CRAY RESEARCH, INC.
SOFTWARE TRAINING

TR-APML

COURSE NAME:	**IOS**
TYPE OF MATERIAL:	**APML (IOS ASSEMBLER) SELF-STUDY WORKBOOK**
RELEASE LEVEL (include BF):	**V115BF3**
DATE PRINTED:	**DECEMBER 1988**

SECURITY CLASS: **PUBLIC**

CRAY RESEARCH, INC.

APML Assembler Self-Study

Software Training Workbook

TR-APML

This manual is not for further distribution without written approval from the nearest CRAY RESEARCH, INC., regional or country sales office.

RECORD OF REVISION PUBLICATION NUMBER TR-APML

Each time this manual is revised and reprinted, all changes issued against the previous revision in the form of change packets are incorporated into the new version and the new version is assigned an alphabetic level. Between reprints, changes may be issued against the current version in the form of change packets. Each change packet is assigned a numeric designator, starting with 01 for for the first change packet of each revision level.

Requests for copies of Cray Research, Inc. Software Training publications and comments about these publications should be directed to:

Cray Research, Inc.
Software Training
2520 Pilot Knob Road
Mendota Heights MN 55120

Revision	Description
	June, 1987 - Original Printing
A	April, 1988 - Reprint with revision; typographical errors corrected; UNICOS information added where pertinent; answer to exercise 4 of section 3 corrected.

Introduction

This self-study workbook is intended to teach APML, the I/O Subsystem assembler, to Cray Research, Inc. analysts and developers, and Cray customers. It assumes that the student is familiar with the X-MP CAL Assembler and the architecture of the I/O Subsystem, and has access to a CRAY machine to run assembly exercises.

For an introduction to the architecture of the I/O subsystem, the student is referred to the IOS Architecture Self-Study Workbook, publication TR-IA. **This information will be assumed in this present workbook.**

To complete this self-study, you will need the the following manuals and materials:

SM-ØØ36	AMPL Assembler Programmer's Reference Manual
SM-ØØ46	IOS Internal Reference Manual
SM-ØØØ7	IOS Tables Manual
SQ-ØØ59A	APML Reference Card For COS and UNICOS

$APTEXT listing for your site
$KERNEL listing for your site

Access to your CRAY-1 or X-MP to run sample assemblies

This self-study is based...

...on the V1.15 BF3 version of the IOS software. However, because APML is a very stable product, you will find the materials equally applicable to both earlier and later releases of the assembler. A quick glance at the page, in the introduction to the SM-ØØ36 manual, showing record of revisions will demonstrate how infrequently this product has been modified. The biggest change in the last several years is the inclusion of information on accessing APML from UNICOS, which is not really a change in the language itself. There are currently no plans for significant changes to APML. You may be confident that this material will be accurate for future releases of the IOS software.

Table Of Contents

Section 1 – The APML Assembler

Module Objectives: with the aid of all available reference materials, upon completion of this self-study module, the learner is able to:

1. Describe the general characteristics of the APML assembler.

2. Describe where AMPL fits in the hierarchy of high-level to low-level languages.

APML Is Different...

...from any language you have worked with. While it is categorized as an assembler, it includes many features usually associated with a compiler (such as multiple object statement generation from a single source statement). And, due to the requirements of the IOS operating system software, it does NOT do some things a garden-variety assembler would be expected to do (like automatically adding the base register to memory references, since it is more common to reference data areas OUTSIDE the current overlay, not INSIDE it).

Its syntax is also unusual for an assembler. The typical APML source statement looks very much like a FORTRAN assignment statement. The APML assignment statement is a very rich, versatile construct, with capabilities far more sophisticated than the usual assembler source statement, though not as sophisticated as the typical compiler. For instance, a single APML assignment statement could add the contents of one of the IOP's operand registers to the contents of a parcel in memory, shift the sum 4 bits right, and place the result in the B register. This would require several source statements in a lesser assembly language. While a FORTRAN assignment statement provides automatic order-of precedence evaluation, APML simply processes from left to right. Nor does APML provide the libraries of advanced math functions one would expect to find with a true high-level language.

APML also provides a sophisticated mechanism for making virtually any statement conditional. The syntax for conditions is very rich, rivaling the capabilities of many compilers.

In summation, APML's unique characteristics place it somewhere between the typical macro assembler and the typical compiler language. It includes many features normally associated with a compiler, while retaining the "flavor" and closeness-to-the-architecture of an assembler.

Programming Language Levels

COMPILER - FORTRAN, Pascal, BASIC, C

APML - in a class by itself...

MACRO ASSEMBLER

SIMPLE ASSEMBLER

MACHINE LANGUAGE

Compiler languages totally insulate the programmer from the actual machine environment and the instruction set of the machine. One source statement may generate many object statements.

Assembly languages require a greater degree of hardware-specific knowledge on the part of the programmer. One source statement generates one object statement. Source and object statements have similar syntax.

Macro assemblers permit the programmer to include pre-written assembly language routines in their program, to be modified by programmer-supplied parameters, and assembled as part of their program.

APML combines characteristics of a macro assembler and a simplified compiler. One APML source statement may generate many object statements. Low-level machine language instructions may also be included, and macro expansion is supported.

The Name APML...

...stands for "A-Processor Middle Language." "A-Processor" was the original engineering name for what became known as the I/O Processor. This also explains how $APTEXT, the IOS system text, gets its name, and the occasional references to the IOPs as AØ - A3 in certain comments and symbols. The designation of APML as a "Middle Language" reflects its intermediate status between an assembler and a compiler. **Current literature simply refers to APML as the I/O Subsystem Assembler.**

The APML Assembler...

...executes in the Cray mainframe, NOT in the I/O subsystem. The binaries that are generated will eventually be disposed to expander tape or disk, from which they will be loaded into the IOS at start-up time for execution. This makes the testing of IOS software somewhat trickier than the testing of software designed to run in the Cray. In the IOS, there are no jobs. All IOS software is system software, which must be integrated into the IOS operating system. In order to run a simple stand-alone APML routine, you must either get dedicated time, during which you can run your stand-alone program as if it were the complete IOS operating system, or test with CSIM, the Cray Simulator, in IOP simulation mode. This is a much more practical approach to testing IOS code in a training situation, and is in fact what we do in the IOS internals class in Mendota Heights for most of our exercises. In this workbook, our exercises ask you to produce a clean assembly only.

An APML Source Listing...

...doesn't look anything like a similar CAL listing. You would never guess that these two assemblers are produced from the same source code! The major similarity is the overall layout of the page, with source on the right and object on the left, in typical assembly listing fashion. We will examine listings in detail on later pages.

APML vs. CAL Assembly Listings

```
CALX                                                          CRAY XMP
       CAL 1


                                                  IDENT       CALX
                                                  START       BEGIN
           0    00000000000000000012  NUM        CON         10
           1                      1   SUM        BSS         1
                                2a+   BEGIN      =           *
          2a    1001 00000000+                   A1          NUM,0
           c    <opdef>                          A2          1
           d    <opdef>                          A3          2
          3a    <opdef>                          A4          0
           b    031110                LOC        A1          A1-1
           c    030442                           A4          A4+A2
           d    030233                           A2          A3+A2
          4a    030001                           A0          A1
           b    011 00000003b+                   JAN         LOC
           d    1104 00000001+                   SUM,0       A4
          5b    <macro>                          ENDP
                                                  END
```

```
APMLX                                                         IOP
APML


                                                  IDENT       APMLX
                                     0      SC    EQUALS      0
                                     1      REG1  EQUALS      1
                                     2      REG2  EQUALS      2
                                     3      REG3  EQUALS      3
                                                  SCRATCH     SC
           0    010012  024001                   R!REG1=12
           2    010001  024002                   R!REG2=1
           4    010000  024003                   R!REG3=0
           6    027001                LOC        R!REG1=R!REG1-1
           7    020003  022002  024003           R!REG3=R!REG3+R!REG2
          12    010002  025002                   R!REG2=R!REG2+2
          14    020001  107007                   P=LOC,R!REG1#0
          16    014000 /000024  024000           (SUM)=R!REG3
                020003  034000
          23    001000                           EXIT
          24                          SUM        <1>
                                                  END
```

APML Features

1. Extremely flexible assignment and conditional syntax

2. Free Field Format - same as in CAL
   Columns 1 to 72 are scanned
   a. Location field begins in column 1
   b. Result field begins with the first non-blank character after location
   c. Operand field begins with the first non blank character after result
   d. Comment field begins with a period
   e. A line beginning with * or . is a comment line

3. Limited set of mathematical functions reflects environment where entities being calculated are addresses or counters

Assembly Process

The APML assembler executes in the mainframe under COS or UNICOS

APML is loaded into central memory and begins executing as a result of a COS JCL statement or UNICOS command

Control Statement parameters specify options and datasets for the assembler run

Source statements may generate more than one object statement

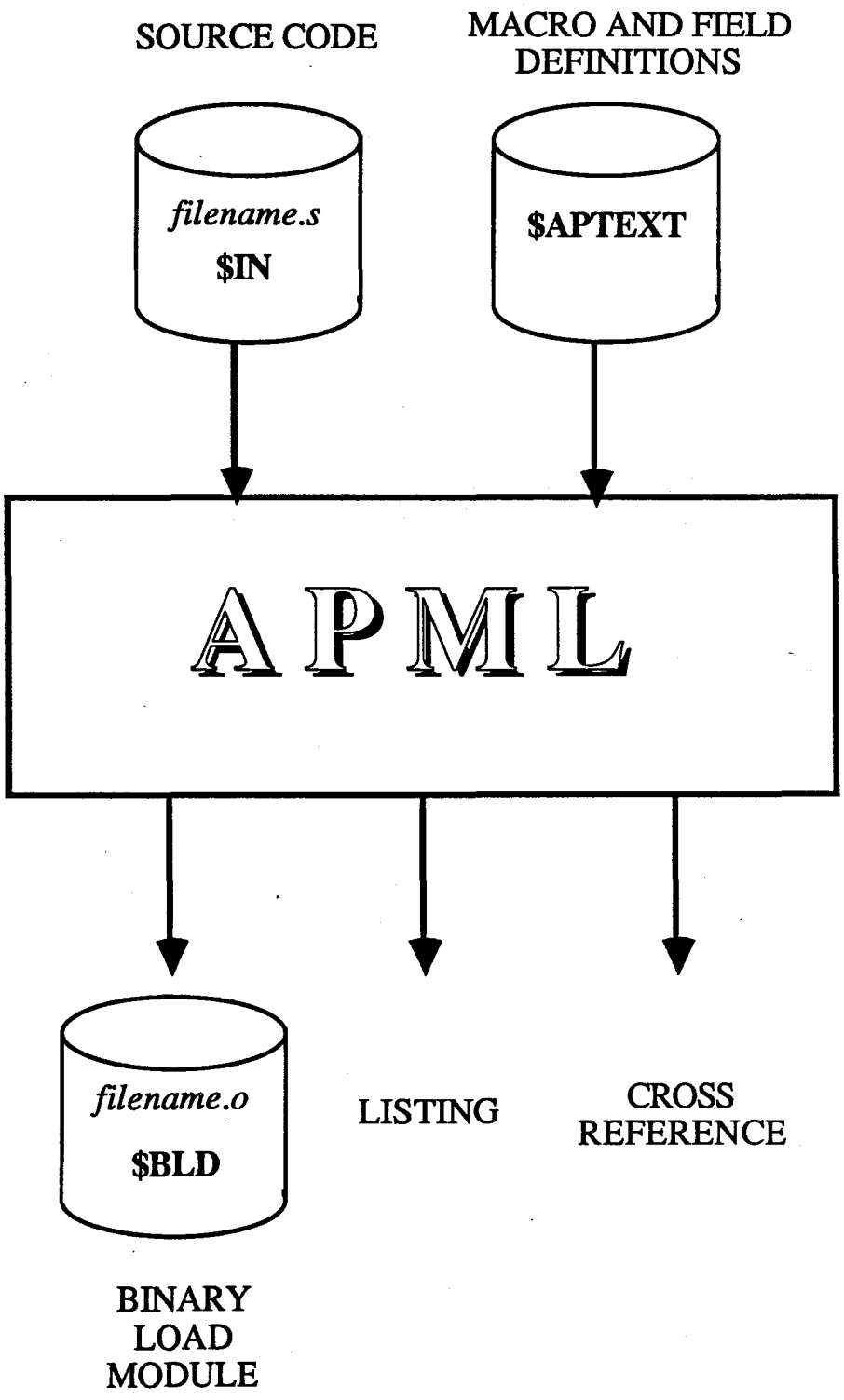
Assembles each module as encountered; BIND will later resolve externals

Two passes are made by the assembler for each module:

Pass ONE:
- Assembler reads each statement
- Expands complex assignment, conditional code
- Expands macros and assigns memory blocks
- Breaks sequences of code into 'pages'
  - Optimize code to one parcel when possible
  - Optimize to one parcel jumps
  - Jumps outside a page are two parcel *dd* jumps

Pass TWO:
- Assigns BLOCK origins
- Substitute values for constants and symbols
- Generate object code and listings

APML Assembler



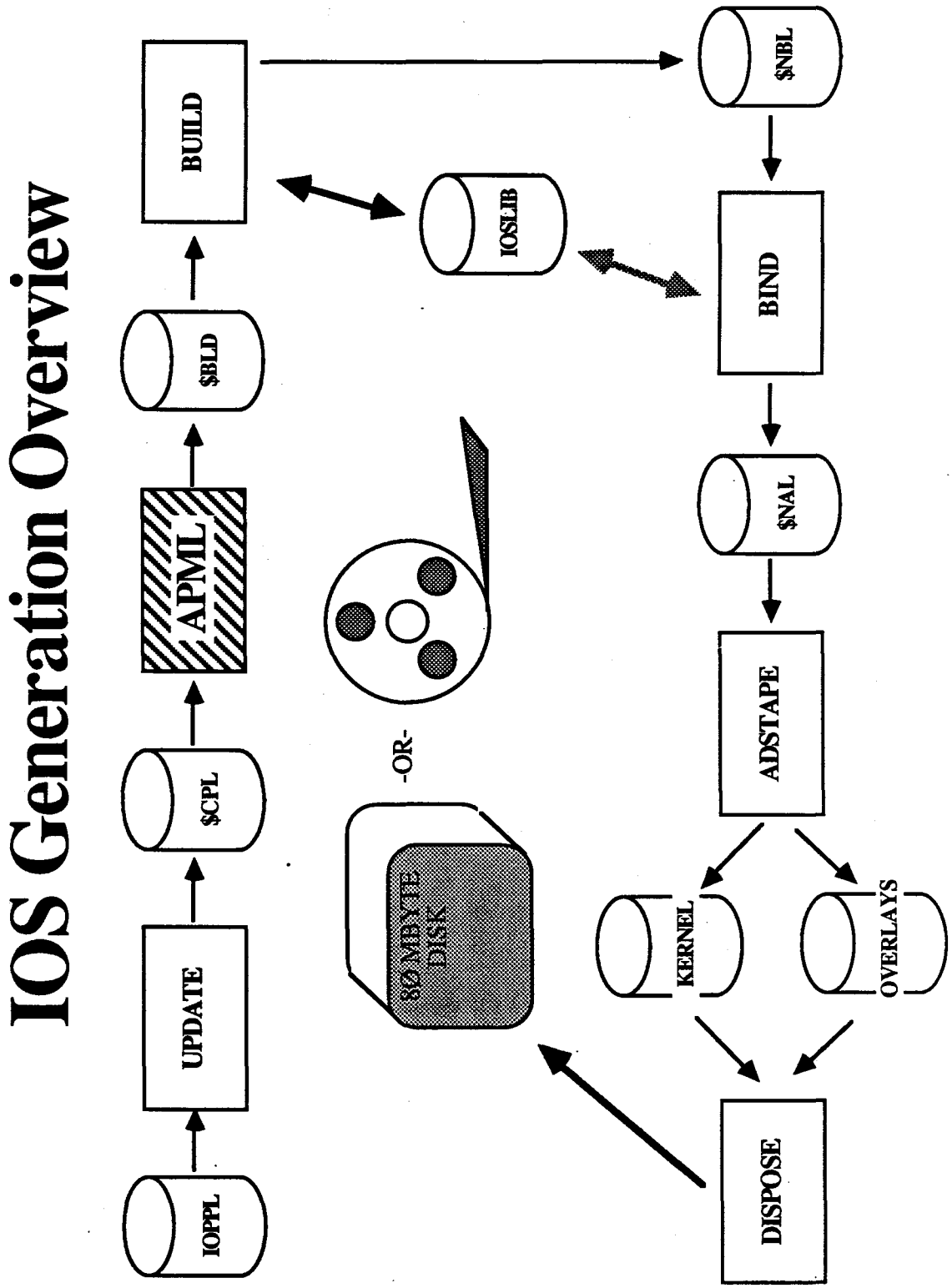
APML is used by...

...developers, almost exclusively. In fact, the original IOS system software was written by about five people. The total number of people who have ever written APML code for a living is extremely small - certainly less than twenty. One side effect of this very small audience is that APML is very seldom upgraded or enhanced. It has no major bugs in it, but is definitely not "polished" in many ways. Changes to make the language prettier are seldom if ever made. Why expend developer time on a product that only a handful of people will use? This "language written for twenty users" syndrome gives APML some unique idiosyncracies, which we will note in our studies.

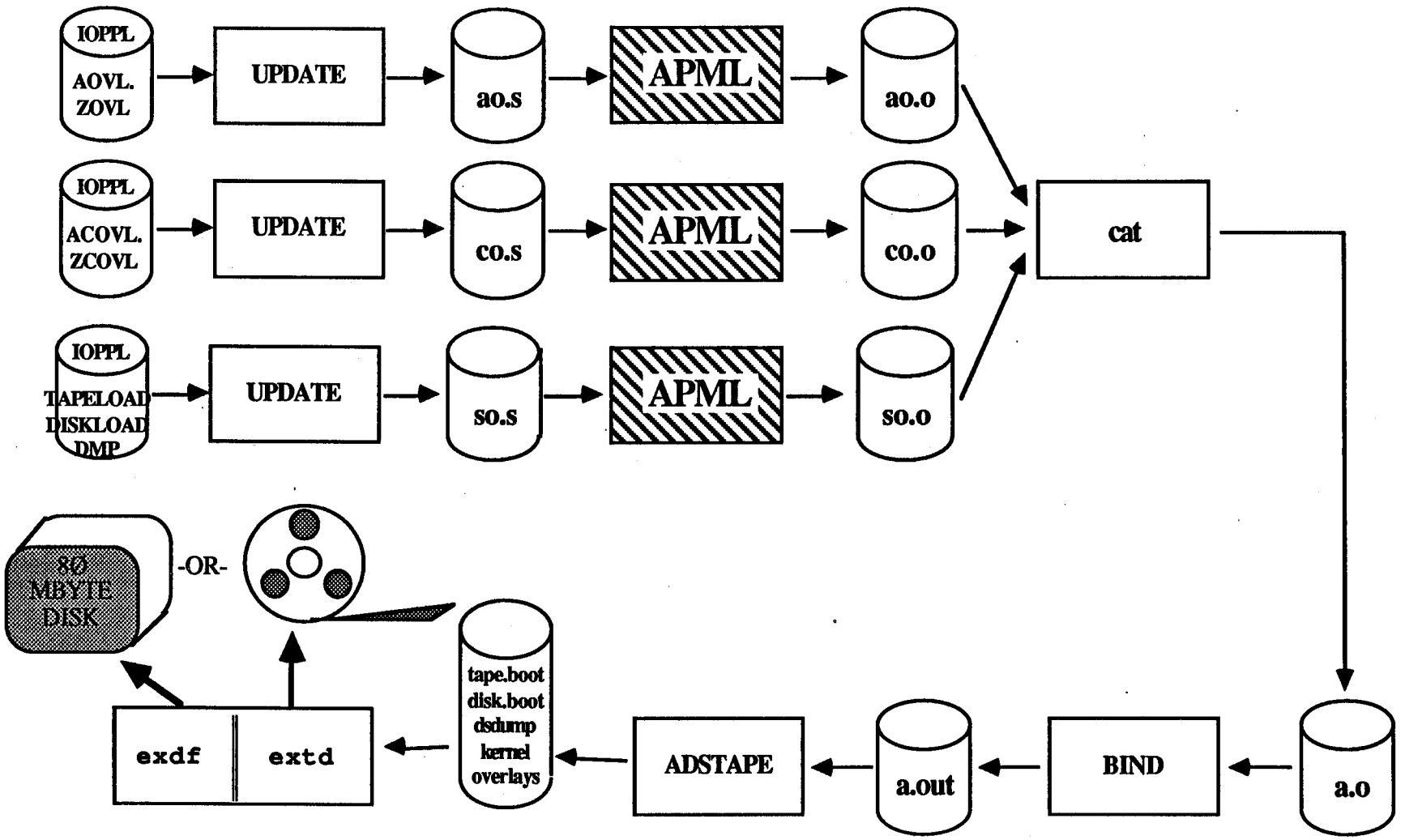
As an analyst or customer, you will probably never have occasion to actually write an overlay to integrate into the IOS software. You are quite likely to need to *read* APML listings for debugging purposes, however. And the best way to learn to read a language is still to write something in it. To this end, this self study includes several programming exercises, in which you will write APML programs.

