CRAY
RESEARCH, INC.

CRAY X-MP™ AND CRAY-1®
COMPUTER SYSTEMS

APML ASSEMBLER
REFERENCE MANUAL

SM-0036

Copyright© 1980, 1981, 1986 by CRAY RESEARCH, INC. This manual or parts thereof may not be reproduced in any form without permission of CRAY RESEARCH, INC.

RECORD OF REVISION **CRAY RESEARCH, INC.** PUBLICATION NUMBER SM-0036

Each time this manual is revised and reprinted, all changes issued against the previous version are incorporated into the new version and the new version is assigned an alphabetic level.

Every page changed by a reprint with revision has the revision level in the lower righthand corner. Changes to part of a page are noted by a change bar in the margin directly opposite the change. A change bar in the margin opposite the page number indicates that the entire page is new. If the manual is rewritten, the revision level changes but the manual does not contain change bars.

Requests for copies of Cray Research, Inc. publications should be directed to the Distribution Center and comments about these publications should be directed to:

CRAY RESEARCH, INC.
2520 Pilot Knob Road
Suite 310
Mendota Heights, Minnesota 55120

Revision	Description
	November 1980 - Original printing.
A	June 1981 - Reprint with revision. This version obsoletes the previous edition and brings the manual into agreement with the 1.10 release. Major changes include the addition of the TEXT, ENDTEXT and MODULE pseudo instructions. The manual has also been reorganized.
A-01	April 1982 - Change packet. Brings the manual into agreement with version 1.11 of the APML assembler. Major changes include the deletion of the MODULE pseudo instruction, the addition of the WRP, NWRP, WMR, and NWMR options to the APML control statement and the LIST pseudo instruction, and the addition of two warning errors: Y1 - EXTERNAL DECLARATION ERROR and Y2 - MACRO REDEFINED. Miscellaneous technical and editorial changes are also included.
A-02	March 1983 - Change packet. Brings the manual into agreement with version 1.12 of the APML assembler. A major change allows externals within absolute assembly. Changes also include the addition of CPU time and release level and assembly date to logfile messages; changing APML message prefix from CA to AP; changing APML message for memory and I/O use from octal to decimal; introduction of new predefined micro and new CPU type on control card, new listed output option where L takes precedence over E on control statement; and miscellaneous technical and editorial changes.

B March 1986 - Reprint with revision. This reprint brings the manual into agreement with APML 2.0 running under COS 1.15 and APML 2.1 running under UNICOS 1.0. Section 3 is deleted; see the IOS hardware manuals for the information this section covered. Section 3 now covers APML invocation and execution. The information on channel interface functions has been moved from appendix C to section 7. The information in appendix E has been moved to appendix C. All previous versions are obsolete.

B-01 October 1986 - This change packet brings the manual into agreement with APML version 3.0 running under UNICOS 2.0. The changes to the UNICOS command line are: -h and -i options have been added for the processing of list pseudos, the -o option has been added allowing for the specification of the binary object file, and the -y option has been changed to -g. The syntax of the -s option and the handling of intermediate files were changed. Miscellaneous technical and editorial changes are also included. All trademarks are now documented in the record of revision.

CRAY, CRAY-1, and SSD are registered trademarks and APML, CFT, CFT77, CFT2, COS, CRAY-2, CRAY X-MP, CSIM, IOS, SEGLDR, SID, SUPERLINK, SUPERLINK/ISP, and UNICOS are trademarks of Cray Research, Inc. The UNICOS system is derived from the AT&T UNIX system; UNIX is a registered trademark of AT&T.

CDC is a registered trademark of Control Data Corporation.

PREFACE

The APML assembler allows you to express symbolically all hardware functions of the Cray Research, Inc. (CRI) I/O Subsystem (IOS). This detailed and precise level of programming is useful when tailoring programs to the architecture of the IOS and writing programs requiring code optimized to the hardware.

The pseudo instructions provided with APML's instruction set allow a variety of options for generating macro instructions, controlling list output, organizing programs, and so on.

The following CRI publications provide supplemental information on the IOS:

SM-0007	IOS Table Descriptions Internal Reference Manual
HR-0030	I/O Subsystem Model B Hardware Reference Manual
SG-0051	I/O Subsystem (IOS) Operator's Guide For COS
HR-0081	I/O Subsystem Model C Hardware Reference Manual
SG-2005	I/O Subsystem (IOS) Operator's Guide For UNICOS

The following CRI publications may also interest you:

SR-0000	CAL Assembler Version 1 Reference Manual
SR-0011	COS Version 1 Reference Manual
SM-0044	Operational Aids Reference Manual
HR-0077	Disk Systems Hardware Reference Manual
SR-2003	CAL Assembler Version 2 Reference Manual

The IOS Software Internal Reference Manual, CRI publication SM-0046, describes the system macro instructions available with APML.

CONTENTS

PREFACE v

1. INTRODUCTION 1-1

1.1 EXECUTION OF THE APML ASSEMBLER 1-2
1.2 CONVENTIONS 1-2

2. APML ASSEMBLER LANGUAGE 2-1

2.1 SOURCE LINE FORMAT 2-1
2.2 STATEMENT FORMAT 2-1
2.2.1 Comment statement 2-1
2.2.2 Symbolic APML instruction format 2-2
2.2.3 Pseudo instruction format 2-2
2.3 CODING CONVENTIONS 2-3
2.4 LINE EDITING 2-3
2.4.1 Concatenation 2-4
2.4.2 Micro substitution 2-4
2.5 NAMES 2-4
2.6 SYMBOLS 2-5
2.6.1 Symbol definition 2-5
2.6.2 Symbol attributes 2-5
2.7 SYMBOL REFERENCE 2-7
2.7.1 Qualified symbols 2-7
2.8 GLOBAL DEFINITIONS 2-8
2.9 SPECIAL ELEMENTS 2-8
2.10 DATA NOTATION 2-9
2.10.1 Numeric constants 2-9
2.10.2 Character constants 2-10
2.10.3 Data items 2-11
2.11 PREFIXED SYMBOLS AND CONSTANTS 2-13
2.11.1 Parcel address prefix - P. 2-13
2.11.2 Word address prefix - W. 2-13
2.12 EXPRESSIONS 2-14
2.12.1 Adding operators 2-14
2.12.2 Multiplying operators 2-14
2.12.3 Elements 2-15
2.12.4 Terms 2-15
2.12.5 Term attributes 2-15
2.13 EXPRESSION EVALUATION 2-17
2.14 EXPRESSION ATTRIBUTES 2-17
2.14.1 Relocatable, external, or absolute 2-18
2.14.2 Parcel address, word address, or value 2-18
2.15 TABLE METHOD OF EXPRESSION ATTRIBUTE EVALUATION 2-20

3. APML INVOCATION AND EXECUTION 3-1

3.1 COS APML CONTROL STATEMENT 3-1
3.2 UNICOS APML COMMAND LINE 3-4
3.3 SYSTEM TEXT 3-6
3.4 BINARY SYSTEM TEXT 3-6

4. SYMBOLIC APML INSTRUCTION SYNTAX 4-1

4.1 OPERAND NOTATION 4-1
4.2 OPERATORS 4-3
4.2.1 Replacement operator 4-3
4.2.2 Function operators 4-3
4.2.3 Relational operators 4-4
4.2.4 Conditional operator 4-4
4.3 PROGRAM STATEMENT INSTRUCTION FORMAT 4-4
4.3.1 Assignment clauses 4-5
4.3.1.1 Replacement assignment 4-5
4.3.1.2 Jump assignment 4-9
4.3.1.3 Set flag assignment 4-9
4.3.1.4 Special function 4-10
4.3.1.5 Channel function 4-10
4.3.2 Condition clauses 4-11
4.3.2.1 Test accumulator 4-11
4.3.2.2 Test register or memory 4-12
4.3.2.3 Test carry flag 4-13
4.3.2.4 Test channel status 4-13
4.3.3 Syntax graphs for APML program statements . . . 4-13
4.4 DATA GENERATION STATEMENT INSTRUCTION FORMAT 4-18

5. BASIC IOP HARDWARE INSTRUCTION SET 5-1

5.1 INSTRUCTION INDEX 5-1
5.2 CONTROL INSTRUCTIONS 5-3
5.2.1 PASS 5-3
5.2.2 EXIT 5-4
5.2.3 I = 0 5-4
5.2.4 I = 1 5-4
5.3 TRANSMIT TO ACCUMULATOR INSTRUCTIONS 5-5
5.3.1 A = *d* 5-5
5.3.2 A = *k* 5-5
5.3.3 A = *dd* 5-6
5.3.4 A = (*dd*) 5-6
5.3.5 A = B 5-6
5.3.6 A = (B) 5-6
5.4 LOGICAL PRODUCT WITH ACCUMULATOR INSTRUCTIONS 5-7
5.4.1 A = A & *d* 5-7
5.4.2 A = A & *k* 5-7
5.4.3 A = A & *dd* 5-7

5.4 LOGICAL PRODUCT WITH ACCUMULATOR INSTRUCTIONS (continued)
5.4.4 A = A & (*dd*) 5-8
5.4.5 A = A & B 5-8
5.4.6 A = A & (B) 5-8
5.5 ADD TO ACCUMULATOR INSTRUCTIONS 5-8
5.5.1 A = A + *d* 5-9
5.5.2 A = A + *k* 5-9
5.5.3 A = A + *dd* 5-9
5.5.4 A = A + (*dd*) 5-9
5.5.5 A = A + B 5-10
5.5.6 A = A + (B) 5-10
5.6 SUBTRACT FROM ACCUMULATOR INSTRUCTIONS 5-10
5.6.1 A = A - *d* 5-10
5.6.2 A = A - *k* 5-11
5.6.3 A = A - *dd* 5-11
5.6.4 A = A - (*dd*) 5-11
5.6.5 A = A - B 5-11
5.6.6 A = A - (B) 5-12
5.7 INCREMENT BY 1 INSTRUCTIONS 5-12
5.7.1 dd = *dd* + 1 5-12
5.7.2 (dd) = (*dd*) + 1 5-12
5.7.3 B = B + 1 5-13
5.7.4 (B) = (B) + 1 5-13
5.8 DECREMENT BY 1 INSTRUCTIONS 5-13
5.8.1 *dd* = *dd* - 1 5-13
5.8.2 (*dd*) = (*dd*) - 1 5-14
5.8.3 B = B - 1 5-14
5.8.4 (B) = (B) - 1 5-14
5.9 ADD TO ACCUMULATOR AND REPLACE OPERAND INSTRUCTIONS . . 5-14
5.9.1 *dd* = A + *dd* 5-15
5.9.2 (*dd*) = A + (*dd*) 5-15
5.9.3 B = A + B 5-15
5.9.4 (B) = A + (B) 5-15
5.10 TRANSMIT FROM ACCUMULATOR INSTRUCTIONS 5-16
5.10.1 *dd* = A 5-16
5.10.2 (*dd*) = A 5-16
5.10.3 B = A 5-16
5.10.4 (B) = A 5-17
5.11 SHIFT INSTRUCTIONS 5-17
5.11.1 End off shifts 5-17
5.11.2 Circular shifts 5-18
5.12 SET CARRY FLAG INSTRUCTIONS 5-18
5.12.1 C = 1, *iod* = DN 5-18
5.12.2 C = 1 5-19
5.12.3 C = 1, *iod* = BZ 5-19
5.12.4 C = 0 5-19
5.12.5 C = 1, IOB = DN 5-19
5.12.6 C = 1, IOB = BZ 5-20
5.13 BRANCH INSTRUCTIONS 5-20
5.14 CHANNEL FUNCTION INSTRUCTIONS 5-21

6. PSEUDO INSTRUCTIONS 6-1

6.1 RULES FOR PSEUDO INSTRUCTIONS 6-1
6.2 TYPES OF PSEUDO INSTRUCTIONS 6-2
6.3 PROGRAM CONTROL PSEUDO INSTRUCTIONS 6-2
6.3.1 IDENT - Identify program module 6-2
6.3.2 END - End program module 6-3
6.3.3 ABS - Assemble absolute binary 6-3
6.3.4 COMMENT - Define Program Descriptor Table comment 6-4
6.3.5 GLOBAL - Declare global symbols 6-4
6.4 CODE CONTROL PSEUDO INSTRUCTIONS 6-5
6.4.1 BASEREG - Declare base operand register 6-5
6.4.2 SCRATCH - Declare APML scratch register 6-6
6.4.3 NEWPAGE - Force a new instruction page 6-7
6.5 LOADER LINKAGE PSEUDO INSTRUCTIONS 6-8
6.5.1 ENTRY - Specify entry symbols 6-8
6.5.2 EXT - Specify external symbols 6-9
6.5.3 START - Specify program entry 6-9
6.6 MODE CONTROL PSEUDO INSTRUCTIONS 6-10
6.6.1 BASE - Declare base for numeric data 6-10
6.6.2 QUAL - Qualify symbols 6-12
6.7 BLOCK CONTROL PSEUDO INSTRUCTIONS 6-14
6.7.1 BLOCK - Local block assignment 6-16
6.7.2 ORG - Set *O counter 6-17
6.7.3 BSS - Block save 6-17
6.7.4 LOC - Set * counter 6-18
6.7.5 BITW - Set *W counter 6-19
6.7.6 BITP - Set *P counter 6-20
6.8 ERROR CONTROL PSEUDO INSTRUCTIONS 6-21
6.8.1 ERROR - Unconditional error generation 6-21
6.8.2 ERRIF - Conditional error generation 6-21
6.9 LISTING CONTROL PSEUDO INSTRUCTIONS 6-23
6.9.1 LIST - List control 6-23
6.9.2 SPACE - List blank lines 6-26
6.9.3 EJECT - Begin new page 6-26
6.9.4 TITLE - Specify listing title 6-27
6.9.5 SUBTITLE - Specify listing subtitle 6-27
6.9.6 TEXT - Begin global text 6-27
6.9.7 ENDTEXT - Terminate global text 6-28
6.10 SYMBOL DEFINITION PSEUDO INSTRUCTIONS 6-29
6.10.1 EQUALS - Equate symbol 6-29
6.10.2 SET - Set symbol 6-30
6.10.3 CHANNEL - Channel symbol 6-31
6.10.4 MICSIZE - Set redefinable symbol to micro size . 6-32
6.11 DATA DEFINITION PSEUDO INSTRUCTIONS 6-33
6.11.1 CON - Generate constant 6-33
6.11.2 BSSZ - Generate zeroed block 6-34
6.11.3 DATA - Generate data words 6-35
6.11.4 PDATA - Generate data parcels 6-36
6.11.5 VWD - Variable word definition 6-36

6. PSEUDO INSTRUCTIONS (continued)

6.12 CONDITIONAL ASSEMBLY PSEUDO INSTRUCTIONS 6-37
6.12.1 IFA - Test expression attribute for assembly condition 6-38
6.12.2 IFE - Test expressions for assembly condition . 6-39
6.12.3 IFC - Test character strings for assembly condition 6-41
6.12.4 SKIP - Unconditionally skip statements 6-42
6.12.5 ENDIF - End conditional code sequence 6-43
6.12.6 ELSE - Toggle assembly condition 6-43
6.13 INSTRUCTION DEFINITION PSEUDO INSTRUCTIONS 6-45
6.13.1 Macro definition format 6-46
6.13.2 MACRO - Macro definition 6-48
6.13.3 LOCAL - Specify local symbols 6-49
6.13.4 ENDM - End macro definition 6-50
6.13.5 OPSYN - Synonymous operation 6-53
6.14 CODE DUPLICATION PSEUDO INSTRUCTIONS 6-54
6.14.1 DUP - Duplicate code 6-54
6.14.2 ECHO - Duplicate code with varying arguments . . 6-55
6.14.3 ENDDUP - End duplicated code 6-56
6.14.4 STOPDUP - Stop duplication 6-57
6.14.5 Examples of duplicated sequences 6-57
6.15 MICRO DEFINITION PSEUDO INSTRUCTIONS 6-59
6.15.1 Micro reference format 6-59
6.15.2 MICRO - Micro definition 6-60
6.15.3 OCTMIC and DECMIC - Octal and decimal micros . . 6-61
6.15.4 Predefined micros 6-62

7. CHANNEL INTERFACE FUNCTIONS 7-1

7.1 INTERFACE CHARACTERISTICS 7-1
7.2 CHANNEL INTERFACE FUNCTION CODES 7-2

8. FORMAT OF ASSEMBLER LISTING 8-1

8.1 PAGE HEADERS 8-1
8.2 SOURCE STATEMENT LISTING 8-1
8.3 CROSS-REFERENCE LISTING 8-3

APPENDIX SECTION

A. CHARACTER SETS A-1

B. HARDWARE INSTRUCTION SUMMARY B-1

B.1 APML OPERAND NOTATION B-1
B.2 INSTRUCTIONS B-3

C. MESSAGES C-1

D. ASSEMBLY ERRORS D-1

FIGURES

4-1 Assignment Syntax 4-14
4-2 Condition Syntax 4-16

TABLES

2-1 Absolute Assembly Element and Term Attribute Evaluation . . . 2-20
2-2 Relocatable Assembly Element and Term Attribute Evaluation . . 2-20
5-1 Instruction Index 5-1
7-1 Channel Functions and Descriptions 7-2
A-1 Character Sets A-1
B-1 Instruction Summary B-3
D-1 Fatal Errors D-1
D-2 Warning Errors D-5

INDEX

1. INTRODUCTION

The Cray Research, Inc. (CRI) I/O Processor (IOP) Language, APML, is a powerful symbolic language that generates object code for the Cray I/O Subsystem's (IOS's) IOPs. An IOS is composed of two through four I/O Processors with Buffer Memory. APML operates on the IOS with either the COS or UNICOS operating system running on the mainframe.

APML source statements consist of symbolic APML instructions and pseudo instructions. The symbolic instructions allow you to express all Cray IOP functions symbolically. Pseudo instructions allow you to control the assembly process.

APML's features include:

- Free-field source statement format: source statement field size is largely controlled by you.
- Control of local blocks: you can assign code or data segments to specific areas.
- Multiple instruction generation: one or more IOP instructions are generated for each symbolic APML instruction.
- Code optimization: the assembler tries to minimize generated code by eliminating unnecessary instructions and by using 1-parcel instructions.
- Preloaded data: you can define data areas during assembly and load them with the program.
- Data notation: you can designate data as integer or character code notation.
- Word and parcel address arithmetic: you can specify addresses as either word or parcel addresses.
- Binary control: you can specify object code as either absolute or relocatable. Relocatable code is not supported by an associated loader for IOP code.
- Listing control: you can control the contents of the assembler listing.
- Micro coding: you can define a character string in a program and substitute for each occurrence of its micro name in the program.

* Macro coding: you can define code sequences in a program and they will be substituted for each occurrence of the macro name in the program using parameters supplied with the macro call.

1.1 EXECUTION OF THE APML ASSEMBLER

The APML assembler executes in the Central Processing Unit under the control of the operating system of either a CRAY X-MP Computer System or a CRAY-1 Computer System with an I/O Subsystem. It has no hardware requirements beyond those required for the minimum system configuration.

The assembler is loaded into Central Memory and begins executing as a result of an invocation statement. Parameters specify characteristics of an assembler run such as the dataset containing source statements and list output.

The source statements may comprise more than one APML program module. The assembler assembles each program module as it is encountered on the source dataset. The assembler makes two passes for each program module to be assembled. During the first pass, the assembler reads each source language statement instruction, expands sequences such as macro instructions, generates the machine function codes, and assigns memory. The assembler also breaks instruction sequences into groups of instructions called *pages* during Pass 1. The assembler then optimizes code within a page. For instance, all jumps within a page are optimized to single-parcel jump instructions; jumps outside the page are 2-parcel instructions. During the second pass, the assembler assigns block origins, substitutes values for symbolic operands and addresses, and generates the object code and an associated listing.

1.2 CONVENTIONS

This manual uses the following conventions:

Convention	Description
Italics	Indicates variable information supplied by the operator
Boldface	Identifies UNICOS command verbs, directory names, or file names
dataset	Refers both to COS datasets and UNICOS files
Choice 1 Choice 2	Stacked items indicate two or more literal options when only one choice may be used

2. APML ASSEMBLER LANGUAGE

This section presents general rules and statement syntax for APML programs.

2.1 SOURCE LINE FORMAT

An APML source statement consists of one to eight source lines. A source line is a maximum of 90 characters. The entire line is recorded in the list output dataset generated during an APML assembly. The assembler interprets only the first 72 columns of a line. A maintenance utility program uses remaining character positions for sequencing information.

A comma in column 1 indicates a continuation line. Columns 2 through 72 are then a continuation of the previous line. Up to seven continuation lines are allowed for source statements. Statements generated by APML in a MACRO or DUP expansion can have any number of continuation lines.

2.2 STATEMENT FORMAT

Statement format is essentially free-field. APML supports three types of statements: a comment statement, a symbolic APML instruction, and a pseudo instruction.

2.2.1 COMMENT STATEMENT

An asterisk as the first nonblank character indicates a comment statement. The assembler lists comment statements, but they have no other effect.

2.2.2 SYMBOLIC APML INSTRUCTION FORMAT

Each symbolic APML instruction consists of a location field, an assignment field, and a comment field as described in section 4, Symbolic APML Instruction Syntax.

A symbolic APML instruction is a statement that generates I/O Processor (IOP) instructions. Also included are certain instructions that generate data without the use of pseudo instruction mnemonic names. Section 4, Symbolic APML Instruction Syntax and section 5, Basic IOP Hardware Instruction Set, describe symbolic APML instructions.

2.2.3 PSEUDO INSTRUCTION FORMAT

Each pseudo instruction consists of a location field, an assignment field, a result field, an operand field, and a comment field. A mnemonic name is in the result field. Each field's contents are as follows:

Field	Contents Description
Location	Begins in column 1 of a line and is terminated by a blank. If column 1 is blank, the location field has no entry. The contents of the location field consist of a name or a symbol and depends upon the requirements of the result field or assignment field.
Assignment	Begins with the first nonblank character following the location field. It cannot begin before column 2 or after column 63. The assignment field has an entry if there are any nonblank characters between the location field and column 64. The assignment field is terminated by the comment field or the end of the statement.
Result	Begins with the first nonblank character following the location field. It cannot begin before column 2 or after column 63. A blank terminates the result field. The result field has an entry if there are any nonblank characters between the location field and column 64.
Operand	Begins with the first nonblank character following a nonempty result field and is terminated by the comment field or the end of the statement. The contents of the result field determine whether an entry is required in the operand field.

Field	Contents Description
Comment	Optional. Begins with a period. A period can appear in certain APML symbols; however, in such cases it is always preceded by a nonblank character. Therefore, it is conventional and good practice to precede the period at the beginning of the comment with a blank. The comment field may be the only field supplied in a statement.

Section 6, Pseudo Instructions, further describes pseudo instructions.

2.3 CODING CONVENTIONS

Although APML statements are essentially free-field, the conventions suggested here provide for a more uniform and more readable listing.

Beginning Column	Field
1	Blank, asterisk, comma, or location field entry left-justified
10	Result or assignment field entry, left-justified
20	Operand field entry, left-justified
35	Beginning of comment field

2.4 LINE EDITING

APML processes source statements sequentially from the source dataset. A macro definition is not immediately interpreted but is saved and interpreted each time it is called. Before interpreting a statement, APML performs two operations referred to as editing. These operations are concatenation and micro substitution.

2.4.1 CONCATENATION

APML examines each line for the underscore (concatenation) character and deletes it so that the two adjoining columns are linked before the statement is interpreted.

2.4.2 MICRO SUBSTITUTION

The APML assembler searches for double quotation marks ("), which serve to delimit micro names. The first " indicates the beginning of a micro name; the second " identifies the end of a micro name. Before a statement is interpreted, APML replaces the micro name by the character string comprising the micro.

2.5 NAMES

A name consists of from 1 to 8 characters. The first character of a name must be alphabetic (A through Z), a dollar sign ($), a percent sign (%), or an at sign (@). Characters other than the first may be decimal digits (0 through 9).

Use names to identify the following types of information:

- Program modules
- Blocks
- Macro instructions
- Micro character strings
- Conditional sequences
- Duplicated sequences
- Symbol qualifiers

Unlike symbols, a name does not have a value or an attribute associated with it and cannot be used in expressions.

Different types of names do not conflict with each other or with symbols. For example, a micro can have the same name as a macro and a program module can be named the same as a block.

2.6 SYMBOLS

A symbol is 1 to 8 characters that identifies a value and its associated attributes. The first character of a symbol must be alphabetic (A through Z), a dollar sign ($), a percent sign (%), or an at sign (@). Characters other than the first may also be decimal digits (0 through 9).

2.6.1 SYMBOL DEFINITION

The process of associating a symbol with a value and attributes is known as symbol definition. This can occur in a number of ways.

A symbol used in the location field of a symbolic APML instruction or certain pseudo instructions is defined as an address having the current value of the location counter and having attributes of parcel address, word address, relocatable, or absolute.

A symbol used in the location field of a symbol defining pseudo instruction is defined as having the value and attributes derived from an expression in the operand field of instruction. The type of symbol defining pseudo instruction used may cause the symbol to have an attribute of redefinability. When a symbol is redefinable, a second attempt to define it must be through use of a redefinable pseudo, which causes the symbol to be assigned a new value and attributes.

A symbol defined in a program module other than the module being currently assembled can be defined as having the attribute of external in the current program module. The true value of an external symbol is not known within the current program module.

2.6.2 SYMBOL ATTRIBUTES

Two or more attributes are assigned to a symbol when it is defined. Possible attributes are as follows:

- Word address, parcel address, or value

  Each symbol is assigned an attribute of word address, parcel address, or value. A word is a 64-bit quantity; a parcel is a 16-bit quantity. A symbol is assigned a word address attribute if it appears in the location field of a pseudo instruction such as VWD, CON, BSS, or BSSZ which defines words or if it is equated to an expression having a word-address attribute.

A symbol is assigned a parcel-address attribute if it appears in the location field of a symbolic APML instruction or certain pseudo instructions.

A symbol has a value attribute if it does not have a word-address or parcel-address attribute. A 64-bit value is associated with such a symbol.

- Relocatable, external, or absolute

  Each symbol is assigned the attribute of relocatable, external, or absolute.

  A symbol is assigned an attribute of relocatable if it appears in a relocatable assembly in the location field of a machine instruction, BSS pseudo instruction, or data generation pseudo instruction such as BSSZ, CON, and so on. A symbol is also relocatable if it is equated to an expression that is relocatable.

  A symbol is assigned the attribute of external if it is defined by an EXT pseudo instruction. An external symbol defined in this manner has a value attribute and a value of 0. A symbol is also assigned the attribute of external if it is equated to an expression that is external. Such a symbol assumes the value of the expression and may have an attribute of parcel address, word address, or value.

  A symbol is assigned the attribute of absolute in a relocatable assembly if it is neither relocatable nor external. In an absolute assembly, symbols that would be relocatable in a relocatable assembly are assigned the attribute of absolute. An exception occurs when the absolute program module is divided into local blocks through use of BLOCK pseudo instructions. In this case, symbols defined in local blocks other than the initial (nominal) block are assigned an attribute of relocatable during Pass 1 and absolute during Pass 2. See subsection 6.7, Block Control Pseudo Instructions, for more information on block control.

- Redefinable

  In addition to its other attributes, a symbol is assigned the attribute of redefinable if it is defined by the pseudo instructions SET or MICSIZE. A redefinable symbol may be defined more than once in a program module and may have different values and attributes at different times during as assembly. When such a symbol is referenced, its most recent definition is used by the assembler.

2.7 SYMBOL REFERENCE

The occurrence of a symbol in a field other than the location field constitutes a reference to the symbol and causes the value and attributes of the symbol to be used in place of the symbol.

A symbol may generally be referenced anywhere in the program. However, certain symbol references require that the symbol be previously defined. In such cases, APML generates an undefined error even though the symbol is defined later in the program. A symbol occurring in any expression in a pseudo instruction, except the data expression fields of CON, VWD, and ERRIF, must refer to previously defined symbol.

A symbol used in the location field of a symbolic APML instruction is not defined until an instruction *page boundary* occurs. APML does code optimization within an instruction page, so instruction address symbols are not defined until all instruction generation is fixed at the next *page boundary*. See section 4, Symbolic APML Instruction Syntax, for more details about *page boundary* conditions.

A symbol reference may contain a prefix, such as W. or P., which causes the usual value and attributes associated with the symbol to be altered according to the prefix. The prefix affects only the specific reference with which it occurs. See subsection 2.11, Prefixed Symbols and Constants, for details.

2.7.1 QUALIFIED SYMBOLS

You can render a symbol other than a global symbol unique to a code sequence by specifying a symbol qualifier to be appended to all symbols defined within the sequence. The option to qualify symbols is initiated by one QUAL pseudo instruction and terminated by the next. If a symbol defined in the sequence is referred to from within the sequence, it can be referred to without qualification. If, however, the symbol is referred to from outside of the code sequence in which it was defined, it must be referred to in the form */qualifier/symbol,* where *qualifier* is a 1- to 8-character name and is defined through the use of a QUAL pseudo instruction.

2.8 GLOBAL DEFINITIONS

Before the first IDENT pseudo instruction and between program modules (that is, between the END pseudo that terminates one program module and the IDENT that begins the next program module), APML recognizes sequences of instructions that do not generate code but define symbols, macro instructions, and micros.

Definitions occurring prior to an IDENT pseudo instruction are considered global and can be referred to without redefinition from within any of the program modules that occur subsequent to the definition. Redefinable symbols and symbols of the form %%xxxxxx, where x is any nonblank character, represent an exception; while they can occur in such sequences, they are local to the program module that follows and are not known to the assembler after the next END pseudo instruction is encountered. Global symbols cannot be qualified.

2.9 SPECIAL ELEMENTS

The following designators can occur as elements of expressions and have special meaning to the assembler.

Designator	Description
*	Denotes a value equal to the current location counter with parcel-address attribute and absolute or relocatable attribute depending on type of assembly
*O	Denotes a value equal to the current value of the origin counter with parcel-address attribute and absolute or relocatable attribute
*P	Denotes a value equal to the current value of the parcel-bit-position counter with absolute and value attributes
*W	Denotes a value equal to the current value of the word-bit-position counter with absolute and value attributes

****

CAUTION

The special elements *, *O and *W when used as location or origin address counters, should not be used except in the expression field of CON, VWD, and ERRIF. When used elsewhere, the value of these special elements is required in Pass 1 but not defined until Pass 2. These elements may be used, however, after an instruction page boundary, such as when a new page is forced by a NEWPAGE pseudo instruction and preceding any executable APML symbolic instructions.

****

Subsection 2.12.3, Elements, describes expression elements. Section 6, Pseudo Instructions, describes counters.

2.10 DATA NOTATION

Data is presented in the form of numeric or character constants and data items. Numeric values and character strings are presented to the assembler based on the following notation.

