)/macrocode/imp4360
PASSWD=$(MAC)/password.mbn
LOADER=$(MAC)/lod.mbn
IMP=$(MAC)/imp.mbn

default: ; echo No default. ; exit 1

        $(CFG)/config_mic $* $(MIC)
        $(BIN)/ma $*.mic -o $*.mbn -l $*.lst && tab $*.lst

*.itape:      $(IMP)
*.ttape:      $(IMP)

41.mbn: $(FULLUCODE) $(MACROS)

```

APPENDIX C Sample Program and Makefile

C.1 Sample Program: NMFS Profiling Package

```
.stitle  
.title "NMFS Profiling Package"  
.wide  
  
.comment
```

NMFS Profiling Package

Table of Contents

- \* description
- \* edit history
- \* prologue
- \* definitions
- \* startup
- \* initialization
- \* timeout
- \* routines
- \* constants and variables

```
.endcomment
```

## .stitt Description

```
;*****  
;* Description *  
;*****
```

```
.comment  
OVERVIEW
```

This package provides a means for monitoring locations in the IMP. The user can choose to monitor the program counter, the value in a memory location, the length of a non-NMFS queue, or the PCB or priority of the current process.

The profiling process takes periodic samples of the selected variable. If the sample falls within a specified range of values, it is recorded as a "hit" in a 512 bin array of counters. The range is partitioned equally among the bins of the array. On request, the bin counts are printed out, together with the percentage which each count represents of the total number of hits.

Optional process and priority qualifiers are available. The process qualifier restricts hits to the process with the selected PCB; similarly, the priority qualifier restricts hits to processes running at the selected priority level.

The user-supplied parameters are:

## Lower and upper bounds-

These two parameters describe the range within which the specified variable is to be monitored. Only samples which fall within the bounds will be counted as "hits". Since the amount of profile array storage is fixed, this range is collapsed into n-word units (where n is a power of 2) in order to fit all hits into the array. Thus, any hit within a unit increments that element of the array.

## Frequency-

Governs the rate of which samples are taken, in units of 1.6 ms.

**Hit limit-**

Controls the number of hits to be recorded. If nonzero, the sampling stops after that number of hits (NOT samples.) If zero, the sampling continues until explicitly stopped by the user.

**Process qualifier-**

This parameter, if nonzero, further qualifies valid hits to processes with a specified PCB or at a particular NMFS priority level. Otherwise, hits are valid for all processes.

A priority level is always less than 32., whereas a PCB address is always greater than 32. This allows the package to distinguish between the two meanings of this parameter.

**Type-**

Determines the type of monitoring desired, as follows:

- 1 monitor PC
- 2 monitor selected memory location
- 3 monitor length of non-NMFS queue
- 4 monitor PCB (process control block) address
- 5 monitor process priority

**Start/stop flag-**

Used to turn sampling on and off. Sampling begins when the flag is set to the value prof.init, and stops at the next sample period after the flag is set to the value prof.off.

**Hook-**

Intended for use by developers who wish to hook their own patches into the profile process. The location prof.hook contains the address of a routine which is called every time that the profile package wakes up to take a sample.

The routine should return at an offset from the point at which it was called, as follows:

- +1 => no hit
- +2 => hit
- +3 => no effect

If the routine declares a hit, it should return the hit value in A.

The results produced by the program are:

Array unit size-

Granularity of the profile array storage with respect to sample values, expressed as the exponent of the power of 2 unit size. For example, 0 means each array element represents one sample value; 3 means each element represents eight values. In the latter case, the first element of the array would count any hit in the range between "lower bound" and "lower bound"+7, the second element any hit in the next 8 values, etc.

Sample count-

Double precision count of all samples taken since the profiling run was started.

Hit count-

Double precision count of the number of hits since the profiling run was started.

Overflow count-

Double precision count of the number of bin overflows since the profiling run was started. The sum of all bins, added to the overflow count, should be equal to the sample count.

Profile array-

Array of single precision counts representing hits within the corresponding units.

## OPERATION

To use the package, first load it into a selected IMP using the standard NU command "load":

```
load nXX -b prof.pkg
```

where "XX" is the number of the IMP selected.

The package is controlled by Eric Rosen's "profile" utility, which runs under NU. Following is a description of "profile", adapted from an online help file which he wrote. The current version of the online help file can be obtained by typing "profile \?\?" on the appropriate NOC machine.

"Profile" can be used to set the parameters of the package, to turn it on or off, and to read the data gathered by the package, printing it in a reasonably readable form. Functions are controlled via command line switches. The default function (if no switches are given) is to read and reduce the profiling data. Command line format is:

```
profile [IMPs] [switches]
```

where the IMPs can be specified by name, number, or alias. For example:

```
profile n63 -stop -read
```

will stop profiling on IMP 63 and cause the data to be gathered and printed on the terminal.

IMPs will be handled one at a time in the order in which their identifiers appear. The environment must have MonPrefix set appropriately.

This program can be terminated at any time with "kill" (send signal 15, the default), or with the interrupt character if it is running in the foreground. Hang-up signal is ignored if output is not to a terminal.

To get the documentation for a particular switch only, place "-\?switch" on the command line. To get a list of switches without the accompanying documentation, type "-\?". To force the program to terminate after outputting documentation, double the question mark. (This is useful if the program is being invoked only to produce the documentation, particularly if the documentation is being sent to a file or pipe.)

The program will ignore any incorrect switch settings, and will prompt for corrections, if necessary. After outputting documentation, it will prompt for additional command line arguments. (Note that command line arguments given on the same line as a documentation request will be processed in the ordinary fashion and will take effect when the program executes.)

To give a switch, just put it on the command line PRECEDED by

a dash, as in the example above. No spaces are permitted between the dash and the switch name. To assign a value to a switch, follow the switch with an equals sign (=) and then the value. (E.g., "-low=4000" sets the lower bound on the sampling range to 4000.) Spaces around the equal sign are optional. Switches may be identified by unique initial substrings of their names, as well as by particular nicknames which are given below on the same line as the switch names.

Numerical values assigned to switches will be interpreted in decimal unless the first digit in the number is a zero, in which case the number is assumed to be octal.

Here is the list of acceptable switches:

low:

(Specify numeric value) Value to store in prof.low, the low bound on the sampling range.

high:

(Specify numeric value) Value to store in prof.hi, the high bound on the sampling range.

frequency:

(Specify numeric value) Value to store in prof.freq, the sampling frequency.

limit:

(Specify numeric value) Value to store in prof.lim, the hit limit.

location:

(Specify numeric value) Value to store in prof.loc, the monitored location.

qualifier:

(Specify numeric value) Value to store in prof.qual, the optional PCB or priority qualifier.

type:

(Specify numeric value) Value to store in prof.type, the type of monitoring.

- pc: (No value allowed) Monitor the program counter. Equivalent to setting prof.type to 1.
- var: (No value allowed) Monitor the value of a variable. Equivalent to setting prof.type to 2.
- queue: (No value allowed) Monitor the length of a non-NMFS queue. Equivalent to setting prof.type to 3.
- pcb: (No value allowed) Monitor the process PCB. Equivalent to setting prof.type to 4.
- pri: (No value allowed) Monitor the process priority. Equivalent to setting prof.type to 5.
- start: (No value allowed) Set package parameters and turn on the package.
- stop: (No value allowed) Turn off the package.
- read: (No value allowed) Read and reduce package data. This is automatically performed if neither the start nor stop switches are given.
- preserve: (No value allowed) If this switch is given along with the start switch, then any parameters not explicitly mentioned by the user will retain their old values when the package is turned on. Otherwise, the parameters not assigned values by the user will be set to default values.
- gateway: (Specify string value) Monitor offnet entity.

## control:

(Specify numeric value) Specify the location of a pointer to the profile control area. This switch is used to override the default pointer location.

## strip:

(No value allowed) Produce concise, numeric output. This is useful when the profile data is to be passed to another piece of software rather than to a human user.

Following are some command line examples:

```
profile n41 -pc -low=014000 -high=014777 -freq=4 -start -limit=1000
```

```
profile n41 -type=2 -loc=0303 -low=0 -high=30 -freq=4 -start
```

```
profile n41 -pc -qual=046315 -low=0 -high=0173777 -freq=4 -start
```

```
profile n41 n46
```

The first example is PC monitoring, with hits confined to the range 14000 to 14777. The package will stop itself when 1000 hits have been recorded. In the second example, variable monitoring, memory location 303 will be watched. A PCB qualifier has been specified in the third example; hits will be restricted to samples taken when the selected process was active. The last example causes data to be collected from IMPs 41 and 46.

Note that the package should be stopped before data is read, because data collected when the package is running will be inaccurate.

## EXAMPLE

Following is an example of output from an actual profile run. Profiling began with

```
profile n41 -sta -pc -qual=15 -low=0 -high=0173777 -freq=47
```

was terminated with

```
profile n41 -stop
```

and the data was read using

profile n41 -read

The package monitored the program counter, and counted as hits all PC values which corresponded to a process at priority level 15 (decimal). This happens to be background's priority level. Since the experiment was done on an idle IMP, it is not surprising that most of the samples were hits.

.endcomment

\*\*\*\*\*

profile: 2/9/84 20:40

For documentation, type "profile \?\?"

Using prefix ".bbn-imp."

41. TEST1 (TEST1)

Thu Feb 9 21:25:07 1984

Parameters:

Low = 000000

High = 173777

Frequency = 057 (every 75.20 ms.)

Limit = 0, Qualifier = 17, Loc = 0, Type = PC

Package is OFF

000600-000777:	9. (.73%)	012000-012177:	2. (.16%)
013600-013777:	87. (7.03%)	014000-014177:	298. (24.07%)
014200-014377:	553. (44.67%)	014600-014777:	12. (.97%)
015000-015177:	76. (6.14%)	015200-015377:	14. (1.13%)
016600-016777:	11. (.89%)	023000-023177:	4. (.32%)
023200-023377:	11. (.89%)	025400-025577:	9. (.73%)
026000-026177:	104. (8.40%)	026200-026377:	2. (.16%)
027200-027377:	34. (2.75%)	062200-062377:	12. (.97%)

Total hits = 1238.

Total overflows = 0.

Total samples = 1339.

\*\*\*\*\*

"Overflows" means bin wraparound events, i.e., the number of times that a hit was scored for any bin which had already reached 177777 (octal). If the overflow count is nonzero, then it is best to simply discard the data.

Below is the same data, but in the stripped format. Numeric output is useful if you are interested in doing further processing on the profile output. (If not, skip the rest of this section.) After the standard header, the first numeric line contains eight parameters: low bound, high bound, frequency, hit limit, location, qualifier, monitoring type, and start/stop flag. The parameters are printed in octal, in eight digit fields (%8o, for those of you who know C.)

The 512. bins come next, in five digit decimal fields. (I have suppressed most of them to save space.) Last are four decimal parameters, field width ten: bin totals, total hits, total overflows, and total samples. Total hits should equal the bin totals plus total overflows.

```
*****
profile: 2/9/84 20:40
For documentation, type "profile \?\?"
Using prefix ".bbn-imp."
```

41. TEST1 (TEST1)			Thu Feb 9 21:25:40 1984		
0 173777	57	0	0	17	1 0
0					
0					
0					
...					
0					
0					
0					
1238					
1238					
0					
1339					

\*\*\*\*\*

## IMPLEMENTATION

This package is implemented as a stand-alone process executing at priority level 0. It is important that application processes not be run at level 0, so that sampling can be accurate and not influenced by the execution patterns of application processes.

When this process is goaded, it initializes itself using the supplied parameters. It then times itself for the prescribed period.

Upon waking up, the process samples the value of the variable on which monitoring is being performed. This value is checked against lower and upper bounds on the range, and optionally against the PCB or priority of the interrupted process. If there is a hit, the hit count and profile array element are incremented. The sample count is always incremented. If a hit limit is specified and that hit limit has been exceeded, the process suspends itself permanently. Otherwise, the process suspends itself to time out until the next sample.

The package usually needs to figure out which process it interrupted. There are two different ways to do this, depending on which version of the microcode is running. Version m7u12 saves the PCB of the interrupted process in a particular location in micromemory, which profile can read using an RSM instruction.

Version m7u11 provides no such convenient hook. To find out which process was last active, profile searches the NMFS pending queues starting with priority 1 and ascending until it finds a nonempty queue. The first PCB on this queue presumably corresponds to the previously active process.

There is a serious hole in this scheme. Namely, it is possible for a higher priority process to time out and enter a pending queue while the profile process is scanning the queues. IMP version 4320 runs on m7u11 microcode, and later IMP versions run on m7u12 or later versions of the microcode. The user is therefore forewarned that the 4320 version of the profile package is not guaranteed to produce accurate results.

If the process is goaded while it is suspended and timing, it will go through the initialization phase (prfini) again and wipe out any previously accumulated results.

.endcomment

.stittl Edit History

\*\*\*\*\*  
;\* Edit History \*  
\*\*\*\*\*

.comment  
Edit history, 1984

15-feb. wg

Converted package to 20 bit assembler.

09-feb. wg

Eliminated backward compatible support for the 4320 version, in order to get rid of some clutter and take advantage of the extended instruction set. It was convenient until now to have two identical (almost) sources for both versions.

07-feb. wg

Moved package control area from page 0 into the package itself. The package will now use a single location on page 0, prof.control, to point to the control area. Aside from saving page 0 space, this also allows for expansion of the control area in future development.

Added protection against NU errors in the case of package shutdown. If NU asks us to shut down twice, the flag prsflg tells us to ignore the second shutdown request.

Edit history, 1983

22-sep. wg

Fixed bug in package shutdown procedure. The package needs to remove the profile PCB from the idle queue after dpr'ing the process.

09-sep, wg

Implemented overflow detection. If an array bin wraps around to zero, the bin will be set back to -1. All such hits will also be recorded in a double precision overflow counter.

Developers can now hook their own patches into the profile process. The instruction "call prof.hook i" will be executed every time a sample is taken.

Gave the profile process an eight word stack and replaced all jst instructions with calls.

06-sep, wg

Added PCB and priority monitoring.

05-aug, wg

Modified the package to allow user control via the 'profile' utility, NU software created by Eric Rosen.

11-jul, wg

Fixed bug in package shutdown procedure. The package will now shut itself down if the shutdown control word has a value of '1'. Previously, any nonzero value would cause shutdown.

Edit history, 1982

06-dec, sae

Added memory location and non-NMFS queue length monitoring.

30-nov, sae

Fitted the package to standard NMFS IMP package format.

BBN Report No. 5000

Sample Program and Makefile  
Sample Program

03-nov, pjs

Debugged the package by including it in unused space within the X.25 L3 module, running with IMP 4310.

24-sep, pjs

Wrote the package in order to provide a useful software engineering tool and to have a sample application-context-free program for BBN Report 5000, Appendix C.

.endcomment

## .stitl Prologue