If you attend the IOS Internals class, usually given in Mendota Heights, you will write a series of overlays, and test them, first with CSIM, and then in dedicated lab time. From this exercise, you will also learn important concepts of how the IOS system software is structured. IOS operating system concepts are beyond the scope of this self-study.

APML, as you would expect, is just one of several important utility routines needed to produce the IOS operating system software. The diagram on the opposite page summarizes the flow of the generation process under COS. The following page diagrams the flow of the IOS *make* process under UNICOS. The functions of the other routines are beyond the scope of this self study workbook. For details of generating IOS software from either COS or UNICOS, see the IOS Internals class.


IOS Generation Overview
IOPPL
UPDATE
$CPL
APML
$BLD
BUILD
$NBL
IOSLIB
BIND
$NAL
ADSTAPE
-OR-
80 MBYTE DISK
KERNEL
OVERLAYS
DISPOSE

IOS Make Overview



4/13/88

To Invoke The APML Assembler...

From COS, write a job that includes an APML control statement. This statement and its parameters are shown on the following page. The parameters are identical with those used in the CAL control statement.

From UNICOS, write a script that includes an APML command. This command and its parameters are shown on the following page. The parameters are identical with those used in the CAL command.

Reading Assignment 1

At this point, please read the following sections in the SM-ØØ36 APML Reference:

Section 1: Introduction

Section 2: APML Assembler Language
2-1 thru 2-6

Section 7: Channel Interface Functions

Section 8: Format of Assembly Listing

COS APML Control Statement

```
APML,CPU=type , I=idn , L=ldn , B=bdn , E=edn , T=bst ,
ABORT,DEBUG,LIST=name , S=sdn , SYM=sym , X=xdn .
```

		default
CPU=	MUST be IOP	IOP
I=	Source Input Dataset	$IN
L=	List Output Dataset	$OUT
B=	Object Binary Output	$BLD
E=	Error Listing	None
ABORT	Abort on Assembly Error	Do Not Abort
DEBUG	Clears Fatal Error Flag	Set Flag
LIST	All List Pseudos Activated	List Pseudos Ignored
LIST=	Matching Name Not Ignored	
S=	System Text Definitions	$APTEXT
SYM	Symbol Tables	None
T=	Binary System Text	None
X=	Global Cross Reference	None

(Note that these parameters are identical to the COS CAL control statement.)

UNICOS APML Command Line

```
apml [-t bsys] [-r xref] [-g sym] [-l listing] [-m tmwords] [-L]
[-s text1,text2,text3,...,textn] [h] or [-i nlist] [-o binary] name.s
```

		default
-t	Binary System Text	None
-r	Binary Cross Reference	None
-g	Symbol Table	None
-l	Assembler Listing	None
-m	Memory for Table Manager	65,476 words
-L	Statistical Logfile Messages	Not Reported
-s	System Text Definitions	None
-h	Process List Pseudos	All
-i	Process List Pseudos	Those Specified
-o	Binary Object File	name.o
name.s	Source Code File	

Sample COS job to assemble an APML source program from $IN:

```
JOB,JN=APML,US=xxx,T=2.
ACCOUNT,AC=nnn,UPW=xxx.
APML.
/EOF
  (YOUR PROGRAM GOES HERE)
/EOF
```

Sample COS job to assemble an APML source program from IOPPL:

```
JOB,JN=OVLIST,US=.
ACCOUNT,AC=,UPW=.
ACCESS,DN=IOPPL,OWN=SYSTEM,ID=V115BF1.
UPDATE,P=IOPPL,I=0,N=0,Q=AMAP:ICOM:CONMAN.
APML,I=$CPL,L=LIST,LIST,B=0.
DISPOSE,DN=LIST,MF=TB,TEXT='/u1/sjh/ios/ovlist'.
```

UNICOS commands to assemble an APML source program from IOPPL:

```
cp /usr/src/ios/pl/ioppl /yourpath/ioppl
update -p ioppl -s aptext.s -q AT
/lib/apml -t aptext -m 150000 aptext.s
update -p ioppl -s icom.s -q ICOM
/lib/apml -s aptext -m 150000 -l listfile -h icom.s
```

Review Questions for Section 1

1) What characteristics of APML make it difficult to classify in the traditional high level/low level hierarchy?

2) How do IOS binaries get from the Cray, where they are assembled, to the I/O Subsystem, where they execute?

3) What makes APML code more cumbersome to test than similar CAL code?

Section 2 – APML Syntax I: Assignment Statement

Upon completion of this module the student will be able to:

1. Identify valid and invalid APML source statements, and locate errors in syntax.

2. Recognize the standard APML notation symbols used in SM-ØØØ7 and SQ-ØØ59, and in this workbook.

3. Follow program logic in an APML assembly listing, using both the source and object listings.

The table on the opposite page summarizes the possible source statement formats in APML.

An APML Source Statement may be one of the following

1. An assignment statement or a conditional assignment statement (format "a" or "b" on the following page)

2. A pseudoinstruction, specifying assembly options or conditions to the APML assembler (format "f")

3. A comment line (format "d")

4. Data Definition (format "c")

5. A macro call (format "f")

Format "e" on the following page shows a neat way to assign a symbol to the beginning of a routine. It says, in essence, "Assign the current assembly location counter value to the label, and do not advance the assembly counter." This results in the following statement being assembled at the address specified by label. If we then branch to label, we branch to the statement following the format "e" statement.

The advantage to this is that it allows us to easily add statements to the beginning of the routine, by inserting them after the statement format "e". We do not need to remove the label from the former first statement in the routine.

APML Source Statement Formats

	Location	Result	Comment
a)	L	assign	.comment
b)	L	assign,condition	.comment
c)	L	data1,data2,data3...	.comment
d)	*	comment	
e)	L	*	.comment
f)	L	name params	.comment

L	optional statement label must begin in column 1
assign	assignment always has an = or :
condition	assignment condition
data1	data item or see PDATA pseudo
name	pseudo or macro name
params	any parameters or arguments needed
.comment	Always preceded by a period
,	A comma means 'IF' when used to delimit the condition portion of an assignment statement. A comma in column one indicates a continuation
*	Asterisk indicates entire line is a comment, or assigns current location counter to L

The APML Assignment Statement...

...is the primary statement format. It is used for moving data within the IOP, for branching, for setting flags, even for issuing channel functions. In its most common form, it looks much like a FORTRAN assignment statement, with which it has much in common.

The form of the assignment statement is:

```
R = expression
```

where **R** is the result field, where the result will be put, and **expression** is any valid APML expression, defined on the following pages. The **expression** will be evaluated (strictly left-to-right, no order-of-precedence), and the result placed in **R**.

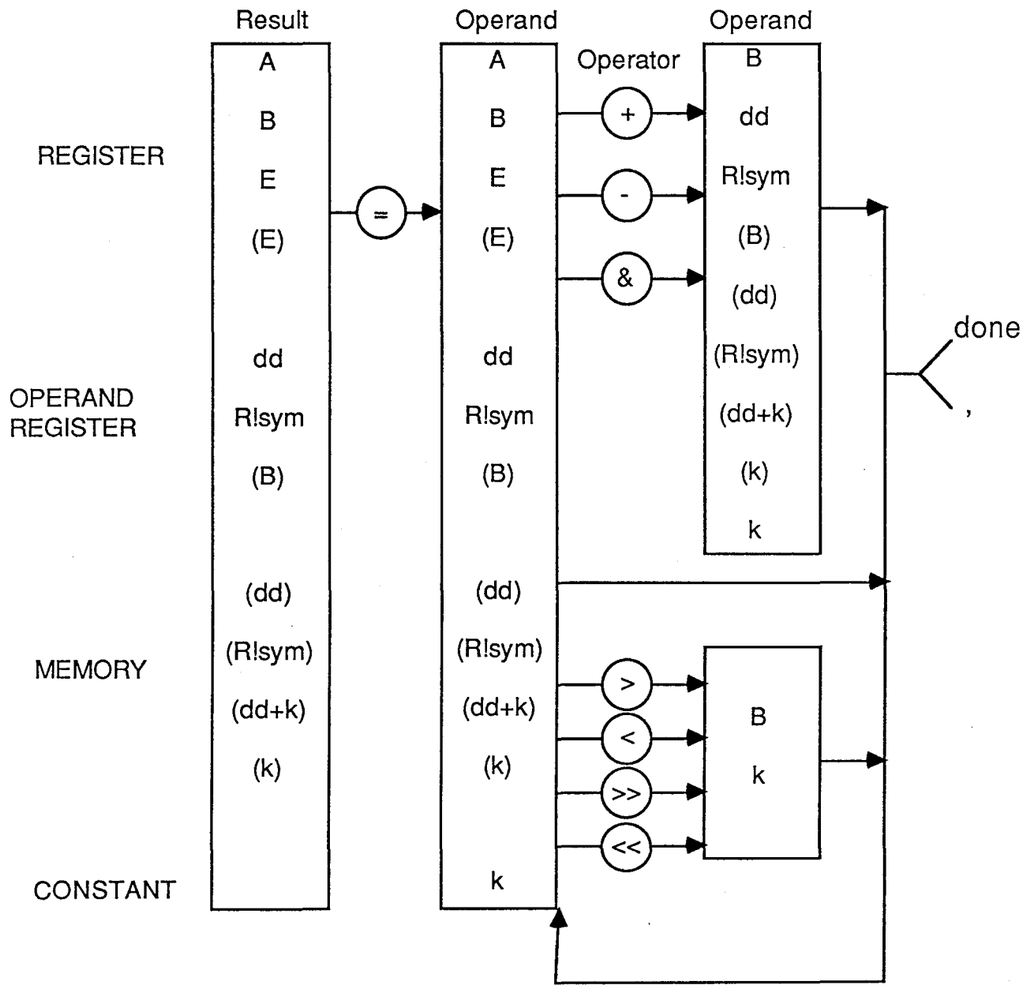
The structure of the APML expression is simpler than that of the FORTRAN expression, reflecting the relatively simpler range of functions needed in the context of the I/O Processor. The general form of an expression is:

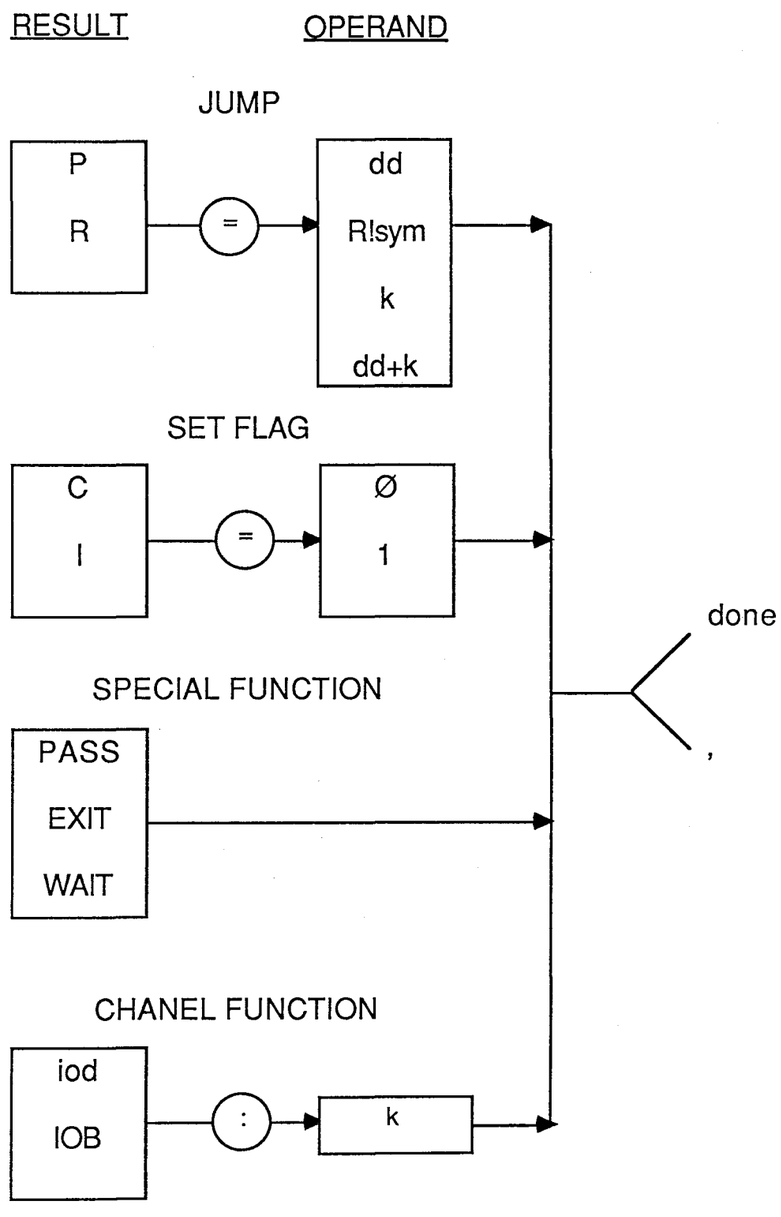
```
operand operator operand [operator operand...]
```

where **operand** is a valid APML operand (usually representing an IOP register or local memory address), and **operator** is a valid APML operator, as listed on the facing page in the column labeled "ASSIGNMENT". Not all operations may be performed on all operands. Legal constructs are defined by the APML Assignment Statement "Railroad Diagram", found by folding out this page to the left.

Fold Out>>>>

APML Assignment Syntax





Any assignment statement may be made conditional by adding a comma and a valid APML condition. Operators for conditions are defined at the bottom of this page, in the column labeled "CONDITION". Legal condition constructs are defined by the APML Condition "Railroad Diagram", on a fold-out at the end of the following chapter. These fold-outs are placed in such a way that you may refer to them both while you read the rest of these two chapters.

Symbol Meanings

Symbol	Assignment	Condition
=	Equal	Equal
#		Not Equal
+	Add	Add
-	Subtract	Subtract
&	Logical Product	Logical Product
:	Channel Function	
<	Shift Left	Less Than
>	Shift Right	Greater Than
<<	Circular Shift Left	
>>	Circular Shift Right	
<=		Less Than or Equal
>=		Greater Than or Equal

Operand Notation...

... is summarized on the facing page. There are several things to notice about how operands are specified.

First, note that the only way to access local memory is to store the Local Memory (LM) address in an operand register. You cannot simply refer to a location by symbol, as you would in virtually any other language that comes to mind. This reflects the fact that, in the IOS operating system software, overlays must be re-entrant. Upon request, the KERNEL will allocate data areas out of a pool of so-called "free memory", and return a pointer to this work area in an operand register. Since there is no variable data in the overlay itself, and constants are more practically established as equates, there is little motivation to access a data area within the confines of the current overlay. With 512 (decimal) operand registers in each IOP, there is no particular shortage, and we see that a great many of them are kept loaded with the addresses of pertinent local memory structures. For a fuller discussion of how operand registers are allocated, how local memory is allocated, operand register pointers, and KERNEL functions, see the IOS Internals class.

Notice also that **there are three ways to refer to the contents of an operand register (abbreviated OR).** The first method is to equate the operand register number to a **two-character symbol.** Such symbols are given special treatment by the APML assembler. When a two-character symbol is used, it is assumed to mean the contents of the operand register with that number. (Any other symbol resolves to the value equated to it.) The second method is to precede the symbol with "**R!**". This is the preferred method of referring to the contents of an operand register. (As you examine IOS system software, code that makes use of two-character symbols is the old code.) To indirectly refer to the contents of an operand register, you may place the number of the operand register into the B register, and use the notation **(B)**.

APML NOTATION

SYMBOL	MEANING
A	ACCUMULATOR
B	B REGISTER
C	CARRY BIT
(B)	CONTENTS OF OPERAND REGISTER ADDRESSED BY B
DD	CONTENTS OF OPERAND REGISTER (2-CHARACTER SYMBOL)
R!sym	CONTENTS OF OPERAND REGISTER
[DD]	OPERAND REGISTER NUMBER OF DD
(DD)	CONTENTS OF LOCAL MEMORY ADDRESSED
(DD+K)	CONTENTS OF MEMORY ADDRESSED BY SUM OF CONTENTS OF OPERAND REGISTER DD AND K (WHICH MAY BE A CONSTANT, OR, AND DISPLACEMENT ADDRESS)
E	EXIT STACK POINTER
(E)	EXIT STACK ENTRY ADDRESSED BY E
I	INTERRUPT ENABLE FLAG
IOB	I/O CHANNEL ADDRESSED BY B
IOD	I/O CHANNEL MNEMONIC DEFINED BY
K	POSITIVE NUMERIC OR CHARACTER
(K)	CONTENTS OF MEMORY ADDRESSED BY K
P	P REGISTER (PROGRAM ADDRESS REGISTER)
R	INDICATES RETURN JUMP
<n>	RESERVE n PARCELS
<<n>>	RESERVE AND INITIALIZE n PARCELS

The IOP Instruction Set...

...is documented both in the HR-ØØ81 IOS Hardware Reference Manual, and in the SQ-ØØ59A APML Reference Card. Look now at the reference card, pages 2-4. Pages 2 and 3 show the machine instructions, with columns for octal instruction code ("IOP"), symbolic description ("APML"), and verbal description. The operand notation used in the "APML" column is indicated at the top of page 4. Note that it is a subset of the notation listed on the previous page.

At the bottom of page 4 the two possible machine instruction formats are shown. The majority of APML instructions occupy 1 parcel of memory, and consist of a 7-bit *f*-field, and a 9-bit d-field. A smaller number of machine instructions occupy 2 parcels of memory, and add a 16-bit k-field.

The *f*-field always contains the machine instruction code, as defined in the "IOP" column on pages 2 and 3 of SQ-ØØ59A.

The d-field contains one of the following:

1) the number of an operand register (ØØØ-777). Note that the 9-bit size of this field is just the right size to hold an operand register number. (For example, op code Ø2Ø.)

2) Immediate data for the instruction to use. (For example, op code Ø1Ø.)

3) For channel functions, op codes 14Ø-157, the channel number on which the function will be issued.

4) For a relative branch instruction (op codes Ø7Ø-Ø73, and 1ØØ-117), a count of the number of parcels to branch forward or backward.