( )	Indicates optional information
[ ]	Indicates required information

2.10.1 NUMERIC CONSTANTS

You can express a numeric constant in integer notation. An integer constant has the following format:

(*prefix*) [*integer*] (*binary scale*)

prefix The numeric base used for the *integer, fraction, decimal exponent,* and *binary scale.* If no prefix is used, base is determined by the default mode of the assembler or by the BASE pseudo instruction. *prefix* can be one of the following:

- O' Octal (default)
- D' Decimal
- X' Hexadecimal

integer and/or *fraction*
A nonempty string of digits as required by *prefix*

binary scale
Indicates that the *integer* and/or *fraction* is to be multiplied by a power of 2

An integer constant is evaluated as a 64-bit twos complement integer.

Example:

Location	Result	Operand	Comment
1	10	20	35
NUMBER	EQUALS	O'50	
	CON	D'300	
	VWD	40/0,D'24/ADDR	
	A = O'177752		
	CON	1S63	.sign bit

2.10.2 CHARACTER CONSTANTS

Character constants are expressed using the following notation:

(*prefix*) ['*character string*'] (*suffix*)

prefix Character set used for stored constant:

- A ASCII character set (default)
- C Control Data Corporation (CDC)® Display Code
- E EBCDIC character set

character string

A string of 0 or more characters from the ASCII character set. Two consecutive apostrophes (excluding the delimiting apostrophes) indicate a single apostrophe. See appendix A, Character Sets.

suffix

Indicates justification and fill of character string:

- H Left-justified, blank-fill (default except in APML symbolic data generation instructions and PDATA)
- L Left-justified, zero-fill
- R Right-justified, zero-fill
- Z Left-justified, zero-fill, at least one trailing binary zero character guaranteed (default on strings used in APML symbolic instructions, data generation, and PDATA)

Example:

Location	Result	Operand	Comment
1	10	20	35
	CON	A'ABC'L	
	VWD	24/'OUT'	.(Default to H suffix - blank fill)
	DD = 'AB'		.(Default to Z suffix - blank fill)

2.10.3 DATA ITEMS

You can use a data item in the operand field of the PDATA, DATA, CON, and VWD pseudo instructions and in APML symbolic data generation instructions. The length of the data field occupied by a data item is determined by its type, size, and where it is used.

An integer data item has the following format:

(*sign*) (*prefix*) [*integer*] (*binary scale*)

If you use an integer data item in a PDATA pseudo instruction or APML symbolic data generation instruction, it generates 1 parcel (16 bits); in a DATA pseudo, it generates 1 word (64 bits).

A character string data item has the following format:

(*prefix*) ['*character string*'] (*count*) (*suffix*)

In the preceding notation, descriptions given for numeric and character constants apply. The two added options, *sign* for numeric data items and *count* for character string data items, have the following significance:

sign Data item is to be stored ones or twos complemented or uncomplemented; can only be used in a DATA or PDATA pseudo instruction.

+ or omitted	Uncomplemented
-	Negated (twos complemented)
#	Ones complemented; allowed on integer constants only.

count The length of the field in number of characters into which the data item is to be placed. If no *count* is supplied, the length is the number of words or parcels needed to hold the character string. If a count field is present, the length is the character count times the character width, so that the field length is not necessarily an integral number of words or parcels. The character width is 8 bits for ASCII or EBCDIC and 6 bits for CDC Display Code.

If an asterisk is in the count field, the actual number of characters in the string is used as the count. A single character is counted when two apostrophes represent a single apostrophe.

If the base is M (mixed), APML assumes that the count is decimal. See section 6, Pseudo Instructions, for a description of mixed base.

Example:

Location	Result	Operand	Comment
1	10	20	35
	DATA	'ERROR IN DSN'	
	DATA	-D'1.5E2	
	DATA	+0'20	
	VWD	40/0,24/0'200	

2.11 PREFIXED SYMBOLS AND CONSTANTS

A symbol, constant, or special element may be prefixed by a P. or a W. to cause the value to assume an attribute of parcel or word address, respectively, in the expression in which the reference appears.

A prefix does not permanently alter the attribute of a symbol; the effect of a prefix is for the current reference only.

2.11.1 PARCEL ADDRESS PREFIX - P.

A symbol, special element, or constant may be prefixed by P. to specify the attribute of parcel address. If a symbol (*sym*) has the attribute of word address, the value of P.*sym* is the value of *sym* multiplied by 4. A P. prefix to a symbol with value attribute or to a constant does not cause the value to be multiplied by 4, but it can be used to assign the parcel address attribute.

Example:

Location	Result	Operand	Comment
1	10	20	35
ADDR	CON	P.ADDR	

2.11.2 WORD ADDRESS PREFIX - W.

A symbol, special element, or constant may be prefixed by W. to specify the attribute of word address. If a symbol (*sym*) has the attribute of parcel address, the value of W.*sym* is the value of *sym* divided by 4. A W. prefix to a symbol with value-address attribute or to a constant does not cause the value to be divided by 4, but it can be used to assign the word-address attribute to the symbol or constant.

Example:

Location	Result	Operand	Comment
1	10	20	35
	A0 = W.ADDR		
	A4 = W.BUFF+O'100		

2.12 EXPRESSIONS

Expressions are used in the operand field of many APML pseudo instructions. An expression consists of one or more terms joined by special characters referred to as adding operators. A blank or a comma terminates an expression. A term, in turn, consists of one or more elements joined by special characters referred to as multiplying operators. Thus, an expression can be diagrammed as follows:

add op_1 (optional)	$TERM_1$	add op_2	$TERM_2$. . .	add op_n	$TERM_n$	comma or blank

Any term in an expression can be diagrammed as follows:

$ELEMENT_1$	mult op_1	$ELEMENT_2$ . . .	mult op_m	$ELEMENT_m$

2.12.1 ADDING OPERATORS

An adding operator joins two terms or precedes the first term of an expression. The two adding operators are as follows:

+ Addition
- Subtraction

2.12.2 MULTIPLYING OPERATORS

A multiplying operator joins two elements. Multiplying operators are as follows:

* Multiplication
/ Division

Multiplication and divisions are performed first, followed by addition and subtractions.

2.12.3 ELEMENTS

An element is a symbol, constant, or special element. It may also be one of these preceded by a # complement operator. An element preceded by #, however, must be absolute.

Examples:

SIGMA	Symbol
*	Special element
*W	Special element
0'77S3	Numeric constant
A'ABC'R	Character constant

The attributes of elements are assigned by the use of SET or EQUALS to define the attributes or by implication when the element is used.

2.12.4 TERMS

A term is an element or two or more elements joined by multiplying operators. Only one relocatable or external element may occur in a term. The following rules apply:

- Two consecutive elements are illegal.
- The element to the right of a / must be an absolute element; that is, a constant or an absolute symbol or, in an absolute assembly, a special element as well.
- A term containing a / must have an attribute of absolute up to the point at which the / is encountered (see subsection 2.12.5, Term Attributes).
- Division by 0 produces an error.
- An external symbol, if present, must be the only element of the term and if preceded by an adding operator, that operator must be +.
- An element cannot be null; that is, two consecutive multiplying operators or a multiplying operator not followed by an element is illegal.

2.12.5 TERM ATTRIBUTES

Attributes assigned to a term depend on the elements and operators comprising the term.

Every term is assigned an attribute of either external, absolute, or relocatable. A term assumes the attribute of external if it consists of a single external symbol. A term assumes the attribute of absolute if it contains only absolute elements. A term assumes an attribute of relocatable if it contains one relocatable element and no external symbols.

Every term assumes an attribute of parcel address, word address, or value. The term attribute may vary as each element in the term is evaluated. The term's final attribute will be that in effect when the final (rightmost) element of the term is evaluated. As APML encounters each element in the left to right scan of a term, it assigns an attribute to the term based on the operator, if any, preceding the element, the attribute of any previous partial term, and the attribute of the element currently being evaluated.

In the following rules, consider that P, W, and V denote an element being incorporated into the term and having an attribute of parcel address, word address, or value, respectively. Consider, also, that *pterm*, *wterm*, and *vterm* denote the attribute of the partial term resulting from all elements evaluated prior to the current element. The following rules apply.

- Following evaluation of the element, a new partial term is assigned a parcel-address attribute if the partial term, operator, and new element are one of the following combinations:

  P
  *pterm**V
  pterm/V
  *vterm**P

- Following evaluation of the element, a new partial term is assigned a word-address attribute if the partial term, operator, and new element are one of the following combinations:

  W
  *wterm**V
  wterm/V
  *vterm**W

- Following evaluation of the element, a new partial term is assigned a value-address attribute if the partial term, operator, and new element are one of the following combinations:

  V
  *vterm**V
  pterm/P
  wterm/W
  vterm/V

- In addition, any of the following combinations results in an attribute of value being assigned but accompanied by a warning error.

  *pterm**W
  *wterm**P
  pterm/W
  wterm/P
  vterm/P
  vterm/W
  *pterm**P
  *wterm**W

2.13 EXPRESSION EVALUATION

Expressions are evaluated from left to right. Each term is evaluated from left to right, with APML performing 64-bit integer multiplication or division as each multiplying operator is encountered. When a complete term has been evaluated, it is added or subtracted from the sum of the previous terms.

The assembler treats each element as 64-bit twos complement integer. Character constants are left- or right-justified within a field width equal to the destination field. Complemented elements are complemented in the rightmost bits in a field width equal to the destination field.

A relocatable term has a 64-bit integer coefficient associated with it equal to the value of the term obtained when a 1 is substituted for the relocatable element. The value of a relocatable term is the value of the relocatable element multiplied by the coefficient.

The coefficient of each relocatable term is added or subtracted to the coefficient maintained for the corresponding relocatable block represented in the expression.

2.14 EXPRESSION ATTRIBUTES

The assembler can assign the following attributes to an expression:

- Relocatable, external, or absolute
- Parcel address, word address, or value

2.14.1 RELOCATABLE, EXTERNAL, OR ABSOLUTE

An expression is relocatable if the coefficient is 0 for every block represented in the expression, except for one block, which must have a coefficient of +1 (positive relocation). An expression error occurs if a coefficient does not equal 0 or +1, or if more than one coefficient is nonzero.

An expression is external if the expression contains one external term and if the coefficients of all relocatable blocks are 0. An expression error occurs if more than one external term is present.

An expression is absolute if no external terms are present and the coefficients of all relocatable blocks are 0.

2.14.2 PARCEL ADDRESS, WORD ADDRESS, OR VALUE

An expression has a parcel-address attribute if at least one term has parcel-address attributes and all other terms have a value- or parcel-address attribute.

An expression has word-address attribute if at least one term has word-address attribute and all other terms have value- or word-address attribute.

All other expressions have value-address attribute. A warning error occurs if an expression has terms with both word-address attribute and parcel-address attribute.

An expression value is truncated to the field size of the expression destination. A warning error occurs if the leftmost bits lost in truncation are not one of the following:

All zeros
All ones with the leftmost remaining bits also 1 (that is, a negative quantity)

A null (empty) expression is treated as an absolute value of 0.

If an error other than a warning error occurs in evaluating an expression, the expression is treated as an absolute value of 0.

Examples of expressions:

ALPHA	An expression consisting of a single term
*W+BETA	Two terms; *W and BETA.
GAMMA/4+DELTA*5	Two terms, each having two elements
0'100+=0'100	Two terms; a constant and the address of a literal.
MU-NU*2+*	Three terms, the first consisting only of MU, the second consisting of NU*2, and the third consisting only of the special element *

In the following examples, R and S are relocatable symbols in the same block, X and Y are external, and A and B are absolute. The location counter is currently in the block containing R and S.

The following expressions are relocatable:

*	
W.*+B	
R+2	
2**-R-S	2** cancels -R and -S
X+R	Error; external and relocatable.
R+S	Error; relocation coefficient of 2.
R/16*16	Error; division of relocatable element is illegal.

The following expressions are external:

X+2	
Y-100	
X+R*	R, -* cancels relocation
X+2**-R-S	Relocatable terms 2**, -R, and -S cancel each other
-X+2	Error; external cannot be negated.
X+Y	Error; more than one external.
X/Z	Error; division of an external element is illegal.

The following expressions are absolute:

A+B	
'A'R-1	
2*R-S-*	Relocation of terms all cancel
1/2*R	Equivalent to 0*R
A*(R-S)	Error; parentheses are not allowed.

2.15 TABLE METHOD OF EXPRESSION ATTRIBUTE EVALUATION

Tables 2-1 and 2-2 summarize evaluation of term attributes for absolute and relocatable assembly, respectively.

If a symbol, special element, or constant has the attribute of the left column and is added, subtracted, multiplied, or divided by a symbol, special element, or constant with the attribute of the top horizontal row, the resulting attribute is determined at the intersection by the arithmetic operator position in the upper left corner of the table.

Table 2-1. Absolute Assembly Element and Term Attribute Evaluation

<table>
<tr><th>+|-
*|/</th><th>V</th><th>P</th><th>W</th><th>2nd Term</th></tr>
<tr><td>V</td><td>V|V
V|V</td><td>P|P
P|Ve</td><td>W|W
W|Ve</td><td></td></tr>
<tr><td>P</td><td>P|P
P|P</td><td>P|P
Ve|V</td><td>Ve|Ve
Ve|Ve</td><td></td></tr>
<tr><td>W</td><td>W|W
W|W</td><td>Ve|Ve
Ve|Ve</td><td>W|W
Ve|V</td><td></td></tr>
<tr><td>First Term</td><td></td><td></td><td></td><td>V = Value
P = Parcel
W = Word
e = Warning message</td></tr>
</table>

Table 2-2. Relocatable Assembly Element and Term Attribute Evaluation

<table>
<tr><th>+|-
*|/</th><th>V</th><th>P</th><th>W</th><th>2nd Term</th></tr>
<tr><td>V</td><td>V|V
V|V</td><td>P|E
Ee|Ee</td><td>W|E
E|Ee</td><td></td></tr>
<tr><td>P</td><td>P|P
E|E</td><td>E|Pa
Ee|E</td><td>Ee|Ee
Ee|Ee</td><td></td></tr>
<tr><td>W</td><td>W|W
E|E</td><td>Ee|Ve
Ee|Ee</td><td>E|Wa
Ee|E</td><td></td></tr>
<tr><td>First Term</td><td></td><td></td><td></td><td>V = Value
P = Parcel
W = Word
E = Error message
e = Warning message
a = Absolute</td></tr>
</table>

3. APML INVOCATION AND EXECUTION

Load and execute APML using either the COS APML control statement or the UNICOS APML command line.

3.1 COS APML CONTROL STATEMENT

An APML control statement has the following format:

```
APML,CPU=type,I=idn,L=ldn,B=bdn,E=edn,ABORT,DEBUG,options,

  LIST=name,S=sdn,SYM=sym,T=bst,X=xdn.
```

Parameters are order-independent and none are required. Parameters are processed in the order they appear. If parameter specification is duplicated or contradictory, the last specification is used.

CPU=*type* Only IOP can be specified as *type*. The parameter is optional, since the default is also IOP.

I=*idn* Name of dataset containing source statement input. The default is $IN. APML reads source statements from dataset *idn* until an end-of-file (EOF) is encountered.

L=*ldn* Name of dataset onto which list output is written. The default is $OUT. APML writes one file of output. If L=0, no listing is written.

B=*bdn* Name of dataset to receive binary load data. The default is $BLD. APML writes binary load data to this dataset, one record per program module. An EOF is not written. If B=0, no binary load data is written.

E=*edn* Name of dataset on which error listing is written. The default is no error listing if the list output is on $OUT; otherwise, the default is $OUT. APML writes source statements containing errors to this dataset as one file. Simply specifying E causes an error listing to be generated on a dataset named $OUT. If the error dataset name *edn* is the same as the listing dataset name, list output is written.

ABORT — Abort mode. If this parameter is present and any fatal errors are encountered during assembly, APML aborts the job after assembling all program modules. If this parameter is omitted or if fatal errors are not encountered, APML exits normally and job processing continues with the next control statement in the job deck.

DEBUG — Debug mode. If this parameter is omitted and fatal errors occur in a program, APML writes a binary record containing only a Program Description Table (PDT) with the fatal error flag set. The loader ignores a program module with this flag set.

When the DEBUG parameter is present, APML writes a full binary record with the fatal error flag clear, whether or not fatal errors are encountered. The loader attempts to load and execute the module.

options — Listing control options. You can specify any of the following listing control options to enable or disable a listing feature. Brackets enclose the defaults. The selection of an option on the APML control statement overrides the enabling or disabling of the corresponding feature on a LIST pseudo instruction. See section 6, Pseudo Instructions, for the description of the LIST pseudo and for more details about these options.

Option	Description
[ON]	Enables source statement listing
OFF	Disables source statement listing
[XRF]	Enables cross-reference
NXRF	Disables cross-reference
[XNS]	Includes unreferenced local symbols in the cross-reference
NXNS	Does not include unreferenced local symbols in the cross-reference
[DUP]	Enables listing of duplicated statements
NDUP	Disables listing of duplicated statement
MAC	Enables listing of macro expansions
[NMAC]	Disables listing of macro expansions
MIF	Enables macro conditioning listing
[NMIF]	Disables macro conditional listing

options (continued)	[MIC]	Enables listing of generated statements before editing
	NMIC	Disables listing of generated statements before editing
	LIS	Enables listing of LIST pseudo instructions
	[NLIS]	Disables listing of LIST pseudo instructions
	[WEM]	Enables warning errors
	NWEM	Disables warning errors
	TXT	Enables global text source listing
	[NTXT]	Disables global text source listing
	[WMR]	Enables warning error message for macro redefinition
	NWMR	Disables warning error message for macro redefinition

LIST=*name* Name of LIST pseudo instructions to be processed. A LIST pseudo instruction with a matching location field name is not ignored. A LIST pseudo instruction with a nonblank location field name that does not match a name specified on the APML control statement is ignored. *name* can be a single name or can be a list of names separated by colon (for example, LIST=TASK1:TASK2:TASK7). If just LIST is specified, all LIST pseudo instructions are processed, regardless of the location field name.

S=*sdn* Name of dataset containing system text file. The default is $APTEXT. If S=0 is specified, no system text is used. *sdn* can be a single dataset name or can be a list of up to 10 dataset names separated by a colon (for example, S=$APTEXT:OURTXT:MYTXT). The system texts are processed in order of appearance.

SYM=*sym* Name of dataset where the optional symbol text is to be written. The default is no symbol table generated by APML. If just SYM is specified, the symbol text is written to the same dataset as the binary load data.

T=*bst* Binary system text. Specifies dataset where all global macros, symbols, and OPSYN assignments are written. The default, equivalent to specifying T=0, is no binary system text written. If T is specified alone, the binary dataset is written to $BST.

X=*xdn* Binary symbol table records for the global cross-reference generator SYSREF. Each record contains cross-reference information for the global symbols in one particular program unit. The default, equivalent to specifying X=0, is to write no global cross-reference records. If X is specified alone, the information is written to $XRF.

Example APML statement:

```
APML(I=$IN,E,ABORT)
```

This APML statement specifies that source statements are in $IN, errors are written to $OUT, list output is suppressed, binary load data is written to $BLD, the system text is in $APTEXT, and no binary system text is written. The job aborts if fatal errors are encountered.

COS APML invocation example:

```
JOB,JN=APMLJOB,T=150.
ACCESS,DN=$PL,PDN=IOPPL.
UPDATE,F.                        CREATE $CPL COMPILE DATASET CONTAINING 3 *
*.                               FILES.
APML,S=0,I=$CPL,T=$APTEXT,E.     ASSEMBLE BINARY SYSTEM TEXT, $APTEXT.
APML,I=$CPL,E.                   ASSEMBLE PROGRAMS ON SECOND FILE OF $CPL.
APML,I=$CPL,E.                   ASSEMBLE PROGRAMS ON THIRD FILE OF $CPL.
ADSTAPE.                         GENERATE DEADSTART TAPE BINARY DATASETS.
DISPOSE,DN=$DS,DC=ST.
DISPOSE,DN=$OVL,DC=ST.
/eof
  UPDATE directives
/eof
```

3.2 UNICOS APML COMMAND LINE

Under UNICOS, invoke APML using the following command line. All parameters are optional.

Format:

```
apml [-t bsys] [-r xref] [-g sym] [-l listing] [-m tmwords] [-L]

                                            -h
  [-s text1,text2,text3,. . . textn] [-i nlist] [-o binary] name.s
```

-t *bsys*	Names the output file to which APML writes the binary system text. There is no default.
-r *xref*	Names the output file to which APML writes the binary cross-reference file. Default is no cross-reference file.
-g *sym*	Names the output file to which APML writes the Symbol Table. Default is no Symbol Table.
-l *listing*	Names the output file to which APML writes the assembler listing. The default is no listing.
-m *tmwords*	Specifies an integer number of memory words to be reserved for the table manager work area. Default is 65476 words.
-L	L requests that the amount of excess work area to be reported and statistical logfile messages to be sent to stderr. Statistics reported include the assembler's name, assembly time, and so on. The amount of excess work area is reported as 'UNUSED: *nnnnn*'. It should not be necessary to increase the work area except on very large assemblies, such as I/O Subsystem. The work area is not expandable at run time, so if sufficient space is not preallocated with the -m option for the assembly to complete, APML aborts.
-s $text_n$	Any number of system texts; must be separated by commas.
-h	When specified, all list pseudos are processed regardless of the location of the field name.
-i *nlist*	Specifies processing of those list pseudos whose location field names are specified by *nlist*. (*nlist* can be a single name or a list of names separated by commas.)
-o *binary*	Names the binary object file. The default is *name.o* if *name.s* is the input.
name.s	Specifies the file containing the assembler source code.

APML writes warning and error logfile messages (or diagnostic and statistic messages if requested with -L) to **stderr**.

UNICOS APML invocation example:

```
apml -t aptext -m 150000 apt.s
apml -s aptext -m 150000 src.s
apml -s aptext -m 200000 src2.s
cat src.o src2.o > big.o
adstape < big.o
```

3.3 SYSTEM TEXT

System text allows for definition of global macros and commonly used symbols. These macros and symbols are defined in a system text separate from your source statement input. This is assembled before your source. All global definitions contained in the system text are preserved for reference in your programs.

System text symbols referenced by you are identified in the cross-reference listing by the system text dataset name.

System text may contain any APML statements allowed in normal source input. Typically, however, a system text would consist of macro and symbol definitions followed by an IDENT and END pseudo. While assembling system text, APML suppresses writing binary load data and list output, except for statements containing errors.

IDENT and END pseudos are not required at the end of a system text, but their presence facilitates assembling the system text separately as a program module for the purpose of obtaining a listing.

3.4 BINARY SYSTEM TEXT

A binary system text is a preassembled version of a source system text. A binary system text is generated by the T option (COS) or the -t option (UNICOS) on the APML invocation statement. When T or -t is all global macros, symbols, and OPSYN assignments are written to the specified dataset in an internal APML format.

NOTE

Use of binary system text generally reduces assembly time.

This dataset can thereafter be used with the S option, as if the source system text were being used. APML determines whether a system text is in source or in binary format.

Under COS, when multiple system texts are used, binary and source versions can be freely mixed. The effect is as if all of the source versions were present.

Under UNICOS, use only binary format system text with the -s option.

COS examples:

```
1.  APML,I=SOURCE1,S=0,T=BINARY1.
2.  APML,I=SOURCE3,S=0,T=BINARY3.
3.  APML,I=MYPROG,S=BINARY1:BINARY3.
```

UNICOS examples:

```
1.  apml -t binary1 < source1
2.  apml -t binary3 < source3
3.  apml -s binary1 binary3 < myprog
```

In examples 1 and 2, binary versions of source system texts SOURCE1 and SOURCE3 are created.

Under COS, if S=0 had not been specified, APML would have assembled $APTEXT by default; the global macros and symbols in $APTEXT would have been copied into the binary system texts being generated. Under UNICOS, no default is provided.

In example 3, the binary texts generated by examples 1 and 2 are used. The effect is as if the following statement had been written instead of example 3:

COS:

```
APML,I=MYPROG,S=SOURCE1:SOURCE2:SOURCE3.
```

UNICOS:

```
cat source1 source2 source3 myprog > bigsource
apml < bigsource
```

4. SYMBOLIC APML INSTRUCTION SYNTAX

Symbolic APML instructions generate I/O processor (IOP) instruction parcels or data parcels. Each symbolic APML instruction may generate one or more IOP instructions or data parcels.

Those familiar with the IOP instruction set can use a subset of symbolic APML instructions that generate single IOP instructions. You can also use more complex symbolic APML instructions that generate multiple hardware instructions to simplify your task.

In symbolic APML instruction notation, certain symbols are reserved to represent IOP registers and memory. Special characters are used as operators to represent arithmetic and logical operations, conditional branch conditions, data movement, and other functions.

4.1 OPERAND NOTATION

The following reserved names represent the contents of IOP registers or memory:

Name	Description
A	Accumulator
B	Operand register, index register (B register)
(B)	Contents of the operand register addressed by B
C	Carry flag
E	Exit stack pointer
(E)	Exit stack entry addressed by E, the exit stack pointer
I	Interrupt Enable flag
P	Program address register
R	Return jump program address
R!*sym*	Operand register whose index is the value of the symbol *sym*, where *sym* is any symbol with positive absolute value less than 512.

Name	Description
dd	Operand register whose index is the value of the symbol *dd*, where *dd* is a 2 character symbol with positive absolute value less than 512
[*dd*]	Value of symbol *dd*; that is, index of register represented by register symbol *dd*.
(*dd*)	Memory parcel addressed by contents of operand register *dd*
k	An unsigned numeric constant, character constant, or a symbol. In general, *k* may have a positive or negative value with absolute value less than 16,384. In some cases, the range of values for *k* is further restricted.
d	An unsigned numeric constant, character constant, or a symbol. In general, *d* may have a positive or negative value with absolute value less than 512. In some cases, the range of values for *d* is further restricted.
(*k*)	Memory parcel addressed by the value of *k*
(*dd*+*k*)	Memory parcel addressed by the sum of the contents of operand register *dd* and constant *k*

NOTE

Instructions referencing the operand register *dd* contain the register index in the *d* field, the lower 9 bits of the instruction parcel.

The following reserved names represent other operands used in symbolic APML instructions:

Name	Description
IOB	I/O channel reference using the contents of the B register as the channel designator
iod	I/O channel reference, where the value of symbol *iod* is the channel designator. Symbol *iod* must be defined by the CHANNEL pseudo instruction. Conventionally, *iod* is a 3-character symbol.
BZ, DN	IOP channel status. A channel busy flag, BZ, and done flag, DN, may be tested with certain instructions.

Name	Description
EXIT	Name of subroutine return function, which generates an IOP instruction which exits from a subroutine
WAIT	Name of branch function which loops until a test condition is satisfied
PASS	Name of function which generates an IOP pass or no-operation instruction

4.2 OPERATORS

The following characters are used in symbolic APML instructions as operators with special significance in the instruction syntax.

4.2.1 REPLACEMENT OPERATOR

The replacement operator, =, indicates that the subject to left of the equal sign is to be replaced by the value generated on the right side.

4.2.2 FUNCTION OPERATORS

The function operators are as follows:

Operator	Description
+	Addition
-	Subtraction
&	Logical product
>	Right shift, end off
<	Left shift, end off
>>	Right shift, circular
<<	Left shift, circular

4.2.3 RELATIONAL OPERATORS

Relational operators are used in a conditioned clause. The subject to the left of the operator is compared with the value generated on the right side according to the relation implied by the operator:

Operator	Description
=	Equal
#	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

4.2.4 CONDITIONAL OPERATOR

A comma (,) is the conditional operator. It introduces a conditioned clause. The assignment clause to the left of the comma is executed only if the conditional clause to the right is evaluated as true.

4.3 PROGRAM STATEMENT INSTRUCTION FORMAT

Symbolic APML instructions fall into two classes, program statements and data generation statements. The program statements described in this subsection generate one or more IOP hardware instructions. For more information on data generation statements, see subsection 4.4, Data Generation Statement Instruction Format.

Program statements have one of the following general formats:

Location	Assignment	Comment
label	*assign*	*.comment*
label	*assign, condition*	*.comment*
label	*	*.comment*

label — Optional label, which must be a valid symbol. The label is defined as the value of the location counter. Because APML does code optimization within an instruction page, the label is not defined until an instruction page boundary occurs.

assign — The assignment clause, represented in this section by *assign*, is required and is either a replacement, jump, set flag, special function, or channel function.

condition — The *condition* clause is optional and is seperated from the assignment clause by a comma. The *assign* clause is executed only when the *condition* clause is true. The *condition* clause represents a test of or comparison of the contents of IOP registers and memory parcels. Any assignment clause may be followed by any conditional clause.

comment — The *comment* is introduced by a period. A period can appear in certain APML operands appearing in the *assign* or *condition* clause, however, in such cases it is always immediately preceded by a nonblank character. Therefore, it is conventional to precede the period with a blank at the beginning of a comment.

* — When an asterisk appears in the assignment field, no data or instructions are generated. The statement serves to define the optional label without also generating code.

4.3.1 ASSIGNMENT CLAUSES

This subsection describes the five types of assignment clauses.

4.3.1.1 Replacement assignment

A replacement clause enters the register or memory parcel designated by the subject with the value expressed on the right side of the equals sign.

The right side of the assignment clause consists of operands and binary operators. Evaluation proceeds strictly from left to right. There is no hierarchy of operators and no grouping of terms other than that implied by their order.

Location	Assignment	Comment
label	*subject* = $operand_1$	.*comment*
	or	
label	*subject* = $operand_1$ op_1 $operand_2$	.*comment*
	or	
label	*subject* = $operand_1$ op_1 $operand_2 \ldots op_{n-1} operand_n$	.*comment*

subject The subject of an assignment clause may be any of the following:

Register	A, B, E, or (E)
Operand register	*dd* or (B)
Memory parcel	(*dd*),(*dd*+*k*), or (*k*)

$operand_i$ The operands may be any of the following:

Register	A, B, E, or (E)
Operand register	*dd* or (B)
Memory parcel	(*dd*),(*dd*+*k*), or (*k*)
Constant	*k*

Additional rules concerning operands in the assignment clause are as follows:

- The accumulator, A, exit stack-pointer, E, and exit stack entry, (E), may only appear as the first operand, $operand_1$.
- After the shift operator (>, <, >>, <<), only the index register B or constant *k* may appear.

op_i The operator may be any of the following characters representing IOP operations:

Arithmetic	+ or -
Logical	&
Shift	>, <, >>, or <<

In general, the assignment clause generates IOP instructions to load the accumulator with $operand_1$. After that, the additional operations indicated in the subsequent operators and operands are performed using the accumulator and carry flag to hold intermediate results. The resulting accumulator value is stored in the register or memory parcel indicated by the subject.

If (*dd+k*) or (*k*) appear anywhere in the assignment, scratch registers are used to preload the memory address before loading the first operand.