```

*****
;* Prologue *
*****
;
; Package loading prologue (see /usr/c30sys/doc/packages.doc)
;
: get 4360 symbols
      .insym /usr/c30sys/macrocode/imp4363/imp/imp4363.sym
; include(macrodefs)
      : get macro definitions

      0000060      . = bittab+14
0000060 0010000      pbit14      ;package bit

      0120302      . = pcwd14
0120302 0000000      0      ;shutdown control word
;
; package unhooks
;
      0120202      . = ini14.hook
0120202 0055000      callretn      ;init unhook

      0120242      . = bck14.hook
0120242 0055000      callretn      ;background unhook

      0000005      . = zero
0000005 0000000      0      ;busy flag
;
; buffer stealing
;
      0000063      buf1 = 63      ;first buffer to steal
      0000012      bufn = 12      ;(octal) number of
      0000010      bufd = 10      ;buffers to steal
      ;start at this displace-
      ;ment into 1st buffer

      0243004      prfbuf = buf0 + (buf1 - 1) * buf1
      ;start of 1st buffer

```

Sample Program and Makefile  
Sample Program

```

0243014      prf000 = prfbuf + bufd
0244470      prf777 = prfbuf + (bufn * buf1)
                                ;start of package
                                ;first free loc after
                                ;package

0055450      . = bufrq2
0055450 0244470      buf0 + (buf1 - 1) * buf1 + (bufn * buf1)
0055447 0243004      . = bufrq1
                                buf0 + (buf1 - 1) * buf1
0055450 0055450      . = bufrq2
                                0
0055450 0000000      . = bufrq1
                                0
0055447 0000000      . = bufrq1
                                0

0120342      . = packvers14
0120342 0000020      20                                ;package version
    
```

## .stilt Definitions

```

*****#
: * Definitions *
: *****#

:
: NOC parameters
:
:     0000001      ncc.shutstart=1          ;NOC shutdown request
:     0000002      ncc.shutdown=2         ;signal shutdown complete
:
: NMFS parameters
:
:     0100000      idlstate=100000        ;value- idle state
:     0000037      primsk=32.-1.          ;mask- priority level
:
: As of ucode version m7u12, the PCB of an interrupted process
: is saved in umemory at location 17R. These parameters tell
: the findpr routine how to find this information.
:
:     0000012      uvers=12                ;slick ucode version
:     0000004      memtyp=4                 ;umemory type for rsm
:     0000017      memadd=17                ;umemory address for rsm
:
: for queue monitoring. limit the maximum queue depth to which we
: will scan
:
:     0000050      qlenmax=50
:
:     0001000      arraysize=1000          ;size of profile array
:
:
: Values of prof.go flag. Prof.off requests the profiling process to
: stop, and prof.init requests the process to start. After starting
: up, the process writes the value prof.on into the flag.
:
:     0000000      prof.off=0
:     0000001      prof.init=1
:     0000002      prof.on=2

```

```
; ascii values for .lev directive. Profile runs at priority 0.
;
      0000060      pro.lev=60      ;profiling process
;                               ; LEV value
;                               ;(60 is ascii "0")
      .stit! Startup
```

```
;*****
;* Startup *
;*****
```

```
.comment
```

```
Init section
```

This section is called from the IMP's initialization code. Initialization is done by a call from the IMP background loop. This code merely gets the attention of that routine by setting a flag indicating that an initialization is needed.

```
.endcomment
```

```
      .lev bck
      .lck all
      0243014      . = prf000      ;beginning of code

0243014 0140040 9 0 pri0:   cra
0243015 0111021 9 0      sta [priflg] i      ;signal init needed
0243016 0111022 9 0      sta [prsflg] i      ;clear shutdown flag
0243017 0111023 9 0      sta [prof.go] i      ;clear start/stop flag
0243020 0100002 9 0      retn

0243021 0243422 9 0 .constants
0243022 0243423 9 0
0243023 0243357 9 0
```

```
.comment
```

## Background section

This section is called every background loop from the IMP's background code, and as such it executes at background's priority level. Prb will check the "init-needed" flag (priflg), and if 0, will perform initialization. Priflg is set to 0 in one of two ways: either from the package image itself when the package is loaded, or by the init section after an IMP restart.

Note that there are two flavors of initialization. When the package is loaded or the IMP restarts, prb calls prfapr to create the profile process and set up a few pointers. In addition, each time a new measurement run is initiated, the profile process initializes its variables at prfini.

This section also monitors the package control word. If the value of this word is equal to ncc.shutstart, the package will be shut down, i.e., the profiling process will be deactivated (dpr'ed). After finishing, the shutdown routine prshut writes ncc.shutdone into the control word. Prshut is smart enough so that if NU asks the package to shut down twice, the second request will be ignored.

If neither initialization nor shutdown is required, this section checks the start/stop flag. If the value of the flag is prof.init, the profile process is goaded.

.endcomment

```

        .lev bck
0243024 0005422 9   prb0:  lda priflg           ;initialization needed?
0243025 0100040 9           size20
0243026 0003031 9           jmp prb1             ;no, continue
0243027 0035235 9           call prfapr          ;yes, activate profile
                                ;   process
0243030 0003045 9           jmp prbret          ;and exit

0243031 0105334 9   prb1:  lda [pcwd14] i       ;check shutdown control
                                ;   word
0243032 0012044 9           era [pcwd.shutstart]
                                ;shutdown requested?
0243033 0100040 9           size20
0243034 0003037 9           jmp prb2             ;no, continue
0243035 0035262 9           call prshut          ;yes, do it

```

```
0243036 0003045 9          jmp prbret          ;and exit
0243037 0005357 9  prb2:  lda prof.go          ;check start flag
0243040 0012044 9          era [prof.init]      ;startup requested?
0243041 0100040 9          size20
0243042 0003045 9          jmp prbret          ;no, so exit
0243043 0073335 9          idx [profpcb]      ;profile PCB address
0243044 0000043 9          gpr          ;goad the process
0243045 0100002 9  prbret: retn          ;and return to background
```

## .stidl Initialization

```

*****
;* Initialization *
*****

;
; This section of code is executed whenever the process is goaded.
; Start by clearing the result variables, including the array.
;
; .lev pro.lev

0243046 0140040 0   prfini: cra           ;clear result variables
0243047 0011360 0           sta prof.unitsize ;array unit size
;
; Samples, hits, and overflows are all double precision counters,
; with the low order bits in the first word.
;
0243050 0011361 0           sta prof.samples ;sample count
0243051 0011362 0           sta prof.samples+1
0243052 0011363 0           sta prof.hits ;hit count
0243053 0011364 0           sta prof.hits+1
0243054 0011365 0           sta prof.ovfl ;overflow count
0243055 0011366 0           sta prof.ovfl+1
;
; clear the profile array, using an IMP utility routine
;
0243056 0004055 0           lda [arraysize] ;size of area
0243057 0073336 0           idx [array] ;beginning of area
0243060 0135337 0           call [clear_area] i
;clear the area
;
; Copy upper and lower bounds in prof.low and prof.hi to proflow and
; profhi, respectively. This guards against accidental alteration of
; the variables by the user during a profile run, which could cause
; the IMP to crash.
;
0243061 0005350 0           lda prof.low ;low bound 0243062
0011424 0           sta proflow
0243063 0005351 0           lda prof.hi ;high bound
0243064 0011425 0           sta profhi
;
;

```

```

; Compute the range between the upper and lower bound qualifiers, and
; then determine the unit size by comparing the range with the array
; size

```

```

;
; Note: "shyft" is used because the normal spelling is a reserved word
; for the m4 preprocessor, even when used in comments.
;

```

```

0243065 0005425 0      lda profhi
0243066 0017424 0      sub proflow      ;result is 20-bit
                                ;absolute range
0243067 0141206 0      aoa      ;bounds are included
0243070 0101400 0      smi20     ;(in case range
                                ;    > 19 bits)

                                unitloop:
0243071 0022055 0      cas [arraysize] ;will current
                                ;unit size fit?
0243072 0003075 0      jmp unitincr  ;range > array size,
                                ; inc unit size
0243073 0101000 0      nop      ;range = array size,
                                ; or
0243074 0003101 0      jmp setshyft ;range < array size,
                                ; go set shyft

                                unitincr:
0243075 0141206 0      aoa      ;round up in next
                                ;instruction

0243076 0040477 0      lgr 1      ;cut range in half
0243077 0025360 0      irs prof.unitsize ;and double unit size
0243100 0003071 0      jmp unitloop ;try again

```

```

;
; save the shyft count for later use
;

```

```

0243101 0005360 0      setshyft:lda prof.unitsize;get shyft count
                                ; = exponent of 2
0243102 0140407 0      tca      ;two's complement
                                ; as required
0243103 0011426 0      sta profshyft ;use later for
                                ; array indexing

```

```

;
; set up hit limit and initial timeout
;

```

```
0243104 0005353 0      lda prof.lim      ;make hit limit a
                        ; negative count
0243105 0140407 0      tca                ;(or leave it 0)
0243106 0011427 0      sta countdown
0243107 0005352 0      lda prof.freq      ;number of 1.6 ms.
                        ; ticks to wait
0243110 0000203 0      tpr                ;for timeout starting now
;
; signal initialization complete
;
0243111 0004045 0      lda [prof.on]     ;package is running
0243112 0011357 0      sta prof.go      ;start/stop flag
;
; fall into timeout loop on the next page
;
```

## .stilt Timeout

```

;*****
;* Timeout *
;*****
;
; This is the top of the timeout loop. We get here by either falling
; through after initialization (prfini), or by jumping back at the
; end of processing a sample.
;
; The code first suspends, waiting for the timeout. Upon waking up,
; it increments the sample count, and checks the value of the
; variable that
;
; the user wants to monitor. If the value matches the hit criteria,
; the hit count is incremented, and checked against the hit limit to
; see if it is time to quit.
;
;
; .lev pro.lev
;
; suspend execution and wait for the timeout
;
0243113 0000103 0    timloop: spr
0243114 0100000 0                skp                ;timeout resumed the process
0243115 0003046 0                jmp prfini        ;we were goaded,
; re-initialize
;
; In order to keep to the prescribed frequency, we must renew the
; timeout right away, provided that the start/stop flag still
; contains the value prof.on. Other possible values are prof.off,
; indicating a shutdown request, or prof.init, indicating a restart
; request. In both of those cases, we go to sleep. Background will
; poke us if this is a restart request.
;
0243116 0005357 0                lda prof.go        ;check start/stop flag
0243117 0012045 0                era [prof.on]     ;keep running?
0243120 0100040 0                size20
0243121 0003113 0                jmp timloop        ;no, suspend
0243122 0005352 0                lda prof.freq     ;non-zero, renew timeout
0243123 0000203 0                tpr                ;starting now
;
; increment sample count

```

```

;
0243124 0025361 0      irs prof.samples ;increment low order bits
0243125 0100000 0      skp              ;no overflow
0243126 0025362 0      irs prof.samples+1
                                ;overflow, inc.
                                ; high order bits
0243127 0101000 0      nop              ;(in case of 40-bit
                                ; overflow)
;
; Call the "hook" routine. This is provided as a facility for IMP
; programmers who wish to hook their own patches into the profile
; process.
;
; The routine should return at an offset from the point at which it
; was called, as follows:
;
;      +1      => no hit
;      +2      => hit
;      +3      => no effect
;
; If the routine declares a hit, the hit value should be returned in
; A.
;
0243130 0135370 0      call prof.hook i
0243131 0003113 0      jmp timloop      ;no hit, suspend
0243132 0003152 0      jmp chkrng       ;score the hit
;
; Find the process that would be running if we were not. As of ucode
; version 12, the PCB address of the interrupted process is stored in
; location 17R, and can be extracted using the RSM instruction (read
; special memory).
;
; RSM requires the memory type (ucode register memory, in this case)
; to be in B, and the memory address to be in X; the value is
; returned in A.
;
0243133 0004046 0      lda [memtyp]     ;ucode register memory
0243134 0000201 0      iab              ;put it into B
0243135 0072070 0      ldx [memadd]     ;location of PCB address
0243136 0000013 0      rsm              ;A ← PCB address
0243137 0010000 0      sta 0            ;transfer to X

```

```

:
: Check process qualifier, if one was specified. This routine expects
: the qualifier in A and the PCB in X, and preserves only X.
:
0243140 0005355 0      lda prof.qual
0243141 0101040 0      snz20                ;qualifier was specified
0243142 0003145 0      jmp noqual          ;no, continue
0243143 0035206 0      call chkqual
0243144 0003113 0      jmp timloop          ;no hit, suspend
:
: select type of monitoring
:
0243145 0005356 0      noqual: lda prof.type
0243146 0015340 0      add [profdisp-1] ;dispatch table
0243147 0141700 0      lai
0243150 0010001 0      sta ireg          ;dispatch address
0243151 0134001 0      call ireg i        ;do it
:
: code converges again here for range check of sampled value
:
0243152 0010001 0      chkrng: sta ireg          ;save the value
0243153 0017424 0      sub proflow        ;convert to absolute
                                ; displacement
0243154 0100021 0      soc                ;no overflow => OK
0243155 0003113 0      jmp timloop          ;fails the check,
                                ; no hit, suspend
0243156 0005425 0      lda profhi          ;check against high bound
0243157 0016001 0      sub ireg          ;subtract sample value
0243160 0100021 0      soc                ;no overflow => OK
0243161 0003113 0      jmp timloop          ;fails the check,
                                ; no hit, suspend
:
: Convert sampled value to array index. First subtract the lower
: bound, then scale by shyfting right, using a shyft count computed
: at init time. The array index is used to locate the appropriate
: bin in which to record the hit.
:
0243162 0004001 0      lda ireg          ;sample value
0243163 0017424 0      sub proflow        ;subtract lower bound
0243164 0073426 0      ldx profshyft      ;load shyft count
0243165 0040400 0      lgr 0              ;scale the value
0243166 0010000 0      sta 0              ;use result as index
0243167 0065430 0      irs array x        ;count unit hit

```