The k-field of 2-parcel instructions contains one of the following:

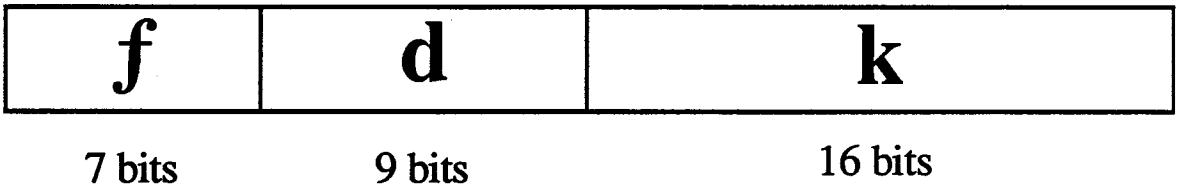
1) The branch address (relative to the base register indicated in the d-field) for an absolute branch instruction (For example, op code Ø75.)

2) Immediate data for the instruction to use (For example, op code Ø14.)

Machine Instruction Formats

1-parcel Instructions:

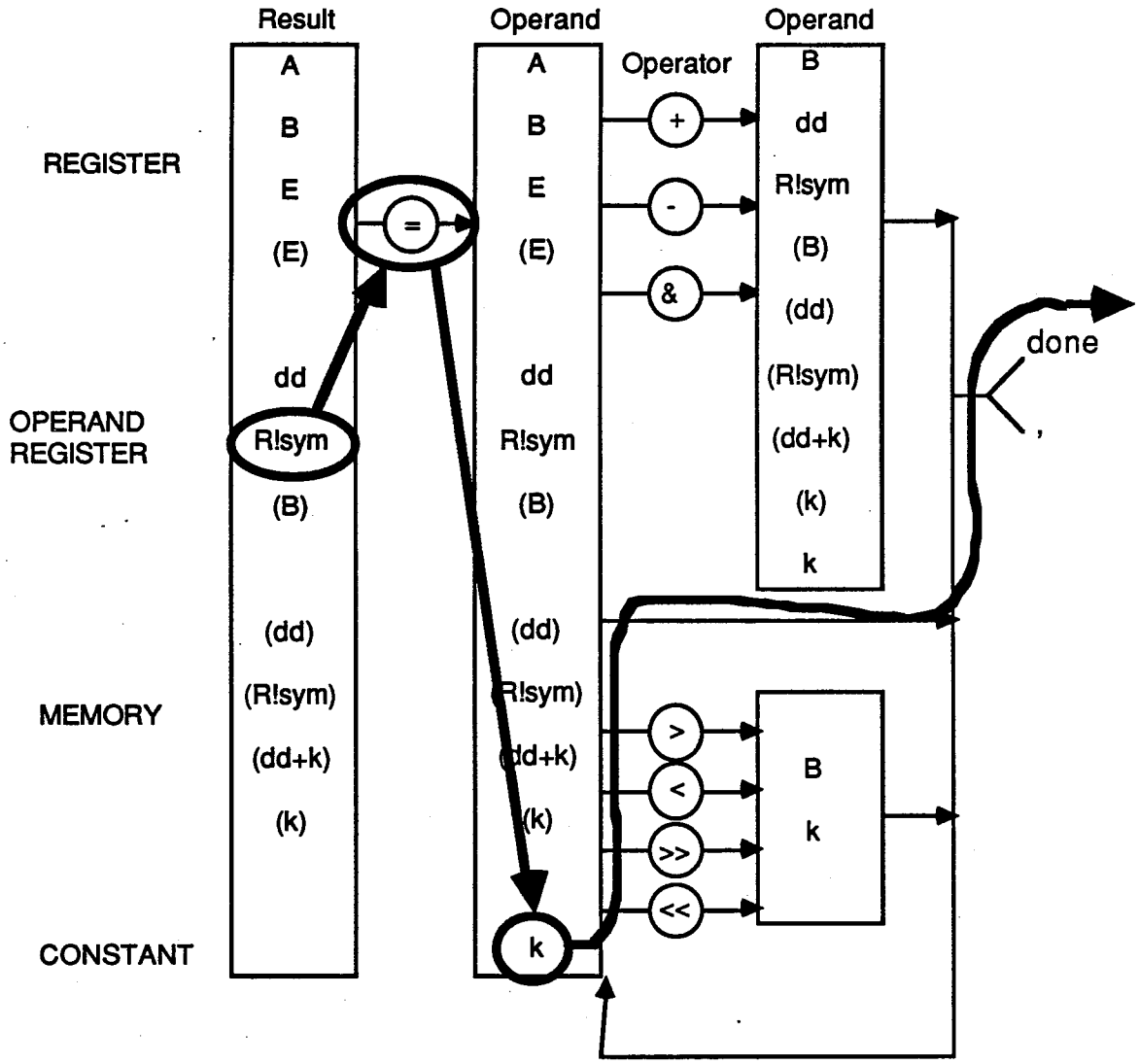
2-parcel Instructions:



Reading The Assignment Statement Syntax Diagram...

Start by selecting the receiving field from the list on the left. Following any arrow, you next select a first operand to follow the equals sign, then (usually) an operation and a second operand. This may then be the end of the statement, or you may select an additional operation/operand pair, and so on until complete. Any assignment statement may be made conditional by following it with a comma and a condition. Condition syntax is described in the next section of this self-study.

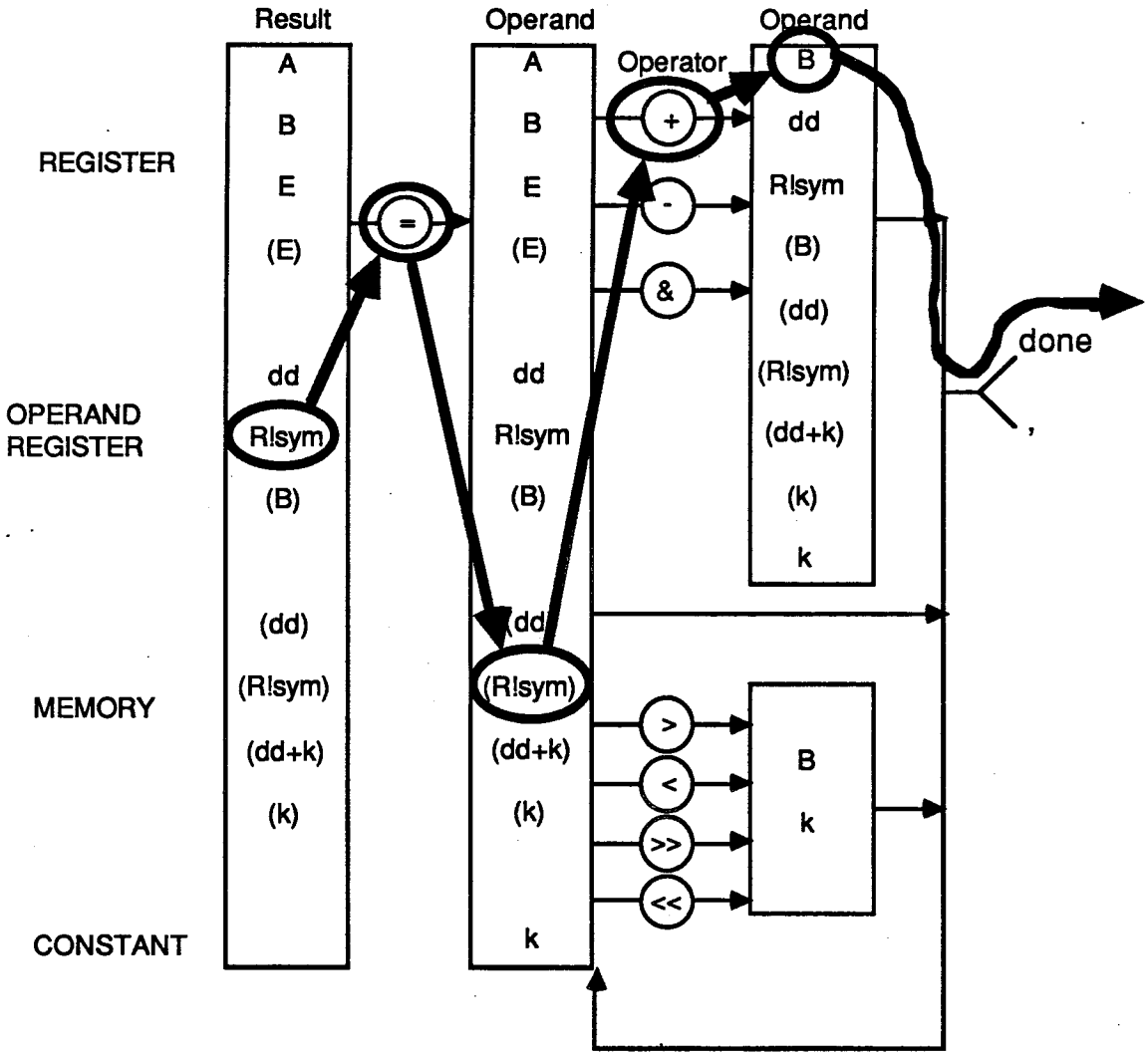
APML Assignment Syntax



This example shows how to traverse the diagram for the statement:

```
R!VAL = Ø
```

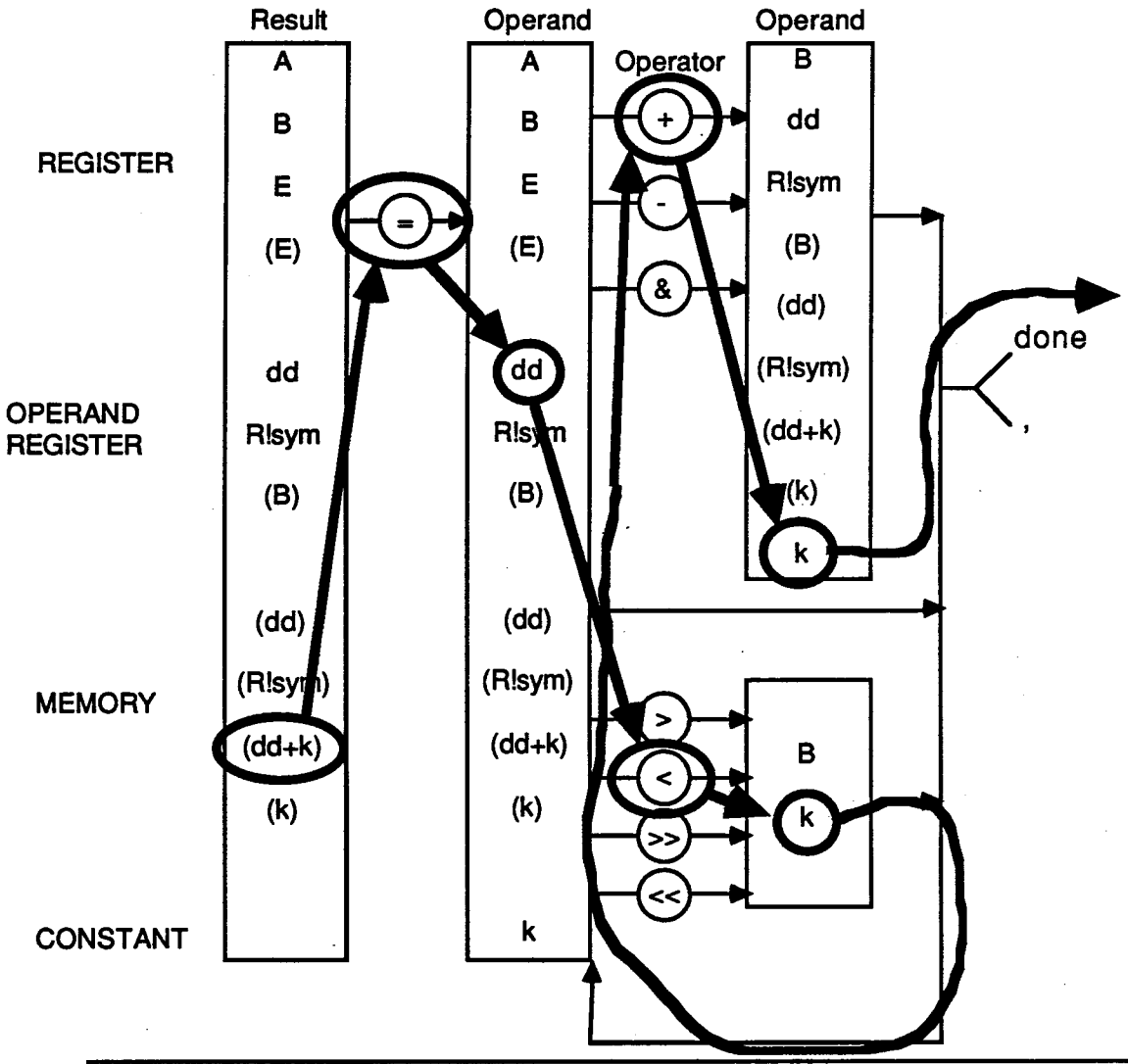
APML Assignment Syntax



This example show how to traverse the diagram for the statement:

```
R!THIS = (R!THAT) + B
```

APML Assignment Syntax

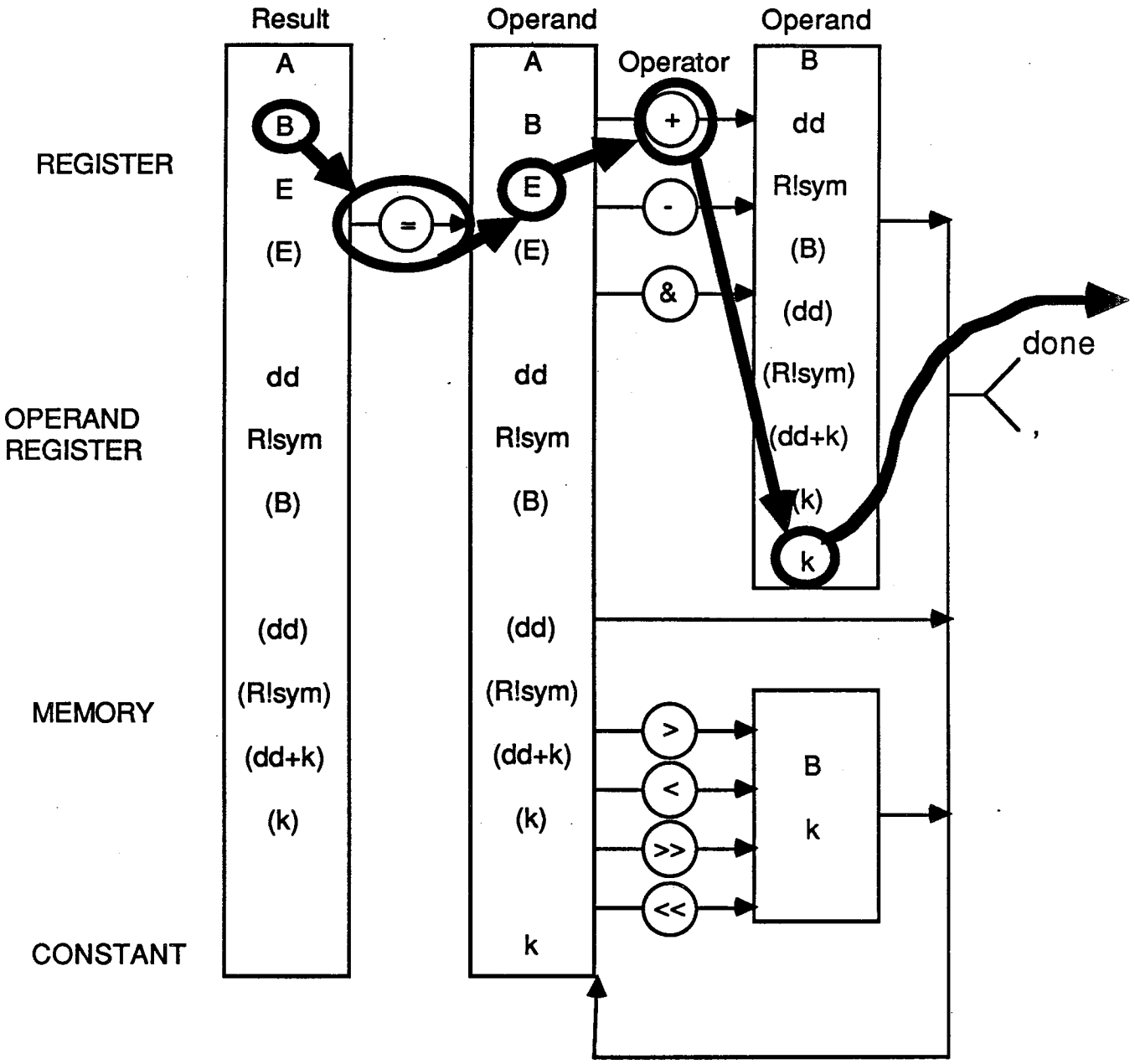


This example show how to traverse the diagram for the statement:

```
(R!TBL+FLD) = LS<2+TAG
```

An Odd Quirk of APML, Part One:

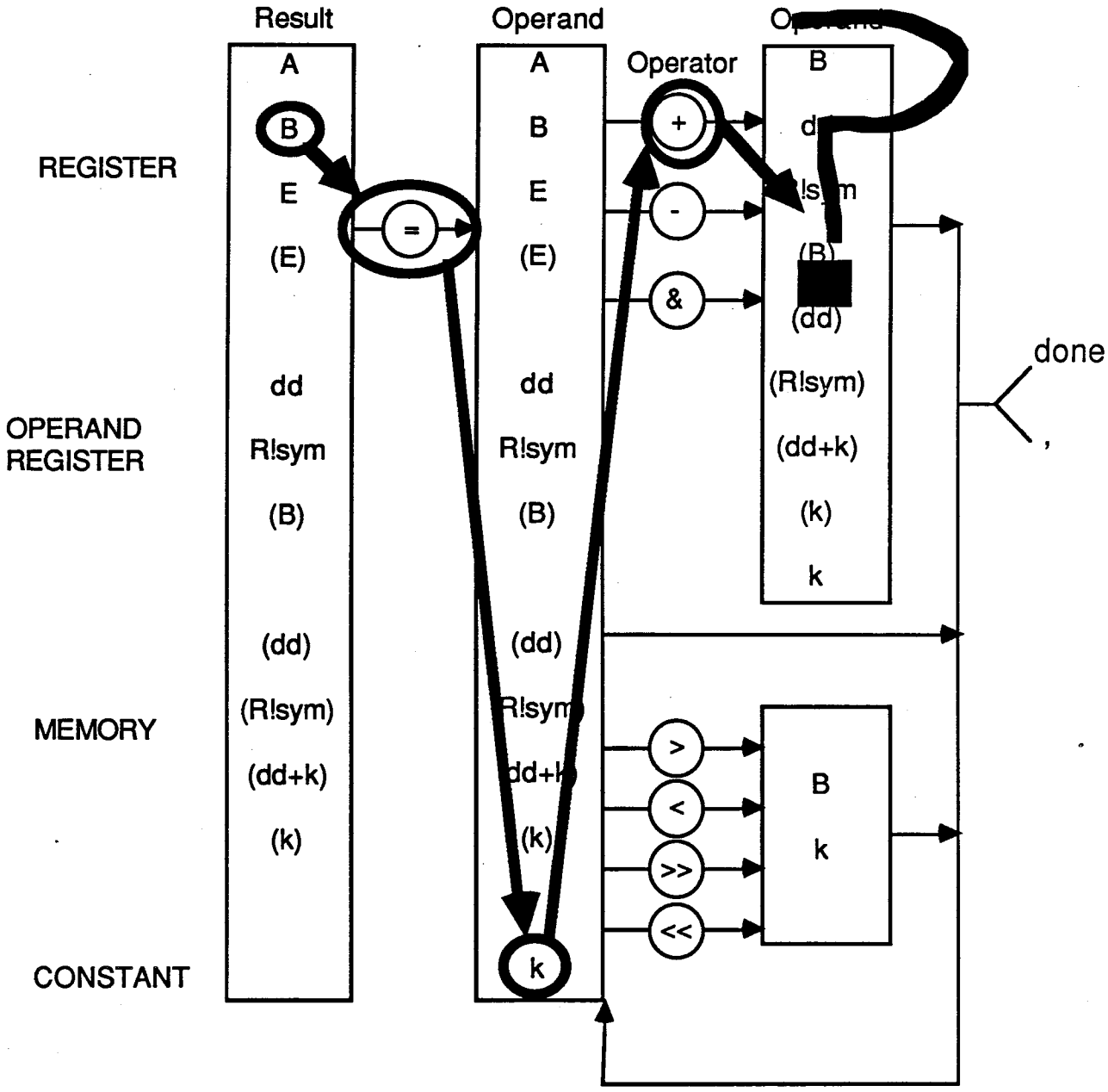
APML Assignment Syntax



B = E+5 works fine, but...

