CENTURION

CPU-5/CPU-6

CENTURION PROGRAMMING LANGUAGE

December 1981

Includes 6/15/83 Revisions

CENTURION COMPUTER CORPORATION
1780 Jay Ell Drive
Richardson, Texas 75081

Printed in the USA

TABLE OF CONTENTS
CPU-5/CPU-6
CENTURION PROGRAMMING LANGUAGE - CPL

Chapter Three - PROGRAM CONTROL

Chapter Four - COMPILER DIRECTIVES

Chapter Six- RESERVING MEMORY

Chapter Seven - CPL ASSIGNMENT STATEMENTS

Chapter Nine - TRANSFER OF CONTROL

REFERENCE TABLE
CPU-5/CPU-6
READY REFERENCE

REFERENCE TABLE
CPU-5/CPU-6
CPL COMMANDS

ABS (VIII.1)	-	ABS (arg)
ADRLST (XIV.11)	-	ADRLST (address, address, ...)
BUFFER (VI.5)	-	BUFFER label (n) [,label (n), label (n), ...]
CALL (IX.6)	-	CALL subroutine [(name, name, ...)]
CLOSE (X.3)	-	CLOSE file [,file, file, ...]
COPY (IV.6)		COPY label [SYSn]
CPL (IV.2)	-	CPL
CURB (XIV.1)	-	CURB (file, number)
CURP (XIV.1)	-	CURP (file, column, line)
CURS (XIV.1)	-	CURS (file, number, string)
CURSOR (XIV.1)	-	CURSOR (file, line, column)
DECODE (XI.5)	-	DECODE (string, format) variable, variable, ...
DECR (VII.3)	-	DECR integer [,n]
DECREMENT (VII.3)	-	DECREMENT integer [,n]

DEFINE (VI.4)	-	DEFINE label:'string' [,label:'string', label:'string', ...]
DIRECT (IV.2)	-	DIRECT
DUMP (XIV.2)	-	DUMP (address1, address2)
EJECT (IV.4)	-	EJECT
ENCODE (XI.6)	-	ENCODE (string, format) variable, variable, ...
END (III.4)	-	END
END DO (IX.12)	-	END DO
ENDFILE (X.4)	-	ENDFILE file [,file, file, ...]
END LOOP (IX.5)	-	END LOOP
ENDREC (XII.1)	-	ENDREC
ENTRY (III.2)		ENTRY
ENTRYPOINT (V.2)		ENTRYPOINT label [,label, label, ...]
EQUATE (XIV.5)		EQUATE label, expression
EXTERNAL (V.1)		EXTERNAL label [,label, label, ...]

FILE (X.1)	-	FILE file: SYSccc, [access], [CLASS=n], [BUFFER=n, buffer], [RECSIZ=n], [KEY=integer], [FILTYP=c], [LSR=routine]
FORMAT (XI.1)	-	FORMAT format: specification, specification, ...
FREE (XII.4)	-	FREE (file)
FRER (XIII.3)	-	CALL FRER (file)
GETR (XIII.1)	-	CALL GETR (file, record)
GO TO (Conditional) (IX.2)	-	GO TO (label, label, ...) ON expression
GO TO (Unconditional) (IX.2)	-	GO TO label
GTIME (Integer) (XIV.6)	-	GTIME (INTEGER, integer)
GTIME (String) (XIV.6)	-	GTIME (STRING, string)
HLDR (XIII.3)	-	CALL HLDR (file)
HOLD (XII.4)	-	HOLD (file)
IF (IX.10)	-	IF (value operator value) action

IF-DO
(IX.10)
	-	IF (value operator value) DO
		actions
		END DO

IF-DO-ELSE
(IX.10)
	-	IF (value operator value) DO
		actions
		END DO
		ELSE action

IF-DO-ELSE DO
(IX.10)
	-	IF (value operator value) DO
		actions
		END DO
		ELSE DO
		actions
		END DO

IF-DO-Null-ELSE
(IX.10)
	-	IF (value operator value) DO
		actions
		END DO
		ELSE

IF-ELSE (IX.10)	-	IF (value operator value) action ELSE action

IF-ELSE DO
(IX.10)
	-	IF (value operator value) action
		ELSE DO
		actions
		END DO