```

0243170 0100000 0      skp                ;no overflow
0243171 0035226 0      call ovflow          ;record the overflow
0243172 0025363 0      irs prof.hits        ;count total hits
0243173 0100000 0      skp                ;no overflow
0243174 0025364 0      irs prof.hits+1    ;overflow, inc.
                                ; high order bits
0243175 0101000 0      nop                ;(in case of 40-bit
                                ; overflow)
;
; if the user specified a hit limit, check to see if we have reached
; it
;
0243176 0005427 0      lda countdown
0243177 0101040 0      snz20                ;are we free-running?
0243200 0003113 0      jmp timloop         ;yes, suspend
0243201 0025427 0      irs countdown      ;no, count down hit limit
0243202 0003113 0      jmp timloop         ;not at limit, suspend
;
; We have reached the hit limit. Set the start/stop flag to stop
; profiling after the next timeout.
;
0243203 0140040 0      cra 0&prof.off
0243204 0011357 0      sta prof.go          ;start/stop flag
0243205 0003113 0      jmp timloop         ;suspend
0243113 0000103 0      timloop: spr
0243114 0100000 0      skp                ;timeout resumed
                                ; the process
0243115 0003046 0      jmp prfini          ;we were goaded,
                                ; re-initialize
;
; In order to keep to the prescribed frequency, we must renew the
; timeout right away, provided that the start/stop flag still contains
; the value prof.on. Other possible values are prof.off, indicating a
; shutdown request, or prof.init, indicating a restart request. In
; both of those cases, we go to sleep. Background will poke us if
; this is a restart request.
;
0243116 0005357 0      lda prof.go          ;check start/stop.flag
0243117 0012045 0      era [prof.on]        ;keep running?
0243120 0100040 0      sze20
0243121 0003113 0      jmp timloop         ;no, suspend
0243122 0005352 0      lda prof.freq       ;non-zero, renew timeout

```

```

0243123 0000203 0          tpr          ;starting now
:
; increment sample count
:
0243124 0025361 0          irs prof.samples ;increment low order bits
0243125 0100000 0          skp          ;no overflow
0243126 0025362 0          irs prof.samples+1 ;overflow, inc.
:                               ; high order bits
0243127 0101000 0          nop          ;(in case of 40-bit
:                               ; overflow)
:
; Call the "hook" routine. This is provided as a facility for IMP
; programmers who wish to hook their own patches into the profile
; process.
:
; The routine should return at an offset from the point at which it
; was called, as follows:
:
:       +1      => no hit
:       +2      => hit
:       +3      => no effect
:
; If the routine declares a hit, the hit value should be returned in
; A.
:
0243130 0135370 0          call prof.hook i
0243131 0003113 0          jmp timloop      ;no hit, suspend
0243132 0003152 0          jmp chkrng      ;score the hit
:
; Find the process that would be running if we were not. As of ucode
; version 12, the PCB address of the interrupted process is stored in
; location 17R, and can be extracted using the RSM instruction (read
; special memory). RSM requires the memory type (ucode register
; memory, in this case) to be in B, and the memory address to be in X
; the value is returned in A.
:
0243133 0004046 0          lda [memtyp]      ;ucode register memory
0243134 0000201 0          iab          ;put it into B
0243135 0072070 0          idx [memadd]      ;location of PCB address
0243136 0000013 0          rsm          ;A ← PCB address
0243137 0010000 0          sta 0          ;transfer to X

```

```

:
: Check process qualifier, if one was specified. This routine expects
: the qualifier in A and the PCB in X, and preserves only X.
:
0243140 0005355 0      lda prof.qual
0243141 0101040 0      snz20                ;qualifier was specified
0243142 0003145 0      jmp noqual          ;no, continue
0243143 0035206 0      call chkqual
0243144 0003113 0      jmp timloop          ;no hit, suspend
:
: select type of monitoring
:
0243145 0005356 0      noqual: lda prof.type
0243146 0015340 0      add [profdisp-1] ;dispatch table
0243147 0141700 0      lai
0243150 0010001 0      sta ireg          ;dispatch address
0243151 0134001 0      call ireg i        ;do it
:
: code converges again here for range check of sampled value
:
0243152 0010001 0      chkrng: sta ireg          ;save the value
0243153 0017424 0      sub proflow        ;convert to absolute
                                ; displacement
0243154 0100021 0      soc                ;no overflow => OK
0243155 0003113 0      jmp timloop          ;fails the check,
                                ; no hit, suspend
0243156 0005425 0      lda profhi          ;check against high bound
0243157 0016001 0      sub ireg          ;subtract sample value
0243160 0100021 0      soc                ;no overflow => OK
0243161 0003113 0      jmp timloop          ;fails the check,
                                ; no hit, suspend
:
: Convert sampled value to array index. First subtract the lower
: bound, then scale by shyfting right, using a shyft count computed at
: init time. The array index is used to locate the appropriate bin in
: which to record the hit.
:
0243162 0004001 0      lda ireg          ;sample value
0243163 0017424 0      sub proflow        ;subtract lower bound
0243164 0073426 0      ldx profshyft      ;load shyft count
0243165 0040400 0      lgr 0            ;scale the value
0243166 0010000 0      sta 0            ;use result as index

```

```

0243167 0065430 0      irs array x      ;count unit hit
0243170 0100000 0      skp                ;no overflow
0243171 0035226 0      call ovflow        ;record the overflow
0243172 0025363 0      irs prof.hits      ;count total hits
0243173 0100000 0      skp                ;no overflow
0243174 0025364 0      irs prof.hits+1 ;overflow, inc.
                                ; high order bits
0243175 0101000 0      nop                ;(in case of 40-bit
                                ; overflow)
:
; if the user specified a hit limit, check to see if we have reached it
:
0243176 0005427 0      lda countdown
0243177 0101040 0      snz20                ;are we free-running?
0243200 0003113 0      jmp timloop         ;yes, suspend
0243201 0025427 0      irs countdown      ;no, count down hit limit
0243202 0003113 0      jmp timloop         ;not at limit, suspend
:
; We have reached the hit limit. Set the start/stop flag to stop
; profiling after the next timeout.
:
0243203 0140040 0      cra 0&prof.off
0243204 0011357 0      sta prof.go        ;start/stop flag
0243205 0003113 0      jmp timloop         ;suspend

```

## .sttl Routines

```

:*****
:*  Routines  *
:*****
:
: CHKQUAL
:
: Check process qualifier, if specified. Skip on return if the
: qualifier matches or if no qualifier was specified.
:
: The qualifier can be either a PCB or priority level. We can
: distinguish the two by comparing the qualifier with primsk, the
: maximum priority level. A PCB will be always be larger than primsk,
: and a priority level will be less than or equal to primsk.
:
: Arguments:  qualifier in A, PCB in X
:
0243206 0016051 0   chkqual:sub [primsk+1]   ;PCB or priority qualifier?
0243207 0101021 0           sos             ;<= max priority
0243210 0003217 0           jmp pcbqual    ;> max priority
:                                     ; => PCB qualifier
:
: check priority level of interrupted process against qualifier
:
0243211 0044003 0           lda pcb.pri x   ;priority of
:                                     ; interrupted process
0243212 0007341 0           ana [primsk]   ;mask out state bits
0243213 0013355 0           era prof.qual  ;does priority match?
0243214 0100040 0           sz20          ;yes
0243215 0003223 0           jmp chkbad    ;no, failure
0243216 0003222 0           jmp chkgood   ;match
:
: check PCB of interrupted process against qualifier
:
0243217 0005355 0   pcbqual:lda prof.qual  ;specified PCB
0243220 0012000 0           era 0          ;does PCB match?
0243221 0101040 0           snz20        ;no, failure
0243222 0100010 0   chkgood: its         ;skip on return
0243223 0100002 0   chkbad: retn        ;return
:

```

```

; NOPHOOK
;
; Developers have the capability of hooking their own patches into the
; profile process, using the location prof.hook. Every time a sample
; is taken, the instruction "call prof.hook i" is executed. This
; dummy routine is provided as the default.
;
; The routine should return at an offset from the point at which it
; was called, as follows:
;
;      +1      => no hit
;      +2      => hit
;      +3      => no effect
;
0243224 0100010 0    nophook:its
0243225 0100003 0                sretn                ;return with no effect
;
; OVFLOW
;
; Array bin overflowed. Record the overflow in ovcnt, and reset the
; bin to -1.
;
; Arguments:  array index in X
;
0243226 0004017 0    ovflow: lda [-1]
0243227 0051430 0                sta array x                ;reset the bin to -1
0243230 0025365 0                irs prof.ovfl                ;count the overflow
0243231 0100000 0                skp
0243232 0025366 0                irs prof.ovfl+1 ;overflow counter
; overflowed
0243233 0101000 0                nop                ;ignore 32-bit overflow
0243234 0100002 0                retn
;
; PRFAPR
;
; Create and activate profile process.
;
; Check to see that the package does not exceed the space allocated
; for it. Prf777 is the first location following the stolen buffers;
; prfend is the first free location after the end of the package.
;

```

```

; This test should really be done at assembly time, but C30ASM
; unfortunately provides no convenient way to do an unsigned
; comparison.
;
0243235 0005342 0   prfapr: lda [prf777]           ;check the assembly
0243236 0017343 0           sub [prfend]         ;does package end
                                           ; before buffers do?
0243237 0100021 0           soc                   ;yes, OK
0243240 0000000 0           hlt                   ;no, so crash -
                                           ; not enough space
;
0243241 0005336 0           lda [array]
0243242 0011367 0           sta prof.array        ;init pointer to array
0243243 0005344 0           lda [nophook]
0243244 0011370 0           sta prof.hook        ;init hook to dummy routine
;
; set up a pointer to the profile control area, used by the profile
; utility
;
0243245 0005345 0           lda [prof.low]
0243246 0010117 0           sta prof.control     ;reserved pg 0 loc
;
; set up profile process
;
0243247 0004146 0           lda idlqi           ;A -> idle queue header
0243250 0073335 0           idx [profpcb]       ;X -> profile PCB
0243251 0000002 0           enq                 ;put profile PCB
                                           ; on idle queue
0243252 0101000 0           nop                 ;(don't care if
                                           ; queue was empty)
0243253 0004063 0           lda [idlstate]      ;state <- idle,
                                           ; priority <- 0
0243254 0050003 0           sta pcb.pri x      ;in profiling PCB
0243255 0005346 0           lda [profstk]      ;put stack address
0243256 0050013 0           sta pcb.sp x      ;into PCB
0243257 0000003 0           apr                 ;activate the
                                           ; profiling process
;
0243260 0025422 0           ; irs priflg       ;signal init done
0243261 0100002 0           ; retn            ;and return to caller
;
; PRSHUT

```

```

; Profile shutdown. Deactivate the profile process.
;
; Before doing anything, we check prsflg to see if NU has mistakenly
; asked us to shut down twice, which can happen if a package unload
; is interrupted.
;
0243262 0005423 0    prshut: lda prsflg          ;shutdown flag
0243263 0100040 0          sze20              ;have we shut down already?
0243264 0003273 0          jmp prsret        ;yes, don't do it again
;
; Clear pointer to control area, so that the NU utility cannot
; accidentally write into active buffers after the package is unloaded.
;
0243265 0010117 0          sta prof.control
;
0243266 0073335 0          idx [profpcb]      ;address of profile PCB
0243267 0000023 0          dpr                ;deactivate the
; profile process,
0243270 0000042 0          rmq                ;and remove it
; from the idle queue
0243271 0101000 0          nop                ;(don't care if
; queue becomes empty)
;
; Signal shutdown complete, both in the package control word which NU
; looks at, and in the internal flag prsflg.
;
0243272 0025423 0          irs prsflg        ;internal flag
0243273 0004045 0    prsret: lda [pcwd.shutdone] ;shutdown signal
0243274 0111334 0          sta [pcwd14] i    ;package control word
0243275 0100002 0          retn              ;return to caller
;
; Monitoring Routines. Each routine returns the sampled value in A.
;
;
; PC monitoring
;
0243276 0044006 0    pcmon: lda pcb.pc x      ;program counter
0243277 0100002 0          retn

```

```

;
; Variable monitoring
;
0243300 0105354 0   varmon: lda prof.loc i   ;prof.loc points
                                ; to location
0243301 0100002 0           retn

;
; Queue length monitoring
;
; The parameter qlenmax imposes a limit on the queue depth to which we
; will scan. This prevents the profiling process from consuming
; enormous amounts of time scanning a very long queue, or worse yet,
; scanning a looped queue in an infinite loop.
;
0243302 0140040 0   qmon:   cra
0243303 0010002 0           sta jreg           ;init queue length counter
0243304 0105354 0           lda prof.loc i 0&q.forw
                                ;get first item
0243305 0101040 0           snz20
0243306 0100002 0           retn             ;zero => empty queue
0243307 0024002 0   qmonlp: irs jreg        ;not at end yet.
                                ; so increment queue length
0243310 0010001 0           sta ireg        ;save queue pointer
0243311 0004002 0           lda jreg        ;load queue length
0243312 0023347 0           cas [qlenmax]   ;have we reached
                                ; limit on scan depth?
0243313 0100002 0           retn           ;yes
0243314 0100002 0           retn           ;yes
0243315 0104001 0           lda ireg i 0&q.forw
                                ;no, advance to
                                ; next item on queue
0243316 0100040 0           sze20          ;at end of queue
0243317 0003307 0           jmp qmonlp     ;not yet, continue
0243320 0004002 0           lda jreg      ;load queue length
0243321 0100002 0           retn

;
; PCB monitoring
;
0243322 0004000 0   pcbmon: lda 0         ;get PCB address
0243323 0100002 0           retn

```

```
;
; Priority monitoring
;
0243324 0044003 0   primon: lda pcb.pri x   ;priority of
;
0243325 0007341 0           ana [primsk]   ; interrupted process
0243326 0100002 0           retn          ;mask out state bits
```

.stitl Constants and Variables