An Odd Quirk of APML, Part Two:

APML Assignment Syntax



...B = 5+E doesn't!

E (the Exit Stack Pointer) is legal as a first operand, but not as a second operand. APML apparently is not aware of the commutative law of addition....

APML Branch Instructions...

...are typically pretty easy to read - they look like an assignment statement with the P-register as the receiving field. A return branch (subroutine call) has a receiving field of R rather than P.They can be made conditional in the usual way: just follow with a comma and a condition.

The machine branch instructions, as shown on the APML Reference Card, pages 2 and 3, come in two flavors - relative and absolute. The distinction is no doubt familiar to you - Absolute branches specify an *address* to branch to, while relative branches specify an *offset* from the current p-register value. When we code a branch instruction, the APML assembler examines it and decides whether it will generate a relative or absolute machine branch instruction to accomplish it. As shown on the following page, the APML assembler becomes quite testy if we attempt to hard-code a relative branch instruction. In the example, we code a branch instruction as:

```
P=P+2.
```

This certainly seems legitimate on the face of it. After all, there is a hardware instruction with exactly this format (function code Ø7Ø). And yet, APML flags this statement with an error S7, a syntax error indicating "illegal operand following P= or R=" (see SM-ØØ36, p. D-3). Consulting our railroad diagram for an assignment statement with P as receiving field, we find that, indeed, the statement does not conform. But why not?

The answer lies in the fact that, at the source code level, the displacement of 2 is ambiguous. Does the displacement refer to 2 parcels, or two instructions (which may occupy a great many parcels)? Between this ambiguity, and the fact that hard-coding a relative branch is a dreadful programming practice anyway, the APML assembler forbids it. The second example shows the legal way of coding the desired branch. We specify the destination as a symbol. The APML assembler examines the statement, and determines that a forward displacement of 2 parcels will accomplish the branch (object code Ø7ØØØ2).

Correct Usage Of P Register:

```
                                        IDENT    PRULES
S7     0  070000                        P=P+2
       1  050000                        A=B
       2  054000              CAT       B=A
                                        END



                                        IDENT    PUSAGE
       0  070002                        P=DOG
       1  050000                        A=B
       2  054000              DOG       B=A
                                        END
```


The annotated APML listings on the following fold-out pages...

...demonstrate how to interpret the object code generated by the assembler, and how to relate the source code you create to the generated object code. In addition, several useful tips/tricks/traps are discussed. Please examine them in detail.

Fold Out>>>>

Annotated APML Listing 1

The "=" pseudoinstruction from CAL, for assigning assembly-time values to symbols, becomes the "EQUALS" pseudoinstruction in APML. This is one of the few differences in APML pseudos vs. CAL pseudos. The symbol in the label field of the EQUALS statement is replaced by the expression in the result field of the statement, wherever it is used in the assembly. In this example, the symbols SC, REG1, REG2, REG3 will be used to refer to Operand Registers Ø-3.

The SCRATCH pseudoinsrtuction assigns up to 5 operand registers for APML's use to hold intermediate results, or for memory pointers. This is one of the pseudos which is unique to APML, and does not appear in CAL. See the chapter on pseudos for a more complete discussion.

```
APMLX                                                          OP   APML

                                                  IDENT      APML
        0              SC          EQUALS     0
        1              REG1        EQUALS     1
        2              REG2        EQUALS     2
        3              REG3        EQUALS     3
                                   SCRATCH    SC
 0   010012  024001                R!REG1=12
 2   010001  024002                R!REG2=1
 4   010000  024003                R!REG3=0
 6   027001                LOC     R!REG1=R!REG1-1
 7   020003  022002  024003        R!REG3=R!REG3+R!REG2
12   010002  025002                R!REG2=R!REG2+2
14   020001  107007                P=LOC,R!REG1#0
16   014000 /000024  024000        (SUM)=R!REG3
      020003  034000
23   001000                        EXIT
24                         SUM     <1>
                                   END
```

Look up any new opcodes in this group, and see how the generated object code relates to the given source statements.

Two new concepts are involved in this source statement. First, this is a branch instruction. Second, it is conditional. We might paraphrase this statement in this way: "Branch to LOC if operand register REG1 is not = Ø." Examine the object code to see how this is accomplished.

Here a single object statement has generated 5 parcels of object code, including the first 2-parcel instruction we have encountered. The second parcel (k-field) of a 2-parcel instruction is easy to identify in the object listing, because it will be preceeded by a slash. This statement also contains the first memory reference we have encountered: the receiving field of this statement is the memory location at symbol SUM. Note that the symbol MUST be in parentheses, to indicate "contents of memory." As we follow the object code, we see the accumulator being loaded with the address of symbol SUM (24), and this address stored into operand register Ø.You will recall that we need an operand register pointer to access local memory. But why was operand register Ø used? It is not named anywhere in the source statement. How did the assembler know to use that particular register to hold the pointer? The answer lies in the SCRATCH pseudo at the beginning of the listing. This statement tells the assembler to use SC (operand register Ø) as a work register.

The first three executable (that is, non-pseudo) instructions in this sample program are assignment statements, which load operand registers with initial values. Note that each of these assignment statements generates 2 separate 1-parcel machine instructions.
The first generated machine instruction in each pair is opcode Ø1Ø, which says: "Load the accumulator with the contents of the d-field of this instruction." In each case, the value loaded is the constant value from the right side of the equals sign in the source statement.
The second generated machine instruction in each pair is opcode Ø24, which says: "Store the contents of the accumulator into the operand register named in the d-field." In each case, the operand register stored into is the register number equated to the symbol from the left side of the equals sign in the source statement. For instance:

```
0    010012   024001                              R!REG1=12
```

Annotated APML Listing 2

```
1CONDX                                          IOP  APML 2.1(02/23/87)  03/08/87 08:26:09  Page 1
                                                                                              (1)

                                                IDENT    CONDX
 0    <macro>                                   REGDEFS  ,,(SOURCE,DEST,COUNT,WORK)
 0    072023                        START       R=SETSRC
 1    020432  034430                LOAD        (R!SOURCE)=R!COUNT
 3    026430                                    R!SOURCE=R!SOURCE+1
 4    027432                                    R!COUNT=R!COUNT-1
 5    020432  107005                            P=LOAD,R!COUNT#0
 7    072014                                    R=SETSRC
10    014000 /000102  024431                    R!DEST=TABLE2
13    030430  034431                MOVE        (R!DEST)=(R!SOURCE)
15    026430                                    R!SOURCE=R!SOURCE+1
16    026431                                    R!DEST=R!DEST+1
17    027432                                    R!COUNT=R!COUNT-1
20    020432  107006                            P=MOVE,R!COUNT#0
22    001000                                    EXIT
23    014000 /000032  024430    SETSRC          R!SOURCE=TABLE1
26    014000 /000050  024432                    R!COUNT=T1LEN
31    001000                                    EXIT
32                              TABLE1          <50>
102                             T1END           *
                                                NEWPAGE
                          50    T1LEN           EQUALS   T1END-TABLE1
102                             TABLE2          <50>
                                                END
```

From the object code, we see that OR43Ø has been assigned to the symbol SOURCE, and OR432 has been assigned to symbol COUNT. REGDEFS starts assigning registers with 43Ø. Notice that here the CONTENTS of register COUNT is stored in memory pointed to by register SOURCE. Compare the object code here to the statement at address 13.

This conditional statement is the exit test for this "load the table" loop. It is one of the simpler forms, since it simply loads the accumulator and brances on non-zero.

Here the length of the table is loaded into register COUNT. See the code below in which this value is calculated. Compare the object code of this instruction to the previous one, where the address of a symbol is loaded into an operand register. The form is identical!

Here the length of TABLE1 is calculated. The symbol T1END marks the next available parcel after TABLE1. The EQUALS pseudo calculates the difference between the start and end of the table to determine the length. The NEWPAGE pseudo forces the assembler to assign addresses to the symbols, so that the EQUALS can use them.

The REGDEFS macro assigns available operand registers for use by an overlay. In addition, it sets up certain work registers, including the scratch registers needed by the APML assembler for memory pointers and intermediate results. Examine the object code in this example to determine the registers assigned. For further information, see SM-ØØ46.

APML is smart enough to see that, for these operations, there is a single instruction that will accomplish it. If we were adding or subtracting, say, 5 to the register, the object code would have to be somewhat longer.

End of the program.

Here we load register SOURCE with the ADDRESS of TABLE1. Since we have not specified a base register with a BASEREG pseudo, the d-field of this instruction is Ø. When we test this program with CSIM as a stand-alone program, as opposed to integrating it as an overlay into the IOS operating system, it will load at address Ø, and execute correctly.

<<<<Fold Out

Review Questions for Section 2

1) Name three ways to refer to the contents of an operand register in APML syntax.

2) What does the f-field of an IOP machine instruction contain?

3) What might the d-field of an IOP machine instruction contain?

4) What might the k-field of an IOP machine instruction contain?

5) How can an APML assignment statement be made conditional?

6) How do you refer to the contents of the B Register in APML syntax?

7) Explain the difference between the notation **[DD]** and the notation **(DD)**.

Section 3 – APML Syntax II: Condition Syntax

Module Objectives: with the aid of all available reference materials, upon completion of this self-study module, the learner is able to:

1. Identify valid and invalid examples of APML conditions.

2. Construct conditional assignment statements.

3. Relate conditional assignment source statements to the object code produced for them.

Any APML assignment statement may be made conditional...

...by adding a comma and a valid APML Condition. Condition syntax is defined by the APML Condition Syntax Diagram, found on a fold-out page at the end of this section. The fold-out on this page contains an annotated assembly showing conditional statements. Please locate and fold out these diagrams now.

Fold Out>>>>

APML Annotated Listing 3

Operand registers assigned to each symbol by REGDEFS

Object code says: "load the accumulator with contents of OR 430 (R!THIS). Subtract contents of OR 431 (R!THAT). Branch ahead 26_8 parcels if acc is zero (ie, if two values were equal).

As always, condition is evaluated first. Object code says: "load the accumulator with OR 434 (R!POINT). Subtract contents of OR 433 (R!COUNT). Branch ahead 4 parcels (ie, skip next 3 parcels) if carry bit is clear (ie, if R!POINT < R!COUNT). Branch ahead 3 parcels (ie, skip next 2 parcels) if Acc = Ø (ie, if R!POINT = R!COUNT). If we don't take either branch, then R!POINT > R!COUNT, and we want to do the next two instructions. which accomplish the assignment. Load a value of 5 into the accumulator. Add the accumulator to OR 433 (ie R!COUNT), and place the result back in OR 433."

```
1CONDX2                                          IOP  APML 2.1(02/23/87)  03/08/87 10:40:02  Page 1
                                                                                               (1)
                                                 43Ø   431   432   433   434    435
                                          IDENT     C
 0     <macro>                            REGDEFS   ,(THIS,THAT,WHAT,COUNT,POINT,TABLE)
                                   ***********************************************
                                   *     JUST A BUNCH OF CONDITIONAL STATEMENTS    *
                                   *     TO SHOW HOW THEY ARE ASSEMBLED. THEY DO   *
                                   *          NOT ADD UP TO A PROGRAM.             *
                                   ***********************************************
 0     020430  023431  102026             P=TAG,R!THIS=R!THAT
 3     020434  023433  100004             R!COUNT=R!COUNT+5,R!POINT>R!COUNT
        102003  010005  025433
11     020435  016000 /000004              (R!TABLE+FIELD)=(R!POINT)+THIS>>3-(R!TABLE+FIELD),
                                 ,                   (R!TABLE+FIELD)#E>5
        024410  150002  004005
        024411  030410  023411
        102006  030434  012430
        006003  033410  034410
30     050000  013066  103002   TAG       IOR:10,B=66
        150000
34     001000                             EXIT
                          4     FIELD     EQUALS      4
                                          END
```

Object code says: "Load Acc with contents of B register. Subtract 66. If acc is non-zero, branch ahead 2 parcels (ie, skip next parcel). Issue channel function 1Ø (why?) on channel Ø."

Here's a really enormous one. Object code says: "Load the acc with contents of OR 435 (R!TABLE). Add the constant 4 (FIELD) to the acc, and save the result in scratch register OR 41Ø (REGDEFS assigns regs 41Ø-414 as scratch). OR 41Ø now points to (R!TABLE+FIELD) in local memory. The following opcode (15Ø) falls in the range 14Ø-157 seen on the APML Quick Reference, indicating that this is a channel function. But what function is it? The d-field contains a 2, indicating the channel on which the function will be issued. To determine the function being issued, subract the14Ø from the opcode, giving a 1Ø. We are therefore issuing channel function 1Ø on channel 2. Channel 2 is the exit stack channel, and function 1Ø, accourding to our handy APML card, is... well, you can see for yourself. The Accumulator now holds the E pointer. The next parcel will right shift the acc 5 bits. We then store the result (E>5) into scratch register 411. The next parcel loads the acc with the parcel of local memory that OR 410 points to - (R!TABLE+FIELD), you will recall. We the subtract the contents of scratch register 411 (E>5), and then branch ahead 6 parcels (ie, skip the next 5 parcels) of the acc=Ø (ie, if (R!TABLE+FIELD)= E>5)." We have now interpreted the parcels of this object code that implement the condition portion of this source statement. The last 5 parcels implement the assignment statement. Their interpretation is left as an exercise for the student.

Comparing the condition syntax diagram to the actual conditional machine instructions...

...it is clear that the source syntax gives us a much richer choice of conditions. All of the conditional **machine** instructions test either the accumulator equal/not equal to Ø, or the carry bit set/clear (see the APML Quick Reference Card SQ-0059). However, at the **source** level, we may test any register for any value, and even compare a register to the result of an expression. This is an extremely powerful and easy-to-use feature of APML. It is both convenient to code, and easy to interpret when reading a listing. The generated code is also very efficient.

Note that the condition syntax defined here will also be used in certain logic-structure macros, where a condition is required ($IF and $UNTIL, in particular).

Reading conditional statements

Please see the table "Symbol Meanings" in the previous chapter for an explanation of notation.

`R!POINTER=R!TABLE+FIELD,R!TABLE#Ø` is read: "POINTER gets TABLE plus FIELD if TABLE is non-zero."

`MOS:5,R!LENGTH>5` is read: "Issue channel function five to buffer memory if LENGTH is greater than 5."

`B=R!CHAN,(COUNT)>THRESHLD` is read: "Put the channel number into the B register if the local memory variable COUNT is greater than the constant THRESHLD."

A Modest Proposal: In reading APML to yourself, don't bother to say "Contents of Operand Register such-and-such" all the time. Instead, just say "such-and-such" when the operand is in an operand register. Be specific if a operand is something *other* than the contents of an operand register. Say, "The field such-and-such," or "The local memory variable such-and-such," or "The constant such-and-such"(for equates), and so on. Just a suggestion.

Don't Use A E xplicitly As An Operand ...

```
1NOAOPRND                                          IOP  APML 2.1(03/17/87)  03/20/87 09:14:12
Page  1

(1)

                                        IDENT    NOAOPRND
 0    <macro>                           REGDEFS  ,,(PNT,INDEX)
 0                              START    *
 0   010005                             A=5                    .LOAD ACC WITH VALUE 5...
 1   014000 /000023  024410             (LOC)=A                ...AND IMMEDIATELY KILL IT! SEE OBJ CODE
     034410
 5   010010                             A=10                   .ALSO PROB IF MEM ACCESS ON RIGHT OF ASG
 6   014000 /000023  024410             B=A+(LOC)              .ALWAYS CALCS MEM ADR FIRST
     032410  054000
13   020430  004002                     A=R!PNT>2
15   020430  013010  100003             R!INDEX=A,R!PNT>10     .LOOKS OK, BUT CHECK OUT THE OBJECT CODE
      102002  024431
                               .ACC IS CLOBBERED BECAUSE OBJ CODE FOR CONDITION IS GENERATED FIRST
22   001000                             EXIT
```

1. APML most closely resembles a _____ language.

2. Suppose Operand Register 17 is equated to both R1 and VAL, and the contents of Memory Location 1ØØØ is 2241. What is the Accumulator value after each instruction in the following sequence? Assume that all registers retain values from previous instructions.

R1=1ØØØ	,A=_____
A=R1	,A=_____
A=[R1]	,A=_____
A=(R1)	,A=_____
A=VAL	,A=_____
A=(R!VAL)	,A=_____
A=R!VAL	,A=_____

3. Suppose the symbol CAT has a value of 25 and Operand Register 25 is denoted by OR2. What is the accumulator value after each instruction in the following sequence?