IF-Null-ELSE (IX.10)	-	IF (value operator value) action ELSE
IF(x) (IX.10)	-	IF (value) action
IFSTRING or IFS (IX.11)	-	IFSTRING (string operator string) action
IFSTRING-DO or IFS-DO (IX.11)	-	IFSTRING (string operator string) DO actions END DO
IFSTRING-DO-ELSE or IFS-DO-ELSE (IX.11)	-	IFSTRING (string operator string) DO actions END DO ELSE action

IFSTRING-DO-ELSE DO or IFS-DO-ELSE DO (IX.11)	-	IFSTRING (string operator string) DO actions END DO actions END DO

IFSTRING-DO-Null-ELSE
or
IFS-DO-Null-ELSE
(IX.11)
	-	IFSTRING (string operator string) DO
		actions
		END DO
		ELSE

IFSTRING-ELSE or IFS-ELSE (IX.11)		IFSTRING (string operator string) actio ELSE action
IFSTRING-ELSE DO or IFS-ELSE DO (IX.11)		IFSTRING (string operator string) actio ELSE DO actions END DO
IFSTRING-Null-ELSE or IFS-Null-ELSE (IX.11)		IFSTRING (string operator string) actio ELSE
INCR (VII.3)		INCR integer [,n]
INCREMENT (VII.3)	-	INCREMENT integer [,n]
INTEGER (VI.1)		INTEGER label [,label, label, ...]
LDATE (XIV.7)		LDATE (form) [,label]
LEN (VIII.2)		LEN (arg)
LOAD (XIV.3)		LOAD (mask, label, option) [(name, name, ...)]

LOOP (IX.3)	-	LOOP j (a, b, c)
LOOP WHILE (IX.4)	-	LOOP WHILE (value operator value)
MAX (VIII.3)	-	MAX (arg1, arg2, ...)
MIN (VIII.4)	-	MIN (arg1, arg2, ...)
MOD (VIII.5)	-	MOD (arg1, arg2)
NOTE (XI.7)	-	NOTE (file, integer)
OPEN (X.2)	-	OPEN access file or OPEN access (file, ...) [,access (file, ...) ,...]
ORIGIN (XIV.4)	-	ORIGIN address
PAGE EJECT (IV.4)	-	PAGE EJECT
POINT (XI.8)	-	POINT (file, integer)
PRINT OFF (IV.3)	-	PRINT OFF
PRINT OFF,COM (IV.3)	-	PRINT OFF [,COM]
PRINT ON (IV.3)	-	PRINT ON
PUTR (XIII.2)	-	CALL PUTR (file, record)

READ (XI.2)	-	READ (file, format) variable, variable, ...
READB (XII.2)	-	READB (file, record)
RECORD (XII.1)	-	RECORD record (n)
RESET (X.12)		RESET file [,file,file,...]
RETRIEVE (IX.9)		RETRIEVE (type, location [,location, ...])
RETURN (IX.8)	-	RETURN
RETURN TO (IX.8)		RETURN/ TO label
REWIND. (X.5)		REWIND file [,file, file, ...]
REWRITE (XI.9)		REWRITE (file, format) variable, variable, ...
ROUND (VIII.6)	-	ROUND (arg1, arg2)
SDATE (XIV.7)	-	SDATE (form, label)
SET (VI.2)		SET label : n [,label:n, label:n, ...]
SETFORM (CPU-6) (X.6)		SETFORM (file [,n] [,HOLD] [,FREE])
SGN (VIII.7)		SGN (arg)
SKIP (X.11)		SKIP (file,n)

SPACE (IV.5)	-	SPACE n
STOP (III.3)	-	STOP [code]
STRING (VI.3)	-	STRING label (n) [,label (n) label (n), ...]
SUBROUTINE (IX.7)	-	SUBROUTINE label
SYSTEM (CPU-5) (III.1)	-	SYSTEM [name] [(parameters)]
SYSTEM (CPU-6) (III.1)	-	SYSTEM [name] [(parameters)]
TABLE (Integer) (VI.6)	-	TABLE name (n) [,name (n), name (n), name (n), ...]
TABLE (String) (VI.6)	-	TABLE name (len,n) [,name (len,n), name (len,n) ...]
TBLGET (XIV.9)	-	TBLGET table (integer)
TBLPUT (XIV.10)	-	TBLPUT table (integer) [:'string'] or [:value]
TITLE (IV.1)	-	TITLE 'title'
WRITE (XI.3)	-	WRITE (file, format) variable, variable, ...
WRITEB (XII.3)	-	WRITEB (file, record)

WRITEN (XI.4)	-	WRITEN (file, format) variable, variable, ...
WRITN (XI.4)	-	WRITN (file, format) variable, variable, ...