```

;*****
;*  Constants and Variables  *
;*****
    
```

.lev con

```

;
; Dispatch table for routines to handle different monitoring types.
;
    
```

profdisp:

```

0243327 0243276 C      pcmon          ;program counter
                                ; monitoring
0243330 0243300 C      varmon         ;variable monitoring
0243331 0243302 C      qmon          ;non-NMFS queue
                                ; monitoring
0243332 0243322 C      pcbmon        ;PCB address
                                ; monitoring
0243333 0243324 C      primon        ;process priority
                                ; monitoring
    
```

```

0243334 0120302 C      .constants
0243335 0243371 C
0243336 0243430 C
0243337 0055542 C
0243340 0243326 C
0243341 0000037 C
0243342 0244470 C
0243343 0244430 C
0243344 0243224 C
0243345 0243350 C
0243346 0243421 C
0243347 0000050 C
:confix(-3)
    
```

.lev var

```

;
; Package control area
;
    
```

; control variables

```

;
0243350          V      prof.low:      .block 1 ;lower bound qualifier
    
```

```

0243351      V  prof.hi:      .block 1  ;upper bound qualifier
0243352      V  prof.freq:    .block 1  ;sampling frequency
0243353      V  prof.lim:     .block 1  ;hit limit
0243354      V  prof.loc:     .block 1  ;location to be
                                ; monitored
0243355      V  prof.qual:    .block 1  ;optional PCB or
                                ; priority qualifier
0243356      V  prof.type:    .block 1  ;type of monitoring
                                ; wanted
0243357 0000000 V  prof.go:     .block 1,0
                                ;start/stop flag

; computed results
;
0243360      V  prof.unitsize: .block 1      ;exponentzef array unit
;
; The sample count, hit count, and overflow count are all double
; precision. In each case, the first word contains the lower order
; bits, and the second word contains the high order bits.
;
0243361      V  prof.samples: .block 1  ;sample count
0243362      V  prof.hits:    .block 1  ;hit count
0243363      V  prof.ovfl:    .block 1  ;overflow count
0243364      V  prof.array:   .block 1  ;pointer to profile
                                ; array
0243370      V  prof.hook:    .block 1  ;hook for experimental
                                ; patches

;
; Profile process PCB. The process type (0) and initial PC (prfini)
; are initialized at assembly time, since they will remain valid even
; if the application resets itself via a NMFS instruction.
;
0243371      V  profpcb:      .block 5
0243376 0000000 V  profpcb:    .block 1,0      ;non-I/O process type
0243377 0243045 V  profpcb:    .block 1,prfini-1 ;PC (GPR will increment it)
0243400      V  profpcb:    .block 10

0243421      V  profstk=+.7   .block 10      ;address of profile stack
0243412      V  profstk=+.7   .block 10      ;(grows upward)
;

```

```

: miscellaneous variables
:
0243422 0000000 V   priflg:  .block 1,0       ;initialization flag
0243423 0000000 V   prsflg:  .block 1,0       ;shutdown flag
0243424          V   proflow:  .block 1       ;copy of prof.low
0243425          V   profhi:  .block 1       ;copy of prof.hi
0243426          V   profshyft:.block 1      ;shyft count for
: array indexing
0243427          V   countdown:.block 1      ;negative hit limit counter

0243430          V   array:  .block arraysize ;profile array

          0244430   prfend = .               ;prfend marks end of
: used buffer space,                          ;beginning of patch space

:
: standard package hooks
:
          .lev con
          . = bck14.hook                      ;background
0120242 0243024 C   prb0
          0120202   . = ini14.hook           ;init
0120202 0243014 C   pri0
  
```

## Cross Reference

.	120203	PRF000	243014
ALL	60	PRF777	244470
ARRAY	243430	PRFAPR	243235
ARRAYSIZE	1000	PRFBUF	243004
BCK	71	PRFEND	244430
BCK14.HOOK	120242	PRFINI	243046
BITTAB	44	PRI0	243014
BUFO	233000	PRI FLG	243422
BUF1	63	PRIMON	243324
BUFD	10	PRIMSK	37
BUFL	122	PRO.LEV	60
BUFN	12	PROF.ARRAY	243367
BUFRQ1	55447	PROF.CONTROL	117
BUFRQ2	55450	PROF.FREQ	243352
CALLRETN	55000	PROF.GO	243357
CHKBAD	243223	PROF.HI	243351
CHKGOOD	243222	PROF.HITS	243363
CHKQUAL	243206	PROF.HOOK	243370
CHKRNG	243152	PROF.INIT	1
CLEAR_AREA	55542	PROF.LIM	243353
CON	103	PROF.LOC	243354
CONCNT	0	PROF.LOW	243350
COUNTDOWN	243427	PROF.OFF	0
DOT	243350	PROF.ON	2
IDLQI	146	PROF.OVFL	243365
IDLSTATE	100000	PROF.QUAL	243355
INI14.HOOK	120202	PROF.SAMPLES	243361
I REG	1	PROF.TYPE	243356
J REG	2	PROF.UNITSIZ	243360
MEMADD	17	PROFDISP	243327
MEMTYP	4	PROFHI	243425
NCC.SHUTDOWN	2	PROFLOW	243424
NCC.SHUTSTAR	1	PROFPCB	243371
NOPHOOK	243224	PROFSHYFT	243426
NOQUAL	243145	PROFSTK	243421
OVFLOW	243226	PRSFLG	243423
PACKVERS14	120342	PRSHUT	243262
PBIT14	10000	PRSRET	243273
PCB.PC	6	Q.FORW	0
PCB.PRI	3	QLENMAX	50
PCB.SP	13	QMON	243302

BBN Report No. 5000

Sample Program and Makefile  
Sample Program

PCBMON	243322	QMONLP	243307
PCBQUAL	243217	SETSHYFT	243101
PCMON	243276	TIMLOOP	243113
PCWD . SHUTDON	2	UNCON	243350
PCWD . SHUTSTA	1	UNITINCR	243075
PCWD14	120302	UNITLOOP	243071
PRB0	243024	UVERS	12
PRB1	243031	VAR	126
PRB2	243037	VARMON	243300
PRBRET	243045	ZERO	5

## Cross Reference

	.	12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 14(9,0), 28(C), 30(C), 30(C):	28(C), 29(V), 29(V)
ALL	-		14(9)
ARRAY	29(V):		17(0), 21(0), 24(0), 25(0)
ARRAYSIZE	13:		17(0), 17(0), 29(V)
BCK	-		14, 15(9,0)
BCK14.HOOK	-		12, 30(C)
BITTAB	-		12
BUFO	-		12, 12, 12
BUF1	12:		12, 12, 12
BUFD	12:		12
BUFL	-		12, 12, 12, 12, 12
BUFN	12:		12, 12
BUFRQ1	-		12, 12
BUFRQ2	-		12, 12
CALLRETN	-		12, 12
CHKBAD	23(0):		23(0)
CHKGOOD	23(0):		23(0)
CHKQUAL	23(0):		21(0)
CHKRNG	21(0):		21(0)
CLEAR_AREA	-		17(0)
CON	-		28(0), 30(V)
CONCNT	28(C):		28(C)
COUNTDOWN	29(V):		19(0), 22(0), 22(0)
DOT	28(C):		28(C), 28(C)
IDLQI	-		25(0)
IDLSTATE	13:		25(0)
INI14.HOOK	-		12, 30(C)
I REG	-		21(0), 21(0), 21(0), 21(0), 21(0), 27(0), 27(0)
J REG	-		27(0), 27(0), 27(0), 27(0)
MEMADD	13:		21(0)
MEMTYP	13:		21(0)
NCC.SHUTDOWN	13		
NCC.SHUTSTAR	13		
NOPHOOK	24(0):		25(0)
NOQUAL	21(0):		21(0)
OVFLOW	24(0):		22(0)
PACKVERS14	-		12
PBIT14	-		12
PCB.PC	-		27(0)
PCB.PRI	-		23(0), 25(0), 27(0)

PCB.SP	-	25(0)
PCBMON	27(0):	28(C)
PCBQUAL	23(0):	23(0)
PCMON	27(0):	28(C)
PCMD.SHUTDOWN	-	26(0)
PCMD.SHUTSTA	-	16(9)
PCMD14	-	12, 16(9), 26(0)
PRB0	16(9):	30(C)
PRB1	16(9):	16(9)
PRB2	16(9):	16(9)
PRBRET	16(9):	16(9), 16(9), 16(9)
PRF000	12:	14(9,0)
PRF777	12:	25(0)
PRFAPR	25(0):	16(9)
PRFBUF	12:	12, 12

## Cross Reference

PRFEND	29(V):	25(0)
PRFINI	17(0):	20(0), 29(V)
PRI0	14(9,0):	30(C)
PRIFLG	29(V):	14(9,0), 16(9), 25(0)
PRIMON	27(0):	28(C)
PRIMSK	13:	23(0), 23(0), 27(0)
PRO.LEV	13:	17(9), 20(0)
PROF.ARRAY	29(V):	25(0)
PROF.CONTROL	-	25(0), 26(0)
PROF.FREQ	28(V):	19(0), 20(0)
PROF.GO	28(V):	14(9,0), 16(9), 19(0), 20(0), 22(0)
PROF.HI	28(V):	17(0)
PROF.HITS	28(V):	17(0), 17(0), 22(0), 22(0)
PROF.HOOK	29(V):	21(0), 25(0)
PROF.INIT	13:	16(9)
PROF.LIM	28(V):	19(0)
PROF.LOC	28(V):	27(0), 27(0)
PROF.LOW	28(V):	17(0), 25(0)
PROF.OFF	13:	22(0)
PROF.ON	13:	19(0), 20(0)
PROF.OVFL	29(V):	17(0), 17(0), 24(0), 24(0)
PROF.QUAL	28(V):	21(0), 23(0), 23(0)
PROF.SAMPLES	28(V):	17(0), 17(0), 20(0), 20(0)
PROF.TYPE	28(V):	21(0)
PROF.UNITSIZ	28(V):	17(0), 18(0), 19(0)
PROFDISP	28(C):	21(0)
PROFHI	29(V):	17(0), 17(0), 21(0)
PROFLOW	29(V):	17(0), 17(0), 21(0), 21(0)
PROFPCB	29(V):	16(9), 25(0), 26(0)
PROFSHYFT	29(V):	19(0), 21(0)
PROFSTK	29(V):	25(0)
PRSFLG	29(V):	14(9,0), 26(0), 26(0)
PRSHUT	26(0):	16(9)
PRSRET	26(0):	26(0)
Q.FORW	-	27(0), 27(0)
QLENMAX	13:	27(0)
QMON	27(0):	28(C)
QMONLP	27(0):	27(0)
SETSHYFT	19(0):	17(0)
TIMLOOP	20(0):	20(0), 21(0), 21(0), 21(0), 21(0), 22(0), 22(0), 22(0)
UNCON	-	28(C)

UNITINCR	17(0):	17(0)
UNITLOOP	17(0):	18(0)
UVERS	13	
VAR	-	28(C)
VARMON	27(0):	28(C)
ZERO	-	12
[0000001]	-	16,16
[0000002]	-	19,20,26
[0000004]	-	21
[0000017]	-	21
[0000037]	23:	27
[0000040]	-	23
[0000050]	27	
[0001000]	-	17,17
[0055542]	17	
[0100000]	-	25

Cross Reference

[0120302]	16:	26
[0243224]	25	
[0243326]	21	
[0243350]	25	
[0243357]	14	
[0243371]	16:	25, 26
[0243421]	25	
[0243422]	14	
[0243423]	14	
[0243430]	17:	25
[0244430]	25	
[0244470]	25	
[3777777]	-	24

BBN Report No. 5000

Sample Program and Makefile  
Sample Program

Assembly Statistics

Assembled: Tue Jul 10 15:41:25 1984

0:48 run time in 2:44

No Errors

## C.2 Makefile

```
.SUFFIXES:      .tmp .mbn .bin .c30 .lst .sym .pl30 .diff  
.IGNORE:
```

```
PREV=/usr/c30sys/macrocode/imp4340
```

```
imp.bin:        imp.m4 edithistory \  
                def.m4 pg0.m4 ini.m4 m2i.m4 i2m.m4 \  
                h2i.m4 i2h.m4 bck.m4 tsk.m4 zud.m4 \  
                tim.m4 fak.m4 utl.m4 spf.m4 dpy.m4 \  
                heap.m4 noc.m4  
  
nmfsdefs:      cm1defs c18defs m52defs  
  
lod.bin:       lod.m4 \  
                asm20 *.m4 -o *.bin -l *.lst -pl 60  
  
stats.bin:     stats.m4 imp4360.sym  
                pl30 *.m4 -ob *.bin -l *.lst -pl 60  
  
eestats.bin:   eestats.m4 imp4360.sym  
                asm20 *.m4 -o *.bin -l *.lst -pl 60  
  
sfstats.bin:   sfstats.m4 imp4360.sym  
                pl30 *.m4 -ob *.bin -l *.lst -pl 60  
  
htm.bin:       htm.m4 imp4360.sym  
                pl30 *.m4 -ob *.bin -l *.lst -pl 60  
  
sapval.bin:    sap.m4 sapval.m4 imp4360.sym  
  
sapgen.bin:    sap.m4 sapgen.m4 imp4360.sym  
  
sapbch.bin:    sapbch.m4 imp4360.sym  
                asm20 *.m4 -o *.bin -l *.lst -pl 60  
  
prof.bin:     prof.m4 imp4360.sym  
                asm20 *.m4 -o *.bin -l *.lst -pl 60  
  
prfbch.bin:   prfbch.m4 imp4360.sym  
                asm20 *.m4 -o *.bin -l *.lst -pl 60
```

## Makefile