R!OR2=721	,A=_____
A=R!OR2	,A=_____
A=CAT	,A=_____
A=R!CAT	,A=_____

4. Which of the following APML statements are illegal? (O=LETTER,Ø=NUMBER)

a) P=P+17
b) E=B+R1&17
c) R2=E&(B)>2
d) R2=(B)&E>2
e) A=B,B=E&R1
f) A=B,P=1ØØØ
g) A=B,B=R1&E
h) A=B,Ø=Ø+(B)-1Ø&Ø>Ø

Exercises For Section 3 Continued

Each of the numbered items below shows more than one notation that will accomplish a certain type of assignment in APML. Determine the results of the five assignments being illustrated.

```
R1      EQUALS  1
R2      EQUALS  2
R3      EQUALS  3
R5      EQUALS  5
REG1    EQUALS  1
REG2    EQUALS  2
REG3    EQUALS  3
REG4    EQUALS  4
```

1)
```
R!REG1=12
R1=12
```
operand register ____ gets _____

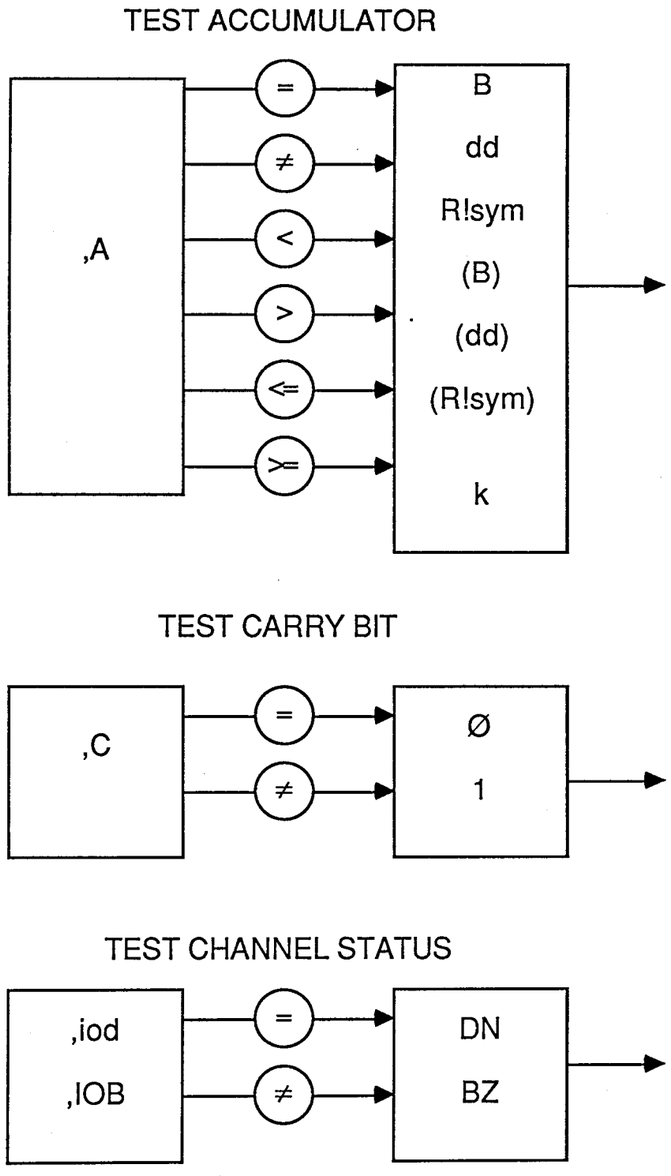
2)
```
R!REG2=R!REG1
R!REG2=R1
R2=R!REG1
R2=R1
```
operand register ____ gets _____

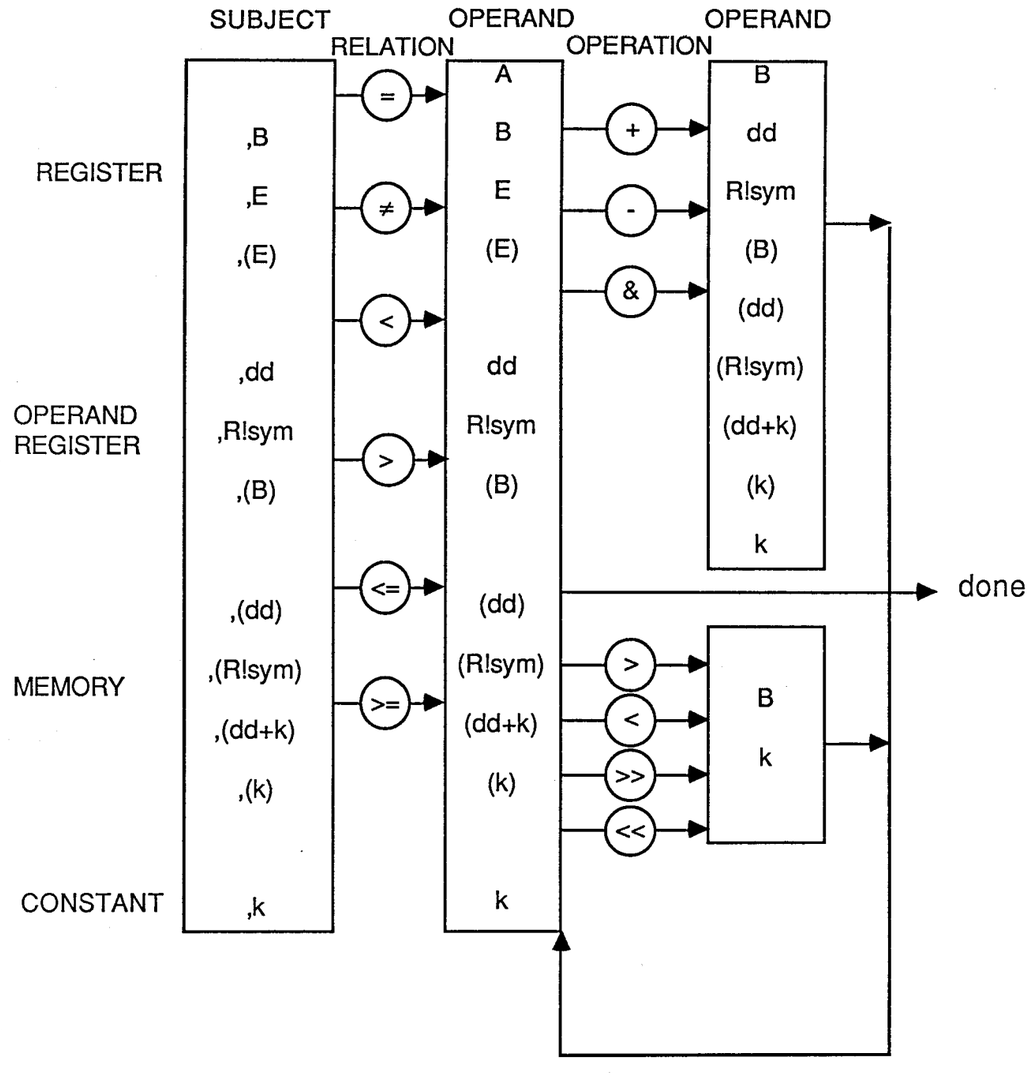
3)
```
R!REG3=(R!REG1)
R3=(R!REG1)
R!REG3=(R1)
R3=(R1)
```
operand register ____ gets _____

4)
```
R!REG4=REG1
R!REG4=[R1]
```
operand register ____ gets _____

5)
```
R5=(REG1)
R5=([R1])
```
operand register ____ gets _____

APML Condition Syntax





APML Condition Syntax Diagram Fold-Out

<<<<Fold Out

Section 4 – APML Syntax IV: Pseudoinstructions

MODULE OBJECTIVES

Upon completion of this module, the learner should be able to:

1. Use basic APML Pseudos in a program
2. Identify those Pseudos which are unique to APML
3. Describe why these Pseudos are unique to APML

APML Pseudos...

...are source statements that instruct the assembler to do something AT ASSEMBLY TIME. This is in contrast to normal source statements which the assembler turns into object code to execute AT RUN TIME.

The typical pseudo generates no object code. Instead, it affects the way object code is generated. (Data generation pseudos are the exception to this.)

The pseudos available in the CAL assembler are also available in the APML assembler, with the following exceptions:

a) The OPDEF pseudo is not available. (This is because the APML assignment statement is in fact implemented as an OPDEF.)

b) The COMMON pseudo is not available.

c) The "=" pseudo becomes "EQUALS." This reflects the special usage of the equals sign in the APML assignment statement.

In addition, APML includes several unique pseudos, which are not available from CAL.

In this self-study workbook, we will examine only those pseudos that are unique, different from those used in CAL, or required. The student is referred to section 6 in SM-ØØ36, the APML Reference, and the appropriate materials on the CAL assembler for further information.

APML Pseudoinstructions List by Class

CLASS	PSEUDOS 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, IFC, IFE, ENDIF, ELSE, SKIP
Instruction Definition	MACRO, LOCAL, ENDDUP, STOPDUP, ENDM
Micro Definition	MICRO, OCTMIC, DECMIC

(**BOLD** indicates unique pseudos, or those different from CAL)

Required Pseudos...

...must be present in every APML program for correct assembly. These pseudos are: IDENT, END, ABS, SCRATCH (SCRATCH is also a unique pseudo - i.e., not found in CAL).

IDENT and **END** mark the beginning and end, respectively, of a module to be assembled.

Location	Result	Operand
	IDENT **END**	**name**

name Name of this program module

These pseudos function in the same way as they do in CAL. They form "bookends" around an assembly module. You may include several such modules in the input dataset to APML, but each must begin with IDENT and end with END.

In the operating system, on overlay must have an IDENT and an END. The IDENT pseudo is generated automatically by the OVERLAY macro, which is used in every overlay in the system. However, you must remember to code an END explicitly at the end of the overlay.

Required Pseudos (Cont.)

ABS declares a module as an absolute binary

Location	Result	Operand
	ABS	

Since all overlays must be absolute, the OVERLAY macro includes an ABS pseudo. Note also that any code that is to be tested with CSIM must also be declared absolute.

ABS must follow IDENT, and precede everything else.

Required Pseudos (Cont.)

SCRATCH is a unique pseudo used to declare the scratch operand registers needed to generate code from complex APML statements. Scratch registers hold memory pointers and intermediate results. If SCRATCH is omitted, an error (type F - "too many entries") will be generated wherever a scratch register is needed, but not available.

The REGDEFS macro, used in all overlays, automatically assigns scratch registers for the programmer.

Location	Result	Operand
	SCRATCH	**R1,R2,R3,R4,R5**

R1-R5 Up to 5 previously defined symbols. Symbols must be defined elsewhere,and must not have been defined with SET

example:

```
SCRATCHX

                                          IDENT    SCRATCHX
                          51       S1     EQUALS   51
                          52       S2     EQUALS   52
                          53       S3     EQUALS   53
                          54       S4     EQUALS   54
                          55       S5     EQUALS   55
                          56       S6     EQUALS   56
                          57       S7     EQUALS   57
                          60       S10    EQUALS   60
                                          SCRATCH  S1,S2,S3,S4,S5
0   014000 /000017  024051                (HOLD)=(TEMP)+(HOLD)-46,.(TEMP)<(HOLD)
    014000 /000016  024052
    030051  033052  101005
    030051  032052  013046
    034052
15  001000                                EXIT
16                            HOLD      <1>
17                            TEMP      <1>
                                        END
```

Unique Pseudos...

...are SCRATCH (previous page), EQUALS, PDATA, BASEREG, and NEWPAGE

PDATA

Logically identical to DATA generation pseudo

Allows unrestricted use of two character symbols as data

Location	Result	Operand
L	PDATA	data1,data2,data3...

L — Statement label with parcel attribute

data1 — Can be any one of the following:

number
symbol
character string
use as many parcels as necessary
parcel storage reservation
* assigns current parcel counter to L

Unique Pseudos (Cont.)
BASEREG

A base register is required for two parcel jumps and for referencing data in a relocatable piece of code (overlay).

Two-parcel (Absolute) branches are generated whenever the destination of the jump is within a different PAGE of the module. The whole concept of pages comes up, in a round-about way, because of the fact that, in the IOP, some jump instructions occupy 1 parcel, while others occupy 2 parcels. Notice the problem encountered when we wish to branch to some point in our program. The assembler must decide whether to generate a 1-parcel relative branch (the preferred method), or a 2-parcel absolute branch (if the displacement is greater than the 7778 maximum that can be held in the d-field of the branch). It therefore needs to determine the distance to the destination before it can determine the length of the instruction. BUT... if the branch destination address is greater than the address of the branch instruction, then it must determine the length of this branch instruction, AND ALL OTHER BRANCH INSTRUCTIONS BETWEEN, before it can assign an address to the destination! This chicken-or-egg problem must be resolved, and the PAGE mechanism is the method chosen to break the deadlock. As the assembler progresses in examining source statements, it keeps a table of unresolved branch destinations. Whenever the assembler encounters a PAGE BOUNDARY (defined below), it determines, for each branch in the preceding page, whether the destination also falls within the page. It can now determine the size (1 or 2 parcels) of all branches in that page, and then assign values to all symbols in the page.

A page, then, is a logical construct of the APML assembler. It is a block of code within which all branches are relative (single parcel jumps). Maximum page size is 512_{10} parcels - the maximum displacement that could be specified in the d field of a relative branch instruction.

Two parcel jumps, DD+K are generated by the assembler for branch points outside of the current 'page'.

Unique Pseudos (Cont.)
Pages are delimited by 'page boundaries' which are formed as follows:

By an IDENT statement

Every 512_{10} parcels

By a pseudo instruction which forces a word boundary (i.e. BSS/BSSZ)

By a PDATA pseudo with a label

By a NEWPAGE pseudo instruction

The BASEREG pseudo is used to declare a base operand register, which the assembler then uses for all two-parcel branches

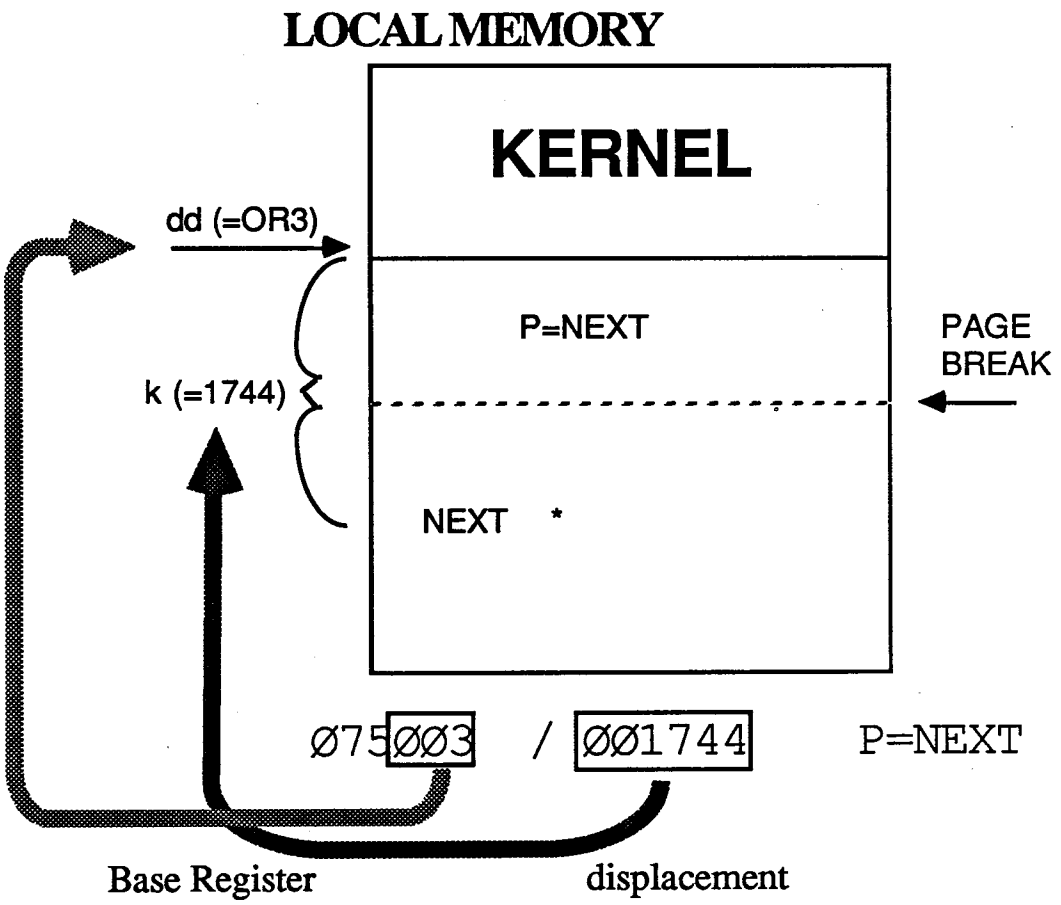
Location	Result	Operand
	BASEREG	**r**

r operand register used for base

Unique Pseudos (Cont.)
BASEREG (cont.)

example:

```
BREGX
                                                IDENT    BREGX
                                3       %B      EQUALS   3
                                                BASEREG  %B
    0    075003 /001744                         P=NEXT
    2                                           <1742>
 1744                                   NEXT    *
 1744    001000                                 EXIT
                                                END
```



Unique Pseudos (Cont.)
NEWPAGE

Allows the programmer to force a new page

This is useful primarily in two situations:

First, it can be used to prevent a page break from falling in the middle of a loop. A small loop will execute more rapidly if its branches are the single-parcel (relative) type, which permit an "in-stack" condition. If a page break should fall within the loop, two-parcel (absolute) branches must be generated, which do not permit an "in-stack" condition.

Second, NEWPAGE forces the assembler to assign values to symbols so that they may be used to calculate values for other symbols, as shown in the examples on the following page.

Location	Result	Operand
	NEWPAGE	

Unique Pseudos (Cont.)
NEWPAGE (cont.)

examples:

Without NEWPAGE - Assembly Error:

```
NPAGEX1
                                                IDENT     NPAGEX1
                                    3     %B    EQUALS    3
                                                BASEREG   %B
             0                              TBL   <110>
           110                              TBLEND *
XW6                                         TBLLEN EQUALS    TBLEND-TBL
           110   001000                             EXIT
                                                    END
```

When the assembler encounters the symbol TBLLEN during its first pass, it has not yet encountered its first page break, and therefore has not yet assigned values to the symbols TBLEND and TBL, on which the value of TBLLEN depends. The assembler gives the unhelpful error code XW6: "Expression Error; Location Field Symbol Not Defined." The real problem is that, at this point in the assembly, the symbols have not had their values assigned to them.

With NEWPAGE - Clean:

```
NPAGEX2
                                                 IDENT     NPAGEX2
                                    3     %B     EQUALS    3
                                                 BASEREG   %B
             0                              TBL   <110>
           110                              TBLEND *
                                                 NEWPAGE
                                  110       TBLLEN EQUALS    TBLEND-TBL
           110   001000                             EXIT
                                                    END
```

By adding a NEWPAGE pseudo before TBLLEN, we force the assembler to go back and assign values to the symbols that precede. Now the assembler knows the values of TBLEND and TBL, and can successfully calculate the value for TBLLEN. Notice, from the location counter values, that NEWPAGE does NOT insert any slack parcels - it simply causes the assembler to perform calculations that would otherwise be postponed.

Review Questions for Section 4

1) List the pseudoinstructions that appear in APML, but not in CAL.

2) Why is the concept of the "page" needed in APML?

3) How are the operand registers defined as SCRATCH used by the APML assembler?

Section 5: – Macros and $APTEXT

Module Objectives:

With the aid of all furnished reference materials, upon completion of this module, the learner should be able to:

1. Interpret a Macro defined in $APTEXT
2. Use the REGDEFS, OVERLAY, LISTOP, and exit stack manipulating macros in a program
3. Interpret and code data structures using the FIELD macro
4. Use the PUT, GET, RPUT, RGET, ADDRESS and STORE macros in a program to manipulate data structures defined with the FIELD macro
5. Locate, in $APTEXT, data structures defined with the FIELD macro

Introductory Reading Assignment:

SM-0036 Section 6 pp. 6-45 – 6-63

... noting the discussion of MICROs, DECMIC, IFA and IFC in particular.

SM-0046 section describing $APTEXT macros...

...skim to get a feel for what's there, and how it's laid out.

$APTEXT...

... is the IOS system text. It includes macros, field and table definitions, and constant values. In this self-study, we will concern ourselves with studying certain important macros and field definitions. A more full presentation of the contents of $APTEXT is a part of the IOS Internals course.

Macros...

...used most frequently in the IOS code will be found in $APTEXT. But many overlays use special-purpose macros, not needed by other overlays. These macros will be defined in the overlay that uses them. The bottom line is: most BUT NOT ALL macros are defined here in $APTEXT.

The exact contents of $APTEXT changes with new versions of the IOS Software, but its overall layout stays the same. Please follow along in your site's $APTEXT listing as we locate its major sections.

The first page of the $APTEXT assembly listing...

...looks like this (again, find the page in your own listing),

```
 1  1$ A P T E X T - SYSTEM TEXT FOR COS I/O SUBSYSTEM                IOP  APML 2.0 (04/08/86) 04/09/86 12:34:14  Page 1
 2                                                                                                             (1)
 3
 4                                        *       Copyright Cray Research, Inc., 1979 thru 1983                  AT.3
 5

 7                                        *       List options for $APTEXT (all tables are listed by default):   AT.5
 8                                        *                                                                      AT.6
 9                                        *       $ALL      All of APTEXT                                        AT.7
10                                        *       MACROS    All macros                                           AT.8
11                                        *       AT        All of APTEXT (except the macros)                    AT.9
12                                        *       KSR       Kernel Service requests                             AT.10
13                                        *       KDEF      Kernel data definitions                             AT.11
14                                        *       DD49      DD49 data definitions                               AT.12
15                                        *       EXP       Expander data definitions                           AT.13
16                                        *       SFE       Station/Front End data definitions                  AT.14
17                                        *       FILE      File management definitions                         AT.15
18                                        *       UCHN      User channel definitions                            AT.16
19                                        *       BMX       BMX data definitions                                AT.17
```

This page shows symbols that can be used in a LIST parameter to specify that only specified tables in $APTEXT are to be produced.