****

CAUTION

The accumulator, A, and carry flag, C, should not, in general, be explicitly used. The APML assembler uses these registers to execute the assignment and condition clauses, and they may be used by APML when not immediately obvious to you.

****

Example:

```
A=EE
(MN+5)=A
```

should be written

```
(MN+5)=EE
```

so that the accumulator is not destroyed when forming the value of MN+5 in the accumulator before storing the address in a scratch register.

Example:

```
|Code generated            |Location |Result    |Operand        |Comment
|                          |1        |10        |20             |35
|                          |         |          |               |
|                          |         |IDENT     |TRY            |
|        231               |R0       |EQUALS    |231            |
|        235               |R6       |EQUALS    |235            |
|                          |         |SCRATCH   |R0,R6          |
|          7               |R1       |EQUALS    |7              |
|         10               |R2       |EQUALS    |10             |
|         11               |R3       |EQUALS    |11             |
|         12               |R4       |EQUALS    |12             |
|         13               |R5       |EQUALS    |13             |
|                          |*        |          |               |
|                          |*        |          |               |
|                          |*        |          |               |
|                          |* A LOST IN ASSIGNMENT              |
|010007                    |         |A=7       |               |
|020007  012025  024231    |         |B=A+(R1+25)               |
|032231  054000            |         |          |               |
|                          |*        |          |               |
|                          |* A LOST IN CONDITION               |
|010010                    |         |A=10      |               |
|020011  013037  102002    |         |B=A,R3#37 |               |
|054000                    |         |          |               |
|                          |*        |          |               |
|                          |* PROGRAMMER ATTEMPTING TO USE      |
|                          |* P AS AN OPERAND     |               |
|070000                    |         |P=P+2     |               |
|050000                    |         |A=B       |               |
|054000                    |CAT      |B=A       |               |
|                          |*        |          |               |
|                          |* ASSEMBLER USING P AS AN OPERAND   |
|070002                    |         |P=DOG     |               |
|050000                    |         |A=B       |               |
|054000                    |DOG      |B=A       |               |
|                          |*        |          |               |
|                          |* SAMPLE STATEMENTS   |               |
|                          |*        |          |               |
|                          |*   SAMPLE ASSIGNMENT STATEMENTS    |
|000334                    |LOC      |334       |               |
|020013  062000  004010    |         |A=R5+(B)>10&B             |
|051000                    |         |          |               |
|014000 /000021  024231    |         |(LOC)=E+R3-(BOG)          |
|014000 /000044  024235    |         |          |               |
|150002  022011  033235    |         |          |               |
|034231                    |         |          |               |
|074012                    |         |P=R4      |               |
```

4.3.1.2 Jump assignment

You can use two formats for the jump assignment: a jump using P and a return jump using R for subroutine calls.

Replacement of P alters the current instruction sequence. Execution continues at the address specified by the new value of P.

Replacement of R suspends execution of the current instruction sequence and begins execution of a subroutine at the specified address. The E register is incremented by 1. The address of the next sequential instruction parcel is entered in the exit stack. If 16_8 is entered in E, an IOP interrupt is generated.

Location	Assignment	Comment
label	P = *address*	.*comment*
label	R = *address*	.*comment*

address The jump destination address may be any of the following operands:

- Operand register *dd*
- Constant *k* (typically a program label)

  The use of *k* as a jump address is restricted to symbols. Numeric and character constants are not allowed.

- Operand register + constant *dd* + *k*

  The use of an operand register + constant, *dd* + *k*, is not allowed if you defined a base register with a BASEREG pseudo instruction. This example would be ambiguous because, in this case, you have asked APML to form all 2-parcel jumps with IOP jump instructions of the form P = R! *basereg* + *k*, not P = *dd* + *k*.

4.3.1.3 Set flag assignment

You may set the carry flag or system interrupt enable flag to 0 or 1 with a set flag assignment clause.

Location	Assignment	Comment
label	C = 0	.*comment*
label	C = 1	.*comment*
label	I = 0	.*comment*
label	I = 1	.*comment*

4.3.1.4 Special function

Certain names are reserved for special APML instructions.

Location	Assignment	Comment
label	PASS	.*comment*
label	EXIT	.*comment*
label	WAIT	.*comment*

PASS — Generates an IOP no-operation or a pass instruction

EXIT — Generates a subroutine exit instruction. If the E register contains a 0, the IOP interrupt is generated; otherwise, the address stored in the program exit stock entry indicated by the E register is entered in the P register. The contents of the E register are decremented by 1.

WAIT — Generates code to wait for the conditional clause to be true. If no conditional clause is present, the program loops forever at the current instruction parcel address.

4.3.1.5 Channel function

A channel function assignment clause generates an IOP channel function instruction. The channel for which the instruction is to be performed is indicated either by a channel mnemonic symbol, *iod*, or by the contents of the B register.

Location	Assignment	Comment
label	*iod*:*k*	.*comment*
label	IOB:*k*	.*comment*

iod — Channel mnemonic symbol defined by a CHANNEL pseudo instruction. The value of the symbol *iod* is stored in the low-order 9 bits of the IOP channel function instruction.

k Channel function, a constant *k* with a positive absolute value less than 20_8. The value of *k* is added to 140_8 or 160_8 to form the IOP instruction operation code for the *iod:k* or IOB:*k* instruction, respectively.

IOB Indicates the contents of the B register is to be used as the index of the channel to be functioned.

4.3.2 CONDITION CLAUSES

The condition clause is optional in an APML program statement. The assignment clause is executed only when the condition clause represents a true condition.

This subsection describes the four condition clauses.

4.3.2.1 Test accumulator

This clause compares the accumulator contents with the contents of a register, memory parcel, or a constant.

Location	Assignment	Comment
label	*assign*,A *rel operand*	.*comment*

rel A relational operator (=, #, >, <, >=, or <=)

operand The operand may be any of the following:

Register	B
Operand register	*dd* or (B)
Memory parcel	(*dd*)
Constant	*k*

In general, the code for a condition clause is generated before the assignment clause. The indicated condition is then tested and a jump is generated around the assignment clause if the condition is false.

In the test accumulator clause, the carry bit is cleared. The operand is subtracted from the accumulator and a jump is generated around the assignment clause if the relation is false.

4.3.2.2 Test register or memory

This clause compares the contents of a register or memory parcel with the value expressed by the operands and operators on the right side of the relation.

```
|Location|Assignment                                         |Comment
|        |                                                   |
|label   |assign, subj rel operand1,                         |.comment
|        |                    or                             |
|label   |assign, subj rel operand1 op1 operand2             |.comment
|        |                    or                             
|label   |assign, subj rel operand1 op1 operand2 ... op(n-1) operandn
```

subj The subject (*subj*) of the condition may be any of the following:

Register	B, E, or (E)
Operand register	*dd* or (B)
Memory parcel	(*dd*), (*dd+k*), or (*k*)

rel A relational operator (=, #, <, >, <=, or >=)

$operand_i$ The operands may be any of the following:

Register	A, B, E, or (E)
Operand register	*dd* or (B)
Memory parcel	(*dd*),(*dd+k*), or (*k*)
Constant	*k*

Additional rules concerning operands in the assignment clause are as follows:

- The accumulator, A, exit stack-pointer, E, and exit stack entry, (E), may only appear as the first operand, $operand_1$.
- After the shift operator (>, <, >>, or <<), only the index register B or constant *k* may appear.

op_i The operator may be any of the following characters representing IOP operations:

Arithmetic	+ or -
Logical	&
Shift	>, <, >>, or <<

The value represented on the right side of the relation is evaluated in the same manner as the right side of a placement assignment clause.

4.3.2.3 Test carry flag

This clause tests the value of the carry flag for a zero or one.

Location	Assignment	Comment
label	*assign*,C=0	.*comment*
label	*assign*,C=1	.*comment*
label	*assign*,C#0	.*comment*
label	*assign*,C#1	.*comment*

4.3.2.4 Test channel status

This clause tests the state of the busy or done flag for a channel indicated by a channel mnemonic *iod* or by the contents of the B register.

Location	Assignment	Comment
label	*assign, iod relstate*	.*comment*
label	*assign,* IOB *relstate*	.*comment*

iod Channel mnemonic symbol

relstate Channel flag state:

=BZ	Channel busy flag set
=DN	Channel done flag set
#BZ	Channel busy flag clear (not busy)
#DN	Channel done flag clear (not done)

4.3.3 SYNTAX GRAPHS FOR APML PROGRAM STATEMENTS

Figures 4-1 and 4-2 graphically represent the rules for forming APML program statements.

Replacement

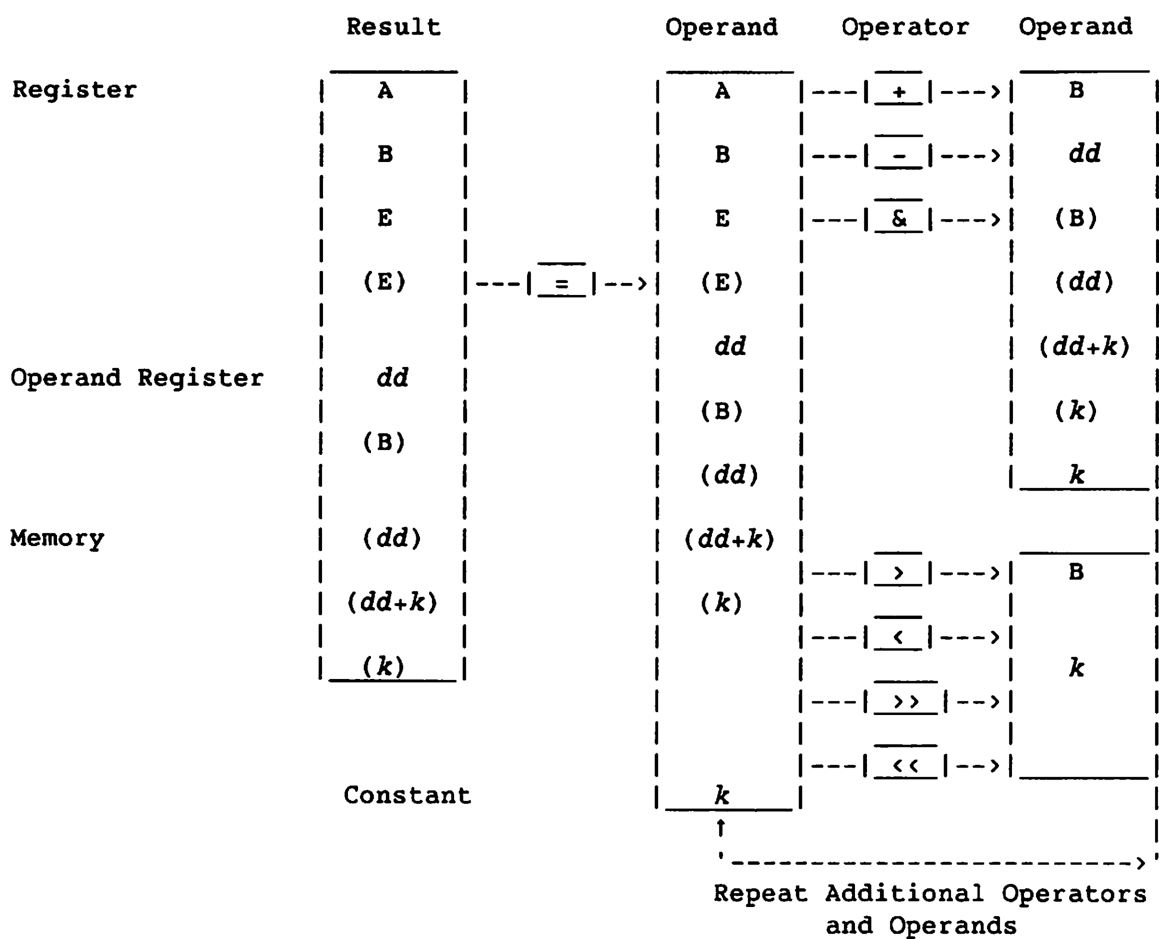


Figure 4-1. Assignment Syntax

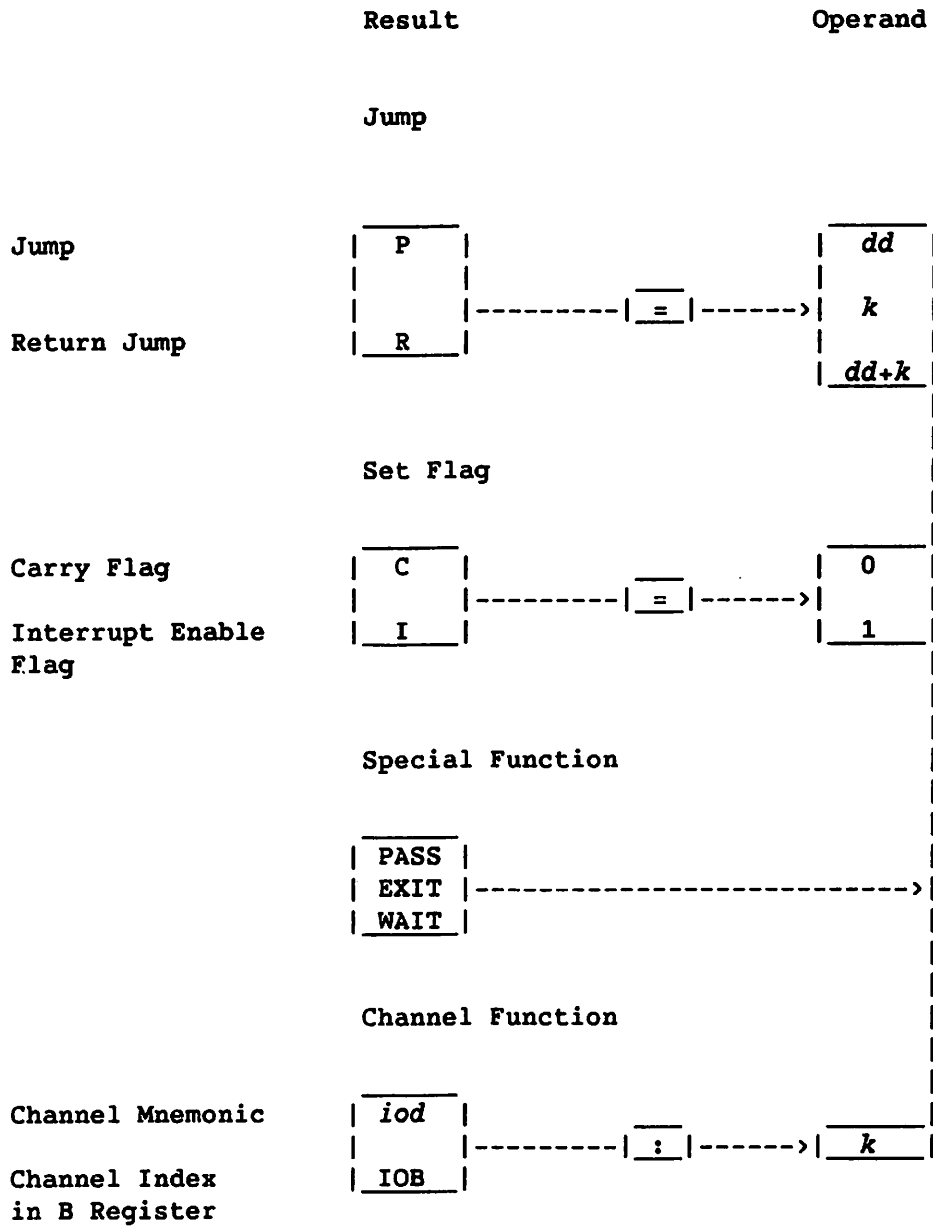


Figure 4-1. Assignment Syntax (continued)

Test Register or Memory

```
               Subject     Relation    Operand    Operator     Operand
              ___________      ___      _________     ___      _________
Register     | ,B        |---| = |--->| A        |---| + |--->| B        |
             |           |          |          |          |          |
             | ,E        |          | B        |          | dd       |
             |           |     ___  |          |     ___  |          |
             | ,(E)      |---| # |--->| E        |---| - |--->| (BB)     |
             |           |          |          |          |          |
             |           |          | (E)      |          | (dd)     |
             |           |     ___  |          |     ___  |          |
Operand      | ,dd       |---| > |--->| dd       |---| & |--->| (dd+k)   |
Register     |           |          |          |          | (k)      |
             | ,(B)      |          | (B)      |          |_k________|
             |           |     ___  |          |     ___   _________
             |           |---| < |--->| (dd)     |---| > |--->|          |
             |           |          |          |     ___  |    B     |
Memory       | ,(dd)     |          | (dd+k)   |---| < |--->|          |
             |           |     ____ |          |     ____ |          |
             | ,(dd+k)   |---| >= |-->| (k)      |---| >> |-->|          |
             |           |          |          |     ____ |    k     |
             | ,(k)      |         /|_k________|---| << |-->|__________|
             |           |        /     ↑                            |
             |           |    ___/      |                            |
             |           |   | <= |     |   Repeat Additional        |
             |           |  /           | Operators and Operands     |
             |           | /            '----------------------->---'
Constant     | ,k        |/
             |___________|
```

Figure 4-2. Condition Syntax

```
                              Subject     Relation    Operand

                                  Test Accumulator

Accumulator                    _______      ___      _______
                              |       |-----|_=_|---->| B     |
                              |  ,A   |      ___      |       |
                              |       |-----|_#_|---->| dd    |
                              |       |      ___      |       |
                              |       |-----|_>_|---->| (B)   |
                              |       |      ___      |       |
                              |       |-----|_<_|---->| (dd)  |
                              |       |      ____     |       |
                              |       |-----|_>=_|--->| k     |
                              |       |      ____     |_______|
                              |_______|     |_<=_|

                                  Test Carry Flag

                               _______      ___      _______
                              |       |-----|_=_|---->|   0   |
Carry Flag                    |  ,C   |      ___      |       |
                              |_______|-----|_#_|---->|___1___|

                                  Test Channel Status

                               _______      ___      _______
Channel Mnemonic              | ,iod  |-----|_=_|---->|  BZ   |
                              |       |      ___      |       |
Channel Index in              |_,IOB__|-----|_#_|---->|__DN___|
B Register
```

Figure 4-2. Condition Syntax (continued)

4.4 DATA GENERATION STATEMENT INSTRUCTION FORMAT

Symbolic APML instruction fall into two classes, program statements and data generation statements. Data generation statements, described in this subsection, generate 1 or more parcels of data. For more information on program statements, see subsection 4.3, Program Statement Instruction Format.

Location	Assignment	Comment
label	$data_1, data_2, \ldots, data_n$	*.comment*

label Optional label, which must be a valid symbol. The label is defined as the value of the location counter. If a label is present, a new instruction page is forced by APML.

$data_i$ Parcel data item, which can be any of the following:

- Numeric data item. APML generates a 16-bit parcel containing the value. Example:

```
0'42
0'74
57
```

- A character data item. APML generates as many parcels as needed to contain the string. If no suffix is present, the string is left-justified, zero-filled with at least 8 bits of trailing binary zeros. Examples:

```
'THIS IS A MESSAGE'
'BLANK FILL THIS STRING' H
```

- A symbol, whose value is defined elsewhere. The value of the symbol is generated in a single 16-bit parcel.

- <k> reserves *k* parcels of storage; *k* may be a numeric constant or a symbol with absolute value.

- <<k>> generates *k* parcels of zeros; *k* may be a numeric constant or a symbol with absolute value.

5. BASIC IOP HARDWARE INSTRUCTION SET

This section describes the AMPL instructions that generate instructions in the basic I/O Processor (IOP) hardware instruction set. For ease of reference, these hardware instructions are grouped with instructions of similar function.

5.1 INSTRUCTION INDEX

Table 5-1 shows the APML instructions described in this section. Designed for quick reference, it gives the general function of a set of instructions, shows the IOP instructions, the APML symbolic instruction, and the subsection that gives detailed information on the instructions.

Table 5-1. Instruction Index

Instruction Function and Subsection	IOP Instruction	APML Symbolic Instruction
Control (5.2)	000	PASS
	001	EXIT
	002	I = 0
	003	I = 1
Transmit to Accumulator (5.3)	010	A = *d*
	014	A = *k*
	020	A = *dd*
	030	A = (*dd*)
	050	A = B
	060	A = (B)
Logical Product with Accumulator (5.4)	011	A = A & *d*
	015	A = A & *k*
	021	A = A & *dd*
	031	A = A & (*dd*)
	051	A = A & B
	061	A = A & (B)

Table 5-1. Instruction Index (continued)

Instruction Function and Subsection	IOP Instruction	APML Symbolic Instruction
Add to Accumulator (5.5)	012	A = A + *d*
	016	A = A + *k*
	022	A = A + *dd*
	032	A = A + (*dd*)
	052	A = A + B
	062	A = A + (B)
Subtract from Accumulator (5.6)	013	A = A - *d*
	017	A = A - *k*
	023	A = A - *dd*
	033	A = A - (*dd*)
	053	A = A - B
	063	A = A -(B)
Increment by 1 (5.7)	026	*dd* = *dd* + 1
	036	(*dd*) = (*dd*) + 1
	056	B = B + 1
	066	(B) = (B) + 1
Decrement by 1 (5.8)	027	*dd* = *dd* - 1
	037	(*dd*) = (*dd*) - 1
	057	B = B - 1
	067	(B) = (B) - 1
Add to Accumulator and Replace Operand (5.9)	025	*dd* = A + *dd*
	035	(*dd*) = A + (*dd*)
	055	B = A + B
	065	(B) = A + (B)
Transmit from Accumulator (5.10)	024	*dd* = A
	034	(*dd*) = A
	054	B = A
	064	(B) = A
Shift (5.11)	004	A = A > *d*
	005	A = A < *d*
	044	A = A > B
	045	A = A < B
	006	A = A >> *d*
	007	A = A << *d*
	046	A = A >> B
	047	A = A << B

Table 5-1. Instruction Index (continued)

Instruction Function and Subsection	IOP Instruction	APML Symbolic Instruction
Set Carry Flag (5.12)	040	C = 1, *iod* = DN
	041	C = 1, *iod* = BZ
	042	C = 1, IOB = DN
	043	C = 1, IOB = BZ
Branch (5.13)	070 - 137	P = *dd*
		R = *dd*
		P = *k*
		R = *k*
		P = *dd* + *k*
		R = *dd* + *k*
Channel (5.14)	140 - 157	*iod* : *k*
	160 - 177	IOB : *k*

5.2 CONTROL INSTRUCTIONS

PASS, EXIT, I=0, and I=1 are the control instructions.

5.2.1 PASS

This instruction performs no operation. It fills program fields with null operations where desired.

APML	Description	IOP Instruction
PASS	No operation	000000

5.2.2 EXIT

This instruction terminates execution of the current program sequence and returns to the sequence that was suspended in calling this subroutine. The current P register value is discarded. The beginning address for the reinitiated sequence is obtained from the program exit stack at the location currently pointed by E. The value of E is then decremented by 1. If the value of E was previously 0, the decrementing is blocked and the Exit Stack Boundary flag is set. The Exit Stack Boundary flag causes an interrupt of the program sequence for restructuring the contents of the program exit stack.

If the EXIT instruction follows a modification of the program exit stack or of the E pointer, at least 5 clock periods (CPs) must elapse between the last modification and the EXIT instruction.

APML	Description	IOP Instruction
EXIT	Exit from subroutine	001000

5.2.3 I = 0

This instruction clears the System Interrupt Enable flag.

The APML assembler generates two instruction parcels for this instruction: 002000/000000. The 000 pass instruction is included because of a hardware anomoly by which an instruction following the 002000 may sometimes be skipped.

APML	Description	IOP Instruction
I = 0	Disable instruction interrupts	002000/000000

5.2.4 I = 1

This instruction sets the System Interrupt Enable flag. The setting of the flag is delayed until after the execution of a nonbranching instruction. This prevents an interrupt from occurring between this instruction and the following one, which is probably a branch or exit instruction. If the following instruction clears the system interrupt enable flag, that instruction takes precedence over the preceding one.

The delay in setting the flag for this instruction allows the interrupt program to reenable the interrupt mode and then exit to the interrupted program.

APML	Description	IOP Instruction
I = 1	Enable system interrupts	003000

5.3 TRANSMIT TO ACCUMULATOR INSTRUCTIONS

These instructions enter a value in the accumulator. The carry flag is cleared.

5.3.1 A = *d*

This instruction enters the *d* designator in the accumulator as a 9-bit positive integer. The high-order bits are 0.

APML	Description	IOP Instruction
A = *d*	Transmit *d* to A	010--*d*

5.3.2 A = *k*

This instruction enters the 16-bit *k* field in the accumulator.

APML	Description	IOP Instruction
A = *k*	Transmit *k* to A	014000/-----*k*

5.3.3 A = *dd*

This instruction enters the contents of operand register *d* in the accumulator.

APML	Description	IOP Instruction
A = *dd*	Transmit operand register *d* to A	020--*d*

5.3.4 A = (*dd*)

This instruction enters the contents of a memory location in the accumulator. The memory address is obtained from operand register *d*.

APML	Description	IOP Instruction
A = (*dd*)	Transmit contents of memory addressed by register *d* to A	030--*d*

5.3.5 A = B

This instruction enters the B register contents in the accumulator as a 9-bit positive integer. The high-order bits are 0.

APML	Description	IOP Instruction
A = B	Transmit B to A	050000

5.3.6 A = (B)

This instruction enters the contents of operand register B in the accumulator and then clears the carry flag.

APML	Description	IOP Instruction
A = (B)	Transmit operand register B to A	060000

5.4 LOGICAL PRODUCT WITH ACCUMULATOR INSTRUCTIONS

These instruction form the bit-by-bit logical product of the previous accumulator contents and a value obtained from the instruction for *k* fields, a register contents, or memory contents. The result is placed in the accumulator and the carry flag is cleared.

5.4.1 A = A & *d*

This instruction forms the logical product of the previous accumulator contents and the *d* designator.

APML	Description	IOP Instruction
A = A & *d*	Logical product of A and *d* to A	011--*d*

5.4.2 A = A & *k*

This instruction forms the logical product of the previous accumulator contents and the 16-bit *k* field.

APML	Description	IOP Instruction
A = A & *k*	Logical product of A and *k* to A	015000/-----*k*

5.4.3 A = A & *dd*

This instruction forms the logical product of the previous accumulator contents and the contents of operand register *d*.

APML	Description	IOP Instruction
A = A & *dd*	Logical product of A and operand register *d* to A	021--*d*

5.4.4 A = A & (*dd*)

This instruction forms the logical product of the previous accumulator contents and the contents of a memory location. The memory address is obtained from operand register *d*.

APML	Description	IOP Instruction
A = A & (*dd*)	Logical product of A and contents of memory addressed by register *d*, result to A	031--*d*

5.4.5 A = A & B

This instruction forms the logical product of the previous accumulator contents and the 9-bit B register contents.

APML	Description	IOP Instruction
A = A & B	Logical product of A and B to A	051000

5.4.6 A = A & (B)

This instruction forms the logical product of the previous accumulator contents and the contents of operand register B.

APML	Description	IOP Instruction
A = A & (B)	Logical product of A and operand register B to A	061000

5.5 ADD TO ACCUMULATOR INSTRUCTIONS

These addition instructions add a value to the previous accumulator contents. The carry flag is complemented if a carry is propagated from the accumulator in the addition process.

5.5.1 A = A + *d*

This instruction adds the *d* designator to the previous accumulator contents. The *d* designator is treated as a 9-bit positive integer.

APML	Description	IOP Instruction
A = A + *d*	Add *d* to A	012--*d*

5.5.2 A = A + *k*

This instruction adds the 16-bit *k* field to the previous accumulator contents.

APML	Description	IOP Instruction
A = A + *k*	Add *k* to A	016000/-----*k*

5.5.3 A = A + *dd*

This instruction adds the contents of operand register *d* to the previous accumulator contents.

APML	Description	IOP Instruction
A = A + *dd*	Add operand register *d* to A	022--*d*

5.5.4 A = A + (*dd*)

This instruction adds the contents of a memory location to the contents of the accumulator. The memory address is obtained from operand register *d*.

APML	Description	IOP Instruction
A = A + (*dd*)	Add contents of memory addressed by register *d* to A	032--*d*

5.5.5 A = A + B

This instruction adds the 9-bit B register contents to the previous accumulator contents.

APML	Description	IOP Instruction
A = A + B	Add B to A	052000

5.5.6 A = A + (B)

This instruction adds the contents of operand register B to the previous accumulator contents.

APML	Description	IOP Instruction
A = A + (B)	Add operand register B to A	062000

5.6 SUBTRACT FROM ACCUMULATOR INSTRUCTIONS

These instructions subtract a value from the previous accumulator contents. The subtraction is performed by complementing the 16-bit value to be subtracted, and adding the result to the previous accumulator contents. 1 is then added to the result. The carry flag is complemented if a carry is propagated from the accumulator during either addition process.

5.6.1 A = A - *d*

This instruction subtracts the *d* designator from the previous accumulator contents. The *d* designator is treated as a 9-bit positive integer.

APML	Description	IOP Instruction
A = A - *d*	Subtract *d* from A	013--*d*

5.6.2 A = A - *k*

This instruction subtracts the 16-bit *k* field from the previous accumulator contents.

APML	Description	IOP Instruction
A = A - *k*	Subtract *k* from A	017000/-----*k*

5.6.3 A = A - *dd*

This instruction subtracts the contents of operand register *d* from the previous accumulator contents.

APML	Description	IOP Instruction
A = A - *dd*	Subtract operand register *d* from A	023--*d*

5.6.4 A = A - (*dd*)

This instruction subtracts the contents of a memory location from the contents of the accumulator. The memory address is obtained from operand register *d*.

APML	Description	IOP Instruction
A = A - (*dd*)	Subtract contents of memory addressed by register *d* from A, result to A	033--*d*

5.6.5 A = A - B

This instruction subtracts the 9-bit B register contents from the previous accumulator contents.

APML	Description	IOP Instruction
A = A - B	Subtract B from A	053000

5.6.6 A = A - (B)

This instruction subtracts the contents of operand register B from the previous accumulator contents.

APML	Description	IOP Instruction
A = A - (B)	Subtract operand register B from A	063000

5.7 INCREMENT BY 1 INSTRUCTIONS

These instructions add 1 to the contents of a register or memory location. The carry flag is cleared at the beginning of the operation and a 1 is entered in the accumulator. The contents of the register or memory location is then added to the accumulator. The carry flag is set if a carry is propagated from the accumulator in the addition process. The result is returned to the register or memory location.

5.7.1 dd = *dd* + 1

This instruction replaces the contents of operand register *d* with the previous contents increased by 1.

APML	Description	IOP Instruction
dd = *dd* + 1	Transmit register *d* to A, add 1, result to operand register *d*	026--*d*

5.7.2 (dd) = (*dd*) + 1

This instruction increments the contents of a memory location by 1. The memory address is obtained from operand register *d*.