NOTE: CPL commands must be included on one line. Those commands which are included on more than one line in this reference reference chart are listed in such a manner for purposes of documentation only.

REFERENCE TABLE
CPU-5/CPU-6
CPL COMMANDS - FORMATTING CONVENTIONS

The following conventions are followed in the format of a CPL command.

1. A CPL command may be from 1 to 300 characters in length. However, the text editor will truncate lines at 132 characters.

2. Capital letters indicate the actual words/phrases to be entered into the computer.

3. Lower case letters indicate the variable portions of a command.

   NOTE: Unless otherwise specified, a lower case "n" represents a numeric literal; a lower case "c" represents an alphanumeric character.

4. Brackets [ ] indicate optional words or phrases.

5. When punctuation is shown as part of a CPL command, it is a requirement.

6. A line of CPL is deblanked as it is compiled with individual fields being separated by spaces and/or commas.

   NOTE: Inclusion of space(s) is optional and is done only for textual clarification.

7. Items within a CPL command are not assigned to specific columns.

8. If special characters (i.e. quotation marks, equal signs, parentheses, etc.) are used in the command format, they must be included in the actual command.

Revision 03/15/83

REFERENCE TABLE
CPU-5/CPU-6
CPL NAMING CONVENTIONS - LABELS, INTEGERS and STRINGS

The following guidelines must be followed in assigning names to labels, integers and strings.

NOTE: Where punctuation is a part of a CPL command format, it is a requirement.

1. A label name may be from 1 to 255 characters in length.

2. The initial character of a label may be a "?", "@" or any alphabetical character. The remaining characters may be alphanumeric, "?" or "@".

   NOTE: While the initial character in a label may not be a blank, any of the remaining characters may contain a blank.

3. The initial character of a 4-byte integer may be any alphabetical character. The remaining characters may be alphanumeric, "?" or "@".

4. The initial character of a 6-byte integer is a "?". The remaining characters may be alphanumeric, "?" or "@".

5. The initial character of a CPL operating system label is "@". Although the system will accept programmer supplied labels beginning with "@", usage may result in multiply defined labels within a single program.

6. Labels must not begin with the characters of a CPL command. (SETUP would be an illegal label name since it incorporates the term SET, a CPL command word.) The program will not compile with labels of this sort and a syntax error will result.

7. The final character in a program label must be followed by a colon (:).

REFERENCE TABLE
CPU-5/CPU-6
CPL SYMBOLS

The following symbols are used in CPL programming.

NOTE: Where punctuation is part of a CPL command format, it is a requirement.

\+ the plus sign is used to indicate (a) the addition operation or (b) concatenation in string operations.

\- the negative sign is used to indicate the subtraction operation.

* the asterisk is used to indicate the multiplication operation.

/ the slash is used to indicate the division operation.

= the equal sign is used to indicate the substitution operation.

( ) parentheses are used in arithmetic expressions to indicate a mathematical grouping.

. the period is used to delimit the operator in all forms of IF, IFSTRING/IFS and LOOP WHILE commands. For additional information see IF (IX.10) and IFSTRING/IFS (IX.11).

: the colon is used (a) following the label in a SET (VI.2) or DEFINE (VI.4) statement to separate the label name from the assigned value or (b) following a file name in a FILE (X.1) statement to separate that name from the keywords which follow (c) following the label in a FORMAT (XI.1) statement to separate that label from the field specifications which follow or (d) following the final character in a program label (Labels, IX.1).

, the comma is used to separate multiple labels or values in a CPL statement.

\ the reverse slash allows multiple statements to be coded on the same line (Reverse Slash, IV.9).