The page or two following will be constants, including such things as conversion values, etc.

We then get into macros. Notice that the subtitle line at the top of each page in this section says MACROS/Macros:

```
$ A P T E X T - SYSTEM TEXT FOR COS I/O SUBSYSTEM          IOP  APML 2.0 (04/08/86) 04/09/86 12:34:14  Page 40
MACROS/ Macros                                                                                      (40)
```

This section contains many of the macros most commonly encountered in IOS system code. **Stop now to page through your listing and clip pages with other headings of the form "MACROS xxxxx".**

One heading says "**Macros Used Only By Other Macros.**" These will not be encountered anywhere but within $APTEXT (or in overlays that define their own macros). Another says "**KSR/ KERNEL Service Function Calls.**" These macros set up parameter passing and branching for commonly-used routines in the IOS KERNEL that overlays may execute.

Following the macros are constant, field and table definitions for the various software subsystems, and for the KERNEL. These consist primarily of EQUALS and SET pseudos, NEXT and REGISTER macros, and FIELD macros. **We will examine this section of $APTEXT when we discuss the FIELD macro.**

Descriptions Of Some Important Macros:

On the following pages, you will find discussions of several of the most common or useful macros used in the IOS software.

REGDEFS...

...automatically assigns various operand registers, for use by the system, in overlays. It also assigns registers for the overlay's use.

Most macros in $APTEXT include a comment block that describes the function of the macro, and the parameters it requires, as well as the result it produces. Following is the REGDEFS macro as it appears in the $APTEXT listing:

```
*************************************************************************
*                                                                       *
*        REGDEFS   Define overlay registers                             *
*                                                                       *
* start  REGDEFS   global,pars,local,temp                               *
*                                                                       *
*        start     Optional; specifies starting register number.       *
*                  Default is %GBLREG.                                  *
*        global    List of global registers.                           *
*        pars      List of parameter registers                         *
*        local     List of registers used locally                      *
*        temp      List of temporary registers                         *
*                                                                       *
*************************************************************************

                  MACRO
<prototype>       START   REGDEFS  GLOBAL,PARS,LOCAL,TEMP
                          LOCAL    $$S
<definition>      $$S     SET      %GBLREG
<definition>              IFC      |_START_|,NE,,1
<definition>      $$S     SET      START
<definition>      $$S     REGISTER (GLOBAL)
<definition>              IFE      $REGORG,GT,$$S+%GBLNUM,1
<definition>              ERROR    .Too many global registers defined
<definition>      $$S     SET      $$S+%GBLNUM
<definition>              SCRATCH  %S1,%S2,%S3,%S4,%S5
<definition>      $$S     REGISTER (%S1,%S2,%S3,%S4,%S5)
<definition>        .     REGISTER (%T1,%T2,%T3,%T4,%T5,%T6)
<definition>              REGISTER (%W1,%W2,%W3,%W4,%W5)
<definition>      %P      EQUALS   $REGORG
<definition>              REGISTER (PARS)
<definition>      %NP     EQUALS   $REGORG-%P
<definition>              REGISTER (LOCAL)
<definition>              REGISTER (TEMP)
                  REGDEFS ENDM
```

See also the REGISTER macro, following, which is used by REGDEFS to assign symbols to operand register numbers.

In the discussion of the $IF macro, there is a sample assembly listing, with macros expanded, showing the code generated by REGDEFS.

The REGDEFS macro...

...is used in all overlays to automatically assign operand registers needed by that overlay.

REGDEFS always assigns scratch registers for APML's use to symbols %S1-%S5, which are equated to 410-414. It equates symbols %T1-%T6 to 415-422 for use as temporary registers by macros, and %W1-%W5 to 423-427 for work registers used by KERNEL service requests, or by overlays.

The four parameters in the REGDEFS macro allow the programmer to specify:

a) up to 8 **global registers**, which will be available to all overlays within an activity (see the IOS class for further details on activities), which will be equated to 400-407.

b) **Parameter Registers**, which will contain values passed from a previous overlay. Parameter registers, by convention, are assumed to contain an initial value when an overlay begins execution. Symbols specified in REGDEFS as parameter registers are equated to registers starting with 430.

c) **Local Registers** are registers to be used by the overlay, but whose initial contents is, by convention, undefined. Symbols specified in REGDEFS as local registers are equated to operand registers starting wherever the parameter registers left off.

d) **Temp Registers** are not used in current IOS software. Symbols specified in REGDEFS as Temp registers will be equated to registers starting wherever the local registers left off.

Because the REGDEFS macro assigns operand registers, for the overlay's use, according to established conventions, the programmer is assured that operand register usage by the overlay will not conflict with usage by the KERNEL or various utility routines.

REGISTER Macro

This macro is used in the listing above for the REGDEFS macro, among many other places. Its function is to assign successive operand register numbers to symbols. After one REGISTER macro has assigned values to the specified symbols (by generating EQUATEs), it leaves the value of a symbol SET to the next available value. The next REGISTER macro in the sequence of REGISTER macros then picks this value up, and EQUATES it to the first symbol in its list.

Location	Result	Operand
origin	REGISTER	(sym1,sym2,...)

origin	Starting operand register number in octal.
sym1	List of symbols to be assigned to an operand register.

Generates the following:

```
sym1    EQUALS  origin
sym2    EQUALS  origin+1
sym3    EQUALS  origin+2
sym4    EQUALS  origin+3
```

These equates appear in the listing following the REGISTER macro if the list option REGISTER is specified.

Example: see the $IF sample listing.

REGISTER Macro Listing

```
                        MACRO                                                   AT.1839
<prototype>     REGORG  REGISTER  REGLIST                                       AT.1840
                        LOCAL     $MSIZE                                        AT.1841
<definition>    $REG    IFC       |_REGORG_|,NE,''                             AT.1842
<definition>    $REGORG SET       REGORG                                        AT.1843
<definition>    $REG    ELSE                                                    AT.1844
<definition>            IFA       #DEF,$REGORG,1                                AT.1845
<definition>            ERROR     .Register origin must be specified            AT.1846
<definition>    $REG    ENDIF                                                   AT.1847
<definition>    $REG    ECHO      REG=(REGLIST)                                 AT.1848
<definition>            IFC       |_REG_|,NE,,6                                 AT.1849
<definition>            IFC       |_REG_|,NE,'*',4                              AT.1850
<definition>    $MSIZE  MICRO     'REG          ',D'8                           AT.1851
<definition>    REGS    LIST      MAC                                           AT.1852
<definition>    "$MSIZE" EQUALS   $REGORG                                       AT.1853
<definition>    REGS    LIST      *                                             AT.1854
<definition>    $REGORG SET       $REGORG+1                                     AT.1855
<definition>    $REG    ENDDUP                                                  AT.1856
                REGISTER ENDM                                                   AT.1857
```

NEXT

Assign successive values symbols. This allows such things as lists of error codes, function codes, result codes, etc. to be conveniently maintained. Adding a new item does not require programmer knowledge of the last assigned value. By inserting a new NEXT macro at the end of the current list, the next value in the sequence will automatically be assigned.

Location	Result	Operand
sym	NEXT	value

sym	optional symbol name
value	optional initial value

	Value Present	Value Blank
Symbol Present	symbol=value $next=value+1	symbol=$next $next=$next+1
Symbol Blank	$next=value+1	$next=$next+1

NEXT macro Example: PUNT codes from $APTEXT

These codes appear on the IOP-Ø KERNEL console in the event of a IOP halt. The displayed code identifies the error condition detected by the IOS software, and gives the analyst an indication of where the halt occurred. When the code is displayed on the IOP console in the halt message it is shown as a number. The numeric value assigned to each symbol can be found in the cross reference of $APTEXT. The symbols themselves are used within the source listings, at the locations where the IOS software detects error conditions.

```
                        ************************************************************************      AT.4345
                        *                                                                      *      AT.4346
                        *       Kernel halt error codes                                        *      AT.4347
                        *                                                                      *      AT.4348
                        ************************************************************************      AT.4349
                                                                                                      AT.4350
0    <macro>                        NEXT     -1                                                     AT.4351
0    <macro>                PT$HUH  NEXT     .No error code specified on $PUNTIF macro              AT.4352
0    <macro>                PT$MEMER NEXT    .Local memory error                                    AT.4353
0    <macro>                PT$START NEXT    .MOS error on deadstart                                AT.4354
0    <macro>                PT$MOS  NEXT     .MOS error                                             AT.4355
0    <macro>                PT$HISP NEXT     .High-speed channel error                              AT.4356
0    <macro>                PT$CRAY NEXT     .Invalid message received from Cray-1                  AT.4357
0    <macro>                PT$DISK NEXT     .Invalid parameter in disk request                     AT.4358
0    <macro>                PT$EXEC0 NEXT    .Program was executing at location 0                   AT.4359
0    <macro>                PT$CLOB0 NEXT    .Local memory location zero overwritten                AT.4360
0    <macro>                PT$ACCUM NEXT    .Undefined message received on accumulator channel     AT.4361
0    <macro>                PT$NOLAY NEXT    .Overlay does not exist                                AT.4362
0    <macro>                PT$STACK NEXT    .Stack overflow or underflow                           AT.4363
0    <macro>                PT$LOCAL NEXT    .Local memory buffer not available                     AT.4364
0    <macro>                PT$NOMOS NEXT    .MOS disk buffer not available                         AT.4365
0    <macro>                PT$MAP  NEXT     .Invalid buffer release call                           AT.4366
0    <macro>                PT$CONFG NEXT    .Buffer memory incorrectly configured                  AT.4367
0    <macro>                PT$MSGCH NEXT    .IOP message channels incorrectly configured           AT.4368
0    <macro>                PT$SMDSZ NEXT    .Software stack exceeds allowable size                 AT.4369
0    <macro>                PT$BADAD NEXT    .Bad parameter address.                                AT.4370
0    <macro>                PT$BADPR NEXT    .Bad parameter                                         AT.4371
0    <macro>                PT$KILL  NEXT    .Stop request received from CPU                        AT.4372
0    <macro>                PT$LOSP  NEXT    .Low-speed channel error                               AT.4373
0    <macro>                PT$BLOCK NEXT    .Block number error                                    AT.4374
                                                                                                      AT.4375
0    <macro>                PT$BMCHE NEXT O'40 .Block Mux interrupt processor error                 AT.4376
0    <macro>                PT$BMDE  NEXT    .Block Mux Demon error                                 AT.4377
0    <macro>                PT$BMSIO NEXT    .Block Mux Start I/O error                             AT.4378
0    <macro>                PT$BMACT NEXT    .Illegal tape activity during CRAY startup             AT.4379
0    <macro>                PT$BMDAT NEXT    .Bad data in CPU tape configuration map                AT.4380
```

NEXT macro Example: PUNT codes from $APTEXT (continued)

```
0    <macro>          PT$DD49   NEXT O'50 .DD49 disk driver code error                    AT.4382
0    <macro>          PT$NODBG  NEXT                .The debugger is not loaded          AT.4383
0    <macro>          PT$BMRQ   NEXT                .Bad buffer memory allocation request AT.4384
0    <macro>          PT$HSLA   NEXT                .Bad lm address on hi speed I/O call AT.4385
0    <macro>          PT$IOLEN  NEXT                .Invalid I/O length specified        AT.4386
0    <macro>          PT$NOHSP  NEXT                .No HISP channel for request         AT.4387
0    <macro>          PT$NACT   NEXT                .Illegal activity activation requested AT.4388
0    <macro>          PT$BMDAL  NEXT                .Buffer memory DAL queue exhausted   AT.4389
0    <macro>          PT$BDEMC  NEXT                .Illegal Demon call                  AT.4390
0    <macro>          PT$BKSR   NEXT                .Undefined kernel service request    AT.4391
0    <macro>          PT$IKSR   NEXT                .Illegal kernel service request      AT.4392
0    <macro>          PT$BKSRP  NEXT                .Bad kernel service request parameter AT.4393
0    <macro>          PT$BDEV   NEXT                .Requested device not configured     AT.4394
0    <macro>          PT$BIOP   NEXT                .Illegal IOP requested               AT.4395
0    <macro>          PT$QFULL  NEXT                .Requested queue full                AT.4396

0    <macro>          PT$IOADD  NEXT                .Illegal I/O address specified       AT.4397
0    <macro>          PT$SMER   NEXT                .SMOD error                          AT.4398
0    <macro>          PT$LMSP   NEXT                .Local memory space exhausted        AT.4399
0    <macro>          PT$BOVL   NEXT                .Illegal overlay load requested      AT.4400
0    <macro>          PT$LMCER  NEXT                .Corrupted local memory chain        AT.4401
0    <macro>          PT$LMREL  NEXT                .Bad local memory release            AT.4402
0    <macro>          PT$BDIOP  NEXT                .Bad I/O parameter                   AT.4403
0    <macro>          PT$UXINT  NEXT                .Unexpected interrupt received       AT.4404
0    <macro>          PT$DISKE  NEXT                .Disk error                          AT.4405
0    <macro>          PT$AMAP   NEXT                .AMAP not available to system        AT.4406
0    <macro>          PT$OMAX   NEXT                .Illegal overlay number read         AT.4407
0    <macro>          PT$INIT   NEXT                .IOP initialization error            AT.4408
0    <macro>          PT$MOSC   NEXT                .MOS configuration error             AT.4409
0    <macro>          PT$OVLSZ  NEXT                .Overlay too large for loading       AT.4410
0    <macro>          PT$EOF    NEXT                .Premature tape EOF encountered      AT.4411
0    <macro>          PT$XSOV   NEXT                .Exit stack fault                    AT.4412
0    <macro>          PT$DEV    NEXT                .Illegal device type          P10610BA.4
0    <macro>          PT$LAST   NEXT      .ALL PUNT CODES PRECEDE THIS LINE!!!           AT.4413
```

EXIT STACK MACROS

These macros are required to avoid timing problems associated with manipulating the exit stack. They include necessary delays to ensure that data arrives before it is accessed.

EGET destination

Reads the E pointer and stores it in memory or a register

EPUT source

Stores a memory location or register in the E pointer

EINCR

Increments the E pointer value

EDECR

Decrements the E pointer value

EXSGET destination

Stores the contents of the Exit Stack Entry pointed to by the E pointer into a memory location or a register

EXSPUT source

Loads the contents of a memory location or a register into the Exit Stack Entry pointed to by the E pointer

AN ANNOYING QUIRK:

When a parameter involving parentheses is used, such as contents of a memory location or the operand register indicated in B, the macro processor assumes the parentheses are intended to group several operands together as a single parameter and strips them off, leaving an expression that will be assembled as contents of operand register or contents of B. **To avoid this problem, you must double the parentheses around such operands.**

CONDITIONAL BLOCK MACROS

$IF $ELSEIF $ELSE $ENDIF; $UNTIL and $ENDTIL

These macros allow structured techniques to be applied to APML code. These are among the most frequently used of all macros. Inspection of virtually any page of the KERNEL or overlays will reveal several usages of these macros. $IF structures may be nested up to 1Ø levels, and $UNTIL structures may be nested up to 1Ø levels.

FORMAT:

Location	Result	Operand
L	$IF	(cond1), and/or, (cond2)
	$ELSEIF	(cond1), and/or, (cond2)
	$ELSE	
	$ENDIF	

L	optional statement label
cond1	any valid APML conditional expression
and/or	logical operator 'AND' or 'OR'
	If blank cond2 is ignored

$IF must be the first conditional block macro of a sequence. It may have one or more $ELSIFs following, and may have one $ELSE following. It must have a corresponding $ENDIF.

$ELSEIF allows the construction of CASE-type structures. It occurs between $IF and $ENDIF and must precede any $ELSE.

$ELSE is optional and delimits last block before $ENDIF - the block that will be executed if the specified condition is false.

Sample Listings Demonstrating Macros

On the following pages, you will find listings of the $IF, $ELSE, $ELSEIF and $ENDIF macros, as they appear in $APTEXT, and a small sample program, SHOWIF.

Two separate SHOWIF listings are given: one assembled with **listing option LEVELS**, and one assembled with **listing option MAC**. The first listing, which matches the appearance you will probably want for a typical APML listing, is very short - only a 15 or so lines. The second listing, with macros expanded, spans six pages! This second listing should help to give you a feel for the code generated by the $IF-$ELSE-$ENDIF macros, and also the REGDEFS macro, which in fact is responsible for generating most of the listing.

Shading in these listings is used in two ways. First, it is used to indicate **levels of macro nesting** - macros which invoke other macros. For instance, the REGDEFS macro invokes the REGISTER macro. Second, shading is used to indicate the **nesting of $IF structures** by the programmer. This is most clearly shown in the short listing of SHOWIF.

The **large arrows** at the left margins of the second listing indicate the actual input file statements. Any statement that has no arrow was generated by a macro.

Compare the listing of the REGDEFS macro to the code it generates in the SHOWIF expanded listing. Pay particular attention to how micros are used to generate unique symbol names in nested structures.Do the same with the conditional block macros. By examining these listings, you should get a better understanding of just what these macros do, and how they function.