APML	Description	IOP Instruction
(dd) = (*dd*) + 1	Transmit memory addressed by register *d* to A, add 1, result to same memory location	036--*d*

5.7.3 B = B + 1

This instruction replaces the contents of the B register with its previous contents increased by 1.

APML	Description	IOP Instruction
B = B + 1	Transmit B to A, add 1, result to B	056000

5.7.4 (B) = (B) + 1

This instruction replaces the contents of operand register B with its previous contents increased by 1.

APML	Description	IOP Instruction
(B) = (B) + 1	Transmit operand register B to A, add 1, result to operand register B	066000

5.8 DECREMENT BY 1 INSTRUCTIONS

These instructions subtract 1 from the contents of a register or memory location. The carry flag is cleared at the beginning of this operation. A minus 1 value is entered in the accumulator. The contents of the register or memory location are then added to the accumulator contents. The carry flag is set if a carry is propagated from the accumulator in the addition process. The result is then returned to the register or memory location.

5.8.1 *dd* = *dd* - 1

This instruction replaces the contents of operand register *d* with the previous contents decreased by 1.

APML	Description	IOP Instruction
dd = *dd* - 1	Transmit register *d* to A, subtract 1, result to operand register *d*	027--*d*

5.8.2 (*dd*) = (*dd*) - 1

This instruction decrements the contents of a memory location by 1. The memory address is obtained from operand register *d*.

APML	Description	IOP Instruction
(*dd*) = (*dd*) - 1	Transmit memory addressed by register *d* to A, subtract 1, result to same memory location	037--*d*

5.8.3 B = B - 1

This instruction replaces the contents of the B register with its previous contents decreased by 1.

APML	Description	IOP Instruction
B = B - 1	Transmit B to A, Subtract 1, result to B	057000

5.8.4 (B) = (B) - 1

This instruction replaces the contents of operand register B with its previous contents decreased by 1.

APML	Description	IOP Instruction
(B) = (B) - 1	Transmit operand register B to A, subtract 1, result to operand register B	067000

5.9 ADD TO ACCUMULATOR AND REPLACE OPERAND INSTRUCTIONS

These instructions add the contents of a register (or memory) to the accumulator and place the result in both the accumulator and the register (or memory). The carry flag is complemented if a carry is propagated in the addition process.

5.9.1 *dd* = A + *dd*

This instruction adds the contents of operand register *d* to the previous accumulator contents and replaces the result in the operand register *d*.

APML	Description	IOP Instruction
dd = A + *dd*	Add operand register *d* to A, result to operand register *d*	025--*d*

5.9.2 (*dd*) = A + (*dd*)

This instruction replaces the contents of a memory location with its previous content plus the current accumulator contents. The memory address is obtained from operand register *d*.

APML	Description	IOP Instruction
(*dd*) = A + (*dd*)	Add memory addressed by register *d* to A, result to same memory location	035--*d*

5.9.3 B = A + B

This instruction adds the 9-bit contents of the B register to the previous accumulator contents.

APML	Description	IOP Instruction
B = A + B	Add B to A, result to B	055000

5.9.4 (B) = A + (B)

This instruction adds the contents of operand register B to the previous accumulator contents.

APML	Description	IOP Instruction
(B) = A + (B)	Add operand register B to A, result to operand register B	065000

5.10 TRANSMIT FROM ACCUMULATOR INSTRUCTIONS

The following instructions transmit from the accumulator: *dd* = A, (*dd*) = A, B = A, and (B) = A.

5.10.1 *dd* = A

This instruction stores the accumulator contents in operand register *d*.

APML	Description	IOP Instruction
dd = A	Transmit A to register *d*	024--*d*

5.10.2 (*dd*) = A

This instruction replaces the contents of a memory location with the current accumulator contents. The memory address is obtained from operand register *d*.

APML	Description	IOP Instruction
(*dd*) = A	Transmit A to memory addressed by register *d*	034--*d*

5.10.3 B = A

This instruction replaces the B register contents with the low-order 9 bits of the accumulator contents.

APML	Description	IOP Instruction
B = A	Transmit A to B	054000

5.10.4 (B) = A

This instruction stores the accumulator contents in operand register B.

APML	Description	IOP Instruction
(B) = A	Transmit A to operand register B	064000

5.11 SHIFT INSTRUCTIONS

The shift instructions shift accumulator contents and associated carry flag to the right or left. The carry flag may be regarded as a 17th bit to the left of the accumulator contents for these operations. The shift count is obtained from the low-order 5 bits of the *d* field or the low-order 5 bits of the B register contents.

5.11.1 END OFF SHIFTS

In the end off shifts, bits shifted off are discarded and 0 bits are entered at the opposite end. The accumulator and carry flag are cleared if the shift count is greater than 16.

APML	Description	IOP Instruction
A = A > *d*	Right shift C and A by *d* places, end off	004--*d*

APML	Description	IOP Instruction
A = A < *d*	Left shift C and A by *d* places, end off	005--*d*

APML	Description	IOP Instruction
A = A > B	Right shift C and A by B places, end off	044000

APML	Description	IOP Instruction
A = A < B	Left shift C and A by B places, end off	045000

5.11.2 CIRCULAR SHIFTS

In the circular shifts, bits shifted off are entered at the opposite end.

APML	Description	IOP Instruction
A = A >> *d*	Right shift C and A by *d* places, circular	006--*d*

APML	Description	IOP Instruction
A = A << *d*	Left shift C and A by *d* places, circular	007--*d*

APML	Description	IOP Instruction
A = A >> B	Right shift C and A by B places, circular	046000

APML	Description	IOP Instruction
A = A << B	Left shift C and A by B places, circular	047000

5.12 SET CARRY FLAG INSTRUCTIONS

The following instructions set the carry flag.

5.12.1 C = 1, *iod* = DN

This instruction forces the carry flag to the same state as the channel *d* done flag.

APML	Description	IOP Instruction
C = 1, *iod* = DN	Set carry equal to channel *d* done	040--*d*

5.12.2 C = 1

Channel 000 is always done. You can set the carry flag by setting *d* = 000 in this instruction.

APML	Description	IOP Instruction
C = 1	Set carry flag	040000

5.12.3 C = 1, *iod* = BZ

This instruction forces the carry flag to the same state as the channel *d* busy flag.

APML	Description	IOP Instruction
C = 1, *iod* = BZ	Set carry equal to channel *d* busy	041--*d*

5.12.4 C = 0

Channel 000 is never busy. You can force the carry flag clear by setting *d* to 000 in this instruction.

APML	Description	IOP Instruction
C = 0	Clear carry flag	041000

5.12.5 C = 1, IOB = DN

This instruction forces the carry flag to the same state as the done flag of the channel specified by the B register contents.

APML	Description	IOP Instruction
C = 1, IOB = DN	Set carry equal to channel B done	042000

5.12.6 C = 1, IOB = BZ

This instruction forces the carry flag to the same state as the busy flag of the channel specified by the B register contents.

APML	Description	IOP Instruction
C = 1, IOB = BZ	Set carry equal to channel B busy	043000

5.13 BRANCH INSTRUCTIONS

The branch instructions in the IOP use instruction codes 070_8 through 137_8, comprising 40 different instructions. This large number of branch instructions comes from having a unique instruction code for every combination of the following three variables. For the full set of hardware instructions and their instruction codes, see appendix B, Hardware Instruction Summary.

1. Branch type:

P=X	Jump
R=X	Return jump

2. Branch condition:

---	Unconditional branch
---,C=0	Branch if carry flag is clear
---,C=1	Branch if carry flag is set
---,A=0	Branch if accumulator is zero
---,A#0	Branch if accumulator is nonzero

3. Branch mode:

P=P+*d*†	Branch to a new program address formed by adding the *d* designator to the current instruction address
P=P-*d*†	Branch to a new program address formed by subtracting the *d* designator from the current instruction address

† This APML format is for illustrative purposes only. The assembler does not support this format as a symbolic APML instruction, although the hardware instruction is generated automatically by APML whenever a branch is to a label within the same instruction page.

P=*dd*	Branch to the address in operand register *dd*
P=*dd+k*	Branch to the address formed by adding the *k* field to the contents of operand register *dd*

The execution of a branch instruction does not alter the accumulator contents and carry flag.

For instructions with destination *dd* or *dd+k*, the Program Fetch Request flag is set if the contents of operand register *dd* contain a 0.

5.14 CHANNEL FUNCTION INSTRUCTIONS

The channel function instructions issue a function to the channel specified. In the IOB:*k* instruction, the B register contents specifiy the channel. In the *iod:k* instruction, the channel mnemonic *iod* specifies the channel, where the value of the *iod* symbol is inserted by APML in the 9-bit *d* field of the instruction. The function code *k* must be a positive value less than 20_8, and is added to 140_8 or 160_8 to form the IOP instruction code for *iod:k* or IOB:*k*, respectively.

The channel function instruction may provide accumulator data to the channel interface or may return channel interface data to the accumulator. For additional information about specific channel functions, see section 4, Symbolic APML Instruction Syntax, and appendix C, Messages.

The channel function instructions are as follows:

iod:k	Channel *d* function *k*	140--*d* through 157--*d*
IOB:*k*	Channel B function *k*	160000 through 177000

6. PSEUDO INSTRUCTIONS

APML includes a set of instructions known as pseudo instructions to direct the assembler in its task of interpreting the source statements and generating an object program.

Some pseudo instructions such as IDENT and END are required by the assembler; others are optional. If certain of these optional instructions are not used, the assembler uses a default setting.

6.1 RULES FOR PSEUDO INSTRUCTIONS

Each program module begins with an IDENT instruction and ends with an END instruction. Symbol, micro, and macro definitions occurring within the program module are cleared before assembling the next program module.

You may define a symbol, micro, or macro prior to the first IDENT pseudo instruction or between an END and a subsequent IDENT pseudo instruction. Such a definition is considered global and may be referenced in any subsequent program module. For more information on global definitions, see subsection 2.8, Global Definitions.

Symbolic machine instructions and the pseudo instructions that follow must appear within a program module. They are allowed outside of an IDENT to END sequence only within macro definitions.

ABS	EXT
BASEREG	GLOBAL
BITP	LOC
BITW	MICSIZE
BLOCK	NEWPAGE
BSS	ORG
BSSZ	PDATA
COMMENT	QUAL
CON	SCRATCH
DATA	START
ENDTEXT	TEXT
ENTRY	VWD

In an absolute program module, the ABS pseudo instruction must appear before any symbolic machine instruction or before any of the preceding pseudo instructions. All other pseudo instructions and macro definitions may appear anywhere.

6.2 TYPES OF PSEUDO INSTRUCTIONS

Pseudo instructions are classified according to their applications as follows:

Class	Pseudo Instructions in Class
Program control	IDENT, END, ABS, COMMENT, GLOBAL
Code control	BASEREG, SCRATCH, NEWPAGE
Loader linkage	ENTRY, EXT, START
Mode control	BASE, QUAL
Block control	BLOCK, ORG, BSS, LOC, BITW, BITP
Error control	ERROR, ERRIF
Listing control	LIST, SPACE, EJECT, TITLE, SUBTITLE, TEXT, ENDTEXT
Symbol definition	EQUALS, SET, CHANNEL, MICSIZE
Data definition	CON, BSSZ, DATA, PDATA, VWD
Conditional assembly	IFA, IFE, IFC, SKIP, ENDIF, ELSE
Instruction definition	MACRO, LOCAL, ENDM, OPSYN
Code duplication	DUP, ECHO, ENDDUP, STOPDUP
Micro definition	MICRO, OCTMIC, DECMIC

6.3 PROGRAM CONTROL PSEUDO INSTRUCTIONS

The pseudo instructions described in this subsection define the limits of a program module and define the type of assembly to be performed.

6.3.1 IDENT - IDENTIFY PROGRAM MODULE

The IDENT pseudo instruction identifies a program module and marks its beginning. The name of the module appears in the heading of the listing produced by APML and in the Program Descriptor Table (PDT) of the binary load module.

Format:

Location	Result	Operand
ignored	IDENT	*name*

name Name of the program module; a name must meet the requirements for names given in section 2, APML Assembler Language.

Example:

Location	Result	Operand	Comment
1	10	20	
	IDENT	KOJE	

6.3.2 END - END PROGRAM MODULE

The END pseudo instruction is the final statement of a program module. It causes the assembler to take the following actions:

- Reset the numeric base for assembly to octal
- Clear the base, list, qualification, base register, and block stacks
- Terminate any skipping, macro definitions, or repeated code
- Reset the list control options to those determined by the APML control statement

Format:

Location	Result	Operand
ignored	END	*ignored*

6.3.3 ABS - ASSEMBLE ABSOLUTE BINARY

The ABS pseudo instruction designates that a program module will be assembled as an absolute rather than a relocatable load module. Since there is no loader for processing relocatable APML code, you should always include this pseudo instruction.

Format:

Location	Result	Operand
ignored	ABS	*ignored*

6.3.4 COMMENT - DEFINE PROGRAM DESCRIPTOR TABLE COMMENT

The COMMENT pseudo instruction defines a character string to be entered as an informational comment in the PDT of the binary load data. The character string is entered as 0 to 10 words of left-justified, blank-filled ASCII data, starting in the 12th header word of the PDT.

If a subprogram contains more than one COMMENT pseudo, the character string from the last COMMENT pseudo is inserted in the PDT.

Format:

Location	Result	Operand
ignored	COMMENT	'*character string*'

'*character string*'
ASCII character string of 0 to 80 characters

Example:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	APML	
	COMMENT	'COPYRIGHT CRAY RESEARCH, INC. 1980'	

6.3.5 GLOBAL - DECLARE GLOBAL SYMBOLS

The GLOBAL pseudo instruction declares a symbol to be a global symbol. A symbol declared in this manner is maintained across program modules as if it were a symbol defined in a system text.

Format:

Location	Result	Operand
ignored	GLOBAL	$symbol_1$, $symbol_2$,..., $symbol_n$

symbol$_i$ The name of a symbol. You must define the symbol elsewhere in the program module.

6.4 CODE CONTROL PSEUDO INSTRUCTIONS

The pseudo instructions described in this subsection provide control of the I/O Processor (IOP) code generated by APML.

6.4.1 BASEREG - DECLARE BASE OPERAND REGISTER

The BASEREG pseudo instruction declares the operand register to be used by APML in 2-parcel jump instructions. All IOP jump instructions are either 1- or 2-parcel instructions. One-parcel instructions allow jumping 511 parcels forward or backward from the current address. Two-parcel jump instructions contain an operand register index and a 16-bit address. The jump destination is the sum of the 16-bit address and the contents of the indicated operand register.

You may either explicitly name the operand register in each 2-parcel jump instruction or may specify a base register and allow APML to implicitly use the declared register whenever a 2-parcel instruction is required.

Format:

Location	Result	Operand
ignored	BASEREG	*symbol,bias*

symbol A symbol representing the base register. You must ensure that the declared register contains the proper base address.

bias An expression whose value is a bias against the address contained in the base register (default is 0). This parameter is normally omitted. It is needed only when using the program fetch feature of the IOP to prevent interrupts when a base register would otherwise contain a valid zero address.

If the operand field is blank, a previously declared base register is no longer valid. Two-parcel jumps which do not explicitly name an operand register produce a warning and operand register 0 is used.

bias (continued) If the operand field is an asterisk, the previous base register and bias are popped from the stack. Each occurrence of a BASEREG pseudo instruction other than BASEREG * causes an entry in the stack. Each BASEREG * removes an entry from the stack. If the stack is empty, no base register is declared.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
		IDENT	BASEREG	
1	R1	EQUALS	1	
		BASEREG	R1	
075001 / 001744		P=NEXT		
		<1742>		.Reserve
				1742
				parcels
	NEXT	A=B		
		.		
		.		
		.		
		END		

6.4.2 SCRATCH - DECLARE APML SCRATCH REGISTER

When generating IOP machine instructions from APML statements, APML sometimes uses scratch operand registers to hold memory addresses or intermediate values.

The SCRATCH pseudo instruction declares operand registers that APML uses for this purpose.

Format:

Location	Result	Operand
ignored	SCRATCH	$r_1, r_2, \ldots r_n$

r_i A symbol used as a register name. You may declare from zero to five register symbols. The symbol may be external, relocatable, or absolute with a positive value less than 512.

Each occurrence of the SCRATCH pseudo instruction declares a new set of scratch registers. If APML needs more scratch registers than are declared, an error is generated.

You can determine scratch register usage by APML from a cross-reference listing generated by APML for each line in which a scratch register is used.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
		IDENT	SCRATCH	
1	SHARK	EQUALS	1	
6	DO	SET	6	
		SCRATCH	SHARK,DO,DA	
4	DA	EQUALS	4	
	LOC	<1>		
014000 /000000 024001		(LOC)=(1057)		
014000 /001057 024006				
030006 034001				
		END		

6.4.3 NEWPAGE - FORCE A NEW INSTRUCTION PAGE

The NEWPAGE pseudo instruction causes APML to force an instruction page boundary. All labels appearing on previous APML instructions are defined. Jumps across a page boundary must be 2-parcel jumps. Optimization of the previous block of code occurs.

This instruction forces definition of labels and allows you to control to some extent where page boundaries occur so that the assembler can improve code optimization.

Format:

Location	Result	Operand
ignored	NEWPAGE	

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
		IDENT	NEWPAGE	
1	R1	EQUALS	1	
		BASEREG	R1	
075001 /000002		P=NEXT		
		NEWPAGE		
	NEXT	A=B		
		.		
		.		
		.		
		END		

6.5 LOADER LINKAGE PSEUDO INSTRUCTIONS

The pseudo instructions ENTRY and EXT provide for loading multiple object program modules and linking them into a single executable program.

6.5.1 ENTRY - SPECIFY ENTRY SYMBOLS

The ENTRY pseudo instruction specifies symbolic addresses or values that may be referred to by other program modules linked by the loader. Each entry symbol must be a relocatable or absolute symbol defined within the program module.

Format:

Location	Result	Operand
ignored	ENTRY	$symbol_1, symbol_2, ..., symbol_n$

$symbol_i$ A valid symbol

Example:

Location	Result	Operand	Comment
1	10	20	35
	ENTRY	EPTNME	

6.5.2 EXT - SPECIFY EXTERNAL SYMBOLS

The EXT pseudo instruction specifies linkage to symbols defined as entry symbols in other program modules. They may be referred to from within the program module but must not be defined within the program module. Symbols specified on the EXT instruction have absolute and value attributes with a value of 0.

Format:

Location	Result	Operand
ignored	EXT	$sym_1, sym_2, \ldots, sym_n$

sym_i An unqualified symbol

Example:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	A	
	.		
	.		
	.		
	ENTRY	VALUE	
VALUE	EQUALS	-2	
	.		
	.		
	END		
	IDENT	B	
	EXT	VALUE	
	CON	VALUE	.The 64-bit external value -2 to
			.be stored here by a loader

6.5.3 START - SPECIFY PROGRAM ENTRY

The START pseudo instruction specifies the main program entry. In a relocatable program, this entry is the symbolic address where execution begins following the loading of the program. The named symbol may optionally be an entry symbol specified in an ENTRY pseudo instruction.

You can name only one main program entry in a program module.

Format:

Location	Result	Operand
ignored	START	*symbol*

symbol An entry symbol

6.6 MODE CONTROL PSEUDO INSTRUCTIONS

Mode control pseudo instructions define the characteristics of an assembly. The BASE pseudo determines whether notation for numeric data is assumed to be octal or decimal. The QUAL pseudo instruction permits symbols to be defined as qualified or unqualified.

6.6.1 BASE - DECLARE BASE FOR NUMERIC DATA

The BASE pseudo instruction allows specification of the base of numeric data as being octal, decimal, or mixed when the base is not explicitly specified by an O' or D' prefix. The default is octal.

Format:

Location	Result	Operand
ignored	BASE	*base*

base Required single character, as follows:

O Octal; all numeric data is assumed to be octal.

D Decimal; all numeric data is assumed to be decimal.

M Mixed; numeric data is assumed to be octal except for numeric data used for the following, which is assumed to be decimal:

- Statement counts in DUP and conditional statements
- Line count in SPACE
- Bit position or count in BITW, BITP, or VWD
- Character counts as in MICRO, OCTMIC, DECMIC, and data items

* Reverts to use of the previous base in the stack. Each occurrence of a BASE pseudo instruction other than BASE * causes an entry in the stack. Each BASE * removes an entry from the stack and causes the base in use prior to the current base to be resumed. If the stack is empty when BASE * is encountered, the APML default mode (octal) is used.

Example:

Location	Result	Operand	Comment
1	10	20	35
	BASE	D	.Change base from default (octal)
			.to decimal
	VWD	40/10	.Field size and constant value
			.both decimal
	.	.	
	.	.	
	.	.	
	BASE	M	.Change from decimal to mixed
			.base
	VWD	40/12	.Field size decimal; constant
			.value octal
	.	.	
	.	.	
	.	.	
	BASE	O	.Change base from mixed to octal
	VWD	50/12	.Field size and constant value
			.both octal
	.	.	
	.	.	
	.	.	
	BASE	*	.Resume mixed base
	BASE	*	.Resume decimal base
	BASE	*	.Stack empty; resume octal base

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
		IDENT	BASE	
010012		A=12		.BASE O
		BASE	*	
010012		A=12		.BASE O
		BASE	D	
010014		A=12		.BASE D
		BASE	*	
010012		A=12		.BASE O
		END		

6.6.2 QUAL - QUALIFY SYMBOLS

A QUAL pseudo instruction begins or ends a code sequence in which all symbols defined are either qualified by a qualifier specified by the QUAL or are unqualified. Until the first use of a QUAL pseudo instruction, symbols are defined as unqualified. Global symbols cannot be qualified. Thus, QUAL pseudo instructions must not occur before IDENT.

A qualifier applies to symbols only and does not affect names used for blocks, conditional sequences, duplicated sequences, macros, micros, externals, and formal parameters.

Format:

Location	Result	Operand
ignored	QUAL	*qualification*

qualification

Indicates whether symbols are to be qualified or unqualified; if qualified, indicates the qualifier to be used. The field may contain a *qualifier,* * , or no entry.

qualifier The presence of a 1- to 8-character *qualifier,* where a *qualifier* is a valid name, causes all symbols defined until the next QUAL pseudo instruction to be qualified. Being qualified means that such a symbol can be referenced with or without the qualifier within any sequence in which the qualifier is in effect; however, if the symbol is referenced while some other qualifier is in effect, the reference must be in the form:

/qualifier/symbol

When a symbol is referenced without a *qualifier,* APML first attempts to find it qualified by the *qualifier* in effect. If the qualified symbol is not defined, APML attempts to find it in the list of unqualified symbols. The symbol is undefined if both of these searches fail.

*	An * resumes use of the qualifier in effect previous to the current qualification. Each occurrence of a QUAL other than a QUAL * causes an entry in a qualification stack. Each QUAL * removes an entry from the stack and causes the qualification in effect to be resumed. If the stack is empty when QUAL * is encountered, symbols are defined unqualified.
no entry	If the operand field of the QUAL is empty, symbols are defined as unqualified until the next occurrence of a QUAL pseudo instruction. You can reference an unqualified symbol without qualification from any place in the program module, or in the case of global symbols, from any program module assembled after the symbol definition.

Example:

```
|Location |Result    |Operand      |Comment
|1        |10        |20           |35
|         |          |             |
|         |.         |             |.System default is unqualified
|         |.         |             |
|         |.         |             |
|ABC      |EQUALS    |1            |.ABC is defined unqualified
|         |QUAL      |JVR          |.Symbols will be qualified by JVR
|ABC      |EQUALS    |2            |
|         |P = XYZ   |             |
|XYZ      |A = 23    |             |
|         |.         |             |
|         |.         |             |
|         |.         |             |
|         |QUAL      |DCK          |.Symbols will be qualified by DCK
|ABC      |EQUALS    |3            |
|         |P = /JVR/XYZ            |
|         |.         |             |
|         |.         |             |
|         |.         |             |
|         |QUAL      |*            |.Resume use of JVR
|         |.         |             |
|         |.         |             |
|         |.         |             |
|         |QUAL      |             |.Symbols will be unqualified
|A        |IFA       |DEF,ABC      |.Test for ABC defined
|B        |IFA       |DEF,/JVR/ABC |.Test for /JVR/ABC defined
|C        |IFA       |DEF,/DCK/ABC |.Test for /DCK/ABC defined
```

6.7 BLOCK CONTROL PSEUDO INSTRUCTIONS

You can divide a program, whether assembled into absolute binary or relocatable binary, into sections called blocks. As assembly of a program proceeds, you explicitly or implicitly assign code to specific blocks or reserve areas of a block. The assembler assigns locations in a block consecutively as it encounters instructions or data destined for the block.

By dividing a program into blocks, you can conveniently separate executable sequences of code from nonexecutable data. When no BLOCK pseudo instructions are used, all assignment of code is implicitly designated in the nominal block. Use the nominal block for all code not explicitly contained in a named block.

When a BLOCK pseudo instruction is used, all code generated or memory reserved from the occurrence of one BLOCK instruction up to the occurrence of the next BLOCK instruction is assigned to the designated block. Until the first BLOCK instruction, the nominal block is used. Blocks defined by BLOCK instructions are referred to as local blocks because at program end, all of the blocks are concatenated with the nominal block to form the program block. That is, blocks exist local to the assembly and are invisible to a relocatable loader.

The nominal block is always the first block in the program block. All other local blocks are appended in the order that the blocks are first referenced in a BLOCK instruction.

APML maintains a pushdown stack of block names. It makes an entry in the stack each time a BLOCK pseudo instruction names a block to be used and deletes an entry from the stack each time a BLOCK pseudo contains * to indicate resumption of the block previously in use. The block in use is always the top entry in the stack. If the program contains more BLOCK * instructions than there are entries in the stack, the assembler uses the nominal block.

For each block used in a program, APML maintains an origin counter, a location counter, and a bit position counter. When a block is first established or its use is resumed, APML uses the counters for that block. During pass 1 of the assembler, the origin and location counters for a block are initially 0. During pass 2, as the assembler constructs the program, it assigns an initial value to each local block origin counter and location counter. Thus, expressions containing relocatable symbols are evaluated differently in pass 2 than in pass 1.

The origin counter controls the relative location of the next word to be assembled or reserved in the block. It is possible to reserve blank memory areas simply by using either the ORG or BSS pseudo instructions to advance the origin counter. When the special element *O is used in an expression, the assembler replaces it with the current parcel-address value of the origin counter for the block in use. You may use W.*O to obtain the word-address value of the origin counter.

The location counter is normally the same value as the origin counter and is used by the assembler for defining symbolic addresses within a block. The counter is incremented whenever the origin counter is incremented. The LOC pseudo instruction adjusts the location counter so that it differs in value from the origin counter or so that it refers to the address relative to a block other than the one currently in use. When the special element * is used in an expression, the assembler replaces it by the current parcel address value of the location counter for the block in use. You may use W.* to obtain the word address value of the location counter.

As instructions and data are assembled and placed into a word, APML maintains a pointer indicating the next available bit within the word currently being assembled. This pointer is known as the word-bit-position counter. It is 0 when a new word is begun and is incremented by 1 for each completed bit in the word. Its maximum value is 63 for the rightmost bit in the word. When a word is completed, the origin and location counters are incremented by 1 and the word bit position counter is reset to 0 for the next word.

When you use the special element *W in an expression, the assembler replaces it with the current value of the word-bit-position counter. The normal advancement of the word-bit-position counter is in increments of 16, 32, and 64 as 1- and 2-parcel instructions or words are generated. You can alter this normal advancement, however, through use of the BITW, BITP, and VWD pseudo instructions.

The assembler completes a partial word and sets the word-bit-position and parcel-bit-position counters to 0 if either of the following conditions is true:

- The current instruction is an ORG, LOC, BSS, BSSZ, or CON pseudo instruction

- The current instruction is a DATA or VWD pseudo instruction and the instruction has an entry in the location field

Unused bits in a partial word are filled with binary zeros.

In addition to the word-bit-position counter, APML maintains a counter that points to the next bit to be assembled in the current parcel. This pointer is known as the parcel-bit-position counter. It is 0 when a new parcel is begun and advances by 1 for each completed bit in the parcel. Its maximum value is 15 for the rightmost bit in a parcel. When a parcel is completed, the parcel bit position counter is reset to 0.

When you use the special element *P in an expression, APML replaces it with the current value of the parcel-bit-position counter.

The parcel-bit-position counter is set to 0 following assembly of most instructions. The pseudo instructions BITW, BITP, DATA, and VWD may cause the counter to be nonzero.

The assembler completes a partially filled parcel and sets the parcel-bit-position counter to 0 if the current instruction is a symbolic APML instruction.

6.7.1 BLOCK - LOCAL BLOCK ASSIGNMENT

A BLOCK pseudo instruction establishes or resumes use of a block of code within a program module (a local block). Each block has its own location, origin, and bit-position counters.

Format:

Location	Result	Operand
ignored	BLOCK	*name*

name Name of the block to be used for assembling code until the occurrence of the next BLOCK pseudo instruction

- *bname* Name of local block
- * Return to previous block
- blank Resume use of nominal block

Example:

Location	Result	Operand	Comment
1	10	20	35
	.		.Nominal block in use
	.		
	.		
	BLOCK	A	.Use block A
	.		
	.		
	.		
	BLOCK		.Use nominal block
	.		
	.		
	.		
	BLOCK	*	.Return to use of block A

6.7.2 ORG - SET *O COUNTER

The ORG pseudo instruction resets the location and origin counters to the value specified. The expression must have a value- or word-address attribute. If the expression has a value attribute, it is assumed to be a word address.

The first occurrence of the ORG instruction in an absolute assembly indicates the address at which binary output begins. Subsequent ORG instructions cannot specify a value lower than the first ORG value. If ORG is omitted, an origin of 0 is assumed.

Format:

Location	Result	Operand
ignored	ORG	*exp*

exp New origin word address, a relocatable expression with positive relocation within block currently in use. In an absolute assembly, *exp* must be absolute if in the nominal block. If the expression is blank, the word address of the next available word in the block is used. All symbols used in the expression must be previously defined. A force to word boundary occurs before the expression is evaluated.

Example:

Location	Result	Operand	Comment
1	10	20	35
	ORG	O'200/4	.Absolute assembly
			.Set origin to the word address
			.equivalent to parcel 200_8.

6.7.3 BSS - BLOCK SAVE

The BSS pseudo instruction reserves a block of memory in a program. A force to word boundary occurs and then the number of words specified by the operand field expression is reserved. This pseudo instruction does not generate data. The block of memory is reserved by increasing the location and origin counters.

Format:

Location	Result	Operand
symbol	BSS	*exp*

symbol Optional symbol assigned the word address of the location counter after the force to word boundary occurs

exp An absolute expression with word-address or value-address attribute and with all symbols previously defined. The expression value must be positive. A force to word boundary occurs before the expression is evaluated.