' the single quote is used as a delimiter for a string.(Literal, II.1; Expression, II.3; TITLE, IV.1; DEFINE, VI.4; String Assignement Statement, VII.2)

the double quotation mark is used to indicate an expression contains a string literal. (Expression, II.3.1; String Assignment Statement, VII.2)

? an initial question mark indicates a 6-byte integer.

@ an initial @ generally indicates an operating system label.

the semicolon functions as the comment character. It allows commentary and blank lines to be added to the source listing. (Comment Line, IV.7)

the hyphen functions as the continuation character. It allows a line of CPL code to be divided between two or more records of the source file; that is, one line of code can be entered on two lines. (Continuation Line, IV.8)

REFERENCE TABLE
CPU-5/CPU-6
CPL ABBREVIATIONS AND GENERAL TERMS

1. Operators for Integer and String Comparisons -

   a. .EQ. - equal to
   b. .NE. - not equal to
   c. .LT. - less than
   d. .LE. - less than or equal to
   e. .GT. - greater than
   f. .GE. - greater than or equal to

   NOTE: Before any comparison takes place in a string, trailing blanks are dropped and lower case characters are converted to upper case.

   An "H" placed before the operators listed above (i.e. .HEQ., .HNE., .HLT., .HLE., .HGT. or .HGE.) may be used only with string comparisons. Use of this type of operator does not cause trailing blanks to be dropped and/or lower case characters to be converted to upper case.

2. STATUS - During program execution, CPL checks input and output. The result of each check is stored in STATUS. Since results are stored with each check of I/O, the value of STATUS changes with each CPL operation.

3. EJECT - This form feed character causes top-of-form when sent to a printer; it causes the screen to be cleared when sent to the CRT (additionally, it sends the cursor to home position). EJECT is usually used with WRITE or MSG (an external routine).

4. VTAB - This vertical tab character causes a printer to space to a predetermined position set by the Vertical Format Unit.

   NOTE: Not all Centurion printers are equipped with VFU; the VTAB option is not operational in these instances.

5. BEEP - This character is used to create a sound when output by some CRTs/printers.

REFERENCE TABLE
CPL
CPU-5/CPU-6 COMPATIBILITY

There is no basic difference between a CPU-5 program and a CPU-6 program. However, while it is possible to compile for a CPU-5 running on a CPU-6 system, compiling for a CPU-6 while running on a CPU-5 system is <u>not</u> allowed.

1. CPU-5 - contains a "S Library" with the compiler invoked by S.CPL or S.SCPL.

   CPU-6 - contains a "P Library" with the compiler invoked by P.CPL or P.SCPL.

   For additional information see Introduction (I.2).

2. CPU-5 - may not compile a program out of a library.

   CPU-6 - may compile a program out of a library.

   For additional information see Introduction (I.2).

3. CPU-5 - SYSTEM command parameters are -
   type, LL=n, EXP=A, EXP=B, EXP=C, EXP=D

   CPU-6 - SYSTEM command parameters are -
   type, LL=n, EXP=A, EXP=B, EXP=C, EXP=D, STACK=nnn

   For additional information see SYSTEM (III.1)

4. CPU-5 - will accept ESP (an obsolete predecessor of CPL).

   CPU-6 - will not accept ESP.

   For additional information see DIRECT/CPL (IV.2).

5. CPU-5 - uses the CLPRP utility for copy member processing.

   CPU-6 - copy member processing is done through the compiler.

   For additional information see COPY (IV.6).

6. CPU-5 - COPY member processing handled by CLPRP utility through processing of an "A" Type copy library or a "D" Type private library.

   CPU-6 - COPY member processing handled by the compiler through processing of an "A" Type copy library only.

   Note: For CPU-5/CPU-6 compatibility use discrete Type "A" files.

   For additional information see COPY (IV.6).

7. CPU-5 - may use INTEGER to pass data from one program to the next.

   CPU-6 - INTEGER may not be used to pass data from one program to the next.

   For additional information see INTEGER (VI.1).

8. CPU-5 - the remainder in a division problem is not stored.

   CPU-6 - allows the remainder of the last division performed to be stored in @REM (4-byte integer) or ?@REM (6- or 8-byte integer).