$IF Macro Listing

```
                        MACRO                                                   AT.161
<prototype>             $IF     COND1,ANDOR,COND2                               AT.162
                        LOCAL   YES1,YES2                                       AT.163
<definition>            IFA     #DEF,/"$QUAL"/%%AA,1                            AT.164
<definition>    %%AA    SET     -1              .Initialize IF level            AT.165
<definition>    %%AA    SET     %%AA+1          .Increment IF level             AT.166
<definition>            ERRIF   %%AA,GT,D'9     .Too many IF levels             AT.167
<definition>    %%ZZ    DECMIC  %%AA                                            AT.168
<definition>    LEVEL   LIST    MAC                                             AT.169
<definition>                                                    .>>>>> "%%ZZ"   AT.170
<definition>    LEVEL   LIST    *                                               AT.171
<definition>            IFA     #DEF,/"$QUAL"/%%B"%%ZZ",2                       AT.172
<definition>    %%B"%%ZZ" SET   -1              .Init this IF level counter     AT.173
<definition>    %%C"%%ZZ" SET   -1              .Init this ENDIF level counter  AT.174
<definition>    %%B"%%ZZ" SET   %%B"%%ZZ"+1     .Incr IF level counter          AT.175
<definition>    %%C"%%ZZ" SET   %%C"%%ZZ"+1     .Incr ENDIF level counter       AT.176
<definition>            ERRIF   %%B"%%ZZ",GT,D'9999 .IF macro overflow          AT.177
<definition>    %%YY    DECMIC  %%B"%%ZZ",4                                     AT.178
<definition>    %%XX    DECMIC  %%C"%%ZZ",4                                     AT.179
<definition>    $LASTIF MICRO   '%%"%%ZZ"L"%%XX"'                               AT.180
<definition>    $NEXTIF MICRO   '%%"%%ZZ"N"%%YY"'                               AT.181
<definition>    %%D"%%ZZ" SET   0               .Clear ELSE flag for this level AT.182
<definition>            IFC     'ANDOR',NE,,3                                   AT.183
<definition>            IFC     'ANDOR',NE,'AND',2                              AT.184
<definition>            IFC     'ANDOR',NE,'OR',1                               AT.185
<definition>            ERROR   .Illegal separator                              AT.186
<definition>            P = YES1, COND1                                         AT.187
<definition>            IFC     'ANDOR',EQ,'OR',1                               AT.188
<definition>            P = YES1, COND2                                         AT.189
<definition>            P = /"$QUAL"/"$NEXTIF"                                  AT.190
<definition>    YES1    *                                                       AT.191
<definition>            IFC     'ANDOR',EQ,'AND',3                              AT.192
<definition>            P = YES2, COND2                                         AT.193
<definition>            P = /"$QUAL"/"$NEXTIF"                                  AT.194
<definition>    YES2    *                                                       AT.195
                $IF     ENDM                                                    AT.196
```

$ELSE Macro Listing

```
                MACRO                                                          AT.51
<prototype>     $ELSE                                                          AT.52
<definition>    P = /"$QUAL"/"$LASTIF"                                         AT.53
<definition>  "$NEXTIF" *                                                      AT.54
<definition>    ERRIF   /"$QUAL"/%%AA,LT,0  .No IF encountered yet             AT.55
<definition>    ERRIF   /"$QUAL"/%%D"%%ZZ",EQ,1 .ELSE already encountered      AT.56
<definition>  %%D"%%ZZ" SET   1             .Set ELSE flag for this level     AT.57
              $ELSE     ENDM                                                   AT.58
```

$ENDIF Macro Listing

```
                MACRO                                                          AT.86
<prototype>     $ENDIF                                                         AT.87
<definition>    IFE     /"$QUAL"/%%D"%%ZZ",EQ,0,1                             AT.88
<definition>  "$NEXTIF" *                       .Define $NEXTIF if necessary   AT.89
<definition>  "$LASTIF" *                                                      AT.90
<definition>  %%LL      DECMIC  %%AA                                           AT.91
<definition>  LEVEL     LIST    MAC                                            AT.92
<definition>                                                .<<<<< "%%LL"      AT.93
<definition>  LEVEL     LIST    *                                              AT.94
<definition>            ERRIF   /"$QUAL"/%%AA,LT,0  .No IF encountered yet     AT.95
<definition>  %%AA      SET     %%AA-1              .Decrement IF level        AT.96
<definition>            IFE     %%AA,GE,0,1                                    AT.97
<definition>  %%ZZ      DECMIC  %%AA                                           AT.98
<definition>  %%YY      DECMIC  %%B"%%ZZ",4                                    AT.99
<definition>  %%XX      DECMIC  %%C"%%ZZ",4                                   AT.100
<definition>  $LASTIF   MICRO   '%%"%%ZZ"L"%%XX"'                             AT.101
<definition>  $NEXTIF   MICRO   '%%"%%ZZ"N"%%YY"'                             AT.102
              $ENDIF    ENDM                                                 AT.103
```

$ELSEIF Macro Listing

```
                        MACRO                                                   AT.60
<prototype>             $ELSEIF   COND1,ANDOR,COND2                             AT.61
                        LOCAL     YES1,YES2                                     AT.62
<definition>            P = /"$QUAL"/"$LASTIF"                                  AT.63
<definition>      "$NEXTIF" *                                                   AT.64
<definition>            ERRIF     /"$QUAL"/%%AA,LT,0  .No IF encountered yet    AT.65
<definition>      %%B"%%ZZ" SET     %%B"%%ZZ"+1         .Incr this IF level counter AT.66
<definition>            ERRIF     %%B"%%ZZ",GT,D'9999 .IF MACRO overflow        AT.67
<definition>      %%YY    DECMIC    %%B"%%ZZ",4                                   AT.68
<definition>      $NEXTIF MICRO     '%%"%%ZZ"N"%%YY"'                             AT.69
<definition>            ERRIF     /"$QUAL"/%%D"%%ZZ",EQ,1 .ELSE already encountered AT.70
<definition>            IFC       'ANDOR',NE,,3                                 AT.71
<definition>            IFC       'ANDOR',NE,'AND',2                            AT.72
<definition>            IFC       'ANDOR',NE,'OR',1                             AT.73
<definition>            ERROR     .Illegal separator                            AT.74
<definition>            P = YES1, COND1                                         AT.75
<definition>            IFC       'ANDOR',EQ,'OR',1                             AT.76
<definition>            P = YES1, COND2                                         AT.77
<definition>            P = /"$QUAL"/"$NEXTIF"                                  AT.78
<definition>      YES1  *                                                       AT.79
<definition>            IFC       'ANDOR',EQ,'AND',3                            AT.80
<definition>            P = YES2, COND2                                         AT.81
<definition>            P = /"$QUAL"/"$NEXTIF"                                  AT.82
<definition>      YES2  *                                                       AT.83
                  $ELSEIF ENDM                                                  AT.84
```

Annotated IF-ELSE-ENDIF Nested Structure

```
WIF                                          IOP  APML 2.1(03/19/87)  04/01/87 16:56:08  Page 1
                                                                                        (1)

                                        IDENT   SHOWIF
 0    <macro>                           REGDEFS   ,,(THIS,THAT,THOSE)
 0    <macro>              START        $IF (R!THIS=R!THAT)
                                                                          .>>>>> 0   $IF     1
 4    <macro>                             $IF (R!THIS<100)
                                                                          .>>>>> 1   $IF     1
10   010005  024432                          R!THOSE=5
12    <macro>                              $ELSE
13   010006  024432                          R!THOSE=6
15    <macro>                             $ENDIF
                                                                          .<<<<< 1   $ENDIF  1
15    <macro>                          $ELSE
16   010001  024432                       R!THOSE=1
20    <macro>                          $ENDIF
                                                                          .<<<<< 0   $ENDIF  1
20   001000                            EXIT
                                       END
```

4/13/88

- Notice that the $IF and $ELSEIF macros display a comment of the form ".>>>>> n" and ".<<<<< n". These are very helpful in locating the endpoints of conditional block structures. You must include the list option "LEVEL" in your APML control statement for these comment lines to be generated in the listing. The $ELSE macro generates no such comment, nor does the $ELSEIF.
- Logically, when a $ELSE macro is encountered, it will be paired with first $IF macro that precedes it, that has not already been paired with a $ELSE. We will see how this is accomplished in a few pages, when we examine the code generated by these macros.
- Indentation is for readability only.

On the following pages, this same example is shown with the list option MAC, which shows the expanded macros.

Annotated Cross-Reference For IF-ELSE-ENDIF Example

Next Available Operand Register

Number of global registers - always 10

First global Operand Register - always 400

Number of parameter registers specified in second parameter of REGDEFS - none in this program

Starting Operand Register to assign to first parameter register always 430

%S1-%S5 are the Scratch registers assigned by REGDEFS

%T1-%T6 are the Temp registers assigned by REGDEFS

%W1-%W5 are the Work registers assigned by REGDEFS

Notice that THIS, THAT and THOSE are asigned to operand registers 430, 431, and 432 by REGDEFS

```
433   $REGORG                 1: 2 D    1: 2 F    1: 2
 10   %GBLNUM   $APTEXT       1: 2 F    1: 2
400   %GBLREG   $APTEXT       1: 2
  0   %NP                     1: 2 D
430   %P                      1: 2 D    1: 2
410   %S1                     1: 2      1: 2 D
411   %S2                     1: 2      1: 2 D
412   %S3                     1: 2      1: 2 D
413   %S4                     1: 2      1: 2 D
414   %S5                     1: 2      1: 2 D
415   %T1                     1: 2 D
416   %T2                     1: 2 D
417   %T3                     1: 2 D
420   %T4                     1: 2 D
421   %T5                     1: 2 D
422   %T6                     1: 2 D
423   %W1                     1: 2 D
424   %W2                     1: 2 D
425   %W3                     1: 2 D
426   %W4                     1: 2 D
427   %W5                     1: 2 D
431   THAT                    1: 2 D    1: 4
430   THIS                    1: 2 D    1: 4      1: 6
432   THOSE                   1: 2 D    1: 7      1: 9      1:13
```

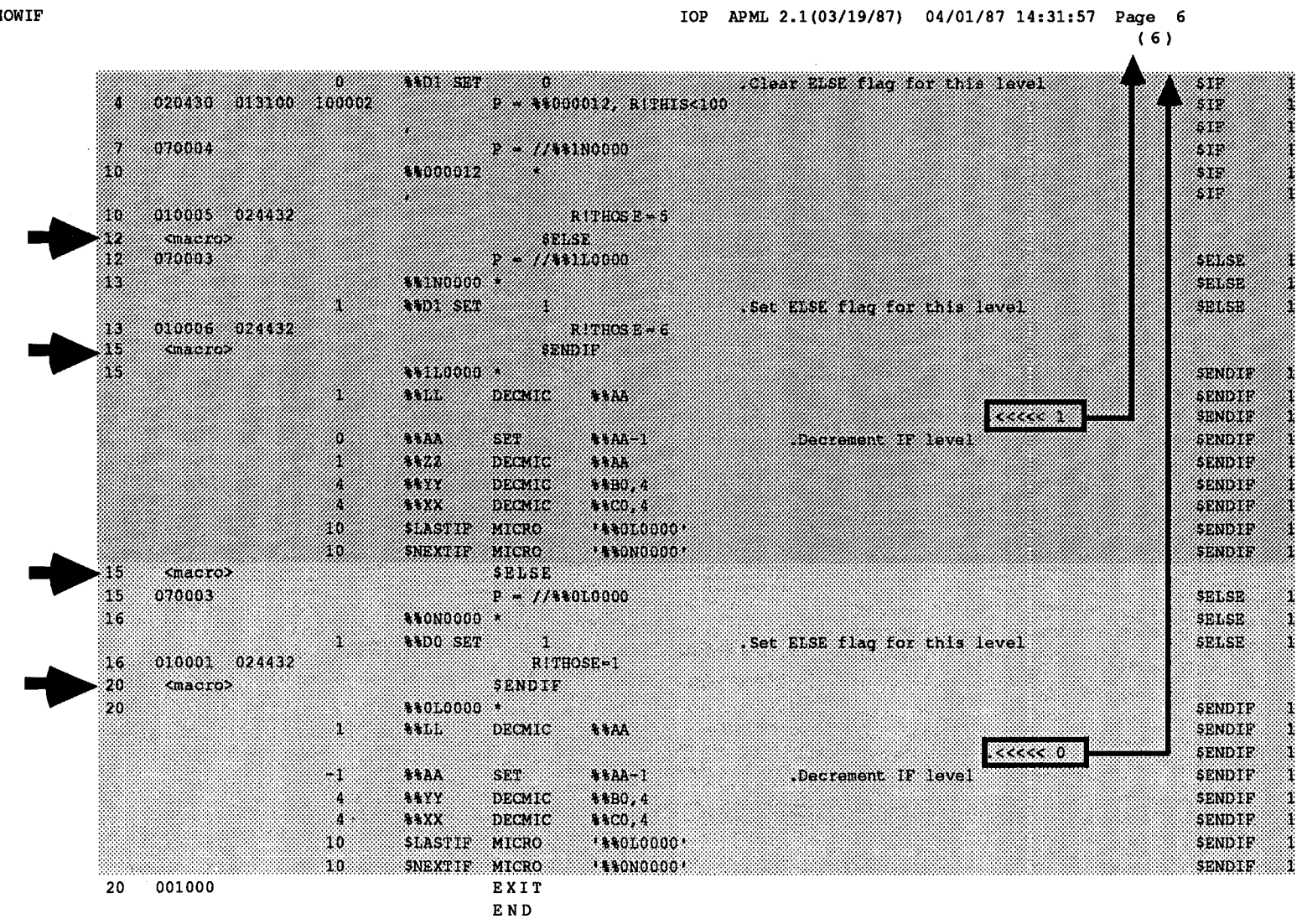
```
1SHOWIF                                    IOP  APML 2.1(03/19/87)  04/01/87 14:31:57  Page  1
                                                                                        ( 1 )

                                    IDENT   SHOWIF
0    <macro>                        REGDEFS  ,,(THIS,THAT,THOSE)
                 400    %%0000000      SET      %GBLREG                                  REGDEFS  1
                                    ,                                                    REGDEFS  1
0    <macro>                        %%0000000    REGISTER  ()                            REGDEFS  1
                 400    $REGORG  SET      %%0000000                                      REGISTER 2
                                    ,                                                    REGISTER 2
                        $REG     ECHO     REG=()                                        REGISTER 2
     <definition>               IFC      |_REG_|,NE,,6                                  REGISTER 2
     <definition>               IFC      |_REG_|,NE,'*',4                               REGISTER 2
     <definition>       %%0000001  MICRO    'REG        ',D'8                           REGISTER 2
                                    ,                                                    REGISTER 2
     <definition>       REGS     LIST     MAC                                           REGISTER 2
     <definition>       "%%0000001" EQUALS   $REGORG                                    REGISTER 2
                                    ,                                                    REGISTER 2
     <definition>       REGS     LIST     *                                             REGISTER 2
     <definition>       $REGORG  SET      $REGORG+1                                     REGISTER 2
                        $REG     ENDDUP                                                REGISTER 2
                 410    %%0000000      SET      %%0000000+%GBLNUM                       REGDEFS  1
                                    ,                                                    REGDEFS  1
                        SCRATCH  %S1,%S2,%S3,%S4,%S5                                    REGDEFS  1
0    <macro>            %%0000000    REGISTER  (%S1,%S2,%S3,%S4,%S5)                    REGDEFS  1
                                    ,                                                    REGDEFS  1
                 410    $REGORG  SET      %%0000000                                     REGISTER 2
                                    ,                                                    REGISTER 2
                        $REG     ECHO     REG=(%S1,%S2,%S3,%S4,%S5)                     REGISTER 2
                                    ,                                                    REGISTER 2
     <definition>               IFC      |_REG_|,NE,,6                                  REGISTER 2
     <definition>               IFC      |_REG_|,NE,'*',4                               REGISTER 2
     <definition>       %%0000002  MICRO    'REG        ',D'8                           REGISTER 2
                                    ,                                                    REGISTER 2
     <definition>       REGS     LIST     MAC                                           REGISTER 2
     <definition>       "%%0000002" EQUALS   $REGORG                                    REGISTER 2
                                    ,                                                    REGISTER 2
     <definition>       REGS     LIST     *                                             REGISTER 2
     <definition>       $REGORG  SET      $REGORG+1                                     REGISTER 2
                        $REG     ENDDUP                                                REGISTER 2
                  10    %%0000002  MICRO    '%S1        ',D'8                           $REG     3
                                    ,                                                    $REG     3
                 410    %S1      EQUALS   $REGORG                                       $REG     3
                 411    $REGORG  SET      $REGORG+1                                     $REG     3
                  10    %%0000002  MICRO    '%S2        ',D'8                           $REG     3
```

```
1SHOWIF                                              IOP  APML 2.1(03/19/87)  04/01/87 14:31:57  Page  2
                                                                                                   ( 2 )
                          ,                                                              $REG      3
                   411    %S2       EQUALS   $REGORG                                     $REG      3
                   412    $REGORG   SET      $REGORG+1                                   $REG      3
                    10    %%000002   MICRO     '%S3          ',D'8                       $REG      3
                          ,                                                              $REG      3
                   412    %S3       EQUALS   $REGORG                                     $REG      3
                   413    $REGORG   SET      $REGORG+1                                   $REG      3
                    10    %%000002   MICRO     '%S4          ',D'8                       $REG      3
                          ,                                                              $REG      3
                   413    %S4       EQUALS   $REGORG                                     $REG      3
                   414    $REGORG   SET      $REGORG+1                                   $REG      3
                    10    %%000002   MICRO     '%S5          ',D'8                       $REG      3
                          ,                                                              $REG      3
                   414    %S5       EQUALS   $REGORG                                     $REG      3
                   415    $REGORG   SET      $REGORG+1                                   $REG      3
0  <macro>                          REGISTER (%T1,%T2,%T3,%T4,%T5,%T6)                   REGDEFS   1
                          $REG      ECHO     REG=(%T1,%T2,%T3,%T4,%T5,%T6)               REGISTER  2
                          ,                                                              REGISTER  2
   <definition>                     IFC      |_REG_|,NE,,6                               REGISTER  2
   <definition>                     IFC      |_REG_|,NE,'*',4                            REGISTER  2
   <definition>           %%000003   MICRO     'REG          ',D'8                       REGISTER  2
                          ,                                                              REGISTER  2
   <definition>           REGS      LIST     MAC                                         REGISTER  2
   <definition>           "%%000003" EQUALS    $REGORG                                   REGISTER  2
                          ,                                                              REGISTER  2
   <definition>           REGS      LIST     *                                           REGISTER  2
   <definition>           $REGORG   SET      $REGORG+1                                   REGISTER  2
                          $REG      ENDDUP                                               REGISTER  2
                    10    %%000003   MICRO     '%T1          ',D'8                       $REG      3
                          ,                                                              $REG      3
                   415    %T1       EQUALS   $REGORG                                     $REG      3
                   416    $REGORG   SET      $REGORG+1                                   $REG      3
                    10    %%000003   MICRO     '%T2          ',D'8                       $REG      3
                          ,                                                              $REG      3
                   416    %T2       EQUALS   $REGORG                                     $REG      3
                   417    $REGORG   SET      $REGORG+1                                   $REG      3
                    10    %%000003   MICRO     '%T3          ',D'8                       $REG      3
                          ,                                                              $REG      3
                   417    %T3       EQUALS   $REGORG                                     $REG      3
                   420    $REGORG   SET      $REGORG+1                                   $REG      3
                    10    %%000003   MICRO     '%T4          ',D'8                       $REG      3
                          ,                                                              $REG      3
```

```
1SHOWIF                                    IOP  APML 2.1(03/19/87)  04/01/87 14:31:57  Page  3
                                                                                        ( 3 )

                    420     %T4       EQUALS    $REGORG                                    $REG    3
                    421     $REGORG   SET       $REGORG+1                                  $REG    3
                     10     %%0000003   MICRO     '%T5          ',D'8                      $REG    3
                            ,                                                              $REG    3
                    421     %T5       EQUALS    $REGORG                                    $REG    3
                    422     $REGORG   SET       $REGORG+1                                  $REG    3
                     10     %%0000003   MICRO     '%T6          ',D'8                      $REG    3
                            ,                                                              $REG    3
                    422     %T6       EQUALS    $REGORG                                    $REG    3
                    423     $REGORG   SET       $REGORG+1                                  $REG    3
0   <macro>                           REGISTER  (%W1,%W2,%W3,%W4,%W5)                      REGDEFS 1
                            $REG      ECHO      REG=(%W1,%W2,%W3,%W4,%W5)                  REGISTER 2
                            ,                                                            REGISTER 2
    <definition>                      IFC       |_REG |,NE,,6                           REGISTER 2
    <definition>                      IFC       |_REG |,NE,'*',4                        REGISTER 2
    <definition>            %%0000004   MICRO     'REG          ',D'8                    REGISTER 2
                            ,                                                            REGISTER 2
    <definition>            REGS      LIST      MAC                                      REGISTER 2
    <definition>            "%%0000004" EQUALS    $REGORG                                REGISTER 2
                            ,                                                            REGISTER 2
    <definition>            REGS      LIST      *                                        REGISTER 2
    <definition>            $REGORG   SET       $REGORG+1                                REGISTER 2
                            $REG      ENDDUP                                           REGISTER 2
                     10     %%0000004   MICRO     '%W1          ',D'8                    $REG    3
                            ,                                                            $REG    3
                    423     %W1       EQUALS    $REGORG                                  $REG    4
                    424     $REGORG   SET       $REGORG+1                                $REG    3
                     10     %%0000004   MICRO     '%W2          ',D'8                    $REG    3
                            ,                                                            $REG    3
                    424     %W2       EQUALS    $REGORG                                  $REG    3
                    425     $REGORG   SET       $REGORG+1                                $REG    3
                     10     %%0000004   MICRO     '%W3          ',D'8                    $REG    3
                            ,                                                            $REG    3
                    425     %W3       EQUALS    $REGORG                                  $REG    3
                    426     $REGORG   SET       $REGORG+1                                $REG    3
                     10     %%0000004   MICRO     '%W4          ',D'8                    $REG    3
                            ,                                                            $REG    3
                    426     %W4       EQUALS    $REGORG                                  $REG    3
                    427     $REGORG   SET       $REGORG+1                                $REG    3
                     10     %%0000004   MICRO     '%W5          ',D'8                    $REG    3
                            ,                                                            $REG    3
                    427     %W5       EQUALS    $REGORG                                  $REG    3
```

```
1SHOWIF                                  IOP  APML 2.1(03/19/87)  04/01/87 14:31:57  Page  4
                                                                                      ( 4 )

                    430     $REGORG  SET     $REGORG+1                                  $REG    3
                    430     %P       EQUALS  $REGORG                                    REGDEFS 1
0   <macro>                          REGISTER ()                                        REGDEFS 1
                            $REG     ECHO    REG=()                                     REGISTER 2
    <definition>                     IFC     |_REG_|,NE,,6                             REGISTER 2
    <definition>                     IFC     |_REG_|,NE,'*',4                          REGISTER 2
    <definition>            %%000005  MICRO    'REG          ',D'8                     REGISTER 2
                            ,                                                          REGISTER 2
    <definition>            REGS     LIST    MAC                                       REGISTER 2
    <definition>            "%%000005" EQUALS  $REGORG                                 REGISTER 2
                            ,                                                          REGISTER 2
    <definition>            REGS     LIST    *                                         REGISTER 2
    <definition>            $REGORG  SET     $REGORG+1                                 REGISTER 2
                            $REG     ENDDUP                                            REGISTER 2
                      0     %NP      EQUALS  $REGORG-%P                                REGDEFS 1
0   <macro>                          REGISTER (THIS,THAT,THOSE)                         REGDEFS 1
                            ,                                                          REGDEFS 1
                            $REG     ECHO    REG=(THIS,THAT,THOSE)                     REGISTER 2
                            ,                                                          REGISTER 2
    <definition>                     IFC     |_REG_|,NE,,6                             REGISTER 2
    <definition>                     IFC     |_REG_|,NE,'*',4                          REGISTER 2
    <definition>            %%000006  MICRO    'REG          ',D'8                     REGISTER 2
                            ,                                                          REGISTER 2
    <definition>            REGS     LIST    MAC                                       REGISTER 2
    <definition>            "%%000006" EQUALS  $REGORG                                 REGISTER 2
                            ,                                                          REGISTER 2
    <definition>            REGS     LIST    *                                         REGISTER 2
    <definition>            $REGORG  SET     $REGORG+1                                 REGISTER 2
                            $REG     ENDDUP                                            REGISTER 2
                     10     %%000006  MICRO    'THIS         ',D'8                     $REG    3
                            ,                                                          $REG    3
                    430     THIS     EQUALS  $REGORG                                   $REG    3
                    431     $REGORG  SET     $REGORG+1                                 $REG    3
                     10     %%000006  MICRO    'THAT         ',D'8                     $REG    3
                            ,                                                          $REG    3
                    431     THAT     EQUALS  $REGORG                                   $REG    3
                    432     $REGORG  SET     $REGORG+1                                 $REG    3
                     10     %%000006  MICRO    'THOSE        ',D'8                     $REG    3
                            ,                                                          $REG    3
                    432     THOSE    EQUALS  $REGORG                                   $REG    3
                    433     $REGORG  SET     $REGORG+1                                 $REG    3
0   <macro>                          REGISTER ()                                        REGDEFS 1
```

```
1SHOWIF                                  IOP  APML 2.1(03/19/87)  04/01/87 14:31:57  Page  5
                                                                                      ( 5 )

                          $REG     ECHO     REG=()                                        REGISTER 2
     <definition>                  IFC      |_REG_|,NE,,6                                 REGISTER 2
     <definition>                  IFC      |_REG_|,NE,'',4                               REGISTER 2
     <definition>         %%000007  MICRO     'REG         ',D'8                         REGISTER 2
                          ,                                                              REGISTER 2
     <definition>         REGS     LIST     MAC                                         REGISTER 2
     <definition>         "%%000007" EQUALS   $REGORG                                    REGISTER 2
                          ,                                                              REGISTER 2
     <definition>         REGS     LIST     *                                           REGISTER 2
     <definition>         $REGORG  SET      $REGORG+1                                   REGISTER 2
                          $REG     ENDDUP                                               REGISTER 2
0    <macro>              START    $IF (R!THIS=R!THAT)
                    -1    %%AA     SET      -1                .Initialize IF level     $IF      1
                     0    %%AA     SET      %%AA+1            .Increment IF level      $IF      1
                     1    %%ZZ     DECMIC   %%AA                                       $IF      1
                                                                  .>>>>> 0               $IF      1
                    -1    %%B0 SET      -1                .Init this IF level counter $IF      1
                    -1    %%C0 SET      -1                .Init this ENDIF level counter $IF   1
                     0    %%B0 SET      %%B0+1         .Incr IF level counter          $IF      1
                     0    %%C0 SET      %%C0+1         .Incr ENDIF level counter       $IF      1
                     4    %%YY     DECMIC   %%B0,4                                     $IF      1
                     4    %%XX     DECMIC   %%C0,4                                     $IF      1
                    10    $LASTIF  MICRO    '%%0L0000'                                 $IF      1
                    10    $NEXTIF  MICRO    '%%0N0000'                                 $IF      1
                     0    %%D0 SET      0                 .Clear ELSE flag for this level $IF   1
0   020430  023431  102002           P = %%000010, R!THIS=R!THAT                       $IF      1
                                                                                       $IF      1
3   070013                         P = //%%0N0000                                      $IF      1
4                        %%000010    *                                                 $IF      1
                                                                                       $IF      1
4    <macro>                        $IF (R!THIS<100)
                     1    %%AA     SET      %%AA+1            .Increment IF level      $IF      1
                     1    %%ZZ     DECMIC   %%AA                                       $IF      1
                                                                  >>>>> 1                $IF      1
                    -1    %%B1 SET      -1                .Init this IF level counter $IF      1
                    -1    %%C1 SET      -1                .Init this ENDIF level counter $IF   1
                     0    %%B1 SET      %%B1+1         .Incr IF level counter          $IF      1
                     0    %%C1 SET      %%C1+1         .Incr ENDIF level counter       $IF      1
                     4    %%YY     DECMIC   %%B1,4                                     $IF      1
                     4    %%XX     DECMIC   %%C1,4                                     $IF      1
                    10    $LASTIF  MICRO    '%%1L0000'                                 $IF      1
                    10    $NEXTIF  MICRO    '%%1N0000'                                 $IF      1
```