```

caswrite.bin:      caswrite.m4
                   pl30 *.m4 -ob *.bin -l *.lst -pl 60

tfhost.bin:       tfhost.m4 imp4360.sym
                   asm20 *.m4 -o *.bin -l *.lst -pl 60

logtables.bin:    logtables.m4 imp4360.sym
                   asm20 *.m4 -o *.bin -l *.lst -pl 60

imp.m4:           macrodefs imp.c30 edithistory
def.m4:           macrodefs def.c30 nmfsdefs
pg0.m4:          macrodefs pg0.c30
noc.m4:          macrodefs noc.c30
ini.m4:          macrodefs ini.c30
m2i.m4:          macrodefs m2i.c30
i2m.m4:          macrodefs i2m.c30
h2i.m4:          macrodefs h2i.c30
i2h.m4:          macrodefs i2h.c30
bck.m4:          macrodefs bck.c30
tsk.m4:          macrodefs tsk.c30
zud.m4:          macrodefs zud.c30
tim.m4:          macrodefs tim.c30
fak.m4:          macrodefs fak.c30
utl.m4:          macrodefs utl.c30
dpy.m4:          macrodefs dpy.c30
spf.m4:          macrodefs spf.c30
heap.m4:         macrodefs heap.c30

lod.m4:          lod.c30
stats.m4:        pmacrodefs stats.pl30
eestats.m4:      macrodefs eestats.c30
sfstats.m4:      pmacrodefs sfstats.pl30
htm.m4:          pmacrodefs htm.pl30
sap.m4:          macrodefs sap.c30
prof.m4:         macrodefs prof.c30
tfhost.m4:       macrodefs tfhost.c30
logtables.m4:    macrodefs logtables.c30

imp.err:         imp.lst
imp.lst:         imp.bin

.c30.m4:         ; /usr/c30sys/bin/al *.c30 ; tab *.c30 ; \
                  m4 *.c30 > *.m4

```

```
.pl30.m4:      ; /usr/c30sys/bin/al *.pl30 ; tab *.pl30 ; \  
                m4 *.pl30 > *.m4  
  
.m4.bin:      ; asm20 *.m4 -o *.bin -l *.lst -e *.err \  
                -f *.sections -f *.traps -f *.locs -f *.crashes \  
                -pl 60  
                sort -n *.sections -o *.sections  
  
.bin.mbn:     ; /usr/c30sys/bin/c20bin *.bin -o *.mbn  
  
.c30.diff:    ; diff -b -c1 $(PREV)/*.c30 *.c30 > *.diff
```

## APPENDIX D. Firmware Crashes

NOTE: The crash code is saved in microregister 25 before a trap or halt.

- 1      HALT. The macrocode executed a HLT instruction, opcode 000000. Look at register 0 of RB.MAIN (LO.PC) to determine the address of the halt. Register L6.CURR in RB.MAIN has the current PCB address, if that helps.
- 2      Instruction not implemented yet. The program attempted to execute an opcode which is defined, but not implemented (e.g., MVQ). Debug it as if it executed an illegal instruction or a halt (i.e., look at LO.PC in RB.MAIN, etc.).
- 3      Watch hit. The user has executed a W command and the associated memory value has changed.
- 4      Illegal instruction. The macrocode executed an unassigned opcode. Debug just like traps #1 or #2, above.
- 5      Illegal stack pointer. The macrocode attempted to access a stack element and SP was 0.
- 6      JST, JMP, or CALL to location 0. The macrocode did a JST, JMP, or CALL which would have gotten it to location 0. In NMFS, this instruction is treated as an illegal instruction with LO.PC, etc. being preserved. Debug it like any other illegal instruction.
- 7      Memory too small. The assembled machine configuration on the cassette defined more memory than that actually available on the machine in use. Edit and reassemble the configuration and rewrite the cassette.
- 10     Bad Special Memory. The value in the B register on execution of an RSM or WSM instruction contained a value outside the range 0 to 5.



- some macromemory bogusness caused by a failing macro program).
- 21 Illegal back pointer. Just like trap #20, different bad word.
  - 22 Illegal header pointer. Just like trap #20, different bad word.
  - 23 Illegal length. Similar to #20-#22, but it means that the length in the queue header did not agree with the contents of the queue. You should proceed through the procedure for #20, but this type of problem is generally more subtle, having to do with macrocode which either smashes the queue's length word or plays very tricky games with the contents without using ENQ/DEQ/RMQ, etc.
  - 24-27 Should never occur.
  - 30 PCB at zero. The macrocode tried to manipulate a PCB which is supposedly at location 000000. Find the offending instruction by tracking down the current macro PC (see trap #1, etc.), then look at the A (L3.A in RB.MAIN) and X (L5.X in RB.MAIN) registers to see what is going on.
  - 31 Should never occur.
  - 32 XDV of idle processes. Macrocode tried to XDV a process which was idle (not APR'ed).
  - 33 RMQ.IDLE, not idle state. Someone (almost certainly the macrocode in this case) tried to remove a PCB from the idle heap (presumably the only time this is done is during an APR) but the PCB did not have the idle bit set in its state word.
  - 34 RMQ.IDLE, not idle queue. Similar to #33. The PCB claims to be in the idle state (has the idle bit set in the state word) but its queue header pointer does not seem to point to the idle queue. Clearly someone is confused.

- 35,36 RMQ.READY, not ready state/queue. Like #33/#34 except referring to the ready queue, not the idle queue. #33 and #34 are almost always caused directly by macrocode instructions. #35 and #36 can also happen as a result of some NMFS manipulations at microcode level. RMQ.READY is the routine to remove a PCB from the READY queue (this occurs whenever a process is GOADed, etc.). The first thing you should do is find out what PCB is in trouble. See L6.PCB in RB.IOOS. Then look at its state word and header pointer. You should see the trouble.
- 37,40 RMQ.TIMING, not timing state/queue. Same as #35/#36, but timing.
- 41,42 RMQ.PENDING, not pending state/queue. Same as #35/#36, but pending.
- 43 RMQ.PENDING, did not see attention bit set. This means that the machine took some PCB off the pending queue and noticed that it was the last (the pending queue is now empty). Since the pending queue went from non-empty to empty, NMFS turned off the "attention bit" associated with that priority level, but found it was already off (implying that the PCB should not have been on the queue in the first place). The attention bits (which live in L10.ATTN1 and L11.ATTN2 of RB.IOOS) are set for each priority that has at least one process which wants to run. This helps NMFS find the highest priority process to run when it needs to context switch. This is almost always caused by the macrocode DEQ or RMQ'ing PCBs from a pending queue (clearly this is illegal).
- 44 ENQ.PENDING, did not see attention bit clear. Similar to #43 but means that when the first PCB was queued on an otherwise empty pending queue, the corresponding attention bit was already set indicating that it should not have been empty. Usually caused by the macrocode ENQ'ing onto a pending queue directly.
- 45 Pending queue was empty but attention bit was set. NMFS went to schedule a process associated with the highest-order attention bit but found no process at that priority. Probably macrocode DEQ'ed or RMQ'ed the

- process off the pending queue which left the attention bit set.
- 46 Illegal PCB.TYPE. The PCB in question (see L6.PCB in RB.IOOS) had an illegal type. Type 0 is software, others are chosen from the same list as in-type and out-type given in section 4.1.1.
- 47 Nothing to run. NMFS could find no processes to run (the attention bits were all zero). Macrocode must always supply at least one runnable process (hopefully the lowest priority one).
- 50,51 Should never occur.
- 52 Empty idle queue at start-up. The macrocode did an NMFS instruction, setting up the addresses of the idle, ready, timing, and pending queues. NMFS went to automatically APR and GPR the first process on the idle queue for you (to get things rolling) but found nothing there. Fix your program.
- 53 PCB.RB messed up. When you create (APR) a process which has a non-zero type, you are creating an I/O process. The PCB.RB field of your PCB determines what register block the microcode should use to do I/O for that device. 1600(8), for example, may correspond to modem #0. The microcode device driver for the device records the address of the associated PCB in one of these registers. Whenever you reference the device (XDV for example), the device driver makes sure your PCB matches the one it was bound to by NMFS. This trap results if that comparison fails. The macrocode has probably stepped on PCB.RB after APR'ing the process. The current PCB is probably the culprit. BASE (as saved in LO.BASE of RB.STATE) is the register block that the device driver was called with (will match PCB.RB of the PCB).
- 54,55 Bad XDV args. These codes are returned when an XDV is done with illegal argument(s). For example, if you try to do function number 100 on a device which supports functions 0-5, you'll get one of these. Find the bogus XDV at the current PC and look at L3.A and L5.X in

RB.MAIN for details.

- 56 Illegal IOCB.SIZE. An IOCB had an illegal transfer size word. Most commonly this is a value of 0, or a value that is not a multiple of the I/O byte size. Find the PCB which owns the IOCB in L6.PCB of RB.IOOS (not necessarily the current one!) and then look at PCB.IOCB to find the address of the offending IOCB.
- 57 Transfer count went negative. This should never happen. It is a serious microcode bug which says that a device driver output or input more than the IOCB said to.
- 60 Illegal IOCB address. The microcode device driver saw an illegal IOCB address (probably 0). Find the current PCB at L6.PCB in RB.IOOS, find the current IOCB at PCB.IOCB, etc.
- 61 uDD attempted to GET second IOCB ("uDD" = microcode device driver). The microcode attempted to set up more than one current IOCB. Find the current PCB at L6.PCB in RB.IOOS, then locate the address of the old IOCB in PCB.IOCB. This trap is given from IODD.GET in NMFS (see IO.MIC). Look at L7.DDUPC in RB.IOOS to find out the microcode address of whoever called IODD.GET. This can also be caused by the macrocode setting PCB.IOCB non-zero itself.
- 62 uDD attempted to PUT non-existent IOCB. Similar to trap #61, above. Track down the confused microcode device driver through L7.DDUPC, etc.
- 63 uDD could not shake IOCB. Serious microcode bug indicating the failure of IODD.PUT to get rid of the current IOCB in use by the microcode device driver.
- 64 Illegal sequence of APRs. Some devices insist on having one direction be in existence before they will let you APR the other direction. Generally, the input side comes first (for those devices which care). Find the offending APR at the current PC.
- 65 Attempt to SPR while INHibited. The macrocode attempted

- to debreak from the current process (SPR) but left the machine INHibited. This is clearly illegal (it makes no sense, either). Find the offending SPR at the current PC.
- 66 uDD saw unexpected interrupt. This is a serious microcode bug which says that the microcode got an interrupt from a device which should not be interrupting; you can generally identify which device from the saved BASE in LO.BASE of RB.STATE. If possible, you will want to look at the hardware I/O registers of that device from the MBB console (using the "I" command) -- find yourself a microcode guru.
- 67 This cannot be done with NMFS off. Find the offending instruction at the current PC.
- 70 Illegal to DPR yourself. A process tried to DPR itself at the current PC.
- 71 The macrocode attempted to APR the wrong side of a device first. Some microcode drivers require that one side or the other of the nmfs device be initialized first.
- 73 A device interrupt was recieved by the microcode for a device which is not in the current configuration. The upc of the interrupt vector is stored in microregister 34, and can be found in a dump at crash.state + 14. Bits 9-7 contain the board number, and bits 6-2 the particular vector location (out of a possible 32 available to each board). Usually, bits 6-3 contains the device number; however, since this trap is probably an io board interrupt logic failure, this cannot be counted on. \*

## APPENDIX E Firmware Programming

Here we present, in detail, the inner workings of the C/30E NMFS firmware. The reader is assumed to be familiar with the facilities provided by NMFS and should have working knowledge of C/30E macro and microprogramming. A source listing of the NMFS firmware would be most helpful in following the information presented here.

Some of the lower level details in this appendix may have changed since the most current update, but the material should still serve as a good guide to the reader interested in the functioning of the NMFS microcode.

In this section, words in CAPITALS refer to specific NMFS firmware symbol names (labels, equates, etc.). One may choose to locate these symbols in the source listing as appropriate.

## Layout

## E.1 Layout

The NMFS firmware is made up of seven basic components:

## USYS

Contains all the hardware control and maintenance functions as well as the basic console command interpreter. Overlaid by the instruction emulation and "console" portions to extend basic USYS functionality to include C/30E NMFS-specific features.

## Instruction Emulation/Dispatch

Contains most of the C/30E instruction set emulation and corresponding C/30E "dispatch memory" assignments.

## Console

Augments the basic USYS console command interpreter to include NMFS-specific commands like "A" (examine emulated A register), etc.

## Crash

Contains the firmware necessary to update the crash area when a program trap is detected.

## Process Manager (PM)

Contains the "brain" of the NMFS process management and (unfortunately) some of the I/O firmware.

## I/O Management Library (IO)

Contains mostly subroutines callable by PM and by the various device drivers. These subroutines are generally concerned with moving IOCBs through NMFS and scheduling the associated processes as necessary.

## Microcode Device Drivers (DDs)

Actually an arbitrary number of modules, each responsible for interfacing some device to the rest of the NMFS firmware and ultimately to the macrocode application.

These components are presented in greater detail below.

## USYS

## E.2 USYS

USYS provides many hardware-related functions for the MBB-based computers, including the C/30E. In many respects, USYS is the "operating system" for all MBB microcode applications, therefore, USYS is strictly application-independent. USYS actually comprises UPROM (in the file `usys-prom.mic`), which is the PROM (non-writable) portion occupying the low 512 words of the microcode address spaces, and URAM (`usys-uram.mic`), which occupies some amount of low URAM space. As with all NMFS microcode files, `usys-uram.mic` leaves the symbol UNEXT equal to the address of the next free microcode word. This address is a function primarily of the number of I/O boards for which I/O vector space is reserved, and secondarily of any changes which may have been made to `usys-uram.mic`. `Usys-uram.mic` assembles a small amount of microcode into the address range 20100-20177, which is unused vector space for board zero. It then assembles beginning at the initial value of UNEXT, which is how the NMFS assembly reserves space for I/O vectors. To reserve space for two boards, UNEXT is set to 20600, and to reserve space for four boards, UNEXT is set to 21200. `Usys-uram.mic` then assembles about 200 additional instructions, and leaves UNEXT pointing just

after itself.

Some of the functions USYS provides are:

- Power-up reset,
- Boot cassette reading,
- Macromemory dynamic memory refreshing,
- Console command processing, and
- Button, Clock, MBM LED control.

The NMFS microprogrammer may be served best by thinking of USYS as a "given" -- a basically unchanging base upon which to build. It is highly unlikely (and undesirable) to modify USYS code itself since it is currently common to many MBB applications (C/30 EXA, C/30E NMFS, C/50 CMOS, C/60 UNIX, C/70 UNIX). For this reason, we will not present USYS's internal structure in detail.

There are a number of "hooks" which the rest of NMFS places into USYS in order to perform certain functions. One such place is at the STARTDUM, one of the locations in USYS's PDT.G routine. PDT.G is run whenever the user issues a "G" command to start the application. NMFS overwrites the NOP instruction at STARTDUM

## USYS

with a branch to START in NMFS's INSTR module. START, in turn, initializes NMFS data structures and sets up for the execution of the first instruction (see Section E.3). Another such hook is the one PM places at CLK.WRRET to catch 1.6 millisecond interrupts seen by USYS. Another is CONSOLE's patch to PDT.URAM.E, the end of USYS's console command decoding chain. NMFS adds its own set of checks for other commands, such as "A", "X", etc.

Control is passed back to USYS after each of the hooks performs its function. USYS is also called by CRASH after the context has been saved. This is done by CRASH's branching to PDT.TRAP, which simply prints the "t" on the console (see also PDT.BKRPNT and ENTER.DDT for other interesting USYS entry points) and attempts to read from the cassette.

The usys-prom.mic file also "includes" the file "ascii.mic" which contains convenient definitions for the ASCII character set.

### E.3 INSTR - Instruction Emulation

INSTR (instr.mic) is responsible for the emulation of most instructions. This includes the all-important "MAIN" instruction entry point fixed at 20050 as per the INTS mechanism of the MBB.

From MAIN, the microcode dispatches based on the macroinstruction, with most dispatching immediately to a specialized routine for that particular instruction.

Memory reference instructions could all go to one of the common routines (EX0, EX01, etc.) to calculate the effective address and read the location addressed. Then, in a second dispatch, they would go to individual routines for the specific memory operation involved (ADD, CAS, etc.).

Originally this was the case, but because STA, STX, and JST did not actually use the contents of the location addressed, a second set of routines (EA0, EA01, etc.) was written just to compute the effective address and leave it in MAR. This saves either one or two cycles, depending on the circumstances. The JMP, JST, and CALL instructions save an additional cycle because all effective address and memory reference routines latch the ALU status when they store the memory address into MAR (the

microinstruction ends in "-> MAR(R) LS" or "-> MAR LS") and leave this status unchanged when they dispatch the second time. The ALU status is thus available to the microcode of the JMP, JST, and CALL instructions; they would otherwise require one more cycle to test the address in MAR in order to issue trap 6 in case it is zero. In certain common cases, the instructions which would otherwise go through the EA routines are optimized still further: the first dispatch for that instruction and the addressing mode combination are pointed to a routine optimized for the specific case. The second dispatch is omitted altogether, and the result is generally one or two more cycles saved.

This "dispatching," of course, is made possible by the special MIR-associated hardware and the associated dispatch memory. An illegal macroinstruction will select a dispatch address of uram 0 (start of microcode ram, currently 20000) from dispatch memory. This jump is trapped with code 4.

The mapping mechanism for the C/30E NMFS MIR daughterboard (MIRDB) is as follows:

## INSTR - Instruction Emulation

CL	M14	MOP	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	Purpose
0	X	X	M9	M8	M7	M6	M5	M4	M3	M2	M1	M0	Load dispatch
2	X	N	0	0	F0	M15	M14	M13	M12	M11	M10	M9	MRI (1st disp)
2	0	0	1	0	M15	M9	C	B	A	M3	M1	M0	Misc. instrs.
2	1	0	1	1	0	M15	M9	M8	M7	M6	M3	M2	Misc. instrs.
3	X	N	0	1	F0	M15	M14	M13	M12	M11	M10	M9	MRI (2nd disp)
10	X	X	1	1	1	0	M5	M4	M3	M2	M1	M0	Unused
14	X	X	1	1	1	1	M5	M4	M3	M2	M1	M0	Unused

## Legend:

D	Dispatch address bit (D0, D1, etc.)
CL	Dispatch class from DISP(CL) -> UPC
M	MIR bit (M0, M1, etc.)
MOP	MIR bits 13-10 ("OP code")
F	MODEF (mode flag, always 1 in NMFS)
X	Don't care
N	Nonzero
A	NOT (M8 OR M6 OR M4)
B	NOT (M7 OR M6)
C	NOT (M5 OR M4)
MRI	Memory reference instruction(s)

The final dispatch address (D0-D9) is then indexed into the dispatch address space as set up by the file "dispatch.mic". The contents of the specified dispatch "cell" are generally gated into the UPC to cause the machine to branch to the microcode designed to emulate that instruction.

Let us assume that the microcode has just finished executing some instruction in the normal way:

```
A) 10.pc + 1 -> 10.pc, mar(r) ;begin fetch of next instr
B) ints -> upc                ;[1] handle ints or next instr
C) nop                        ;[2]
```

In step A we begin the 3-cycle fetch for the next instruction as addressed by the next program counter (LO.PC). In the meantime, we begin a microcode branch through INTS which will nominally result in a branch to MAIN at 20050. By the time we arrive at MAIN on fetch cycle [3], MBR will contain the correct instruction:

```
D) mbr -> mir                ;copy instr to MIR
E) disp(2) -> upc            ;dispatch
F) 10.pc & 177000 -> temp    ;get C/30E "page" number
```

Step D moves the newly-fetched instruction to the MIR where it will be processed by the MIRDB into a dispatch address during step E. We then begin a branch to an appropriate place in the microcode based on the value in MIR and the "2" supplied in as the dispatch class (CL). Step F is concerned with computing the C/30E "page" number for possible use by memory reference instruction's effective address calculation. Perhaps an example would help clarify matters.

We arrive at step D with MBR containing an "LDA 123" instruction, opcode 004123 (no indirection, no indexing, page zero address 123). The MIRDB then computes the corresponding dispatch address for class 2 as 0204. This is why: CL=2, MOP (bits M13-M10) are non-zero (bit M11 is one), bits M9-M15 together have a value of 004 (octal), plus F0 is always 1 in NMFS.

If we look at dispatch.mic for a line which says "DISPATCH 204" we find an "EXP EX0". Therefore, the DISP(2)->UPC will result in branching to EX0 in INSTR.