The left margin of the listing shows the octal word count.

Example:

Location	Result	Operand	Comment
1	10	20	35
	BSS	4	
	.		
	.		
	.		
A	CON	'NAME'	
	CON	1	
	CON	2	
	BSS	A+16-W.*	.Reserve 13 more words

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
		IDENT	BSSBSSZ	
050000		A=B		
12	NON	BSS	12	
4	ZERO	BSSZ	4	
	HERE	*		
		END		

6.7.4 LOC - SET * COUNTER

The LOC pseudo instruction resets the location counter to the first parcel of the word address specified. The location counter assigns address values to location field symbols. Changing the location counter

allows code to be assembled and loaded at one location, controlled by the origin counter, then moved and executed at another address, controlled by the location counter.

Format:

Location	Result	Operand
ignored	LOC	*exp*

exp New location counter word address, a relocatable expression with positive relocation, not necessarily within the block currently in use. The expression may also be absolute. All symbols used in the expression must be previously defined. A force word boundary occurs before the expression is evaluated.

Example:

Location	Result	Operand	Comment
1	10	20	35
*	In this example, the code is generated and loaded at parcel		
*	10000, and moved by your parcel to 200 before execution		
	ABS		
	ORG	10000/4	
	LOC	200/4	
A	A1 = 0		
	.		
	.		
	.		
	P = A		

6.7.5 BITW - SET *W COUNTER

The BITW pseudo instruction sets the current bit position relative to the current word to the value specified. A value of 64 indicates the following instruction is to be assembled at the beginning of the next word (force word boundary). If the counter is set lower than its current value, any code previously generated in the overlapping portion of the word is ORed with any new code.

Format:

Location	Result	Operand
ignored	BITW	*exp*

exp An expression with absolute value attribute with positive value less than or equal to 64. When the base is M (mixed), APML assumes that *exp* is decimal.

Example:

Location	Result	Operand	Comment
1	10	20	35
	BITW	D'39	

6.7.6 BITP - SET *P COUNTER

The BITP pseudo instruction sets the bit position relative to the current parcel to the value specified. A value of 16 forces a parcel boundary. If the current position is in the middle of a parcel, the bit position is set to the beginning of the next parcel; otherwise, the bit position is not changed. If the counter is set lower than its current value, any code previously generated in the overlapping portion of the word is ORed with any new code.

Format:

Location	Result	Operand
ignored	BITP	*exp*

exp An expression with absolute value attribute with positive value less than or equal to 16. When the base is M (mixed), APML assumes that *exp* is decimal.

Example:

Location	Result	Operand	Comment
1	10	20	35
	BITP	D'14	

6.8 ERROR CONTROL PSEUDO INSTRUCTIONS

Two pseudo instructions, ERROR and ERRIF, allow you to generate an assembly error condition.

6.8.1 ERROR - UNCONDITIONAL ERROR GENERATION

The ERROR pseudo instruction unconditionally sets an assembly error flag.

Format:

Location	Result	Operand
error	ERROR	*ignored*

error A valid error flag character as defined in appendix D, Assembly Errors. P is used if this field is null.

Example:

Location	Result	Operand	Comment
1	10	20	35
	IFE	ABC,LT,DEF,1	
	ERROR		
	.		
	.		
	.		

6.8.2 ERRIF - CONDITIONAL ERROR GENERATION

The ERRIF pseudo instruction conditionally sets an assembly error flag.

Format:

Location	Result	Operand
error	ERRIF	exp_1,op,exp_2

error A valid error flag character as defined in appendix D, Assembly Errors. P is used if this field is null.

exp_1, exp_2
: Expressions to be compared. Any symbols must have been defined previously. These expressions are evaluated in pass 2, whereas expressions in other conditional pseudo instructions are evaluated in pass 1. In pass 2, address expressions in local blocks have been relocated relative to the beginning of the program block rather than relative to the local block.

op
: Specifies a relation to be satisfied by exp_1 and exp_2 that causes generation of an error. For LT, LE, GT, and GE, only the values of the expressions are examined. The word-address, parcel-address, or value attributes and the relocatable, external, or absolute attributes are not compared.

  - LT Less than; the value of exp_1 must be less than the value of exp_2.
  - LE Less than or equal to; the value of exp_1 must be less than or equal to the value of exp_2.
  - GT Greater than; the value of exp_1 must be greater than the value of exp_2.
  - GE Greater than or equal to; the value of exp_1 must be greater than or equal to the value of exp_2.
  - EQ Equal; the value of exp_1 must be equal to the value of exp_2. The expressions must either both be absolute, or both be external relative to the same external symbol, or both be relocatable in the same block. The word-address, parcel-address, or value attributes must be the same.
  - NE Not equal; the two expressions, exp_1 and exp_2, do not satisfy the conditions required for EQ previously described.

Example:

Location	Result	Operand	Comment
1	10	20	35
P	ERRIF	ABC,LT,DEF	

6.9 LISTING CONTROL PSEUDO INSTRUCTIONS

The pseudo instructions described in this subsection allow you to control the contents and format of the listing produced by the assembler. These pseudo instructions are not ordinarily listed.

6.9.1 LIST - LIST CONTROL

The LIST pseudo instruction controls the listing. An END pseudo instruction causes options to be reset to the default values.

Format:

Location	Result	Operand
name	LIST	$option_1, option_2, \ldots, option_n$

name Optional list name. If a name is present, the instruction is ignored unless a matching name is specified on a LIST parameter on the APML control statement. For example, if LIST=*name* appears on the APML control statement, LIST pseudos with a matching name are not ignored. If only LIST is specified on the APML statement, all LIST pseudo instructions are processed regardless of the location field name. LIST pseudos with a blank location field are always processed regardless of the control statement LIST parameters.

If L=0 is specified on the APML control statement, listing output is not generated. In this case, LIST pseudos and list options specified on the APML control statement have no effect.

You may specify all of the following option names as APML control statement parameters. The selection of an option on the APML control statement overrides the enabling or disabling of the corresponding feature by a LIST pseudo.

$option_i$ An option name specifying that a particular listing feature be enabled or disabled. You may specify zero, one, or more options. Defaults are enclosed in brackets. If no options are specified, OFF is assumed. The options are as follows:

- * Returns to previous LIST pseudo
- [ON] Enables source statement listing. Source statements and code generated are listed.

$option_i$ (continued)	OFF	Disables source statement listing. Only statements with errors are listed while this option is selected. If LIS option is enabled, listing control pseudo instructions are also listed. Default when operand field is blank.
	[XRF]	Enables cross-reference. Symbol references are accumulated and a cross-reference listing is produced.
	NXRF	Disables cross-reference. Symbol references are not accumulated. If this option is selected when the END pseudo is encountered, no cross-reference is produced. This does not affect the $XRF written by APML.
	[XNS]	Includes unreferenced local symbols in the cross-reference. Local symbols that were not referenced in the listing output are included in the cross-reference listing.
	NXNS	Excludes unreferenced local symbols in the cross-reference. If this option is selected when the END pseudo is encountered, local symbols that were not referenced in the listing output are not included in the cross-reference.
	[DUP]	Enables listing of duplicated statements. Statements generated by DUP and ECHO expansions are listed. Conditional statements and skipped statements generated by DUP and ECHO are not listed unless the macro conditional list feature (MIF) is enabled.
	NDUP	Disables listing of duplicated statements. Statements generated by DUP and ECHO are not listed.
	MAC	Enables listing of macro expansions. Statements generated by macro calls are listed. Conditional statements and skipped statements generated by macro calls are not listed unless the macro conditional list feature is enabled (MIF).
	[NMAC]	Disables listing of macro expansions. Statements generated by macro calls are not listed.

option$_i$ (continued)	MIF	Enables macro conditional listing. Conditional statements and skipped statements generated by a macro call, or by a DUP or ECHO pseudo instruction, are listed. The listing of macro expansions or the listing of duplicated statements must also be enabled. This option does not affect listing of conditional statements and skipped statements in source code (not macro expansions).
	[NMIF]	Disables macro conditional listing. Conditional statements and skipped statements are not listed.
	[MIC]	Enables listing of generated statements before editing. Statements generated by a macro call or by a DUP or ECHO pseudo instruction, and containing a micro reference or concatenation character are listed before and after editing. The listing of macro expansions or the listing of duplicated statements must also be enabled. Statements in source code (not macro expansions) containing a micro reference or a concatenation character are listed before editing regardless of this option.
	NMIC	Disables listing of generated statements before editing. Statements generated by a macro call, or by a DUP or ECHO pseudo instruction, are not listed before editing.
	LIS	Enables listing of listing control pseudo instructions, including LIST, SPACE, EJECT, TITLE, and SUBTITLE. These statements are listed regardless of whether the source statement listing is enabled.
	[NLIS]	Disables listing of listing control pseudo instructions
	[WEM]	Enables warning errors. Each statement containing a warning error is written to the source listing and the error listing. A logfile message is issued giving the number of warning errors.
	NWEM	Disables warning errors; warning errors are ignored.

TXT — Enables global text source listing. Each statement following a TEXT pseudo instruction is listed through the ENDTEXT instruction if the listing is otherwise enabled.

[NTXT] — Disables global text source listing. Statements following a TEXT pseudo instruction through the following ENDTEXT instruction are not listed.

[WMR] — Enables warning error message for macro redefinition. If the name of a macro is the same as a currently defined pseudo instruction or macro, a warning message is issued.

NWMR — Disables warning error message for macro redefinition

6.9.2 SPACE - LIST BLANK LINES

The SPACE pseudo instruction inserts blank lines in the output listing.

Format:

Location	Result	Operand
ignored	SPACE	*count*

count — An absolute expression specifying the number of blank lines to insert in the listing. When the base is M (mixed), APML assumes that count is decimal.

6.9.3 EJECT - BEGIN NEW PAGE

The EJECT pseudo instruction causes a page eject on the output listing.

Format:

Location	Result	Operand
ignored	EJECT	*ignored*

6.9.4 TITLE - SPECIFY LISTING TITLE

The TITLE pseudo instruction specifies the main title that appears on each page of the listing.

Format:

Location	Result	Operand
ignored	TITLE	'*character string*'

'*character string*'
A character string to be printed as the main title on subsequent pages of the listing. A maximum of 64 characters is allowed.

6.9.5 SUBTITLE - SPECIFY LISTING SUBTITLE

The SUBTITLE pseudo instruction specifies the subtitle that appears on each page of the listing.

Format:

Location	Result	Operand
ignored	SUBTITLE	'*character string*'

'*character string*'
A character string to be printed as the subtitle on subsequent pages of the listing. The instruction also causes a page eject. A maximum of 64 characters is allowed.

6.9.6 TEXT - BEGIN GLOBAL TEXT

The TEXT pseudo instruction declares the beginning of global text source. Source lines following the TEXT pseudo instruction up through the next ENDTEXT pseudo instruction are treated as global text source statements. These statements are listed only when the TXT listing option is enabled. A symbol defined in global text source is treated as a system text symbol for cross-reference purposes. That is, such a symbol is not listed in the cross-reference unless there is a reference to the symbol from a listed statement. The */block/* or system text *name* column of the cross-reference listing contains the text name.

Symbols defined in source text are global if the text appears prior to an IDENT pseudo instruction. Symbols in source text are local to a program module if the text appears between IDENT and END pseudo instructions.

The TEXT pseudo instruction is listed if the listing is ON or if the LIS listing option is enabled (regardless of other listing options).

The TEXT and ENDTEXT pseudo instructions have no effect within system text.

Format:

Location	Result	Operand
name	TEXT	'*character string*'

name Optional name of global text. This name is used as the name of the global text source following the TEXT pseudo instruction until the next ENDTEXT pseudo instruction. This name is associated with any symbols defined in the global text, and it is listed in the name column of the cross-reference listing.

'*character string*'
An optional character string to be printed as the subtitle of subsequent pages of the listing. This operand and the TXT option causes a page eject. A maximum of 64 characters is allowed. If the operand field is blank, the original subtitle is not affected and no page eject is performed. If the operand field is nonblank, the preceding subtitle is lost and replaced by the character string in the operand field.

6.9.7 ENDTEXT - TERMINATE GLOBAL TEXT

The ENDTEXT pseudo instruction terminates global text source initiated by a TEXT instruction. An IDENT or END pseudo instruction also terminates global text source. The ENDTEXT instruction is not listed unless the TXT option is enabled. If the LIS option is enabled, the ENDTEXT instruction is listed no matter what other listing options are enabled.

Format:

Location	Result	Operand
ignored	ENDTEXT	

Example (with TXT option off):

Source:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	TEXT	
CAT	EQUALS	17	
TXTNAME	TEXT	'An example.'	
DOG	EQUALS	231	
HAT	EQUALS	2	
	ENDTEXT		
	A=CAT		
	A=DOG		
	END		

Output:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
		IDENT	TEXT	
17	CAT	EQUALS	17	
	TXTNAME	TEXT	'An example.'	
0 010017		A=CAT		
1 010231		A=DOG		
		END		
17 CAT			1: 2 D 1: 4	
231 DOG	TXTNAME		1: 5	

6.10 SYMBOL DEFINITION PSEUDO INSTRUCTIONS

The pseudo instructions EQUALS, SET, CHANNEL, and MICSIZE define symbols used in the program.

Section 2, APML Assembly Language, gives requirements for symbols.

6.10.1 EQUALS - EQUATE SYMBOL

The EQUALS pseudo instruction defines a symbol with the value and attributes determined by the expression. The symbol is not redefinable.

Format:

Location	Result	Operand
symbol	EQUALS	*exp,attribute*

symbol An unqualified symbol. The symbol is implicitly qualified by the current qualifier. The symbol must not be defined already. If the location field is blank, no symbol is defined.

exp Any expression

attribute P, W, or V indicating parcel, word, or value attribute (optional). Attribute, if present, is used instead of the expression's attribute. An expression with word-address attribute is multiplied by 4 if a parcel-address attribute is specified; an expression with parcel-address attribute is divided by 4 if word-address attribute is specified. A relocatable expression cannot be specified as having value attribute.

Example:

Location	Result	Operand	Comment
1	10	20	35
SYMB	EQUALS	A*B+100/4	

6.10.2 SET - SET SYMBOL

The SET pseudo instruction resembles the EQUALS pseudo instruction; however, a symbol defined by SET is redefinable.

Format:

Location	Result	Operand
symbol	SET	*exp,attribute*

symbol An unqualified symbol. The symbol is implicitly qualified by the current qualifier. The symbol must not be defined already. If the location field is blank, no symbol is defined.

exp Any expression

attribute P, W, or V indicating parcel, word, or value attribute (optional). Attribute, if present, is used instead of the expression's attribute. An expression with word-address attribute is multiplied by 4 if a parcel-address attribute is specified; an expression with parcel-address attribute is divided by 4 if word-address attribute is specified. A relocatable expression cannot be specified as having value attribute.

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
100	SIZE	EQUALS	O'100	
22	PARAM	SET	D'18	
10	WORD	SET	*W	
40	PARCEL	SET	*P	
	SIZE	EQUALS	SIZE+1	.(Illegal)
24	PARAM	SET	PARAM+2	.(Legal)

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
		IDENT	EQUSET	
2	R1	EQUALS	2	
		BASEREG	R1	
1024	GEORGE	EQUALS	1024	
17	CAT	SET	17,P	
075002 /000017		P=CAT		
1031	CAT	SET	GEORGE+5	
		END		

6.10.3 CHANNEL - CHANNEL SYMBOL

The CHANNEL pseudo instruction defines a symbol which is recognized in APML symbolic instructions as being a channel mnemonic. By convention, symbols defined as channel mnemonics are 3 characters.

A symbol defined by the CHANNEL pseudo instruction has value attribute and a value which is taken to be the hardware channel number. A channel symbol must be defined before it is used in a symbolic APML instruction.

Format:

Location	Result	Operand
symbol	CHANNEL	*expression*

symbol A 1- to 8-character symbol name; by convention, 3 characters.

expression
An expression with a positive value less than 512

Example:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
		IDENT	CHANNEL	
5	BUF	CHANNEL	5	
140005		BUF:0		
140005		MOS:0		
		END		

NOTE

If an instruction is used that references the exit stack, register E, or the contents of an exit stack entry, designated (E), APML requires that the channel symbol PXS be defined. PXS is normally defined in a system text such as $APTEXT. APML does not assume names for other channels and does not require definition of any other channel symbols.

6.10.4 MICSIZE - SET REDEFINABLE SYMBOL TO MICRO SIZE

The MICSIZE pseudo instruction defines the symbol in the location field as an absolute symbol with a value equal to the number of characters in the micro string whose name is in the operand field. Another SET or MICSIZE instruction with the same symbol redefines the symbol to the new value.

Format:

Location	Result	Operand
symbol	MICSIZE	*name*

symbol An unqualified symbol; the symbol is implicitly qualified by the current qualifier. The location field can be blank.

name The name of a micro string that is previously defined

6.11 DATA DEFINITION PSEUDO INSTRUCTIONS

The pseudo instructions following generate object binary. The only other instructions that are translated into object binary are the symbolic APML instructions.

Pseudo Instruction	Description
CON	Places an expression value into one or more words
BSSZ	Generates one or more words containing zeros
DATA	Generates one or more words of numeric or character data
PDATA	Generates one or more parcels of numeric or character data
VWD	Generates a variable-width field of word-oriented data

6.11.1 CON - GENERATE CONSTANT

The CON pseudo instruction generates one or more full words of binary data. This pseudo always forces a word boundary.

Format:

Location	Result	Operand
symbol	CON	$exp_1,exp_2,...,exp_n$

symbol Optional symbol assigned the word address value of the location counter after the force to word boundary occurs

exp_i An expression whose value is to be inserted into a single 64-bit word. If an expression is blank, a single zero word is generated. A word boundary is forced before any operand field expressions are evaluated. A double-precision floating-point constant is not allowed.

Example:

Code generated	Location	Result	Operand
	1	10	20
0000000000000007777017	A	CON	O'7777017
0404401002004010020040		CON	A

6.11.2 BSSZ - GENERATE ZEROED BLOCK

The BSSZ pseudo instruction causes a block of words containing zeros to be generated. A force to word boundary occurs, and then the number of zero words specified by the operand field expression is generated.

Format:

Location	Result	Operand
symbol	BSSZ	*exp*

symbol Optional symbol assigned the word address value of the location counter after the force to word boundary occurs

exp An absolute expression with word address or value attribute whose value specifies the number of 64-bit words containing zeros to be generated. A blank operand field results in no data generation. The expression value must be positive, and all symbols must be previously defined.

The left margin of the listing shows the octal word count.

Example:

Code generated	Location	Result	Operand
	1	10	20
144		BSSZ	D'100

6.11.3 DATA - GENERATE DATA WORDS

The DATA pseudo instruction generates data from the items listed. The length of the field generated for each data item depends on the type of constant involved. A word boundary is not forced between data items.

Format:

Location	Result	Operand
symbol	DATA	$data_1, data_2, \ldots, data_n$

symbol Optional symbol assigned to the address value of the location counter after a force to word boundary. If no symbol is present, a force to word boundary does not occur.

$data_i$ A numeric or character data item

Example:

Code generated	Location	Result	Operand
	1	10	20
0000000000000000005252		DATA	O'5252,A'ABC'R
0000000000000020241103			
0405022064204010020040		DATA	'ABCD'
0425062164404010020040		DATA	'EFGH'
040502206420		DATA	'ABCD'*
10521443510		DATA	'EFGH'*
0000000000000000000000		DATA	'ABCD'12R
040502206420			
10521443510		DATA	'EFGHIJ'*
044512			
0405022064204010020040	LL2	DATA	'ABCD'
		.	
		.	
0000000000000000000144		DATA	100
		.	
		.	
0521102225144022251440		DATA	
0404402324252324640507			
0424			'THIS IS A MESSAGE'*L
000		VWD	8/0

6.11.4 PDATA - GENERATE DATA PARCELS

The PDATA pseudo instruction is equivalent to the data generation statements described in section 5, Basic IOP Hardware Instruction Set, for symbolic APML instructions. When using the PDATA pseudo instruction, the data items are listed in the operand field, whereas in symbolic APML data generation statements the data items are listed in the assignment field with no mnemonic operation name. By using PDATA, some data items can be used which otherwise would not be allowed. For instance, you cannot use symbolic names A, EXIT, PASS, B, and E in a symbolic APML data definition because of conflicts with special names and registers in APML instruction syntax.

Format:

Location	Result	Operand
symbol	PDATA	$item_1, item_2, \ldots, item_n$

symbol Optional symbol; if present, APML forces an instruction page boundary.

$item_i$ A symbol, numeric constant, character data item, or item of the form <k> or <<k>>, where *k* is a symbol or numeric constant. See section 5, Basic IOP Hardware Instruction Set, for a more detailed description.

Example:

Code generated	Location	Result	Operand
	1	10	20
		IDENT	PDATA
	R1	EQUALS	2
	A	EQUALS	217
000217 000002 000007	DOG	PDATA	A,R1,7
		PDATA	'DATA ITEM'
		PDATA	<10>
		END	

6.11.5 VWD - VARIABLE WORD DEFINITION

The VWD pseudo instruction allows data to be generated in fields from 0 to 64 bits wide. Fields may cross word boundaries. Data begins at the current bit position unless a symbol is used; in which case, a force word boundary occurs and the data begins at the new current bit position.

Format:

Location	Result	Operand
symbol	VWD	$n_1/exp_1, n_2/exp_2, \ldots, n_m/exp_m$

symbol Optional symbol; if present, a force to word boundary occurs.

n_i Field width, specifying the number of bits in the field. A numeric constant or symbol, with absolute and value attributes. The value of n_i must be positive and less than or equal to 64. When the base is M (mixed), APML assumes that n_i is decimal.

exp_i An expression whose value is to be inserted in the field

Example:

In the following example, the value of SIGN is 1, the value of FC is 0, the value of ADD is 653 (octal), and the value of DSN is $IN in ASCII code.

Code generated	Location	Result	Operand
	1	10	20
		BASE	M
	PDT	BSS	0
1000000000000023440515		VWD	1/SIGN,3/0,60/A'"NAM"'R
10000000653		VWD	1/1,6/FC,24/ADD
37	REMDR	EQUALS	64-*W
00011044516		VWD	REMDR/DSN

6.12 CONDITIONAL ASSEMBLY PSEUDO INSTRUCTIONS

The instructions described in this subsection permit optional assembly or skipping of source code. The conditional pseudo instructions IFA, IFC, or IFE determine whether a sequence of instructions following the test is to be skipped or assembled. The end of the conditional sequence is determined by a count of instructions provided on the test instruction or by an ENDIF pseudo instruction with a matching location field name.

The ELSE pseudo instruction provides a means of reversing the effect of a previous IFA, IFE, IFC, SKIP, or ELSE instruction. The SKIP pseudo instruction unconditionally skips following statements.

When skipping under control of a statement count, comment statements (asterisk in column 1) and continuation lines (comma in column 1) are not included in the statement count.

6.12.1 IFA - TEST EXPRESSION ATTRIBUTE FOR ASSEMBLY CONDITION

The IFA pseudo instruction tests an attribute of an expression. If the expression has the specified attribute, assembly continues with the next statement. If the attribute test is failed, subsequent statements are skipped. If a location field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered. Otherwise, skipping stops when the statement count is exhausted.

If an assembly error is detected, assembly continues with the next instruction.

Formats:

Location	Result	Operand
ifname	IFA	*attribute,exp*
	IFA	*attribute,exp,count*

ifname Optional name of conditional sequence of code

attribute A mnemonic signifying an attribute of *exp*. An expression has one and only one of the attributes PA, WA, or VAL and has one and only one of the attributes EXT, REL, or ABS.

An attribute may also be any of the following letters preceded by a complement sign (#) indicating that the second subfield does not satisfy the corresponding condition.

Mnemonic	Significance
PA	The expression *exp* has parcel-address attribute.
WA	The expression *exp* has word-address attribute.
VAL	The expression *exp* has value attribute.
EXT	The expression *exp* has external attribute.
REL	The expression *exp* has relocatable attribute.

attribute (continued)	Mnemonic	Significance
	ABS	The expression *exp* has absolute attribute.
	DEF	All symbols in the expression *exp* have been previously defined.
	SET	The symbol in the second subfield is a redefinable symbol.
	MIC	The name in the second subfield is a micro name.

exp The second subfield must either be a valid expression, symbol, name, or character string depending on the attribute mnemonic.

For PA, WA, VAL, EXT, REL, ABS, and COM, the second subfield must be a valid expression with all symbols previously defined.

For DEF, the second subfield must be a valid expression.

For SET, the second subfield must be a valid defined symbol.

For MIC, the second subfield must be a valid name.

Expressions are evaluated in pass 1. Expressions that are relocatable addresses in local blocks have values relative to the beginning of the local block rather than the program block. Address expressions in a local block other than the nominal block on an absolute assembly are considered relocatable in pass 1.

count Statement count; must be an absolute expression with positive value. When the base is M (mixed), APML assumes that count is decimal. A count parameter is required if *ifname* is missing, otherwise, it is ignored. A missing or null subfield gives a zero count.

6.12.2 IFE - TEST EXPRESSIONS FOR ASSEMBLY CONDITION

The IFE pseudo instruction tests a pair of expressions for a condition under which code is to be assembled if the relation specified by the operation (*op*) is satisfied. That is, if the relationship is true, assembly resumes with the next statement. If the relationship is not satisfied (is false), subsequent statements are skipped. If a location field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered; otherwise, skipping stops when the statement count is exhausted.

If an assembly error is detected, assembly continues with the next statement.

Format:

Location	Result	Operand
ifname	IFE	exp_1,*op*,exp_2
	IFE	exp_1,*op*,exp_2,*count*

ifname Optional name of a conditional sequence of code

exp_1, exp_2
Expressions to be compared. All symbols in the expression must be previously defined.

Expressions are evaluated in pass 1. Expressions that are relocatable addresses in local blocks have values relative to the beginning of the local blocks rather than the program block. Address expressions in a local block other than the nominal block in an absolute assembly are considered relocatable in pass 1.

op Specifies relation to be satisfied by exp_1 and exp_2. It must be one of the following:

LT Less than; the value of exp_1 must be less than the value of exp_2.

LE Less than or equal to; the value of exp_1 must be less than or equal to exp_2.

GT Greater than; the value of exp_1 must be greater than the value of exp_2.

GE Greater than or equal to; the value of exp_1 must be greater than or equal to exp_2.

EQ Equal; the value of exp_1 must be equal to thevalue of exp_2. The expressions must either both be absolute, or both be external relative to the same external symbol, or both be relocatable in the same block. The word-address, parcel-address, or value attributes must be the same.

NE Not equal; the expressions exp_1 and exp_2 do not satisfy the conditions required for EQ described previously.

count Statement count; must be an absolute expression with positive value. When the base is M (mixed), APML assumes that count is decimal. A count parameter is required if *ifname* is missing, otherwise, it is ignored. A missing or null count subfield gives a zero count.

6.12.3 IFC - TEST CHARACTER STRINGS FOR ASSEMBLY CONDITION

The IFC pseudo instruction tests a pair of character strings for a condition under which code is to be assembled if the relation specified by the operation (*op*) is satisfied. That is, if the relationship is not satisfied (is false), subsequent statements are skipped. If a location field name is present, skipping stops when an ENDIF or ELSE pseudo instruction with the same name is encountered; otherwise, skipping stops when the statement count is exhausted.

If an assembly error is detected, assembly continues with the next statement.

Format:

Location	Result	Operand
ifname	IFC	'$char_1$',*op*,'$char_2$'
	IFC	'$char_1$',*op*,'$char_2$',*count*

ifname Optional name of a conditional sequence of code

'$char_1$','$char_2$'

Character strings to be compared. The first and third subfields may be null (empty) indicating a null character string.

The ASCII character code value of each character in $char_1$ is compared with the value of each character in $char_2$, beginning at the left and continuing until an inequality is found or until the longer string is exhausted. A zero value is required for missing characters in the shorter string.

See appendix A, Character Sets, for the ASCII character code values.

Micros and formal parameters may be contained in the character strings.

A character string may be delimited by a character other than an apostrophe. You can use any ASCII character other than a comma or space. Two consecutive occurrences of the delimiting character indicates a single such character. For example,

```
AIF IFC = O'100=,EQ,*ABCD***
```

compares the character strings O'100 and ABCD*.

op Relation to be satisfied by $char_1$ and $char_2$. It must be one of the following:

- LT Less than
- LE Less than or equal to
- GT Greater than
- GE Greater than or equal to
- EQ Equal to
- NE Not equal to

count Statement count; must be an absolute expression with positive value. A missing or null count subfield gives a zero count. If the base is M (mixed), APML assumes that count is decimal. A count parameter is required if *ifname* is missing; otherwise, it is ignored.

6.12.4 SKIP - UNCONDITIONALLY SKIP STATEMENTS

The SKIP pseudo instruction unconditionally skips subsequent statements. If a location field name is present, skipping stops when an ENDIF or ELSE with the same name is encountered. Otherwise, skipping stops when the statement count is exhausted.

Format:

Location	Result	Operand
ifname	SKIP	*count*

ifname Optional name of conditional sequence of code

count Statement count; must be an absolute expression with positive value. If the base is M (mixed), APML assumes that count is decimal. A count parameter is required if *ifname* is missing; otherwise, it is ignored. A missing or null count subfield gives a zero count.

6.12.5 ENDIF - END CONDITIONAL CODE SEQUENCE

The ENDIF pseudo instruction terminates skipping initiated by a IFA, IFE, IFC, ELSE, or SKIP pseudo instruction with the same location field name. Otherwise, ENDIF acts as a do-nothing pseudo instruction. ENDIF has no effect on skipping which is controlled by a statement count.

Format:

Location	Result	Operand
ifname	ENDIF	

ifname Required name of conditional code sequence

NOTE

An END statement encountered while skipping is recognized and terminates skipping.

6.12.6 ELSE - TOGGLE ASSEMBLY CONDITION

The ELSE pseudo instruction terminates skipping initiated by an IFA, IFC, IFE, ELSE, or SKIP pseudo instruction with the same location field name. If statements are currently being skipped under control of a statement count, ELSE has no effect.

If the assembler is not currently skipping statements, ELSE initiates skipping. Skipping is terminated by an ENDIF or ELSE pseudo instruction with a matching location field name.