1SHOWIF
IOP APML 2.1(03/19/87) 04/01/87 14:31:57 Page 6
(6)
<<<<< 1
<<<<< 0

$UNTIL and $ENDTIL

Together, these macros define a loop. $UNTIL marks the start of the loop, and specifies the exit test condition. This condition consists of one or two APML conditions connected by an AND or OR relationship. $ENDTIL marks the end of the loop. The statements in between constitute the repeated procedure. $UNTILs may be nested up to 1Ø levels.

Upon entering the $UNTIL, the condition is checked. If **false**, the routine between $UNTIL and $ENDTIL is executed once, and control passes back to the $UNTIL, where the condition is checked again. When the condition becomes **true**, control passes to the statement following $ENDTIL. If the condition is true initially, the procedure is skipped altogether.

FORMAT:

Location	Result	Operand
	$UNTIL	(cond1), AND/OR, (cond2)
	$ENDTIL	

L	optional statement label.
cond1,cond2	any valid APML conditional expression.
and/or	logical operator 'AND' or 'OR'. If blank cond2 is ignored.

Example:

See virtually any page in the KERNEL.

FIELD

FIELD describes the location of a field within a table.

Location	Result	Operand
sym	FIELD	p,s,w[,L=]

sym — Field symbol name. By convention, this name has the form : TTT@FF, where TTT is a three-letter abbreviation identifying the table to which the field belongs, and FF is a two-letter abbreviation identifying this field in the table. The name may not be longer than 6 characters, because the FIELD macro generates a series of EQUALS pseudos, giving them names in the form: SYM@Z, where SYM is the symbol used on the FIELD macro, and Z is one of the suffix codes shown below. If the SYM is longer than 6 characters, then the addition of the @Z will give a name longer than the maximum 8 characters.

p — Parcel offset of the field from the beginning of the table

s — Starting bit of the field within the parcel (default is Ø)

w — bit Width of Field (default is 16)

L= — parcel length of field

Note that *p*, *s*, and *w* are specified in **decimal.**

The following five symbols are assigned values by EQUALS pseudos generated by the FIELD macro:

...@P	Parcel offset from the beginning of table
...@S	Starting bit of the field (numbered from the left)
...@N	Width of the field
...@M	Mask for field, right justified
...@X	Complement of mask in proper position in field

If P=* then @P is undefined

If S=* then @S, @N are undefined

GET and PUT

These macros make use of the parameters generated by the FIELD macro to greatly simplify the manipulation of the data in the fields. We supply the name of the field, and a pointer to the start of the table in local memory, and the macros will move data to or from them. Any necessary masking or shifting is done for us.

GET	Loads a field from a table into local memory location or an operand register
PUT	Stores data in a field in a table from an operand register or a local memory location

The RPUT and RGET macros are variations, used less frequently, for those situations where the parcel containing the data field has been previously loaded into an operand register, or where there is a pointer, in an operand register, to the specific parcel in local memory containing the field.

RGET	Loads an operand register or memory location from a field in an operand register or memory location
RPUT	Loads a field in an operand register or memory location from an operand register or memory location

FORMATS:

Location	Result	Operand
L	GET	dest,field,base
L	PUT	source,field,base
L	RGET	dest,field,source
L	RPUT	source,field,dest

dest — Destination operand register or pointer to local memory location to receive data from field

field — Field to be loaded, defined by the FIELD macro

base — An operand register containing table base address

source — Source operand register or pointer to local memory location containing data to be placed into field

These field manipulating macros are to be found on nearly every page of the IOS system software. The FIELD macros that define the data areas to be manipulated account for perhaps half of the $APTEXT listing.

ADDRESS, STORE, RSTORE, COPY and CLEAR

ADDRESS returns the local memory address of the designated field.

STORE and RSTORE places a constant value into the designated field.

COPY moves an arbitrary area of local memory to another area of local memory.

CLEAR sets the designated area of local memory to Ø, or to blanks if BLANKS=YES is specified on the macro call.

FORMATS:

Location	Result	Operand
L	ADDRESS	result,field,base
L	STORE	constant,field,base
L	RSTORE	constant,field,dest
L	COPY	source,dest,length
L	CLEAR	START= ex1,COUNT=ex2

L	Optional statement label
ex1	starting address of area to be cleared
ex2	length of area to be cleared
constant	any constant value
source	operand register containing address of data to be copied
dest	operand register containing address of destination
result	operand register or memory location to receive address of field

Exercise For Section 5

Skill: Interpret system macros from $APTEXT and write APML using them

Task:

A. Write Field Macros to define the following data structure

Parcel Ø	16 bits	-Forward Link
Parcel 1	16 bits	-Backward Link
Parcel 2	bits Ø-7	-Message Number
Parcel 2	bits 9-14	-Reply Status
Parcel 3	bits 3-6	-Priority

B. Write a GET macro to read the priority field into register R!PRI. Assume that register R!BASE already contains the base address of the data structure.

C. (OPTIONAL) - Assemble the above statements with APML,MAC. to get a listing of the code generated by the FIELD and TABLE macros. You will have to add a few statements to assemble it without errors.

Related Reading:

SM-ØØ46	chapter 1Ø
SM-ØØ36	chapter 7

Intended Lesson Result: To be able to read and write using the $APTEXT macros and read their text definitions in the $APTEXT listing

Optional Programming Exercise

Write an APML routine to sort a table of parcel values in memory. The table starts at address TABLE, and symbol TABLEX is the next available parcel following the table. Use of discussed macros is encouraged.

Appendix A: Answers To Review Questions and Exercises

Answers To Review Questions for Section 1

1) **What characteristics of APML make it difficult to classify in the traditional high level/low level hierarchy?**

   APML generally retains the flavor of an assembler, with object code shown for each source statement, and ability to write statements at the machine instruction level. However, it also displays many characteristics usually associated with compilers. These characteristics include multiple source statements generated from a single object statement, and a sophisticated mechanism and syntax for making virtually any statement conditional.

2) **How do IOS binaries get from the Cray, where they are assembled, to the I/O Subsystem, where they execute?**

   APML binaries are disposed to expander tape or disk, from which they will be loaded into the IOS at start-up time.

3) **What makes APML code more cumbersome to test than similar CAL code?**

   The fact that there is no mechanism for executing "jobs" in the IOS. All IOS software is system software, and must be integrated with the IOS operating system.

Review Questions for Section 2

1) **Name three ways to refer to the contents of an operand register in APML syntax.**
   a) a two-character symbol
   b) R!symbol
   c) (B) [indirect reference to the contents of the operand register whose number is in the B register]

2) **What does the *f*-field of an IOP machine instruction contain?**
   the 7-bit function code for the instruction

3) **What might the d-field of an IOP machine instruction contain?**
   an operand register number; an immediate value; a channel number; a branch displacement

4) **What might the k-field of an IOP machine instruction contain?**
   and absolute machine address; immediate data

5) **How can an APML assignment statement be made conditional?**
   by following it with a comma and an APML condition

6) **How do you refer to the contents of the B Register in APML syntax?**
   the symbol B

7) **Explain the difference between the notation [DD] and the notation (DD).**
   [DD] gives the **operand register number** for a two-character symbol.
   (DD) gives the **contents of memory** at the address indicated in OR DD

Answers To Exercises For Section 3

1. APML most closely resembles a **macro assembly** language.

2.

R1=1ØØØ	,A=1ØØØ
A=R1	,A=1ØØØ
A=[R1]	,A=17
A=(R1)	,A=2241
A=VAL	,A=17
A=(R!VAL)	,A=2241
A=R!VAL	,A=1000

3.

R!OR2=721	,A=721
A=R!OR2	,A=721
A=CAT	,A=25
A=R!CAT	,A=721

4. Statements with errors are underlined. Look up the error codes (circled, at left of listing) in SM-0036 appendix A.

```
                                        IDENT   APML2
     0    <macro>                       REGDEFS ,,(R1,R2)
                                  *   SOLUTION FOR SECTION 2 EXERCISE, QUESTION 4
(S7)
     0   070000                         P=P+17
     1   050000  022430  011017         E=B+R1&17
         154002
     5   150002  061000  004002         R2=E&(B)>2
         024431
(S9)
    11   070000                         R2=(B)&E>2
    12   150002  021430  024410         A=B,B=E&R1
         050000  023410  103002
         050000
(04)
(S9)
    21   070000                         A=B,P=1000
    22   070000                         A=B,B=R1&E
    23   010000  062000  013010         A=B,0=0+(B)-10&0>0
         011000  004000  024410
         010000  023410  103002
         050000
```

Each of the numbered items below shows more than one notation that will accomplish a certain type of assignment in APML. Determine the five assignments being illustrated.

```
R1      EQUALS  1
R2      EQUALS  2
R3      EQUALS  3
R5      EQUALS  5
REG1    EQUALS  1
REG2    EQUALS  2
REG3    EQUALS  3
REG4    EQUALS  4
```

1) R!REG1=12
R1=12
operand register 1 gets 12

2) R!REG2=R!REG1
R!REG2=R1
R2=R!REG1
R2=R1
operand register 2 gets 12

3) R!REG3=(R!REG1)
R3=(R!REG1)
R!REG3=(R1)
R3=(R1)
operand register 3 gets contents of LM where operand register 1 points

4) R!REG4=REG1
R!REG4=[R1]
operand register 4 gets 1

5) R5=(REG1)
R5=([R1])
operand register 5 gets contents of LM parcel 1

Answers to Review Questions for Section 4

1) **List the pseudoinstructions that appear in APML, but not in CAL.**
   BASEREG, SCRATCH, NEWPAGE, EQUALS, PDATA
2) **Why is the concept of the "page" needed in APML?**
   Since object code instructions may be of varying lengths, and since the assembler must decide whether to generate a 1-parcel relative of 2-parcel absolute branch, the address of a destination symbol is indeterminate during pass 1. A page break forces the assembler to go back and assign addresses to all symbols in the preceding page, allowing a determination of whether a relative or absolute branch will be generated for the branches within that page.
3) **How are the operand registers defined as SCRATCH used by the APML assembler?**
   They will be used to hold pointers to memory address operands, and to hold intermediate results.

Answers To Exercises For Section 5

A & B:

```
JOB,JN=TNGDEF5,T=5,US=.
ACCOUNT,AC=,UPW=.
APML,MAC.
~e
         IDENT    EXS5
         REGDEFS  ,,(BASE,PRI)
EX5@FL   FIELD    0           .START BIT & LENGTH DEFAULT TO 0 AND 16
EX5@BL   FIELD    1
EX5@MSG  FIELD    2,0,8
EX5@REP  FIELD    2,9,6
EX5@PRI  FIELD    3,3,4
*
GETPRI   GET      R!PRI,EX5@PRI,R!BASE
         EXIT
```

Solution To Optional Programming Exercise:

```
 0    <macro>                          IDENT   SAMPLE1
 0                                     REGDEFS ,,(NEXT,TBLC,TMP)
 0                                SORT *
 0   010001  024432                    R!TMP=1                .FORCE 1 TIME THRU LOOP
 2    <macro>                          $UNTIL (R!TMP=0)
 4   010000  024432                        R!TMP=0
 6   014000 /000037  024431                R!TBLC=TABLE       .INIT CURRENT TO START OF TABLE
11   020431  012001  024430                R!NEXT=R!TBLC+1   .START COMPARISON W NEXT ITEM
14    <macro>                             $UNTIL (R!NEXT=TABLEX)
20    <macro>                                  $IF ( (R!NEXT)<(R!TBLC) )
24   030430  024432                                R!TMP=(R!NEXT)     .EXCHANGE TABLE ITEMS, AND
26   030431  034430                                (R!NEXT)=(R!TBLC) .SET EXCHANGE FLAG (TMP)
30   020432  034431                               (R!TBLC)=R!TMP
32    <macro>                                  $ENDIF
32   026431                                    R!TBLC=R!TBLC+1
33   026430                                   R!NEXT=R!NEXT+1    .COMPARE NEXT ITEM IN TBL ON NEXT PASS
34    <macro>                                $ENDTIL
35    <macro>                             $ENDTIL
36   001000                               EXIT
37                                    TABLE *            .VALUES TO BE SORTED
37   000055                               55                              iP
40   000027                               27
41   000072                               72
42   000010                               10
43   000005                               5
44   000036                               36
45   000103                               103
46                                    TABLEX *            .END OF TABLE
                                                          END
```

Partial CSIM Output For Optional Programming Exercise

```
At 17:30:59 on 06/08/87:  Cray CPU/IOS simulation     X.17 CSIM version of 05/13/87  08:22::
DEFIOS,,,,SIM.
DEFCPU,MEM=400000.
START,TNGDEFB.
Dataset TNGDEFB  loaded into MIOP.
RUN,T=35,HT=IOP.
P=000032
A=000003    C=1
B=000
000430   000040
000431   000037
000432   000000
000433   000000
000434   000000
000435   000000
000436   000000
000437   000000
DUMP,FW=0,LW=277,HT=IOP0.
0000000 010001 024432 020432 102033 010000 024432 014000 000037   ) !     )
0000010 024431 020431 012001 024430 020430 017000 000046 102016 ) !   ) !    &
0000020 030430 033431 100002 070007 030430 024432 030431 034430 1 7   p 1 ) 1 9
0000030 020432 034431 026431 026430 071020 071033 001000 000005 ! 9 - - r r
0000040 000010 000027 000036 000055 000072 000104 000000 000000        - : C.
```