```
G) mirfld -> mar(r)           ;fetch pg0 word
H) disp(3) -> upc           ;decode this MRI
I) nop
```

EX0 uses the special source MIRFLD (which is set by the MIRDB to the low 9 bits of the MIR) to begin the fetch of the corresponding macromemory word (step G). In this case, we will be fetching 000123. While this is happening, we dispatch on the type of memory reference instruction we have encountered (in this case LDA) using dispatch class 3, and arrive at 0604.

Again, when we examine DISPATCH 0604 in dispatch.mic, we find "EXP LDA", so we end up branching to LDA. By the time we

get there on cycle 3, location 123 will have been fetched into the MBR as per step G in EX0.

J) 10.pc + 1 -> 10.pc, mar(r) ;start fetch of next  
K) ints -> upc  
L) mbr -> 13.a ;store word into Areg

INSTR also contains some of the logic needed to service the MBB's programmable software interrupt (bit M.PROGINT in G14.MISC and MISC itself). Normally, when there are no device-requested interrupts, INTS has the value MAIN (20050) and the next instruction is begun. It is sometimes desirable to process any and all device interrupts but NOT go to MAIN after their completion. By setting M.PROGINT in MISC, INTS will have the value SOFTINT (20040) instead of MAIN.

There are three things which are multiplexed onto this programmable software interrupt: long instructions, scheduling, and watching the word specified by the "W" command.

The shift and rotate instructions are presently somewhat long-running. In between each iteration, they check for microinterrupts and service them. In order to regain control after the interrupt is serviced, SHFLUP sets M.PROGINT (by calling the general-purpose routine INT.INSTR) and puts its

address in L5.INSPC of RB.SOFTINT. When the software interrupt is received (after all the device interrupts), control passes to SOFT at the end of INSTR which sees L5.INSPC non-zero and branches to the address so identified. This, in turn, gives SHFLUP control instead of fetching a new instruction. SHFLUP will do the next iteration and check for interrupts again, etc. SOFT.INSTR always clears L5.INSPC when it branches.

The second function multiplexed onto the software interrupt is PM's goad queue processor and the NMFS process scheduler. RB.SOFTINT's L7.PMSCAN being non-zero directs SOFT to jump to PM.SCAN in PM (as it clears L7.PMSCAN). PM.SCAN is documented elsewhere. The INSTR subroutine INT.SCAN can be used to set L7.PMSCAN and set M.PROGINT (much like INT.INSTR).

The last function is the memory watcher as enabled by the W console command. If L4.WMAR (RB.SOFTINT) is non-zero, it is interpreted as the MAR to watch. At SOFT.WATCH, we check to be sure it has not changed (under the mask in L3.WMASK). If the value is still pure, we go directly to MAIN and fetch the current instruction. Note that we CANNOT go through INTS here because we will again wind up at SOFT having executed no macro instructions.

If no functions are to be performed, the M.PROGINT flag is turned off, thus enabling normal operation. The current implementation implies that if, say, an instruction debreaks once, it will end up at SOFT twice: the first time to regain control with M.PROGINT still on but L5.INSPC zeroed, the second time SOFT realizes there is nothing left to do and clears M.PROGINT.

The order of these checks is important: we must always finish the current instruction (even if it's long like a shift/rotate) before rescheduling any NMFS processes or watching for memory changes.

Note that unlike the shift/rotate instructions, BLT, CHK, and similar instructions DO NOT use the software interrupt because they are restartable from the "top" when the interrupt finishes. Simply put, these instructions leave L0.PC pointing to themselves until they finish. When seen in this light, instructions of this class may be "executed" a number of times in place -- each time progressing slightly further through their chore. Note that this strategy produces an instruction which can be preempted by higher priority processes, a property necessary for any potentially long-running instruction.

#### E.4 Extended Console Commands

Console.mic hooks into USYS at PDT.URAM.E to add new checks to the command decoding chain. For the commands P, A, X, S, and N, CONSOLE branches to PDT.HREG which translates the command into the appropriate "R" command. "X", for example, is the same as "205R".

The "W" command is handled in CONSOLE, too. If no argument was seen, W calls WATCH.OFF to clear L4.VMAR (in RB.SOFTINT) thus disabling memory watching. If an argument was present, PDT.WATCH sets up L4.VMAR, L6.WMBR, etc., by calling WATCH.GO.

## E.5 Crash Handler

Instead of going directly to PDT.TRAP to print the "t", etc., the NMFS firmware always goes to CRASH (in crash.mic). CRASH saves RB.MAIN, RB.CLOCK, RB.IOOS (PM's main register block), RB.SOFTINT, RB.TID, calls USYS's SAVE.STATE routine to set up RB.STATE, saves it, then saves the register block associated with BASE at the time of the CRASH -> UPC.

These register blocks are saved in macromemory at the addresses specified in the header of crash.mic so they are easily changed. CRASH.XXX is the macromemory address of the saved register block pointed to by BASE (and therefore its "name" is unknown).

Upon completion, CRASH goes directly to PDT.LOAD. It cannot go to PDT.TRAP because PDT.TRAP will re-save the current machine state into RB.STATE which has since been altered by CRASH's execution. In theory, this does not matter since the desired machine state is saved in macromemory. However, it is necessary for consistency since traps issued by USYS (the ones which type "m", "p", etc.) do not store the current machine state in main memory. Thus the state of a non-running C/30E is always kept in

RB.STATE and access to this information does not depend on reading it from the console, which may be off, broken, disconnected, or (at a remote site) unattended.

In order to branch to PDT.LOAD, the ascii character to be typed on the console must be passed in TEMP, and so CRASH takes the trouble to pass an "h" if the macrocode executed a HLT instruction (crash code 1), and otherwise passes a "t" (any other code).

E.6 Process Manager (PM)

In this section we present the structure and function of the NMFS process manager module, PM (pm.mic).

For historical reasons, PM (and some parts of I/O and the microcode device drivers) uses a set of names different from the current NMFS terminology. The translation table below should help:

Real Name	PM's Name
PCB	DCB
APR	ADV
DPR	DDV
GPR	SPK
SPR	RFI
PDV	HPK
XDV	ACT
PM	IOOS

## E.6.1 PCBs, Queues, and the State Word

As the reader is well-aware, NMFS manages the application's various PCBs by arranging them on the NMFS queues (idle, ready, timing, pending 0-31). Most NMFS operations consist of moving a PCB from one NMFS queue to another, possibly with some "side effects." In general, the microcode actually consists of such sequences. There is a family of routines of the form: "IO.RMQ.xxxx" and "IO.ENQ.xxxx" where xxxx is one of IDLE, READY, TIMING, or PENDING. These routines remove (RMQ) or install (ENQ) a given PCB on a given queue. Note that IO.RMQ.PENDING and IO.ENQ.PENDING will actually locate the correct pending queue (0-31) by fetching the PCB's priority field. Conceptually, at least, all the pending queues become one.

Although it is true that the queue header pointer (Q.HEDR) in the PCB identifies which queue the PCB is on, for reasons of efficiency and consistency-checking, NMFS maintains a state word near the top of the PCB which also identifies what state the process is in and therefore which queue its PCB SHOULD be located on.

When one of the RMQ routines is called, it first checks to be sure that the state word of the specified PCB is in agreement with the job to be performed: IO.RMQ.READY, check to see that the STATE.READY bit is set in the state word of the PCB before attempting to remove the PCB. After checking the state word, the RMQ routine will check to be sure that the PCB's header pointer matches the known address for the queue: IO.RMQ.IDLE makes sure that the Q.HEDR word of the PCB is, in fact, the idle queue header. These consistency checks make it very difficult for either the microcode or macrocode to get very far on a bogus data structure.

When one of the ENQ routines is called, the corresponding state bit is set and others cleared. IO.ENQ.TIMING, for example, clears STATE.IDLE, STATE.READY, and STATE.PENDING, and sets STATE.TIMING.

As we will see, STATE.RETIMED and STATE.REPOKED do not represent true states, and thus have no corresponding NMFS queues, etc.

## The Attention Mask

## E.6.2 The Attention Mask

NMFS often wishes to locate the pending process (PCB) which has the highest priority. When the current process executes an SPR, for example, NMFS might have to scan each of the 32 pending queues "looking" for the first one to contain a process. Clearly this is a costly operation: 32 memory fetches just to locate the PCB.

Instead, NMFS maintains 32 bits worth of "attention mask" in RB.IOOS, L10.ATTN1, and L11.ATTN2. For each priority level with at least one pending process, there is a "1" bit in the attention mask. In other words, the attention masks are a 1-bit abbreviation for the lengths of each of the pending queues.

Since IO.ENQ.PENDING and IO.RMQ.PENDING are the only routines in NMFS which change the contents of the pending queues, it is their job to update the corresponding bit of the attention mask. The reader might recall the queuing primitive's particular emphasis on recognizing the cases where ENQ puts something on a previously-empty queue and DEQ/RMQ remove the last thing. This facility of the queuing primitives is used by IO.ENQ/RMQ.PENDING to maintain the attention mask. RB.BITS is used extensively for

this purpose.

At any instant, to locate the first non-empty pending queue (i.e., the pending queue containing the process of the highest priority), NMFS must only locate the most significant "1" in the 32-bit attention mask. This search is made by using the "JFFO" subroutine of INSTR -- the same subroutine called by the FFO instruction. JFFO returns the number (0-31) of the bit. This number is generally passed to IO.FIND.PHEDR, a subroutine in PM, which locates the corresponding pending queue header by adding  $4*n$  to the address of pending header number 0.

### E.6.3 The Goad Queue

As we learned above, it is generally straightforward to move PCBs from NMFS queue to NMFS queue, changing the PCB's state words and the NMFS attention mask accordingly. In the simplest case, this motion is caused by the execution of a particular instruction, such as an APR (activate process), which would move a PCB from the idle queue to the ready queue.

Unfortunately, two of the reasons for PCB motion do NOT occur during instruction emulation: I/O completions and/or timeouts can each cause one or more PCBs to become runnable, and thus may require moving the PCB onto some pending queue. These two cases occur during the processing of a microinterrupt. Microinterrupts are generally serviced between instructions when the firmware does INTS-→UPC and ends up at the microcode interrupt vector for some device (or clock, etc.). If microinterrupts were serviced only between instructions, no difficulty would result; however, there are cases where the firmware must allow microinterrupt processing in mid-instruction.

Consider the case where a process has set a timeout for itself (TPR), queued an IOCB for processing, and is now executing

## The Goad Queue

a debreak (SPR). During the emulation of SPR, NMFS discovers the timeout request and decides to place the PCB onto the timing queue by calling IO.ENQ.TIMING. The IO.ENQ.TIMING routine finds the appropriate location (recall that the timing queue is in time-order) on the queue iteratively with no upper bound on the compute time necessary (although in practice there can only be a finite number of PCBs in memory, let alone on the timing queue). IO.ENQ.TIMING checks for pending microinterrupts and services them between each successive queue iteration. If an interrupt occurs, it will be serviced.

Now, the interrupt which was just granted turns out to cause the completion of this process' I/O and results in the need to goad the PCB. If left unchecked, the device driver (using a NMFS-supplied subroutine) would goad the very PCB which was being inserted into the timing queue back in IO.ENQ.TIMING. This would result in chaos since IO.ENQ.TIMING would then be trying to insert something which was on the pending queue into the timing queue!

The scenario above is not the only way in which interrupt-related goads can disturb the continuity of a long-running operation at instruction level. To solve this "race condition,"

## The Goad Queue

NMFS employs a singly-linked list known as the "goad queue." Whenever a process is to be goaded (either at instruction level as with GPR, or at interrupt level), its PCB is added to the goad queue and the software interrupt of the MBB is set along with L7.PMSCAN (indicating a need for scheduling). When the firmware eventually ends up in the PM.SCAN routine, the goad queue is checked to see if there have been any recent goads which must be serviced. Since the software interrupt is the lowest priority operation of the machine (except for beginning the emulation of a new macroinstruction), it is "safe" to do even the most violent of PCB-related actions at this point.

The subroutine IO.GOAD is called to place a PCB onto the goad queue. IO.GOAD checks to see if the PCB is already queued; if so, it leaves it there. The STATE.SKIP bit of the PCB's state word is used to control whether the PC of the process should be incremented when the process is scheduled. I/O completion (which causes the SPR to skip) enters at IO.GOADSKIP and sets the STATE.SKIP bit. Clock timeout (SPR returns to the next instruction) enters at IO.GOAD which does not set the SKIP bit. If the process being goaded is the current process (L6.DCB in RB.IOOS equals L6.CURR in RB.MAIN), then PM.SCAN increments L0.PC

## The Goad Queue

if the STATE.SKIP bit is set. Note that the head and tail of the goad queue are located in the registers L12.SGQ and L13.EGQ (RB.IOOS), respectively. In addition, the last PCB on the goad queue has minus one as its goad queue pointer (one might expect it to contain zero). The minus one is necessary so that IO.GOAD can easily check whether a PCB is on the goad queue by fetching its goad queue pointer and checking it for a non-zero value.

At the top of the scheduler, PM.SCAN checks L12.SGQ to see if there is anything to be goaded. Then, for each PCB on the queue, it removes the PCB from the goad queue (adjusting SGQ and EGQ accordingly) and checks its DCB.STATE.

If the PCB was pending (STATE.PENDING), the process was obviously goaded while it was already running. In this case, the scheduler simply sets the STATE.REPOKED bit in the STATE word so that when the process eventually SPR's, it will resume immediately.

If the PCB was not pending, we check the STATE.SKIP bit and increment the PCB's saved PC if STATE.SKIP=1. We then RMQ the PCB from whatever queue it is on (must be timing or ready) and put it on the pending queue.

After each iteration on the goad queue, PM.SCAN checks and services microinterrupts. Although this might initially seem strange, it is possible to have too many PCBs on the goad queue; however, no harm is done by adding additional PCBs at the end of the queue while PCBs at the front of the queue are being removed and processed.

## E.6.4 Scheduling

Whenever INT.SCAN is called, NMFS eventually ends up at PM.SCAN to (possibly) service the goad queue and (possibly) schedule a new process for execution. The actual scheduling code begins at SCAN.SCAN.

If the sign bit of L10.FLAGS (RB.MAIN) is set, the macroprogram is executing instructions in "inhibit" mode and does not want NMFS to switch processes. If SCAN.SCAN sees this flag, it goes directly to SCAN.SAME to continue with the current process instead of looking for a higher-priority one.

Otherwise, SCAN.SCAN finds the highest priority process by looking through the attention mask. Once found, the scheduler fetches the forward pointer of the corresponding pending queue header to locate the memory address of the first PCB at the highest priority. If this address matches the current PCB (L6.CURR of RB.MAIN), NMFS branches again to SCAN.SAME without switching the machine's context.

If a different, new PCB is found, SCAN.NEW saves off the current context in the current PCB's context save area and "restores" the context of the new process. Note that if L6.CURR

is zero, there is no current context to save. This happens only once when NMFS is first enabled with the NMFS instruction. When the context is switched, SCAN.NEW "falls into" SCAN.SAME, since now the "current" process is the one we wish to run.

## E.6.5 PM's Prologue

PM begins with a few pages of text and symbol definitions. The symbols like DCB.TYPE and DCB.TIME refer to offsets in the macrocode image of the PCB. The symbols TYPE.xxxx are values in DCB.TYPE. RB.ALL is the mask for DCB.RB. STATE.xxxx defines bits and fields within DCB.STATE (shared with DCB.PRIORITY). PRIORITY.xxxx defines the priority portion of DCB.PRIORITY/DCB.STATE. DD.xxx.OFFSET defines entry vector offsets for the microcode device drivers (see section E.8).

The register blocks which PM uses include: RB.IOOS (PM's main register block), RB.TID (used to manage the timing heap), RB.BITS (a table of powers of two used to manage the ATTN words), and RB.DEV (the device number dispatch table).

Also located in the prologue are the IOERR definitions -- these codes become the program trap error code number of Appendix D when offset by IOERR.0 (nominally 30).

## E.6.6 PM.CLEAR, NMFS Instruction

As part of INSTR's START routine, PM.CLEAR is called to initialize PM variables. This currently consists of the rather trivial task of zeroing out L6.CURR in RB.MAIN. L6.CURR contains a pointer to the currently-executing PCB. If zero, there is no current PCB because NMFS is "off." PM.CLEAR is called at level 2 from START.

The NMFS instruction (IO316.NMFS) is entered directly from MAIN's instruction dispatch because it is a simple (non-MRI) instruction. In this discussion, the reader should note that "NMFS" may now refer to the specific instruction (opcode 30) rather than the Native Mode Firmware System as a whole.

NMFS first calls IO.CLEAR to perform any I/O-related resets (see I/O). It then clears the two attention registers, empties the GOAD queue, sets the NMFS time to 0, and clears the macrointerrupt inhibit flag in L10.FLAGS of RB.MAIN.

NMFS then checks to see whether the A register is zero (turn NMFS off) or non-zero (initialize the NMFS data structures starting at the specified macromemory address). If A=0, we return through INTS to execute the next instruction in-line. If

A is not 0, the firmware computes the addresses for the various NMFS queue headers (idle, ready, timing, pending) and stores them in similarly-named registers of RB.IOOS for quick access in subsequent operation.

The NMFS instruction then moves the PCB at the front of the idle queue to the ready queue and calls IO.GOAD to cause the process to be scheduled.

## INH and ENB

## E.6.7 INH and ENB

The NMFS INH and ENB instructions do the straightforward thing of setting or clearing the sign bit of L10.FLAGS in RB.MAIN to indicate the mode of the machine.

As an optimization for short inhibited sections of macrocode, the firmware maintains a flag, L14.DIRTY in RB.IOOS, which is cleared by INH, set by GOAD, and examined by ENB. If ENB finds the flag zero, it assumes no process has been goaded and thus the current process must still be the one which should run. If DIRTY is set, ENB causes a scheduling scan via INT.SCAN. ENB is further optimized to detect the case where the machine is not inhibited (i.e., a spurious ENB) and goes directly to INTS in that case.

The use of the DUMMY return address for INT.SCAN is presented below in Appendix E.6.10..

## E.6.8 APR (ADV), DPR (DDV), and DD Entries

These routines are entered directly from MAIN's dispatch (IO316.ADV, IO316.DDV). They each do the obvious thing: move a PCB to or from the idle queue. ADV initializes some fields of the PCB to lessen the chance of truly mystifying behavior when the macrocode fails to do so.

Note that DDV, like most other PCB-manipulating instructions, checks to see what state the PCB is in so it knows which IO.RMQ.xxxx routine to call. For no good reason, issuing a DDV of an idle PCB causes the PCB to shuffle to the end of the idle queue.

In both ADV and DDV, any microcode device driver (DD) corresponding to this PCB is given the chance to initialize (ADV) or shut down (DDV) cleanly through its DD.ADV.OFFSET or DD.DDV.OFFSET entry points. See Appendix E.8 for more information about this mechanism.

## E.6.9 XDV (ACT) and PDV (HPK)

These instruction handlers simply call any associated microcode device driver at either DD.ACT.OFFSET or DD.HPK.OFFSET, the former passing the emulated A register (L3.A, RB.MAIN) in GO.TEMP for the DD's easy access. See Appendix E.8 for more information on this linkage. These routines also make sure the PCB is not idle because it is illegal to manipulate idle devices in this way.

## E.6.10 GPR (SPK) and DUMMY

GPR could not be simpler: it merely calls IO.GOADSKIP to cause the corresponding process to become runnable at its SPR+2, or have its STATE.REPOKED bit set if it is already pending.

Note the use of "DUMMY" as a return address from IO.GOADSKIP. Since anything (like GOADSKIP) which results in a call INT.SCAN forces the machine to PM.SCAN BEFORE the next instruction, there is no need for SPK to pre-fetch the next instruction. It simply makes sure that LO.PC is updated, calls GOADSKIP (which calls INT.SCAN), and winds up at DUMMY, which does an INTS->UPC, branching to PM.SCAN (through SOFT) which is guaranteed to fetch the next instruction from either the current (SCAN.SAME) or new (SCAN.NEW) process.

## TPR (TDV)

## E.6.11 TPR (TDV)

TPR simply sets up DCB.TIME to contain the current time (L15.TIME in RB.IOOS) plus the value supplied by the macroprogram in the A register. In addition, TPR sets the STATE.RETIMED bit as a flag for SPR to place the process on the timing queue rather than the ready queue.

## E.6.12 SPR (RFI)

SPR makes sure the macrocode is not running in inhibited mode: if it were in inhibit mode, no subsequent scheduling could occur and the machine would be left with no current process (because it just suspended itself) and no new process (because PM.SCAN could not schedule any).

Inhibits aside, SPR checks the state of the current process. If the DCB.STATE has the STATE.REPOKED bit set, the SPR shuffles the PCB to the end of its pending queue and forces a scheduling scan (by calling INT.SCAN). SPR must move the current PCB to the end of the queue to insure that round-robin scheduling occurs at any given priority, even if the current process has been REPOKED. Imagine the situation where there are multiple, compute-bound process running at the same priority level, each one GPR'ing itself and debreaking (SPR'ing) every now and then. If SPR simply skipped on the REPOKED condition, the first of these processes would monopolize the machine forever.

If STATE.REPOKED is off, STATE.RETIMED is examined to see whether the process issued a TPR instruction during its last wakeup. If RETIMED=1, IO.ENQ.TIMING is called, else

IO.ENQ.READY. In either case, as with the REPOKED case, INT.SCAN is called to force a scheduling scan before the next instruction (since the current process is not necessarily runnable).

In either case, SPR always updates L0.PC (+2 if REPOKED, +1 otherwise) and writes its new value into DCB.PC.

## IO.CLOCK

## E.6.13 IO.CLOCK

Every 1.6 milliseconds, USYS "calls" PM's IO.CLOCK routine. "Calls" is perhaps the wrong word since PM overlays the CLK.WRRET location in USYS and thus gives USYS very little choice in the matter!

IO.CLOCK "ticks" L15.TIME (the NMFS timebase) and checks to see if the PCB at the front of the timing queue is set to timeout at a time equal to the current time. If the times are equal, IO.CLOCK calls IO.GOAD to place the PCB on the goad queue for later movement to the appropriate pending queue and checks the next PCB on the queue since multiple processes may be scheduled to wakeup at the same tick.

Although no clock ticks can ever "slip by", IO.CLOCK is willing to skip over those entries at the front of the timer queue which appear to be in the past. This would happen only if the machine were completely bogged down for 100 microseconds processing I/O interrupts, and therefore had not gotten around to removing the entry from the front of the timing queue after it had been GOADED on the previous clock interrupt.

## E.7 I/O Library

The I/O library (io.mic) contains subroutines called from microcode device drivers. The subroutines are called through global register G6.TEMP and immediately save BASE and switch to RB.IOOS. Because of these precautions, a DD can call an I/O subroutine at any point without fear of side effects.

Other than IO.CLEAR (a general-purpose reset routine called by INSTR's START routine and PM's NMFS instruction), there are two subroutines in the I/O library.

IODD.GET attempts to get a new IOCB for a DD to use. This consists of trying to DEQ an IOCB from the device's GET queue and, if successful, setting DCB.IOCB to point to this new IOCB, and filling in the IOCB GET queue timestamp if timestamping was requested (i.e., if the timestamping bit was on) in the flags word of this IOCB. IODD.GET is called with the return address in G6.TEMP and the PCB address in TEMP. It then places the IOCB GET queue timestamp into the IOCB if timestamping was requested in the flags word of this IOCB.

It returns with GO.TEMP pointing to the new IOCB, or 0 if none was obtained. In either case, the value of GO.TEMP is

latched into the C/30E's ALUST register (n→G0.TEMP LS) for easy testing by the caller. IODD.GET also checks to be sure that DCB.IOCB is zero when called, preventing the device driver from trying to establish two "current" IOCBs.

IODD.PUT is the opposite of IODD.GET: it moves the current IOCB to the PUT queue, fills in the IOCB PUT queue timestamp if timestamping was requested, and clears DCB.IOCB. In addition, it tests for appropriate interrupt conditions and calls IO.GOADSKIP if it determines an interrupt is necessary. This determination is based only on device-independent conditions: the interrupt-always bit, and the interrupt-on-error bit, combined with a non-zero completion code in the IOCB.FLAGS word. Device-dependent interrupt conditions (e.g., interrupt-on-EOM, etc.) must be determined by the device.

IODD.PUT has two alternate entry points to assist device drivers in controlling the disposition of a possible completion interrupt. By entering at IODD.IPUT, the device driver can instruct the put subroutine to ALWAYS cause an interrupt, regardless of the conditions which IODD.PUT would normally check. This is how a device driver can cause an interrupt on its own device-dependent conditions such as interrupt-on-EOM. Upon

detecting the EOM condition, the DD branches to IODD.IPUT instead of IODD.PUT.

The third entry point, IODD.NPUT, NEVER causes an interrupt, regardless of IODD.PUT's normal considerations. If used improperly, IODD.NPUT can cause serious violations of the normal NMFS IOCB handling: if the IOCB says "interrupt-always", but the DD goes to IODD.NPUT, no interrupt will occur and the macroprogrammer could become confused. Normally, IODD.NPUT is used only in exceptional conditions such as in the processing of XDV "reset" operations where the macrocode is asking the DD to give back the current IOCB and reset, but does not need to receive a completion for this aborted transfer.

When in doubt, the device driver should use IODD.PUT.

It then places the IOCB PUT queue timestamp into the IOCB if timestamping was requested in the flags word of this IOCB. Finally, IODD.PUT exits.

Other than IODD.PUT's examination of the interrupt-always, interrupt-on-error, timestamping, and completion code bits, neither IODD.PUT nor IODD.GET examine or alter the value in IOCB.FLAGS. Device-specific bits (such as end-of-message) are

managed outside of the I/O library routines. When a driver wishes to signal a completion with some status bit set (call it DEV.EOM), it OR's DEV.EOM into IOCB.FLAGS before calling IODD.PUT. When, on output, it wishes to examine the EOM condition, it does so by fetching IOCB.FLAGS and looking at the DEV.EOM bit.

## E.8 Microcode Device Drivers

Each microcode device driver is responsible for controlling exactly one of a particular kind of device (such as an "MII board 1822 interface"). There may be a choice of two or more different drivers for a particular type of device. For example, one might want to use a high-throughput driver on fast MII synchronous lines and a lower-efficiency, high-functionality version on the slower lines.

The device driver proper must be reentrant if it is to control more than one instance of the device. Generally, this is accomplished by giving each instance of the device a different block of microregisters. The reentrant driver operates identically, but independently for all devices under its control, with each maintaining its current state, etc., in a unique register block.

In addition to a register block, each instance of the device must have a set of microinterrupt vector locations. These vector locations generally set up BASE to the appropriate register block, and branch to a common, reentrant point in the driver.

It is important to note that the register block and interrupt vector assignments "tailor" an otherwise reentrant driver to a particular instance of the device type.

## E.8.1 The Driver Entry Table and IO.DD.CALL

Each device driver must provide the address of a driver entry table (DET) to NMFS in RB.DEV. RB.DEV is indexed by the device type (from DCB.TYPE) to yield the base of the DET for the device. The DET, in turn, yields a number of entry points into the driver. NMFS calls on these entry points when particular events occur.

Whenever an APR is given for a process with an I/O type code (DCB.TYPE is non-zero), the corresponding driver is called through its ADV entry in its DET. Similarly, when DPR, XDV, or PDV are issued on a process, the corresponding driver is called.

PM's IO.DD.CALL subroutine is responsible for calling the device driver associated with a process. NMFS calls IO.DD.CALL with a small positive integer in TEMP. IO.DD.CALL finds the associated driver (and its DET) by indexing DCB.TYPE into RB.DEV, then adds TEMP to the DET base address and branches there, thus entering the driver.

Before branching, IO.DD.CALL sets BASE to DCB.RB (or RB.MAIN if DCB.RB is zero), sets G6.TEMP to the PCB address, and passes GO.TEMP to the driver unchanged so that XDV (ACT) can pass the

user's A register to the driver through GO.TEMP.

All device drivers should return from a DET call by branching to "DD.RET" in NMFS. DD.RET automatically restores BASE, etc., and may thus be branched to without serious consideration.

Here is an example of a typical driver entry table:

```
mydev.det:
    mydev.apr -> upc   ;APR
    mydev.dpr -> upc   ;DPR
    mydev.xdv -> upc   ;XDV
    mydev.pdv -> upc   ;PDV

mydev.pdv:                               ;(for example)
    here would code the
    code for handling
    a PDV from NMFS.
    Probably, we would
    call IO.DD.GET, etc.

    dd.ret -> upc     ;return to NMFS
    nop
```

## E.8.2 Bidirectional Devices and Their Drivers

Recall that in NMFS, each side of a bidirectional device is handled by a separate driver. Specifically, this means that there will be two macroprocesses attached to the same hardware via two drivers. As a matter of implementation, the two drivers are generally co-resident in the same microcode source file and often share each other's register blocks, but because they really are two distinct drivers, each has a different device type in DCB.TYPE. It follows that there must be two entries in RB.DEV (one for input and one for output) and thus two driver entry tables. In many cases, some, if not all, of the DET entries branch to the same place for each driver. In fact, sometimes both drivers share the same DET by pointing both RB.DEV entries at one address.

### E.8.3 Organizational Suggestions

Other than supplying a driver entry table pointer in RB.DEV and servicing APR, DPR, PDV and XDV entries, an NMFS microcode device driver has no required structure or function. Generally, the driver will contain firmware to set-up for new input or output operations and receive and process subsequent device microinterrupts. If it controls multiple instances of a device, the DD will generally use unique microregister blocks to maintain state information about each instance.

The organization of a driver which controls more than one instance of a device requires further attention. In particular, it is important for the NMFS firmware to be configurable to the application's needs. One configuration might require 3 CM1's on the first I/O board, another 2 C18's, and 1 CM1, etc. In this section, we will present a popular format for device drivers which easily accommodates varied configurations.

It is often helpful to pattern a new device driver after one which has already been written (such as CM1 or C18). Both CM1 and C18 use the mechanism presented here, albeit in a full-duplex configuration. The reader is urged to consult the sources for

these drivers during this discussion.

One should first write the basic driver code. This would consist of the driver entry table(s) and the basic code to handle each entry. It would also consist of the code to handle the various device microinterrupts but WOULD NOT contain the actual interrupt vector assignments. Imagine that "somehow" your drivers gets control when there is an interrupt for the device. In addition, this portion of the driver may assume that BASE is always set-up for it: NMFS (IO.DD.CALL) sets it to DCB.RB on DET entries. Let us assume that our unspecified interrupt vector code sets BASE to the same value when a microinterrupt occurs.

In essence, this basic driver includes no register allocations, no hard-wired interrupt vector addresses, and no references to physical MBB I/O space addresses. Instead, it will assume that BASE is pre-set by the external environment, that the interrupt vectors are set-up elsewhere, and that the I/O address will appear in some register for the particular device associated with that register block.

After writing the "pure" portion of the driver, one should create a file which produces the missing information for a given

instance of the device. These files are traditionally named "xxx-device.mic" where xxx corresponds to the same name as the "pure" portion of the device (see cm1-device.mic, etc.). This file should contain something like the following (omitting, of course, the parenthetical step letters):

```
(A) rblk == rnext
(B) rnext == rnext + 10

(C) addr == (board_15) + 400 + 20*device
(D) vect == (board_7) ! 20100 + 10*device

(E) uram vect
(F) rblk -> base
(G) myint -> upc
(H) mbr -> g13.mbr

(I) regs rblk + 6
(J) exp addr
```

The job of this file is to transform "BOARD" and "DEVICE" into the appropriate code and data to service an interrupt from I/O board number "BOARD", interface number "DEVICE". BOARD and DEVICE will be supplied to this file by some external source-level configuration scheme (we will discuss this in greater detail).

Steps (A) and (B) are concerned with allocating 8 (decimal) registers for this instance of the device. At the end of the "pure" NMFS assembly, the symbol RNEXT contains the address of the first free microregister. The symbol table from this assembly will be made visible to this file by the unspecified source-level configuration scheme to be presented shortly. In any event, step (A) defines RBLK to point to the first of these free registers, and step (B) advances RNEXT over these now-reserved registers leaving it pointing to the next available register for subsequent users.

Step (C) transforms BOARD and DEVICE into an MBB I/O address. This, of course, requires an intimate knowledge of the device involved. In this example, the I/O address is 400 (octal) away from the BOARD number shifted 15 places left, plus 20 times the DEVICE number (400, 420, 440, etc), see Reference 2 for information about MBB I/O space.

Step (D) computes the address of the corresponding interrupt vector location for BOARD/DEVICE. In this example, it is 100 from BOARD shifted 7 places left plus 10 times the DEVICE number (20100, 20110, ...). Note that the 20000 bit is always set in these addresses: recall that in MBB, all microcode URAM address

have this bit set.

Using the result of step (D), step (E) sets the assembler's current address to be the beginning of the appropriate interrupt vector area, in URAM. Steps (F), (G), and (H) implement the linkage to MYINT, the interrupt handling entry point of this driver. Step (F) changes BASE to the value computed in step (A), step (G) begins the 2-cycle branch, while step (H) saves the previous MBR. In the MBB, MBR must be saved by microinterrupt handlers and restored on interrupt dismissal (INTS->UPC) because the machine may branch directly to MAIN and assume that the MBR contains the next instruction (which it would have if this interrupt had not preempted the branch to MAIN via INTS).

There may be other instructions in the interrupt vector area, but one must be careful not to run over into the next interrupt area by writing a long vector linkage. Although it is board-specific, most devices have four-word interrupt vector areas.

Steps (I) and (J) place the physical I/O space address (from step C) into register "L6" of this device's register block. The driver may then reference the I/O address through this register

(L6.IOADDR -> MAR(RIO)).

Of course the choice of which register to use for the I/O address is completely up to the programmer, as is the number of registers to allocate (steps A and B), and the style of the interrupt vector linkage (with possible constraints on the number of instructions and the use of MBR).

## E.8.4 Source-Level Configuration

Armed with the "pure" NMFS microcode and a collection of "customization" files to set up the registers and interrupt locations, we may now create the necessary binary image for NMFS.

Owing to its purity, the main body of NMFS (including the device drivers) is first assembled using the MBB microassembler "ma" (see E.9). Then a user-supplied source file is assembled into another binary which assigns various BOARD and DEVICE values just before including the appropriate customization file.

The example below should not be considered a real-life implementation, but it serves to illustrate the desired effect.

An MA macro definition file:

```
-----  
_define(MYDEV, '  
    BOARD = $1  
    DEVICE = $2  
    include('mydev-device.mic')  
)  
  
_define(YOURDEV, '  
    BOARD = $1  
    DEVICE = $2  
    include('yourdev-device.mic')  
)
```

The user-supplied configuration specification: -----

```
MYDEV(1,3) ;board #1, device #3
MYDEV(1,4) ;board #1, device #4
MYDEV(2,0) ;board #2, device #0
YOURDEV(4,2) ;board #4, device #2
```

The UNIX Typescript: -----

```
ma config-file.mic -o config-file.mbn
```

The resulting binary image is combined with the "pure" image from the basic NMFS assembly to make up firmware specifically tailored to the desired configuration. This is done by loading the NMFS binary followed by the configuration binary. Normally the two files must be written onto a cassette in loading order; then when the C/30E is booted and reads its cassette, the contents of both assemblies are correctly placed in the various memories of the C/30E. See sections 6.3.1 and 6.5 for more details of the register block allocation scheme.

## E.9 The Microassembler (MA)

MA, the microassembler, was originally implemented on TENEX and TOPS-20, and this implementation is described in section 11 of the MBB Microprogrammer's Handbook, BBN Report No. 4268. The UNIX version uses the identical source program format, but, of necessity, is run differently.

The command line for MA is as follows:

```
ma inputfile [switches]
```

The optional switches are:

- o filename : This specifies the .mbn output file, and is required, unless one only wants a listing.
- a : This causes the output binary to be in human-readable format.
- h : This causes help to be printed.
- l filename : This specifies the listing output file, and is optional.
- p : Use the assembler interactively -- you type in an instruction, and it types back the binary.
- v : Prints a message at the beginning of each pass.