Format:

```
|Location |Result  |Operand
|         |        |
|ifname   |ELSE    |
```

ifname Required name of conditional sequence of code

Conditional assembly examples:

```
|Location |Result  |Operand        |Comment
|1        |10      |20             |35
|         |        |               |
|         |IFA     |#DEF,A,1       |
|A        |EQUALS  |10             |.Define A if not already defined
|         |.       |               |
|         |.       |               |
|         |.       |               |
|BTEST    |IFA     |ABS,SYM        |
|X        |ERROR   |               |.Generate X error if SYM absolute
|BTEST    |ELSE    |               |
|         |CON     |SYM            |.Assemble if SYM not absolute
|BTEST    |ENDIF   |               |
|         |.       |               |
|         |.       |               |
|         |.       |               |
|*        |Assemble BSSZ instruction if W.* is less than BUF,
|*        |otherwise assemble ORG  |
|         |IFE     |W.*,LT,BUF,2   |
|         |BSSZ    |BUF-W.*        |.Generate words of zero to
|*        |        |               |.address BUF
|         |SKIP    |1              |.Skip next statement
|         |ORG     |BUF            |
|         |.       |               |
|         |.       |               |
|         |.       |               |
|         |IFC     |'"L"',EQ,,1    |
|         |ERROR   |               |.Error if micro string defined
|*        |        |               |.by L is empty
|         |.       |               |
|         |.       |               |
|         |.       |               |
|X        |IFC     |'ABCD',GT,'ABC'.ABCD is greater than ABC
|         |.       |               |
|         |.       |               |
|         |.       |               |
|Y        |IFC     |' ',GT,        |.Single space is greater than
|*        |        |               |.null string
|Z        |IFC     |'''',EQ,*'*    |.Single apostrophe equals single
|*        |        |               |.apostrophe
```

6.13 INSTRUCTION DEFINITION PSEUDO INSTRUCTIONS

The APML assembler allows you to identify a sequence of instructions to be saved for assembly at a later point in the source program. When the sequence is defined, APML stores it in a list of definitions but does not assemble the sequence. Each time the defined sequence is referenced, the sequence is placed in the source program and is assembled. Defined sequences are of three types: macro, dup, and echo.

A macro definition identifies a sequence of instructions. This instruction sequence is referenced at a later point in the source program by a single instruction, the macro call. Each time the macro call occurs, the definition sequence is placed in the source program. For a macro call, the name in the result field matches the name associated with the macro. Thus, a macro call resembles a pseudo instruction.

A dup or echo definition identifies a sequence of instructions which is assembled repeatedly, immediately following the definition. The number of times the sequence is assembled depends on the parameters on the DUP or ECHO pseudo.

A macro is defined as global if it occurs before the IDENT that begins the program module. Macro definitions are local if they occur within an IDENT, END sequence. Every local definition is removed from the assembler tables at the end of a program module. A global definition may be referenced in any program module following the definition.

The body of the definition begins with the first instruction following the header. The body consists of a series of APML instructions other than END and can include other definitions and calls. However, a definition used within another definition is not recognized until the definition in which it is contained is called. Therefore, an inner definition cannot be called before the outer definition is called for the first time.

A comment statement identified by an asterisk in the first nonblank column is ignored in the definition header or definition body. Such comments are not saved as a part of the definition sequence. Comment fields on other statements in the body of a definition are saved.

The body of the definition is saved before editing for micros, concatenation marks, and lowercase comments. Editing occurs when the definition is assembled each time it is called. An inner nested definition is not edited until it is called. ENDDUP, ENDM, END, and LOCAL pseudo instructions and prototype statements cannot contain any micros or concatenation characters. These statements are not edited when they occur in a definition.

The end of a macro definition is signaled by an ENDM pseudo instruction with the proper name in the location field. The end of a dup or echo definition is signaled by a statement count or by an ENDDUP with the proper name in the location field.

Each time a definition sequence of code is referenced (called), an entry is made in a pushdown stack called the assembly source stack. The most recent entry indicates the current source of statements to be assembled. When a definition is called within a definition sequence being assembled, another entry is made in the stack, and assembly continues with the new definition sequence belonging to the inner, or nested, call. When the end of a definition sequence is reached, the most recent stack entry is removed and assembly continues with the previous stack entry. When the stack becomes empty, assembly continues with statements from the source file.

An inner nested call may be recursive; that is, it may reference the same definition referenced by an outer call. The depth of nested calls permitted by APML is limited only by the amount of memory available.

An inner definition must be entirely contained within the next outer definition.

Skipping of statements due to conditional assembly must not extend beyond the end of a definition sequence being assembled. An error is generated and skipping is terminated if this condition occurs.

The sequence field in the right margin of the listing shows the definition name and nesting depth for definition sequences being assembled.

Formal parameters are defined in the definition header. Formal parameters recognized are: positional, keyword, and local. Formal parameters are recognized in the definition body whenever they are delimited by a space, comma, beginning or end of a statement, or any of the following characters:

    ! " # & ' ( ) * + - . / < = > | _

There may be from 0 to 511 formal parameters. Positional, keyword, and local parameters must all have unique names within a given definition.

You should not use END, ENDM, ENDDUP, or LOCAL as formal parameter names. When the definition is referenced, substitution of actual arguments will occur in any pseudo instruction with these names contained in any inner definition.

6.13.1 MACRO DEFINITION FORMAT

A macro definition may be called by an instruction of the following format:

Location	Result	Operand
loc	*name*	$a_1,a_2,\ldots,a_j,f_1=b_1,f_2=b_2,\ldots,f_k=b_k$

loc — Location field argument; must be a valid name. If a location field parameter is specified on the macro definition, this symbol is optional. It is substituted wherever the location field parameter occurs in the definition.

If no location field parameter is specified in the definition, this field must be empty.

name — Macro name; must match the name specified in the macro definition.

a_i — Actual argument string corresponding to positional parameters in the definition prototype statement

The first argument a_1 is substituted for the first positional parameter p_1 in the prototype operand field, the second argument a_2 is substituted for the second positional parameter p_2, and so on. If the number of operand subfields is less than the number of positional parameters, null argument strings are used for the missing arguments.

Two consecutive commas indicate a null (empty) argument string.

f_i — A keyword parameter. Each keyword parameter f_i must match a keyword parameter in the macro definition. The keyword parameters may be listed in any order; they do not need to match the order given in the macro definition. The default arguments specified in the macro definition are used as the actual argument for missing keyword parameters.

Keyword parameters are not recognized until after *n* subfields (*n* commas), where *n* is the number of positional parameters in the operand field of the macro definition.

b_i — Actual argument string for keyword parameter f_i. A space or comma following the equal sign indicates a null (empty) argument string.

An actual argument string may consist of any ASCII characters except comma or blank. A comma separates subfields and a blank terminates the operand field.

If the first character of the actual argument is a left parenthesis, the string must be terminated by a matching right parenthesis. Such an argument is called an embedded argument and consists of all characters between the enclosing parentheses. An embedded string may contain commas and blanks and may also contain pairs of matching left and right parentheses.

The actual argument string for each positional and keyword parameter is substituted in the definition sequence wherever the formal parameter occurs. Embedded argument strings are substituted without the enclosing parentheses.

6.13.2 MACRO - MACRO DEFINITION

The MACRO pseudo instruction is the first statement of a macro definition. The macro header consists of the MACRO pseudo instruction, a prototype statement, and optional LOCAL pseudo instructions.

Format:

Location	Result	Operand	
ignored	MACRO		HEADER:
lfp	*name*	$p_1,p_2,\ldots,p_n,e_1=d_1,e_2=d_2,\ldots,e_m=d_m$	Prototype statement
	LOCAL	$sym_1,\ldots,sym_r$	Optional local pseudo instructions
	.		
	.		Definition body
	.		
	.		
name	ENDM		Definition end

Prototype statement parameters:

lfp Optional location field parameter. It must be a valid name. If present, it is a positional parameter.

name Name of the macro; must be valid name. If the name is the same as a currently defined pseudo instruction or macro, this definition redefines the operation associated with the name and a warning message is issued (see appendix C, Messages, and appendix D, Assembly Errors).†

p_i Positional parameter; must be a valid name. There may be none, one, or more positional parameters.

† Warning error depends on the WMR and NWMR features of the APML control statement or the LIST pseudo instruction.

e_i Keyword parameter; must be a valid name. There may be none, one, or more keyword parameters.

d_i Default argument for keyword parameter e_i. An argument string may consist of any string of ASCII characters except comma or blank.

If the first character of the default argument d_i is a left parenthesis, the string must be terminated by a matching right parenthesis. Such an argument is called an embedded argument and consists of all characters between the enclosing parentheses. An embedded string may contain commas and blanks and may also contain pairs of matching left and right parentheses.

A space or comma following the equal sign specifies a null (empty) character string as the default argument.

The default argument for a positional parameter is an empty string.

An inner macro definition must be entirely contained within the outer definition.

6.13.3 LOCAL - SPECIFY LOCAL SYMBOLS

The LOCAL pseudo instruction specifies symbols which are defined only within the macro definition. The LOCAL pseudo instruction also defines any of the named symbols used within an inner definition or call that are not defined as local to that inner usage.

On each macro call and each repetition of a dup or echo definition sequence, the assembler creates a unique symbol for each local parameter and substitutes the created symbol for the local parameter on each occurrence within the definition. The symbol created for local parameters has the form %%*nnnnnn*, where *n* is an octal digit.

A symbol not defined as local in a definition may be referenced outside an assembly of the definition sequence.

One or more LOCAL pseudo instructions may appear in a macro, dup, or echo definition. The LOCAL pseudo instructions must follow the macro prototype statement or DUP or ECHO pseudo instructions, except for intervening comment statements.

Format:

Location	Result	Operand
ignored	LOCAL	$sym_1, sym_2, \ldots, sym_n$

sym$_i$ Symbols that are to be rendered local to the definition

6.13.4 ENDM - END MACRO DEFINITION

An ENDM pseudo instruction terminates the body of a macro definition.

Format:

Location	Result	Operand
name	ENDM	

name Name of a macro definition sequence. The name must match the name appearing in the result field of the macro prototype.

Example 1. Macro with positional parameters:

Macro definition:

Location	Result	Operand	Comment
1	10	20	35
	MACRO		
SYMBOL	NEXT	VALUE	
	IFC	#VALUE#,NE,,1	
$NEXT	SET	VALUE	
	IFC	#SYMBOL#,NE,,1	
SYMBOL	EQUALS	$NEXT	
$NEXT	SET	$NEXT+1	
NEXT	ENDM		

Macro calls:

Location	Result	Operand	Comment
1	10	20	35
ABC	NEXT	3	
	.		
	.		
	.		
ABCD	NEXT		

Macro expansion:

Code generated	Location	Result	Operand
	1	10	20
		IFC	#3#,NE,,1
3	$NEXT	SET	3
		IFC	#ABC#,NE,,1
3	ABC	EQUALS	$NEXT
4	$NEXT	SET	$NEXT+1
		.	
		.	
		.	
		IFC	##,NE,,1
(skipped)	$NEXT	SET	
		IFC	#ABCD#,NE,,1
4	ABCD	EQUALS	$NEXT
5	$NEXT	SET	$NEXT+1

The operand field parameter was omitted on the second macro call, so a null character string was substituted for each occurrence of the parameter value.

Example 2. Macro with positional and keyword parameters

Macro definition:

Location	Result	Operand	Comment
1	10	20	35
	MACRO		
	TABLE	TABN,VAL1=#0,VAL2=,VAL3=0	
	BLOCK	TABLES	
TABN	CON	'TABN'L	
	CON	VAL1	
	CON	VAL2	
	CON	VAL3	
	BLOCK	*	.Resume use of previous block
TABLE	ENDM		

Macro call:

Location	Result	Operand	Comment
1	10	20	35
	TABLE	TABA,VAL3=4,VAL2=A	

Macro expansion:

Location	Result	Operand	Comment
1	10	20	35
	BLOCK	TABLES	
TABA	CON	'TABA'L	
	CON	#0	
	CON	A	
	CON	4	
	BLOCK	*	.Resume use of previous block
TABLE	ENDM		

Example 3. Macro with positional and keyword parameters

Macro definition:

Location	Result	Operand	Comment
1	10	20	35
	MACRO		
BAG	IDLE	COUNT,CAT=6	.Prototype
	LOCAL	XXXXXXXX	
BAG	A=COUNT		.Definition
XXXXXXXX	A=A-1		.Definition
	P=XXXXXXXX,A#0		.Definition
	B=CAT		.Definition
IDLE	ENDM		

Macro call:

Code generated	Location	Result	Operand
	1	10	20
		IDENT	CALL
7	NUM	SET	7
		IDLE	NUM,CAT=24

Macro expansion:

Code generated	Location	Result	Operand
	1	10	20
010007		A=NUM	
013001	%%000000	A=A-1	
107001		P=%%000000,A#0	
010024 054000		B=24	
	SHRIMP	IDLE	NUM
010007	SHRIMP	A=NUM	
	,		
013001	%%000001	A=A-1	
107001		P=%%000001,A#0	
010006 054000		END	

6.13.5 OPSYN - SYNONYMOUS OPERATION

The OPSYN pseudo instruction defines or redefines a name in the location field as being the same as the named operation in the operand field. A previous definition with a name matching the location field name is no longer available. Any pseudo instruction or macro may be redefined in this manner.

An operation defined by OPSYN is global if the OPSYN pseudo occurs before the IDENT pseudo that begins a program module, and it is local if the OPSYN pseudo appears with an IDENT, END sequence. Global operations may be referenced in any program module following the definition. Every local operation is removed at the end of a program module, making any previous global definition with the same name available again.

Format:

Location	Result	Operand
$name_1$	OPSYN	$name_2$

$name_1$ A valid name or the name of a defined operation such as a pseudo instruction or macro. $name_1$ must not be blank.

$name_2$ The name of a defined operation. If $name_2$ is blank, $name_1$ becomes a do-nothing pseudo instruction.

Example:

In the following example, OPSYN redefines the pseudo instruction IDENT with a macro definition.

OPSYN definition:

Location	Result	Operand	Comment
1	10	20	35
IDENTT	OPSYN	IDENT	
	MACRO		
	IDENT	NAME	
	LIST	OFF,NXRF	
NAME	LIST	ON,XRF	.Processed if LIST=NAME
			.on APML statement
	IDENTT	NAME	
IDENT	ENDM		

OPSYN call and expansion:

Location	Result	Operand	Comment
1	10	20	35
	IDENT	A	
	LIST	OFF,NXRF	
A	LIST	ON,XRF	
	IDENTT	A	

6.14 CODE DUPLICATION PSEUDO INSTRUCTIONS

APML provides a set of four instructions (DUP, ECHO, ENDDUP, and STOPDUP), which allow multiple assemblies of sequences of source statements.

6.14.1 DUP - DUPLICATE CODE

The DUP pseudo instruction introduces the definition of a sequence of code which is assembled repetitively immediately following the definition. The dup sequence is assembled the number of times specified on the DUP instruction. The DUP sequence to be repeated consists of statements following the DUP instruction and any optional LOCAL pseudo instructions. Comment statements are ignored; the dup sequence ends when the statement count is exhausted or when a ENDDUP with a matching location field name is encountered.

A nested inner DUP definition must be entirely contained in the outer definition.

You may use STOPDUP to override the repetition count.

Format:

Location	Result	Operand
dupname	DUP	*times*
	or	
	DUP	*times,count*

dupname Name of the DUP sequence, required if the count field is null or missing. Use *dupname* to match an ENDDUP name if no count field is present. Also use *dupname* in the sequence field of the listing for the DUP expansion.

times An absolute expression with positive value, specifying number of times to repeat the code sequence. If the value is 0, the code is skipped.

count Optional absolute expression with positive value, specifying the number of statements to be duplicated. LOCAL pseudo instructions and comment statements (* in first nonblank column) are ignored for the purpose of this count. Statements are counted before expansion of nested macro calls or DUP or ECHO sequences.

6.14.2 ECHO - DUPLICATE CODE WITH VARYING ARGUMENTS

The ECHO pseudo instruction introduces the definition of a sequence of code that is assembled repetitively immediately following the definition. On each repetition, the actual arguments are substituted for the formal parameters until the longest argument list is exhausted. The echo sequence to be repeated consists of statements following the ECHO pseudo instruction and any optional LOCAL pseudo instructions. Comment statements are ignored. The echo sequence ends when an ENDDUP with a matching location field name is encountered.

A nested inner echo definition must be entirely contained in the outer definition.

STOPDUP overrides the repetition count determined by the number of arguments in the longest argument list.

Format:

Location	Result	Operand
dupname	ECHO	e_1=$list_1$,e_2=$list_2$,...,e_n=$list_n$

dupname Name of the echo sequence; must not be empty. This name must match the location field name in the ENDDUP pseudo instruction that terminates the echo sequence.

e_i Formal parameter name. There may be none, one, or more e_i parameters.

$list_i$ List of actual arguments. The list can be a single argument a_{i1} or a parenthesized list of arguments (a_{i1},a_{i2},...,a_{im}), where each a_{ij} is an actual argument to be substituted for e_i in the echo sequence. Each actual argument a_{ij} may be an ASCII character string not containing blanks or commas or may itself be an embedded argument containing a list of arguments a_{ij} enclosed in matching parentheses. An embedded argument may contain blanks or commas and matched pairs of parentheses.

The argument a_{i1} is substituted for e_i in the echo sequence on the first repetition; a_{i2} is substituted for e_i on the second repetition.

A comma immediately followed by another comma or closing right parenthesis specifies a null (empty) character string as the argument.

6.14.3 ENDDUP - END DUPLICATED CODE

The ENDDUP pseudo instruction ends the definition of the code sequence to be repeated. An ENDDUP terminates a DUP or ECHO sequence with the same name. ENDDUP has no effect on DUP or ECHO sequences terminated by a statement count.

Format:

Location	Result	Operand
dupname	ENDDUP	

dupname Name of a DUP sequence

6.14.4 STOPDUP - STOP DUPLICATION

The STOPDUP pseudo instruction stops duplication of a code sequence indicated by a DUP or ECHO pseudo instruction. It overrides the repetition count. Assembly of the current repetition of the DUP sequence is terminated immediately. STOPDUP terminates the innermost DUP or ECHO sequence with the same name. STOPDUP does not affect the definition of the code sequence to be duplicated.

Format:

Location	Result	Operand
dupname	STOPDUP	

dupname Name of a DUP sequence

6.14.5 EXAMPLES OF DUPLICATED SEQUENCES

Example 1. Use DUP to define an array with values 0, 1, 2, and 3.

DUP definition:

Location	Result	Operand	Comment
1	10	20	35
S	EQUALS	W.*	
	DUP	3,1	
	CON	W.*-S	

DUP expansion:

Code generated	Location	Result	Operand	Comment
	1	10	20	35
0000000000000000000000		CON	W.*-S	.(W.*-S=0)
0000000000000000000001		CON	W.*-S	.(W.*-S=1)
0000000000000000000002		CON	W.*-S	.(W.*-S=2)
0000000000000000000003		CON	W.*-S	.(W.*-S=3)

Example 2. Nested duplication

ECHO definition:

Location	Result	Operand	Comment
1	10	20	35
X	ECHO	CHN=(PFR,PXS,LME)	
Y	ECHO	FCN=(0,7)	
	CHN:FCN		
Y	ENDDUP		
X	ENDDUP		

ECHO and DUP expansion:

Location	Result	Operand	Comment
1	10	20	35
	PFR:0		
	PFR:7		
	PXS:0		
	PXS:7		
	LME:0		
	LME:7		

Example 3. Use STOPDUP to terminate duplication

STOPDUP definition:

Location	Result	Operand	Comment
1	10	20	35
T	SET	0	
A	DUP	1000	
T	SET	T+1	
	IFE	T,EQ,3,1	.Terminate duplication when T=3
A	STOPDUP		
	CON	T	
A	ENDDUP		

STOPDUP expansion:

Location	Result	Operand	Comment
1	10	20	35
T	SET	T+1	
	CON	T	
T	SET	T+1	
	CON	T	
T	SET	T+1	
A	STOPDUP		

6.15 MICRO DEFINITION PSEUDO INSTRUCTIONS

Micros allow you to assign a name to a character string and subsequently refer to the character string through use of its name. A reference to a micro results in the character string being substituted for the name before assembly of the source statement containing the reference.

6.15.1 MICRO REFERENCE FORMAT

Refer to a micro by using the micro name enclosed by quote marks anywhere in a source statement other than a comment line. If column 72 of a line is exceeded as a result of a micro substitution, the assembler creates additional continuation lines. No replacement occurs if the micro name is unknown or if one of the micro marks has been omitted.

Example:

A micro named PFX is defined as ID. A reference to PFX is in the location field of a line:

Location	Result	Operand	Comment
1	10	20	35
"PFX"TAG	B = DD		

However, before the line is interpreted, APML substitutes the definition for PFX producing the following line:

Location	Result	Operand	Comment
1	10	20	35
IDTAG B	= DD		

6.15.2 MICRO - MICRO DEFINITION

The MICRO pseudo instruction assigns a name to a character string.

Format:

Location	Result	Operand
name	MICRO	'*character string*',exp_1,exp_2
	or	
name	MICRO	'*character string*',exp_1
	or	
name	MICRO	'*character string*'

name — Micro name. If *name* is previously defined, the previous micro definition is lost.

'*character string*'
: A character string optionally including previously defined micros.

  To specify a single apostrophe in a character string, use two adjacent apostrophes. These are counted as a single character in the string.

  A character string may be delimited by a character other than an apostrophe; use any ASCII character other than a comma or space. Two consecutive occurrences of the delimiting character indicates a single such character. For example, a micro consisting of the single character * could be specified as '*' or ****.

exp_1 — Absolute expression indicating number of characters in the micro character string

  The micro character string is terminated either by the character count or the final apostrophe of the character string, whichever occurs first. The string is considered empty if exp_1 has a 0 or negative value. exp_1 is considered very large if it is null. In this case, the string is terminated by the final apostrophe.

exp_2	Absolute expression indicating starting character. The micro character string is considered to begin with the first character of the character string if exp_2 is null, exp_2 has the value of 0 or 1, or exp_2 is negative.

Example:

```
|Location |Result   |Operand         |Comment
|1        |10       |20              |35
|         |         |                |
|MIC      |MICRO    |'THIS IS A MICRO STRING'
|MIC1     |MICRO    |****            |.Micro string is 1 asterisk
|MIC2     |MICRO    |'"MIC"',1       |.Micro consisting of 1st
|         |         |                |.character of the micro string
|         |         |                |.represented by MIC
|MIC2     |MICRO    |'THIS IS A MICRO STRING' ,1
|MIC4     |MICRO    |'"MIC"',2,2     |.Micro consisting of 2nd and
|         |         |                |.3rd characters of micro string
|         |         |                |.represented by MIC
|MIC4     |MICRO    |'THIS IS A MICRO STRING' ,2,2
|MIC5     |MICRO    |                |.Blank operand field defines
|         |         |                |.an empty string
```

6.15.3 OCTMIC AND DECMIC - OCTAL AND DECIMAL MICROS

OCTMIC and DECMIC convert the value of the expression into a character string that is assigned a micro name.

Formats:

Location	Result	Operand
name	OCTMIC	*exp,count*
name	DECMIC	*exp,count*

name	Micro name
exp	An absolute expression to be converted to up to 8 characters representing the octal (or decimal) value
count	An expression providing an optional character count less than or equal to 8. If this parameter is present, leading zeros are supplied to provide the requested number of characters.

Example of MICSIZE and DECMIC:

	Location	Result	Operand	Comment
	1	10	20	35
26	V	MICSIZE	MIC	.The value of V is the number
				.of characters in the micro
				.string represented by MIC
2	VOCT	DECMIC	V,2	.VOCT is a micro name
				.There are VOCT characters
				.in MIC
				.There are 26 characters in
				.MIC

Example of OCTMIC:

Location	Result	Operand	Comment
1	10	20	35
IP	EQUALS	O'20	
VAL	OCTMIC	IP	
MSG	DATA	'THE VALUE OF IP IS VAL'	
	DATA	'THE VALUE OF IP IS 20'	

6.15.4 PREDEFINED MICROS

In addition to the preceding micros, the APML assembler provides the following predefined micros.

Micro	Description
$DATE	Current date *yy/mm/dd*
$JDATE	Julian date *yy/dd*
$TIME	Time of day *hh:mm:ss*
$MIC	Micro character (quote, ASCII 042)
$CNC	Concatenation character (underline, ASCII 137)
$QUAL	Name of qualifier that is currently in effect (the null string if none)
$CPU	Target machine ('IOP')

Example: Use of predefined micro $DATE

Location	Result	Operand	Comment
1	10	20	35
	DATA	'THE DATE IS "$DATE"'	
	DATA	'THE DATE IS 06/16/81'	

7. CHANNEL INTERFACE FUNCTIONS

Channel interfaces buffer data, generate control signals for peripheral devices, and multiplex several devices into the same I/O Processor (IOP) channel. This section gives the channel interface functions for Cray I/O Subsystem (IOS) Models B and C.

For more detail on any of these channel interfaces, see the following CRI manuals:

HR-0030	I/O Subsystem Model B Hardware Reference Manual
HR-0081	I/O Subsystem Model C Hardware Reference Manual
HR-0077	Disk Systems Hardware Reference Manual

7.1 INTERFACE CHARACTERISTICS

Each IOP provides for I/O channels. These channels are addressed by the *d* designator in the program instruction or by the B register contents. Data can be transferred from the IOP accumulator to a channel interface register or from a channel interface register to the accumulator. You can use the Direct Memory Access (DMA) ports for block transfers of data into or out of Local Memory. Data transfers and channel interface actions are a function of each interface logic control.

Each interface can interpret up to 16 function signals from the IOP program. These functions are generated by instructions 140 through 177. Interpretation of each function is specifically designated by each interface. However, three functions common among the interfaces (except the peripheral expander) are as follows:.

Function	Description
iod : 0 or IOB : 0	Clears the Channel Busy and Done flags and place the channel in an idle status
iod : 6 or IOB : 6	Clears the Channel Interrupt flag for the associated channel, blocking any further interrupt requests from that channel
iod : 7 or IOB : 7	Sets the Channel Interrupt Enable flag for the associated channel and enable the interrupt requests from that channel

Each channel interface provides for a Busy flag, normally set during the active period of the channel and cleared during an idle period. The setting and clearing of this flag depends on the channel interface interpretation of the 16 function codes. The Channel Busy flag can be sensed by the IOP program through execution of instructions 041 and 043.

Each channel interface provides for a Done flag, normally used to signal the IOP program when some step of the channel activity has reached a point requiring program action. Setting and clearing of the flag is normally a function of the interface hardware, but the program can set or clear the flag for special purposes. The program senses the state of this flag through instructions 040 and 042. An interrupt is normally generated by the interface hardware when the Channel Done flag and the Channel Interrupt Enable flag are set. The system must have interrupts enabled to be interrupted. When not enabled, however, it can still sense the interrupt waiting through IOR : 10.

7.2 CHANNEL INTERFACE FUNCTION CODES

Table 7-1 lists all the currently supported peripheral devices and briefly explains each function code interpretation that is implemented. The APML mnemonic identifies the function. Only the first mnemonic of each type is given. The interface functions for disk storage unit channels are not described below; they are described in the Disk Systems Hardware Reference Manual, CRI publication HR-0077.

Table 7-1. Channel Functions and Descriptions

Channel	Function	Description
0 I/O Request	IOR : 10	Read interrupt channel number
1 Program Fetch Request	PFR : 0 PFR : 6 PFR : 7 PFR : 10	Clear Program Fetch Request flag Clear Channel Interrupt Enable flag Set Channel Interrupt Enable flag Read operand register number

Table 7-1. Channel Functions and Descriptions (continued)

Channel	Function	Description
2 Program Exit Stack	PXS : 0	Clear Exit Stack Boundary flag
	PXS : 6	Clear Channel Interrupt Enable flag
	PXS : 7	Set Channel Interrupt Enable flag
	PXS : 10	Read exit stack pointer, E
	PXS : 11	Read exit stack address, (E)
	PXS : 13†	Read history log
	PXS : 14	Enter exit stack pointer, E
	PXS : 15	Enter exit stack address, (E)
	PXS : 16†	Enter diagnostic mode (available in diagnostic mode only)
3 Local Memory Error	LME : 0	Clear Local Memory Parity Error flag
	LME : 6	Clear Channel Interrupt Enable flag
	LME : 7	Set Channel Interrupt Enable flag
	LME :10††	Read error information
4 Real-time Clock	RTC : 0	Clear Channel Done flag
	RTC : 6	Clear Channel Interrupt Enable flag
	RTC : 7	Set Channel Interrupt Enable flag
	RTC : 10	Read real-time clock
5 Buffer Memory	MOS : 0	Clear Channel Busy and Done flags
	MOS : 1	Enter Local Memory address for next transfer
	MOS : 2	Enter upper bits of Buffer Memory address
	MOS : 3	Enter lower bits of Buffer Memory address
	MOS : 4	Read Buffer Memory/enter block length
	MOS : 5	Write Buffer Memory/enter block length
	MOS : 6	Clear the Channel Interrupt Enable flag
	MOS : 7	Set the Channel Enable Interrupt flag
	MOS : 10†	Read bypass modes if accumulator bit 2^1=1; read error bits if accumulator bit 2^0=1.
	MOS : 14	Set control register flags
	MOS : 15†	Set second control register flags
	MOS : 16†	Set bypass modes
6, 10, 12 IOP Input (AIA, AIB, AIC)	AIA : 0	Clear Channel Done flag
	AIA : 6	Clear Channel Interrupt Enable flag
	AIA : 7	Set Channel Interrupt Enable flag
	AIA : 10	Read input to accumulator and resume channel

† Model C only
†† Model B only

Table 7-1. Channel Functions and Descriptions (continued)

Channel	Function	Description
7, 11, 13 IOP Output (AOA, AOB, AOC)	AOA : 0	Clear Channel Busy and Done flags
	AOA : 1	Enter control bits from accumulator
	AOA : 6	Clear Channel Interrupt Enable flag
	AOA : 7	Set Channel Interrupt Enable flag
	AOA : 14	Set Channel Busy flag and output accumulator data
14 Input from Central Memory or Solid State Disk (SSD) (HIA) (100 Mbyte channel)	HIA : 0	Clear Channel Busy and Done flags
	HIA : 1	Enter Local Memory address
	HIA : 2	Enter high-order bits of Central Memory or SSD address; see specific hardware manual for the actual number of bits to enter.
	HIA : 3	Enter low-order 9 bits of Central Memory or SSD address
	HIA : 4	Enter block length; start transfer to Local Memory if Buffer Memory channel not in bypass mode.
	HIA : 6	Clear Channel Interrupt Enable flag
	HIA : 7	Set Channel Interrupt Enable flag
	HIA : 10†	Read syndrome code or error code (available in diagnostic mode only)
	HIA : 14	Enter diagnostic mode (available in diagnostic mode only)
15 Output to Central Memory or SSD (HOA) (100 Mbyte Channel)	HOA : 0	Clear Channel Busy and Done flags
	HOA : 1	Enter Local Memory address
	HOA : 2	Enter high-order bits of Central Memory or or SSD address; see specific hardware manual for the actual number of bits to enter.
	HOA : 3	Enter low-order 9 bits of Central Memory or SSD address
	HOA : 5	Enter block length for transfer; start transfer from Local Memory unless Buffer Memory channel is in bypass mode.
	HOA : 6	Clear Channel Interrupt Enable flag
	HOA : 7	Set Channel Interrupt Enable flag
	HOA : 10†	Read error code (available in diagnostic mode only)
	HOA : 14	Enter diagnostic mode (available in diagnostic mode only)

† Model C only

Table 7-1. Channel Functions and Descriptions (continued)

Channel	Function	Description
50 Mainframe Input (LIA)	LIA : 0†	Clear Channel Busy and Done flags
	LIA : 1†	Enter Local Memory address, start transfer to Local Memory
	LIA : 2†	Enter parcel count for transfer
	LIA : 3†	Clear Channel Parity Error flags
	LIA : 4†	Clear Ready Waiting flag
	LIA : 6†	Clear Channel Interrupt Enable flag
	LIA : 7†	Set Channel Interrupt Enable flag
	LIA : 10†	Read present Local Memory address
	LIA : 11†	Read status (ready waiting, parity error)
51 Mainframe Output (LOA)	LOA : 0†	Clear Channel Busy and Done flags
	LOA : 1†	Enter Local Memory address, start transfer from Local Memory
	LOA : 2†	Enter parcel count for transfer
	LOA : 3†	Clear Error flag
	LOA : 4†	Set/clear external control signals
	LOA : 6†	Clear Channel Interrupt Enable flag
	LOA : 7†	Set Channel Interrupt Enable flag
	LOA : 10†	Read present Local Memory address
	LOA : 11†	Read processor number (0 through 3); read Error flag.
Console Keyboard (TIA - TID) (Accumulator Channel)	TIA : 0	Clear Channel Done flag
	TIA : 3†	Set baud rate, both input and output pair
	TIA : 6	Clear Channel Interrupt Enable flag
	TIA : 7	Set Channel Interrupt Enable flag
	TIA : 10	Read data into accumulator and clear Done flag
Console Display (TOA - TOD) (Accumulator Channel)	TOA : 0	Clear Channel Busy and Done flags
	TOA : 6	Clear Channel Interrupt Enable flag
	TOA : 7	Set Channel Interrupt Enable flag
	TOA : 14	Send accumulator data to display

† Model C only

Table 7-1. Channel Functions and Descriptions (continued)

Channel	Function	Description
Peripheral Expander (EXB) (Accumulator Channel)	EXB : 0	Idle the channel
	EXB : 1	Request A Input register contents (DIA)
	EXB : 2	Request B Input register contents (DIB)
	EXB : 3	Request C Input register contents (DIC)
	EXB : 4	Read Busy/Done flag, interrupt number
	EXB : 5	Load device address
	EXB : 6	Send interface mask (MSKO)
	EXB : 7	Set interrupt mode
	EXB : 10	Read data bus status
	EXB : 11	Read status 1
	EXB : 13	Read status 2
	EXB : 14	Send data to A Output register (DOA)
	EXB : 15	Send data to B Output register (DOB)
	EXB : 16	Send data to C Output register (DOC)
	EXB : 17	Send control
Front-end Input† (CIA - CID) (DMA Channel)	CIA : 0	Clear channel
	CIA : 1	Enter Local Memory address, start input
	CIA : 2	Enter parcel count
	CIA : 3	Clear Channel Parity Error flags
	CIA : 4	Clear Data Waiting flag
	CIA : 6	Clear Interrupt Enable flag
	CIA : 7	Set Interrupt Enable flag
	CIA : 10	Read Local Memory address
	CIA : 11	Read status (ready waiting, parity errors)
Front-end Output† (COA - COD) (DMA Channel)	COA : 0	Clear channel
	COA : 1	Enter Local Memory address
	COA : 2	Enter parcel count
	COA : 3	Clear Error flag
	COA : 4	Set/clear external control signals
	COA : 6	Clear Interrupt Enable flag
	COA : 7	Set Interrupt Enable flag
	COA : 10	Read Local Memory address
	COA : 11	Read status (error) (4-bit channel data)††

† These functions apply only to the MIOP
†† Model B only

Table 7-1. Channel Functions and Descriptions (continued)

Channel	Function	Description
Block Multiplexer Channel (BMA - BMP) (DMA Channel)	BMA : 0	Clear channel control
	BMA : 1	Send reset functions
	BMA : 2	Send commands to control units
	BMA : 3	Read request-in address
	BMA : 4	Clear Channel Done flag; set Channel Busy flag for asynchronous I/O.
	BMA : 5	Delay counter diagnostic
	BMA : 6	Clear Channel Interrupt Enable flag
	BMA : 7	Set Channel Interrupt Enable flag
	BMA : 10	Read Local Memory address
	BMA : 11	Read byte count
	BMA : 12	Read status
	BMA : 13	Read input tags
	BMA : 14	Enter Local Memory address
	BMA : 15	Enter byte count
	BMA : 16	Enter device address
	BMA : 17	Enter output tags
Error Logging Channel for Serial No. 20 and Below† (ERA) (Accumulator Channel)	ERA : 0	Idle channel
	ERA : 6	Clear Interrupt Enable flag
	ERA : 7	Set Interrupt Enable flag
	ERA : 10	Read error status
	ERA : 11	Read error information (first parameter)
	ERA : 12	Read error information (second parameter)
	ERA : 13	Read error information (third parameter)

† Model B only

8. FORMAT OF ASSEMBLER LISTING

The APML assembler generates list output as determined by list pseudo instructions and by options on the APML control statement.

8.1 PAGE HEADERS

Every page of list output produced by the APML assembler contains two 132-character header lines. The first line contains the title, version of APML, time and date of assembly, and a global page number over all programs assembled by the current assembly. The title is taken from a TITLE pseudo instruction if there is one or from the operand field of the IDENT pseudo instruction. The second line contains the subtitle specified by a SUBTITLE pseudo if there is one, a local block name if other than the nominal block, a symbol qualifier if there is one in effect, and a local page number which is reset for each new program unit. The local page number is used in the cross-reference listings generated by APML and SYSREF.

1	66	76	96	105	115
title	*cpu type*	APML *version*	*date*	*time*	Page *nnn*
subtitle	*unused*	Block: *bname*	Qualifier: *qualname*		(*nn*)

8.2 SOURCE STATEMENT LISTING

The listing for source statements comprising an APML program is organized into five columns of information, as follows.

Title line				
Subtitle line				
error code	*location address*	*octal code*	*source line*	*sequence*

error code

The leftmost column contains up to 7 characters indicating errors detected for the current statement. If too many errors occurred to fit in seven columns, the seventh character is a + indicating that not all errors are shown. Appendix C, Messages, describes error codes.

location address

The second column gives the parcel or word address where the current statement is assembled. If the statement is a symbolic APML instruction or PDATA, the address is listed as a parcel address. For word-oriented pseudo instructions, the address is listed as a word address with a W appended.

octal code

The third column of information contains the octal equivalent of the instruction or value.

For symbolic APML instructions, this column contains up to 3 parcels of I/O Processor instructions in octal digits. For 2-parcel instructions, the second parcel is preceded by a / character. If more than 3 parcels of instruction are generated by a statement, the instructions are listed on subsequent lines with a blank source and sequence field.

If the value represents an address, the octal code has a suffix as follows:

+ Positive relocation in program block

- Negative relocation in program block

X External symbol

For a symbol defined through SET, MICSIZE, CHANNEL, or EQUALS, the column contains the octal value of the symbol.

For a BSS or BSSZ instruction, the column contains the octal value of the number of words reserved.

For a MICRO, OCTMIC, or DECMIC instruction, the column contains the number of characters in the micro string.

source line

The fourth column presents columns 1 through 72 of each source line.

sequence The rightmost column either contains the sequence number for the source line as taken from columns 73 through 90 of the source line image or contains an identifier if the line is an expansion of a macro.

8.3 CROSS-REFERENCE LISTING

The assembler generates a cross-reference table with the format as follows. Symbols are listed alphabetically and grouped by qualifier. Each qualified group of symbols is headed by the message SYMBOL QUALIFIER IS *qualname*.

Global symbols which are not referenced are not listed in the cross-reference. Symbols of the form %%xxxxxx, where x is any ASCII character, are not listed in the cross-reference.

```
 ____________________________________________________________________
|        Title line                                                  |
|________Subtitle line___________________|___________________________|
|                |        |              |                           |
|value           |symbol  |name          |symbol references          |
```

value Octal value of *symbol*

symbol A symbol with word-address attribute W appended. A relocatable symbol has a plus (+) suffix if it has positive relocation relative to the program block and a minus (-) suffix if negative relocation relative to the program block. An external symbol has an X suffix. An undefined symbol has a U suffix.

name A global symbol defined by the user is indicated by *GLOBAL*. A global symbol defined in a system text is indicated by the system text dataset name. A symbol defined in global text between TEXT and ENDTEXT pseudo instructions is indicated by the associated text name.

symbol references
This column lists one or more references to the symbol in the following format:

page : *line x*

page Local decimal number of page containing reference. The local page number appears in parentheses at the right end of the second title line, also called the subtitle line.

line Decimal number of line containing reference

x Type of reference, as follows:

blank Symbol value is used at this point.

symbol references (continued)	B	Symbol used as a base register in an APML symbolic jump instruction which required a 2-parcel machine branch instruction
	D	Symbol defined at this reference; that is, it appears in the location field of an instruction or is defined by a SET, EQUALS, or EXT pseudo instruction.
	E	Declares the symbol as an entry name
	F	Symbol used in an expression in a conditioned pseudo instruction such as IFE, IFA, or ERRIF

APPENDIX SECTION

A. CHARACTER SETS

Table A-1 lists the character sets.

Table A-1. Character Sets

CHAR	ASCII	ASCII Card Code	EBCDIC	CDC Display Code
NUL	000	12-0-9-8-1	00	None
SOH	001	12-9-1	01	None
STX	002	12-9-2	02	None
ETX	003	12-9-3	03	None
EOT	004	9-7	37	None
ENQ	005	0-9-8-5	2D	None
ACK	006	0-9-8-6	2E	None
BEL	007	0-9-8-7	2F	None
BS	010	11-9-6	16	None
HT	011	12-9-5	05	None
LF	012	0-9-5	25	None
VT	013	12-9-8-3	0B	None
FF	014	12-9-8-4	0C	None
CR	015	12-9-8-5	0D	None
SO	016	12-9-8-6	0E	None
SI	017	12-9-8-7	0F	None
DLE	020	12-11-9-8-1	10	None
DC1	021	11-9-1	11	None
DC2	022	11-9-2	12	None
DC3	023	11-9-3	13	None
DC4	024	4-8-9	3C	None
NAK	025	9-8-5	3D	None
SYN	026	9-2	32	None
ETB	027	0-9-6	26	None
CAN	030	11-9-8	18	None
EM	031	11-9-8-1	19	None
SUB	032	9-8-7	3F	None
ESC	033	0-9-7	27	None
FS	034	11-9-8-4	1C	None
GS	035	11-9-8-5	1D	None
RS	036	11-9-8-6	1E	None
US	037	11-9-8-7	1F	None
Space	040	None	40	55
!	041	12-8-7	5A	66

Table A-1. Character Sets (continued)

CHAR	ASCII	ASCII Card Code	EBCDIC	CDC Display Code
"	042	8-7	7F	64
#	043	8-3	7B	60
$	044	11-8-3	5B	53
%	045	0-8-4	6C	63
&	046	12	50	67
'	047	8-5	7D	70
(	050	12-8-5	4D	51
)	051	11-8-5	5D	52
*	052	11-8-4	5C	47
+	053	12-8-6	4E	45
,	054	0-8-3	6B	56
-	055	11	60	46
.	056	12-8-3	4B	57
/	057	0-1	61	50
0	060	0	F0	33
1	061	1	F1	34
2	062	2	F2	35
3	063	3	F3	36
4	064	4	F4	37
5	065	5	F5	40
6	066	6	F6	41
7	067	7	F7	42
8	070	8	F8	43
9	071	9	F9	44
:	072	8-2	7A	00
;	073	11-8-6	5E	77
<	074	12-8-4	4C	72
=	075	8-6	7E	54
>	076	0-8-6	6E	73
?	077	0-8-7	6F	71
@	100	8-4	7C	74
A	101	12-1	C1	01
B	102	12-2	C2	02
C	103	12-3	C3	03
D	104	12-4	C4	04
E	105	12-5	C5	05
F	106	12-6	C6	06
G	107	12-7	C7	07
H	110	12-8	C8	10
I	111	12-9	C9	11
J	112	11-1	D1	12
K	113	11-2	D2	13
L	114	11-3	D3	14

Table A-1. Character Sets (continued)

CHAR	ASCII	ASCII Card Code	EBCDIC	CDC Display Code
M	115	11-4	D4	15
N	116	11-5	D5	16
O	117	11-6	D6	17
P	120	11-7	D7	20
Q	121	11-8	D8	21
R	122	11-9	D9	22
S	123	0-2	E2	23
T	124	0-3	E3	24
U	125	0-4	E4	25
V	126	0-5	E5	26
W	127	0-6	E6	27
X	130	0-7	E7	30
Y	131	0-8	E8	31
Z	132	0-9	E9	32
[	133	12-8-2	AD	61
\	134	0-8-2	E0	75
]	135	11-8-2	BD	62
^	136	11-8-7	5F	76
_	137	0-8-5	6D	65
`	140	8-1	79	None
a	141	12-0-1	81	None
b	142	12-0-2	82	None
c	143	12-0-3	83	None
d	144	12-0-4	84	None
e	145	12-0-5	85	None
f	146	12-0-6	86	None
g	147	12-0-7	87	None
h	150	12-0-8	88	None
i	151	12-0-9	89	None
j	152	12-11-1	91	None
k	153	12-11-2	92	None
l	154	12-11-3	93	None
m	155	12-11-4	94	None
n	156	12-11-5	95	None
o	157	12-11-6	96	None
p	160	12-11-7	97	None
q	161	12-11-8	98	None
r	162	12-11-9	99	None
s	163	11-0-2	A2	None
t	164	11-0-3	A3	None
u	165	11-0-4	A4	None
v	166	11-0-5	A5	None

Table A-1. Character Sets (continued)

<table>
<tr><th>CHAR</th><th>ASCII</th><th>ASCII Card Code</th><th>EBCDIC</th><th>CDC Display Code</th></tr>
<tr><td>w</td><td>167</td><td>11-0-6</td><td>A6</td><td>None</td></tr>
<tr><td>x</td><td>170</td><td>11-0-7</td><td>A7</td><td>None</td></tr>
<tr><td>y</td><td>171</td><td>11-0-8</td><td>A8</td><td>None</td></tr>
<tr><td>z</td><td>172</td><td>11-0-9</td><td>A9</td><td>None</td></tr>
<tr><td>{</td><td>173</td><td>12-0</td><td>C0</td><td>None</td></tr>
<tr><td>|</td><td>174</td><td>12-11</td><td>6A</td><td>None</td></tr>
<tr><td>}</td><td>175</td><td>11-0</td><td>D0</td><td>None</td></tr>
<tr><td>~</td><td>176</td><td>11-0-1</td><td>A1</td><td>None</td></tr>
<tr><td>DEL</td><td>177</td><td>12-9-7</td><td>07</td><td>None</td></tr>
</table>

B. HARDWARE INSTRUCTION SUMMARY

This appendix briefly describes APML operand notation and instructions.

B.1 APML OPERAND NOTATION

The following reserved names represent the contents of I/O Processor (IOP) registers or memory:

Name	Description
A	Accumulator
B	Operand register, index register (B register)
(B)	Contents of the operand register addressed by B
C	Carry flag
E	Exit stack pointer
(E)	Exit stack entry addressed by E, the exit stack pointer
I	Interrupt Enable flag
P	Program address register
R	Return jump program address
R!*sym*	Operand register whose index is the value of the symbol *sym*, where *sym* is any symbol with positive absolute value less than 512
dd	Operand register whose index is the value of the symbol *dd*, where *dd* is a 2-character symbol with positive absolute value less than 512
[*dd*]	Value of symbol *dd*; that is, index of register represented by register symbol *dd*.
(*dd*)	Memory parcel addressed by contents of operand register *dd*

Name	Description
k	An unsigned numeric constant, character constant, or a symbol. In general, *k* may have a positive or negative value with absolute value less than 16,384. In some cases, the range of values for *k* is further restricted.
d	An unsigned numeric constant, character constant, or a symbol. In general, *d* may have a positive or negative value with absolute value less than 512. In some cases, the range of values for *d* is further restricted.
(*k*)	Memory parcel addressed by the value of *k*
(*dd* + *k*)	Memory parcel addressed by the sum of the contents of operand register *dd* and constant *k*

NOTE

Instructions referencing the operand register *dd* contain the register index in the *d* field, the lower 9 bits of the instruction parcel.

The following reserved names represent other operands used in symbolic APML instructions:

Name	Description
IOB	I/O channel reference using the contents of the B register as the channel designator
iod	I/O channel reference, where the value of symbol *iod* is the channel designator. Symbol *iod* must be defined by the CHANNEL pseudo instruction. Conventionally *iod* is a 3-character symbol.
BZ, DN	IOP channel status. A channel busy flag, BZ, and done flag, DN, may be tested with certain instructions.
EXIT	Name of subroutine return function, which generates an IOP instruction which exits from a subroutine
WAIT	Name of branch function which loops until a test condition is satisfied
PASS	Name of function which generates an IOP pass, or no-operation instruction

B.2 INSTRUCTIONS

Table B-1 shows IOP and APML instructions and gives an explanation of their functions.

Table B-1. Instruction Summary

IOP	APML	Description
000	PASS	No operation
001	EXIT	Exit from subroutine
002	I = 0	Disable system interrupts
003	I = 1	Enable system interrupts
004	A = A > *d*	Right shift C and A by *d* places, end off
005	A = A < *d*	Left shift C and A by *d* places, end off
006	A = A >> *d*	Right shift C and A by *d* places, circular
007	A = A << *d*	Left shift C and A by *d* places, circular
010	A = *d*	Transmit *d* to A
011	A = A & *d*	Logical product of A and *d* to A
012	A = A + *d*	Add *d* to A
013	A = A - *d*	Subtract *d* from A
014	A = *k*	Transmit *k* to A
015	A = A & *k*	Logical product of A and *k* to A
016	A = A + *k*	Add *k* to A
017	A = A - *k*	Subtract *k* from A
020	A = *dd*	Transmit operand register *d* to A
021	A = A & *dd*	Logical product of A and operand register *d* to A
022	A = A + *dd*	Add operand register *d* to A
023	A = A - *dd*	Subtract operand register *d* from A
024	*dd* = A	Transmit A to operand register *d*
025	*dd* = A + *dd*	Add operand register *d* to A, result to operand register *d*
026	*dd* = *dd* + 1	Transmit *d* to A, add 1, result to operand register *d*
027	*dd* = *dd* - 1	Transmit *d* to A, subtract 1, result to operand register *d*

Table B-1. Instruction Summary (continued)

IOP	APML	Description
030	A = (*dd*)	Transmit contents of memory addressed by register *d* to A
031	A = A & (*dd*)	Logical product of A and contents of memory addressed by register *d*, result to A
032	A = A + (*dd*)	Add contents of memory addressed by register *d* to A, result to A
033	A = A - (*dd*)	Subtract contents of memory addressed by by register *d* to A, result to A
034	(*dd*) = A	Transmit A to memory addressed by register *d*
035	(*dd*) = A + (*dd*)	Add memory addressed by register *d* to A, result to same memory location
036	(*dd*) = (*dd*) + 1	Transmit memory addressed by register *d* to A, add 1, result to same memory location
037	(*dd*) = (*dd*) - 1	Transmit memory addressed by register *d* to A, subtract 1, result to same memory location
040	C = 1, *iod* = DN	Set carry equal to channel *d* done
041	C = 1, *iod* = BZ	Set carry equal to channel *d* busy
042	C = 1, IOB = DN	Set carry equal to channel B done
043	C = 1, IOB = BZ	Set carry equal to channel B busy
044	A = A > B	Right shift C and A by B places, end off
045	A = A < B	Left shift C and A by B places, end off
046	A = A >> B	Right shift C and A by B places, circular
047	A = A << B	Left shift C and A by B places, circular
050	A = B	Transmit B to A
051	A = A & B	Logical product of A and B to A
052	A = A + B	Add B to A
053	A = A - B	Subtract B from A
054	B = A	Transmit A to B
055	B = A + B	Add B to A, result to B
056	B = B + 1	Transmit B to A, add 1, result to B
057	B = B - 1	Transmit B to A, subtract 1, result to B

Table B-1. Instruction Summary (continued)

IOP	APML	Description
060	A = (B)	Transmit operand register B to A
061	A = A & (B)	Logical product of A and operand register B to A
062	A = A + (B)	Add operand register B to A
063	A = A - (B)	Subtract operand register B from A
064	(B) = A	Transmit A to operand register B
065	(B) = A + (B)	Add operand register B to A, result to operand register B
066	(B) = (B) + 1	Transmit operand register to A, add 1, result to operand register B
067	(B) = (B) - 1	Transmit operand register to A, subtract 1, result to operand register B
070	P = P + *d*†	Jump to P + *d*
071	P = P - *d*†	Jump to P - *d*
072	R = P + *d*†	Return jump to P + *d*
073	R = P - *d*†	Return jump to P - *d*
074	P = *dd*	Jump to address in operand register *d*
075	P = *dd* + *k*	Jump to sum of *k* and operand register *d*
076	R = *dd*	Return jump to address in operand register *d*
077	R = *dd* + *k*	Return jump to address sum of *k* and operand register *d*
100	P = P + *d*, C = 0†	Jump to P + *d* if carry = 0
101	P = P + *d*, C # 0†	Jump to P + *d* if carry ≠ 0
102	P = P + *d*, A = 0†	Jump to P + *d* if A = 0
103	P = P + *d*, A # 0†	Jump to P + *d* if A ≠ 0
104	P = P - *d*, C = 0†	Jump to P - *d* if carry = 0
105	P = P - *d*, C # 0†	Jump to P - *d* if carry ≠ 0
106	P = P - *d*, A = 0†	Jump to P - *d* if A = 0
107	P = P - *d*, A # 0†	Jump to P - *d* if A ≠ 0
110	R = P + *d*, C = 0†	Return jump to P + *d* if carry = 0
111	R = P + *d*, C # 0†	Return jump to P + *d* if carry ≠ 0
112	R = P + *d*, A = 0†	Return jump to P + *d* if A = 0
113	R = P + *d*, A # 0†	Return jump to P + *d* if A ≠ 0

† These APML instruction formats are for illustrative purposes; they are not supported by APML even though the hardware instructions are generated by APML.

Table B-1. Instruction Summary (continued)

IOP	APML	Description
114	R = P - *d*, C = 0†	Return jump to P - *d* if carry = 0
115	R = P - *d*, C # 0†	Return jump to P - *d* if carry ≠ 0
116	R = P - *d*, A = 0†	Return jump to P - *d* if A = 0
117	R = P - *d*, A # 0†	Return jump to P - *d* if A ≠ 0
120	P = *dd*, C = 0	Jump to address in operand register *d* if carry = 0
121	P = *dd*, C # 0	Jump to address in operand register *d* if carry ≠ 0
122	P = *dd*, A = 0	Jump to address in operand register *d* if A = 0
123	P = *dd*, A # 0	Jump to address in operand register *d* if A ≠ 0
124	P = *dd* + *k*, C = 0	Jump to address in operand register *d* + *k* if carry = 0
125	P = *dd* + *k*, C # 0	Jump to address in operand register *d* + *k* if carry ≠ 0
126	P = *dd* + *k*, A = 0	Jump to address in operand register *d* + *k* if A = 0
127	P = *dd* + *k*, A # 0	Jump to address in operand register *d* + *k* if A ≠ 0
130	R = *dd*, C = 0	Return jump to address in operand register *d* if carry = 0
131	R = *dd*, C # 0	Return jump to address in operand register *d* if carry ≠ 0
132	R = *dd*, A = 0	Return jump to address in operand register *d* if A = 0
133	R = *dd*, A # 0	Return jump to address in operand register *d* if A ≠ 0
134	R = *dd* + *k*, C = 0	Return jump to address in operand register *d* + *k* if carry = 0
135	R = *dd* + *k*, C # 0	Return jump to address in operand register *d* + *k* if carry ≠ 0
136	R = *dd* + *k*, A = 0	Return jump to address in operand register *d* + *k* if A = 0
137	R = *dd* + *k*, A # 0	Return jump to address in operand register *d* + *k* if A ≠ 0

† These APML instruction formats are for illustrative purposes; they are not supported by APML even though the hardware instructions are generated by APML.

C. MESSAGES

APML supports four classes of messages: abort, fatal, warning, and informative. Under COS, all messages are written to the logfile. Under UNICOS, abort, fatal, and warning messages are written to stderr; APML generates informative messages only if you request them with the -L parameter.

A description of each class follows:

Mesage Class	Description
Abort	APML aborts
Fatal	For UNICOS, APML aborts. For COS, the effect of ABORT and DEBUG options is as follows:
Warning	Possible error detected, no action taken
Informative	Informative message

Effect of ABORT and DEBUG options for Fatal messages under COS:

ABORT Option	DEBUG Option	Result
Off	Off	Permanent Dataset Table (PDT) fatal error flag set
Off	On	PDT fatal error flag clear
On	Off	APML aborts
On	On	APML aborts

This section lists messages issued by APML according to numeric sequence by the message identifier number.

AP000 - [APML] INTERNAL 'APML' ERROR DETECTED AT P = *paddress*

CLASS: Under COS, Abort; under UNICOS, Informative.

CAUSE: APML detects an internal error at parcel address *paddress* and is unable to proceed.

ACTION: Refer the problem to a Cray Research analyst.

AP001 - [APML] APML VERSION *x.xx* (*mm/dd/yy*) - IOP

CLASS: Informative

CAUSE: At the beginning of each assembly, APML issues an informative message indicating the version number *x.xx*, the date *mm/dd/yy* in which APML was assembled, and the type of machine that will execute APML source code, IOP.

ACTION: Not applicable

AP002 - [APML] ASSEMBLY TIME: *nnnnn.nnnn* CPU SECONDS

CLASS: Informative

CAUSE: All programs in the current file of the source dataset are assembled. *nnnnn.nnnn* is the assembly time in floating-point CPU seconds.

ACTION: Not applicable

AP003 - [APML] MEMORY WORDS: *mwords* + I/O BUFFERS: *iobuffers*

CLASS: Informative

CAUSE: All programs in the current file of the source dataset are assembled. *mwords* is the decimal number of memory words required in the user portion of the job field. *iobuffers* is the decimal number of words needed for the I/O table and buffer area of this job field.

ACTION: Not applicable

AP004 - [APML] ASSEMBLY ERRORS

CLASS: Abort

CAUSE: If you set the ABORT flag on the APML control statement and fatal errors are encountered during assembly, APML issues this message followed by an abort.

ACTION: Either remove the ABORT flag from the APML control statement or correct all fatal errors found by APML.

AP010 - [APML] 1 WARNING ERROR, PROGRAM MODULE *pname*
or
AP010 - [APML] *n* WARNING ERRORS, PROGRAM MODULE *pname*

CLASS: Warning

CAUSE: APML issues this message for all source lines in which warning errors are detected, from the previous program module (if any) through program module *pname*. *pname* is equivalent to the name used on a particular IDENT pseudo statement.

ACTION: Correct all warning errors. See appendix D, Assembly Errors, for a list of warning errors.

AP011 - [APML] 1 FATAL ERROR, PROGRAM MODULE *pname*
or
AP011 - [APML] *n* FATAL ERRORS, PROGRAM MODULE *pname*

CLASS: Fatal

CAUSE: APML issues this message for all source lines in which fatal errors are detected, from the previous program module (if any) through program module *pname*. *pname* will be equivalent to the name used on a particular IDENT pseudo statement.

ACTION: Correct all fatal errors. See appendix D, Assembly Errors, for a list of fatal errors.

AP012 = [APML] MISSING IDENT STATEMENT

CLASS: Warning

CAUSE: An END pseudo on the source dataset occurred before an IDENT pseudo instruction.

ACTION: Check the source dataset for matching IDENT and END pseudo instructions.

AP013 - [APML] MISSING END STATEMENT, PROGRAM MODULE *pname*

CLASS: Warning

CAUSE: On the source dataset, an end-of-file (EOF) occurred before an END pseudo instruction corresponding to the IDENT pseudo in program module *pname*. *pname* is equivalent to the name used on that IDENT pseudo statement.

ACTION: Check the source dataset for matching IDENT and END pseudo instructions.

AP014 - [APML] EMPTY SOURCE FILE, DN = *dname*

CLASS: Warning

CAUSE: An EOF or end-of-data (EOD) was encountered on the source dataset before any source statements.

ACTION: Check the job control statements and the source dataset for a problem that causes a null file.

AP015 - [APML] 1 LINE EXCEEDS 90 CHARACTERS, DN = *dname*
or
AP015 - [APML] *n* LINES EXCEED 90 CHARACTERS, DN = *dname*

CLASS: Warning

CAUSE: The given number of records in the named dataset contain more than 90 characters. The most typical cause is UPDATE sequence numbers that extend past column 90. (APML truncates the long records to 90 characters). This message is also issued when a binary dataset is erroneously read.

ACTION: If the records exceed 90 characters, break up the long records with continuation lines.

AP016 - [APML] OPEN ERROR, DN = *dname*

CLASS: Abort

CAUSE: The dataset *dname* was not found in your local environment or in the system directory.

ACTION: Access or create the dataset *dname*.

AP017 - [APML] INVALID CPU TYPE SPECIFIED: *cpu*

CLASS: Warning

CAUSE: The CPU=*type* parameter on the APML control statement is invalid (was specified as something other than IOP).

ACTION: Correct the CPU type on the APML job control statement.

AP030 - [APML] BAD BINARY TEXT, DN = *dname*, (ERROR CODE = *cc*)

CLASS: Fatal

CAUSE: An error was discovered in the binary system text *dname*. The error codes and their meanings are as follows:

Error Code	Meaning
P1	Prologue field BSTTT ≠1
P2	Prologue field BSTWC less than LE@BSTPR
P3	End-of-record (EOR) encountered while prologue was being read
P4	EOF, EOD, or null record encountered while prologue was being read
H1	EOF, EOD, or null record encountered while subtable header was being read
H2	Header field BSTTT ≠1
H3	Header field BSTWC <1
H4	Header field BSTID not recognized
M1	EOR encountered while TMDF was being read
M2	EOF, EOD, or null record encountered while TMDF was being read
M3	Length of TMDF entry <0
M4	Length of TMDF entry =0
M5	Global word count exceeded during TMDF processing

CAUSE: (continued)	Error Code	Meaning
	S1	EOR encountered while TSYM entry was being read
	S2	EOR, EOD, or null record encountered while TSYM entry was being read
	S3	Global word count exceeded during TSYM processing
	E1	Epilogue field BSTWC ≠1
	E2	Global word count not equal to sum of subtable word counts

ACTION: Generate a new binary system text from the original source system text and rerun the job with the new binary system text, rerun the job with the source system text in place of the binary system text, or show listing and DSDUMP output of offending binary system text to a Cray Research systems analyst.

AP031 - [APML] *symbol* DOUBLY-DEFINED IN BINARY TEXT *dname*

CLASS: Fatal

CAUSE: The named *symbol* is defined in the named binary system text but is defined differently in a previous system text.

ACTION: Remove one of the offending definitions from the source system texts, generate a new binary system text, and resubmit job.

AP032 - [APML] MACRO *opsyn* NOT FOUND, BINARY TEXT *dname*

CLASS: Fatal

CAUSE: The named binary system text contains an OPSYN directive of the form name OPSYN *opsyn*, but no macro or pseudo-op with the name *opsyn* is known to the assembler.

ACTION: Correct the spelling of opsyn, remove the OPSYN from the named system text, or define the offending macro in a previous system text or before the OPSYN directive in the named system text.

AP033 - [APML] MACRO *mname* REDEFINED IN BINARY TEXT *dname*

CLASS: Warning

CAUSE: A definition for the named macro appears in the named dataset, but the macro is previously defined.

ACTION: If the redefinition is intentional, the new definition will be used; otherwise, remove the unwanted macro definition.

CA999 - NAME *name* TOO LONG

CLASS: Fatal

CAUSE: One of your file names is longer than 7 characters.

ACTION: Use a shorter name for that file.

D. ASSEMBLY ERRORS

Two types of errors, fatal errors and warning errors, can occur during an assembly. Fatal errors cause APML to abort the job unless a DEBUG parameter is present on the APML control statement. See table D-1 for an explanation of fatal error types. Warning errors have no effect on the assembly process. Table D-2 defines warning errors. An error code consists of a single alpha character, or an alpha character and a digit.

Table D-1. Fatal Errors

Error Type	Definition
C	NAME, SYMBOL, CONSTANT, OR DATA ITEM ERROR Indicates a variety of possible errors. For example: • Illegal character, too many characters, or illegal separator in a name, symbol, constant, or data item • Count field in character constant exceeds 800 • Missing right apostrophe in a character string • Parentheses in an embedded parameter not matched properly • Embedded argument not followed by blank or comma
D	DOUBLE DEFINED SYMBOL OR DUPLICATE PARAMETER NAME • Symbol previously defined; the first definition holds. No error is given if the second definition results in the same value and attributes. • A formal parameter in a definition has the same name as a previously defined parameter. The parameter is ignored.
E	DEFINITION OR CONDITIONAL SEQUENCE ILLEGALLY NESTED
F	TOO MANY ENTRIES • Number of block exceeds 1024 • Number of external names exceeds 4095 • Number of entry names exceeds 5461 • Location or origin counter word address exceeds 4,194,303

Table D-1. Fatal Errors (continued)

Error Type	Definition
I	INSTRUCTION PLACEMENT ERROR The instruction is treated as a null (blank) pseudo instruction. • ABS not allowed after a symbolic machine instruction or restricted pseudo instruction • IDENT not allowed after IDENT without an intervening END • Symbolic APML instruction, or restricted pseudo instruction, appears outside an IDENT, END sequence • END pseudo instruction within a macro expansion
L	LOCATION FIELD ERROR Indicates an invalid name in the location field of a pseudo instruction, macro call, or prototype statement
N	RELOCATABLE FIELD ERROR Indicates an error in a relocatable field. For example, more than one main program entry is named in a program module.
On	OPERAND FIELD ERROR Indicates an error in the operand field of a pseudo instruction Errors O1 through O9 refer to operand or operator errors in a symbolic APML statement. O1 Illegal operand following shift operator O2 Channel function separator must be a colon (:). O3 Channel function must be a constant. O4 One of the following: • Relational operator must follow the subject of a conditional clause • Operand not allowed as subject of conditional clause • An = or # must follow IOB of channel mnemonic in a test for channel busy or done. An = or # must follow C in a test of the carry flag. O5 Unused

Table D-1. Fatal Errors (continued)

Error Type	Definition
	06 Illegal operand follows the subject in a conditional clause. 07 Illegal operator or separator following an operand 08 More than 18 operands appear in an APML statement. 09 One of the following: • 0 or 1 must follow C = or I = • BZ or DN must follow = or # in a conditional clause involving IOB or a channel mnemonic
P	PROGRAMMER ERROR Error generated by ERROR or ERRIF pseudo instruction
R	RESULT FIELD ERROR Indicates a syntax error in result field of a symbolic APML instruction
S*n*	SYNTAX ERROR Indicates a syntax error in an undefined pseudo instruction Errors S1 through S9 indicate syntax errors in symbolic APML instructions S1 Unrecognized operand S2 One of the following: • Illegal operator or operand following (*dd* or (B or (E • Illegal operand following (S3 Unused S4 Missing] following [symbol S5 Operator must be = following subject S6 Illegal subject of assignment clause S7 One of the following: • Illegal operand following P = or R = • Illegal operand in assignment clause S8 Illegal operator when + or - or & or shift operator is required S9 Illegal operand following + or - or &

Table D-1. Fatal Errors (continued)

Error Type	Definition
T	TYPE ERROR Word address, parcel address, or value type not as required for an expression or constant
U	UNDEFINED SYMBOL OR OPERATION Reference to a symbol that is not defined
V	REGISTER EXPRESSION OR FIELD WIDTH ERROR Indicates inconsistency between an expression attribute and field width defined. For example: • Relocatable attribute not allowed for field width • External attribute not allowed for field width • Word-address or parcel-address attribute not allowed for field width • Field width symbol or constant (in VWD) not terminated by slash (/)
X	EXPRESSION ERROR Expression contains illegal attribute, separator value, and so on, for application. For example: • Expression element not terminated by space, comma, or expression operator • Complement (#) of external or relocatable element not allowed • Negative expression value in BSS, BSSZ, ORG, or LOC pseudo instruction • Expression in ORG not relative to current block • Expression is relocatable or external when relocatable or external attribute is not allowed • More than one element in a term is external or relocatable, or external element is not the only element in a term • More than one external element in an expression, or minus sign precedes an external element

Table D-1. Fatal Errors (continued)

Error Type	Definition
	• Expression is relocatable relative to more than one block after cancellation of relocatable terms with opposite signs • Expression is negative relocatable • Expression is both external and relocatable

Table D-2. Warning Errors

Error Type	Definition
W	PROGRAMMER WARNING ERROR Error may be generated by ERROR or ERRIF pseudo instruction
W1	LOCATION FIELD SYMBOL IGNORED Location symbol not used in a pseudo instruction and is ignored
W2	BAD LOCATION SYMBOL Illegal character or too many characters
W3	EXPRESSION ELEMENT TYPE ERROR Value, parcel-address, or word-address attribute not allowed for an element in an expression
W4	POSSIBLE SYMBOLIC APML INSTRUCTION ERROR

Table D-2. Warning Errors (continued)

Error Type	Definition
W5	TRUNCATION ERROR • Expression value exceeds field size, result truncated • Division by 0 (zero result) • External expression in zero width field
W6	LOCATION FIELD SYMBOL NOT DEFINED • Illegal character or too many characters • The expression defining the symbol contains an undefined symbol • The micro name on a MICSIZE instruction is not previously defined
W7	MICRO SUBSTITUTION ERROR A quote mark encountered in APML source was not followed by a previously defined micro name or was not terminated by a second quote mark.
W8	ADDRESS COUNTER BOUNDARY ERROR • * (or *0) used in an expression when the location (or origin) counter is not a parcel boundary. • W.* (or W. *0) used in an expression when the location (or origin) counter is not a word boundary.
W9	BASE REGISTER DECLARATION REQUIRED This error appears, if a base register is not currently declared, on any branch instruction whose destination is outside the current page.
Y2	MACRO REDEFINED A macro name encountered in the APML source was redefined.†

† Warning error depends on the WMR and NWMR features of the APML control statement or the LIST pseudo instruction.

INDEX

Aborting the APML COS job, 3-2
ABS pseudo instruction, 6-3
Absolute
    assembly element and term attribute evaluation, 2-20
    attribute for a symbol, 2-6
    expression attribute, 2-18
    expressions, examples, 2-19
Accumulator
    entering a value, 5-5
    reserved name, B-1
    shift, 5-17
Add to accumulator and replace operand instructions, 5-2, 5-14, 5-15
Add to accumulator instructions, 5-2, 5-8 through 5-10
Adding operators, 2-14
AIA functions, 7-3
AOA functions, 7-4
APML assembler language
    coding conventions, 2-3
    cross-reference listing, 8-3
    data notation
        character constants, 2-10
        data items, 2-11
        numeric constants, 2-9
        overview, 2-9
    examples, 3-7
    execution, 1-2
    expression attributes
        overview, 2-17
        parcel address, word address, or value, 2-18
        relocatable, external, or absolute, 2-18
    expression evaluation, 2-17
    expressions
        adding operators, 2-14
        elements, 2-15
        multiplying operators, 2-14
        overview, 2-14
        term attributes, 2-15
        terms, 2-15
    global definitions, 2-8
    instruction summary, B-3 through B-7
    JCL example
        COS, 3-4
        UNICOS, 3-6
    line editing
        concatenation, 2-4
        micro substitution, 2-4
        overview, 2-3

APML assembler language (continued)
    list output, 8-1
    names, 2-4
    operand notation, B-1
    page headers, 8-1
    prefixed symbols and constants
        overview, 2-13
        parcel address prefix - P., 2-13
        word address prefix - W., 2-13
    qualified symbols, 2-7
    source line format, 2-1
    source statement listing, 8-1
    special elements, 2-8
    statement format
        comment statement, 2-1
        overview, 2-1
        pseudo instruction format, 2-2
        symbolic APML instruction format, 2-2
    symbol reference, 2-7
    symbols
        overview, 2-5
        symbol attributes, 2-5
        symbol definition, 2-5
    table method of evaluation attribute evaluation, 2-20
$APTEXT, default system text file, COS, 3-3
ASCII
    character set, A-1
    representation of characters, 2-10
Assembler listing format, 8-1
Assembly errors, D-1
Assignment clauses
    channel function, 4-10
    jump assignment, 4-9
    overview, 4-5
    replacement assignment, 4-5
    set flag assignment, 4-9
    special function, 4-10
Assignment field, description, 2-2
Assignment syntax, 4-14
Asterisk
    as a special element, 2-8
    introducing comment, 2-1
Attributes, symbol, 2-5

BASE pseudo instruction, 6-10
Base register, 6-5
BASEREG pseudo instruction, 6-5
Basic IOP hardware instructions, section 5
Binary object file, 3-5
Binary symbol table, 3-4

Binary system text
    COS, 3-3
    overview, 3-6
BITP pseudo instruction, 6-20
BITW pseudo instruction, 6-19
$BLD, default COS binary output dataset
  from APML, 3-1
Block control pseudo instructions
    BITW - set *W counter, 6-19
    BITP - set *P counter, 6-20
    BLOCK - local block assignment, 6-16
    BSS - block save, 6-17
    LOC - set * counter, 6-18
    ORG - set *O counter, 6-17
    overview, 6-14
Block Multiplexer (BMA) functions, 7-7
BLOCK pseudo instruction, 6-16
Blocks, definition, 6-14
BMA functions, 7-7
Branch instructions, 5-20
BSS pseudo instruction, 6-17
$BST, default binary system text, COS, 3-3
BSSZ pseudo instruction, 6-34
Buffer memory (MOS) functions, 7-3
Busy flag
    overview, 7-2
    test, 4-13
BZ, reserved name, B-2

Carry flag, reserved name, B-1
CDC Display code character set, A-1
Central memory input (HIA) functions, 7-4
Central memory output (HOA) functions, 7-4
Channel
    functions and descriptions (table), 7-2
    interface function codes, 7-2
Channel Busy flag, 7-2
Channel Done flag, 7-2
Channel function
    assignment clause, 4-10
    instructions, 5-21
Channel Interrupt Enable flag, 7-2
CHANNEL pseudo instruction, 6-31
Character
    constants, data notation, 2-10
    string, justification, 2-11
Character sets, A-1
CIA functions, 7-6
Circular shifts, 5-18
Clauses, see Assignment clauses or
  Condition clauses
$CNC micro, 6-62
COA functions, 7-6
Code control pseudo instructions
    BASEREG - declare base operand
      register, 6-5
    NEWPAGE - force a new instruction page,
      6-7
    overview, 6-5
    SCRATCH - declare APML scratch
      register, 6-6
Code duplication pseudo instructions
    DUP - duplicate code, 6-54
    ECHO - duplicate code with varying
      arguments, 6-55
    ENDDUP - end duplicated code, 6-56
    Examples of duplicated sequences, 6-57
    overview, 6-54
    STOPDUP - stop duplication, 6-57
Coding conventions, 2-3
Command line, see Invocation and execution
Comment field, description, 2-2
COMMENT pseudo instruction, 6-4
Comment statement format, 2-1
Comments, designating, 6-45
Compare
    accumulator, 4-11
    register or memory parcel, 4-12
CON pseudo instruction, 6-33
Concatenation, 2-4
Condition clauses
    overview, 4-11
    test accumulator, 4-11
    test carry flag, 4-13
    test channel status, 4-13
    test register or memory, 4-12
Condition syntax, 4-16
Conditional assembly pseudo instructions
    ELSE - toggle assembly condition, 6-43
    ENDIF - end conditional code sequence,
      6-43
    IFA - test expression attribute for
      assembly condition, 6-38
    IFC - test character strings for
      assembly condition, 6-41
    IFE - test expressions for assembly
      condition, 6-39
    overview, 6-37
    SKIP - unconditionally skip statements,
      6-42
Conditional
    branch instructions, 5-20
    operator, 4-4
Console
    display (TOA - TOD) functions, 7-5
    keyboard (TIA - TID) functions, 7-5
Control instructions
    EXIT, 5-1, 5-4
    I = 0, 5-1, 5-4
    I = 1, 5-1, 5-4
    overview, 5-3
    PASS, 5-1, 5-3
Control statement, see Invocation and
      execution
Conventions used in this manual, 1-2
COS
    APML control statement, 3-1
    APML JCL example, 3-4
    handling of messages, C-1
Cray I/O Subsystem (IOS) Model B, 7-1
Cray I/O Subsystem (IOS) Model C, 7-1
Cross-reference
    information, 3-2
    listing, 8-3

Data definition pseudo instructions
    BSSZ - generate zeroed block, 6-34
    CON - generate constant, 6-33
    DATA - generate data words, 6-35
    overview, 6-33
    PDATA - generate data parcels, 6-36
    VWD - variable word definition, 6-36
Data generation statement instruction format, 4-18
Data items, data notation, 2-11
Data notation
    character constants, 2-10
    data items, 2-11
    numeric constants, 2-9
    overview, 2-9
DATA pseudo instruction, 6-35
$DATE micro, 6-62
DEBUG parameter, and errors, D-1
Debugging information, 3-2
Decimal representation of numbers, 2-10
DECMIC pseudo instruction, 6-61
Decrement by 1 instructions, 5-2, 5-13, 5-14
Direct Memory Access (DMA) ports, 7-1
Display code representation of characters, 2-10
DMA ports, 7-1
DN, reserved name, B-2
Done flag
    overview, 7-2
    test, 4-13
DUP pseudo instruction, 6-54

EBCDIC
    character set, A-1
    representation of characters, 2-10
ECHO pseudo instruction, 6-55
EJECT pseudo instruction, 6-26
Elements, 2-15
ELSE pseudo instruction, 6-43
End off shifts, 5-17
END pseudo instruction, 6-3
ENDDUP pseudo instruction, 6-56
ENDIF pseudo instruction, 6-43
ENDM pseudo instruction, 6-50
ENDTEXT pseudo instruction, 6-28
ENTRY pseudo instruction, 6-8
EQUALS pseudo instruction, 6-29
ERA functions, 7-7
ERRIF pseudo instruction, 6-21
Error code, contents of, D-1
Error control pseudo instructions
    ERRIF - conditional error generation, 6-21
    ERROR - unconditional error generation, 6-21
    overview, 6-21
Error Logging (ERA) functions, 7-7
Error message classes and explainations, C-1
ERROR pseudo instruction, 6-21
Errors, types and effects of, D-1
Evaluation
    absolute assembly element and term attribute, 2-20

Evaluation (continued)
    of terms, 2-16
    relocatable assembly element and term attribute, 2-20
Example JCL for APML, COS, 3-4
Examples, 3-7
EXB functions, 7-6
Execution of the APML assembler, 1-2
Execution, see Invocation and execution
EXIT
    instruction, 4-10
    instruction summary, B-3
    reserved name, B-2
Exit Stack
    Boundary flag, 5-4
    pointer, reserved name, B-1
    stack, 6-32
EXIT, 5-1, 5-4
Expression attributes
    overview, 2-17
    parcel address, word address, or value, 2-18
    relocatable, external, or absolute, 2-18
Expression evaluation, 2-17
Expressions
    adding operators, 2-14
    elements, 2-15
    examples, 2-19
    multiplying operators, 2-14
    overview, 2-14
    terms, 2-15
    term attributes, 2-15
EXT pseudo instruction, 6-9
External attribute for a symbol, 2-6
External expression
    attribute, 2-18
    examples, 2-19

Fatal errors
    effects on assembly, D-1
    table, D-1
Features of APML, 1-1
Fields of a pseudo instruction, 2-2
First pass, overview, 1-2
Format of assembler listing, 8-1
Front-end
    input (CIA) functions, 7-6
    output (COA - COD) functions, 7-6
Function
    codes, 7-2
    operators, 4-3

Global definitions, 2-8
GLOBAL pseudo instruction, 6-4
Global text source, 6-27

Hardware instruction summary, B-1
Hexadecimal representation of numbers, 2-10
HIA functions, 7-4
HOA functions, 7-4

I = 0, 5-1, 5-4, B-3
I = 1, 5-1, 5-4, B-3
I/O channels, 7-1
I/O Request function, 7-2
IDENT pseudo instruction, 6-2
IFA pseudo instruction, 6-38
IFC pseudo instruction, 6-41
IFE pseudo instruction, 6-39
$IN, default COS input dataset to APML, 3-1
Increment by 1 instructions, 5-2, 5-12, 5-13
Instruction definition pseudo instructions
    ENDM - end macro definition, 6-50
    LOCAL - specify local symbols, 6-49
    Macro definition format, 6-46
    MACRO - macro definition, 6-48
    OPSYN - synonymous operation, 6-53
    overview, 6-45
Instruction
    index (table), 5-1
    summaries, B-3 through B-7
Instructions, see Pseudo instructions or
  Basic IOP hardware instruction set
Integer data item format, 2-11
Interface
    characteristics, 7-1
    function codes, 7-2
Interrupt Enable flag, reserved name, B-1
Invocation and execution
    binary system text, 3-6
    COS APML control statement, 3-1
    overview, 3-1
    system text, 3-6
    UNICOS APML command line, 3-4
IOB : 0 - 17, instruction summary, B-7
IOB, as reserved name, B-2
iod : 0 - 17, instruction summary, B-7
IOP
    Input (AIA, AIB, AIC) functions, 7-3
    instruction summary, B-3 through B-7
    Output (AOA, AOB, AOA) functions, 7-4
IOR function, 7-2

JCL example
    UNICOS, 3-6
    COS, 3-4
$JDATE micro, 6-62
Jump
    assignment, assignment clause, 4-9
    destination address, possibilities, 4-9
Justification of character string, 2-11

-L parameter and error messages, C-1
Label, definition, 4-5
Length of field for a data item, 2-12
Length of source line, 2-1
LIA functions, 7-5
Line editing
    concatenation, 2-4
    micro substitution, 2-4
    overview, 2-3
List output, APML, 8-1
List pseudo processing, 3-5
LIST pseudo instruction, 6-23
Listing control options, 3-2
Listing control pseudo instructions
    EJECT - begin new page, 6-26
    ENDTEXT - terminate global text, 6-28
    LIST - list control, 6-23
    overview, 6-23
    SPACE - list blank lines, 6-26
    SUBTITLE - specify listing subtitle,
      6-27
    TEXT - begin global text, 6-27
    TITLE - specify listing title, 6-27
LME functions, 7-3
LOA functions, 7-5
Loader linkage pseudo instructions
    ENTRY - specify entry symbols, 6-8
    EXT - specify external symbols, 6-9
    overview, 6-8
    START - specify program entry, 6-9
LOC pseudo instruction, 6-18
Local
    block, 6-14
    Memory Error (LME) functions, 7-3
    memory transfers, 7-1
LOCAL pseudo instruction, 6-49
Location counter
    overview, 6-15
    setting, 6-17, 6-18
Location field, description, 2-2
Logical product with accumulator
  instructions, 5-1, 5-7, 5-8

Macro definition format, 6-46
MACRO pseudo instruction, 6-48
Main program entry, 6-9
Mainframe
    input (LIA) functions, 7-5
    output (LOA) functions, 7-5
Memory parcel, compare, 4-12
Messages, classes and explanation, C-1
$MIC micro, 6-62
Micro definition pseudo instructions
    DECMIC - decimal micro, 6-61
    MICRO - micro definition, 6-60
    Micro reference format, 6-59
    OCTMIC - octal micro, 6-61
    overview, 6-59
    predefined micros, 6-62
Micro
    names, delimiting, 2-4
    reference format, 6-59
    substitution, 2-4
MICRO pseudo instruction, 6-60
MICSIZE pseudo instruction, 6-32
Mode control pseudo instructions
    BASE - declare base for numeric data,
      6-10
    overview, 6-10
    QUAL - qualify symbols, 6-12
Mode of branch, 5-20
MOS functions, 7-3
Multiplying operators, 2-14

Names
    registers, 4-1
    reserved, B-1, B-2
Nesting, 6-46, 6-55, 6-58
NEWPAGE pseudo instruction, 6-7
Nominal block, definition, 6-14
Numeric constants, data notation, 2-9

Octal representation of numbers, 2-10
OCTMIC pseudo instruction, 6-61
Operand
    field, description, 2-2
    notation, 4-1, B-1
    register B, reserved name, B-1
Operators
    adding, 2-14
    conditional operator, 4-4
    function operators, 4-3
    multiplying, 2-14
    overview, 4-3
    relational operators, 4-4
    replacement operator, 4-3
OPSYN pseudo instruction, 6-53
ORG pseudo instruction, 6-17
Origin counter
    overview, 6-15
    setting, 6-17
$OUT, default COS list output dataset from APML, 3-1
Output, APML, 8-1

P. parcel address prefix, 2-13
Page headers, 8-1
Parcel address
    attribute for a symbol, 2-5
    expression attribute, 2-18
    prefix - P., 2-13
Parcel-bit-position counter, 6-15
PASS
    instruction, 4-10, 5-3, 5-1
    instruction summary, B-3
    reserved name, B-2
Pass, first and second, overview, 1-2
PDATA pseudo instruction, 6-36
PDT, 3-2
Peripheral Expander (EXB) functions, 7-6
PFR functions, 7-2
Predefined micros, 6-62
Prefixed symbols and constants
    parcel address prefix - P., 2-13
    word address prefix - W., 2-13
Program address register, reserved name, B-1
Program control pseudo instructions
    ABS - assemble absolute binary, 6-3
    COMMENT - define Program Descriptor Table comment, 6-4
    END - end program module, 6-3
    GLOBAL - declare global symbols, 6-4
    IDENT - identify program module, 6-2
    overview, 6-2
Program Description Table (PDT), 3-2
Program Descriptor Table (PDT), 6-2
Program Exit Stack (PXS) functions, 7-3
Program Fetch Request (PFR) functions, 7-2
Program Fetch Request flag, 5-21
Program sequence, termination, 5-4
Program statement instruction format
    assignment clauses
        channel function, 4-10
        jump assignment, 4-9
        overview, 4-5
        replacement assignment, 4-5
        set flag assignment, 4-9
        special function, 4-10
    condition clauses
        overview, 4-11
        test accumulator, 4-11
        test carry flag, 4-13
        test channel status, 4-13
        test register or memory, 4-12
    overview, 4-4
    syntax graphs for APML program statements, 4-13
Pseudo instructions,
    ABS, 6-3
    BASE, 6-10
    BASEREG, 6-5
    BITP, 6-20
    BITW, 6-19
    BLOCK, 6-16
    BSS, 6-17
    BSSZ, 6-34
    CHANNEL, 6-31
    COMMENT, 6-4
    CON, 6-33
    DATA, 6-35
    DECMIC, 6-61
    DUP, 6-54
    ECHO, 6-55
    EJECT, 6-26
    ELSE, 6-43
    END, 6-3
    ENDDUP, 6-56
    ENDIF, 6-43
    ENDM, 6-50
    ENDTEXT, 6-28
    ENTRY, 6-8
    EQUALS, 6-29
    ERRIF, 6-21
    ERROR, 6-21
    EXT, 6-9
    fields, 2-2
    format, 2-2
    GLOBAL, 6-4
    IDENT, 6-2
    IFA, 6-38
    IFC, 6-41
    IFE, 6-39
    LIST, 6-23
    LOC, 6-18
    LOCAL, 6-49
    MACRO, 6-48
    MICRO, 6-60
    MICSIZE, 6-32
    NEWPAGE, 6-7
    OCTMIC, 6-61

Pseudo instructions (continued)
OPSYN, 6-53
ORG, 6-17
PDATA, 6-36
QUAL, 6-12
rules, 6-1
SCRATCH, 6-6
SET, 6-30
SKIP, 6-42
SPACE, 6-26
START, 6-9
STOPDUP, 6-57
SUBTITLE, 6-27
TEXT, 6-27
TITLE, 6-27
types, 6-2
VWD, 6-36
PXS functions, 7-3

$QUAL micro, 6-62
QUAL pseudo instruction, 6-12
Qualified symbols
overview, 2-7
referencing, 2-7
Quotation marks, delimiting micro names, 2-4

Real-time Clock (RTC) functions, 7-3
Redefinable attribute for a symbol, 2-6
Register names, 4-1
Register, compare, 4-12
Relational operators, 4-4
Relocatable assembly element and term attribute evaluation, 2-20
Relocatable attribute for a symbol, 2-6
Relocatable expression
attribute, 2-18
examples, 2-19
Replacement
assignment, assignment clause, 4-5
operator, 4-3
Reserved names, B-1, B-2
Result field, description, 2-2
Return jump program address, reserved name, B-1
RTC functions, 7-3
Rules
for terms, 2-15
operands in assignment clause, 4-6

SCRATCH pseudo instruction, 6-6
Second pass, overview, 1-2
Set carry flag instructions, 5-3, 5-18 through 5-20
Set flag assignment clause, 4-9
SET pseudo instruction, 6-30
Shift instructions
circular shifts, 5-18
end off shifts, 5-17
overview, 5-17
Sign for a data item, 2-12
SKIP pseudo instruction, 6-42

Source line
format, 2-1
length of 2-1
Source statement listing, APML, 8-1
SPACE pseudo instruction, 6-26
Special
elements, 2-8
function assignment clause, 4-10
SSD
input (HIA) functions, 7-4
output (HOA) functions, 7-4
START pseudo instruction, 6-9
Statement format
comment statement, 2-1
overview, 2-1
pseudo instruction format, 2-2
symbolic APML instruction format, 2-2
Stderr
error output, C-1
UNICOS, 3-5
STOPDUP pseudo instruction, 6-57
String
characters, 2-11
data item format, 2-11
SUBTITLE pseudo instruction, 6-27
Subtract from accumulator instructions, 5-2, 5-10 through 5-12
Symbol
attributes, 2-5
definition, 2-5
reference, 2-7
text dataset, COS, 3-3
Symbol definition pseudo instructions
CHANNEL - channel symbol, 6-31
EQUALS - equate symbol, 6-29
MICSIZE - set redefinable symbol to micro size, 6-32
overview, 6-29
SET - set symbol, 6-30
Symbolic APML instruction format, 2-2
Symbolic APML instruction syntax
data generation statement instruction format, 4-18
operand notation, 4-1
operators, 4-3
overview, 4-1
program statement instruction format
assignment clauses, 4-5
condition clauses, 4-11
overview, 4-4
syntax graphs for APML program statements, 4-13
Syntax
assignment, 4-14
condition, 4-16
SYSREF, 8-1
System Interrupt Enable flag, clearing and setting, 5-4
System text
overview, 3-6
file dataset, COS, 3-3

Table method of evaluation, attribute
  evaluation, 2-20
Term
    attributes, 2-15
    definition, 2-14
Terminate execution of program sequence, 5-4
Test accumulator condition clause, 4-11
Test carry flag condition clause, 4-13
Test channel status condition clause, 4-13
Test register or memory condition clause,
  4-12
TEXT pseudo instruction, 6-27
TIA functions, 7-5
$TIME micro, 6-62
TITLE pseudo instruction, 6-27
TOA functions, 7-5
Transmit from accumulator instructions,
5-2, 5-16, 5-17
Transmit to accumulator instructions, 5-1,
5-5, 5-6
Truncation of expression value, 2-18

Underscore (concatenation) character, 2-4
UNICOS APML
    command line, 3-4
    handling of messages, C-1
    JCL example, 3-6

Value
    attribute for a symbol, 2-5
    expression attribute, 2-18
VWD pseudo instruction, 6-36

W. word address prefix, 2-13
WAIT
    instruction, 4-10
    reserved name, B-2
Warning errors
    effects on assembly, D-1
    table, D-5
Word address
    attribute for a symbol, 2-5
    expression attribute, 2-18
    prefix - W., 2-13
Word-bit-position counter, 6-15

$XRF, default binary symbol table for
  SYSREF, 3-4

READER COMMENT FORM

APML Assembler Reference Manual SM-0036 B-01

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME ____________________

JOB TITLE ____________________

FIRM ____________________

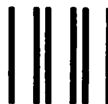
ADDRESS ____________________

CITY __________ STATE ______ ZIP ______

DATE ____________________

CUT ALONG THIS LINE

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120

FOLD

STAPLE

READER COMMENT FORM

APML Assembler Reference Manual SM-0036 B-01

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME ______________________________

JOB TITLE ______________________________

FIRM ______________________________

ADDRESS ______________________________

CITY ________________ STATE ________ ZIP ________

DATE ______________________________

CUT ALONG THIS LINE

FOLD

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120

FOLD

STAPLE

PUBLICATION CHANGE NOTICE

September 1986

TITLE: APML Assembler Reference Manual

PUBLICATION NO. SM-0036 REV. B CHANGE PACKET NO. B-01

This change packet brings the manual into agreement with the APML version 3.0 running under UNICOS 2.0. Please make the following changes to your manual:

Replace:
- Title page through xii
- 1-1 and 1-2
- 2-9 and 2-10
- 3-3 through 3-6
- B-3 through B-6
- Index