   For additional information see Integer Assignment Statement (VII.1).

9. CPU-5 - does not provide partition protection.

   CPU-6 - provides partition protection.

   For additional information see String Assignment Statement (VII.2).

10. CPU-5 - illegal use of RETURN, CALL or GO TO commands may result in eventual destruction of a program.

10. (cont.)
CPU-6 - illegal use of RETURN, CALL or GO TO commands may result in a program abort.

For additional information see RETURN/RETURN TO (IX.8).

11. CPU-5 - utilizes Type "C", VSI, indexed files.

CPU-6 - utilizes Type "I", VSI, indexed files.

For additional information see FILE (X.1).

12. CPU-5 - LSR=5 may result in sector pointer error.

CPU-6 - contains no sector pointers.

For additional information see FILE (X.1).

13. CPU-5 - random file wastes disk space, but is faster than random-spanned; random-spanned file doesn't waste disk space, but is slower than random.

CPU-6 - random file wastes disk space, but the file may or may not be buffered; random-spanned waste disk space, but requires a large buffer.

For additional information see FILE (X.1).

14. CPU-5 - no default exists for RECSIZ=n on Type "A", Type "B" or Type "C" files.

CPU-6 - default for RECSIZ=n on Type "I" file is 400.

For additional information see FILE (X.1)

15. CPU-5 - maximum file record size is -

393 bytes - Type "A" and Type "B" files

15. (cont.)

395 bytes	-	Type "C", random
		Type "C", 4- and 6-byte indexed
2048 bytes	-	Type "C", random-spanned
		Type "C", VSI

CPU-6 - maximum file record size is -

398 bytes	-	Type "A" and Type "B" files
400 bytes	-	Type "C", random
		Type "C", 4- and 6-byte indexed
2048 bytes	-	Type "C", random-spanned
		Type "I", VSI

For additional information see FILE (X.1).

16. CPU-5 - operating system does not check whether the file is open for input or output. It will allow either input or output at any time after the opening no matter how the file was opened.

CPU-6 - operating system checks to see what access opened the file; only that type of access may be used to open the file.

For additional information see OPEN (X.2).

17. CPU-5 - does not allow the SETFORM command.

CPU-6 - allows the SETFORM command.

For additional information see SETFORM (X.6).

18. CPU-5 - READB/WRITEB and HOLD/FREE should not be be used with Type "C" random-spanned or VSI files.

CPU-6 - READB/WRITEB and HOLD/FREE may be used with Type "C" random-spanned or Type "I" VSI files.

For additional information see READB (XII.1), WRITEB (XII.3), HOLD/FREE (XII.4).

19. CPU-5 - use of READB/WRITEB and HOLD/FREE will result in errors when used with Type "C" random-spanned or VSI files. Use GETR/PUTR and HOLDR/FRER in this instance.

    CPU-6 - use of READB/WRITEB and HOLD/FREE will not result in errors when used with Type "C" random-spanned or Type "I" VSI files. However, to insure CPU-5 compatibility, use GETR/PUTR and HOLDR/FRER.

    For additional information see READB (XII.2), WRITEB (XII.3), HOLD/FREE (XII.4), GETR (XIII.1), PUTR (XIII.2) and HLDR/FRER (XIII.3).

20. CPU-5 - maximum number of entries in the system HOLD/FREE table is equal to 6 times the number of partitions on the system.

    CPU-6 - maximum number of entries is system generated.

    For additional information see HOLD/FREE (XII.4).

21. CPU-5 - automatic partition growth/shrinkage is not provided.

    CPU-6 - operating system provides automatic partition adjustment.

    For additional information see LOAD (XIV.3).

22. CPU-4 - stores system time as 1/20,000 of a second

    CPU-5 and CPU-6 - stores system time as 1/10 of a second.

    For additional information see GTIME (XIV.6).

****

WARNING

****

The latest version of CPL (12/81) contains important differences from earlier versions. These differences include:

1. Offsets - offsets must be in brackets to indicate assembler expressions. For example:

   CALL SUBR ([VARI+OFFSET], VAR2)

2. Data fields - data field locations in IF and assignment statements have changed positions. Therefore, in any program containing assembler code which stores data in an IF or assignment statement during execution, it may be necessary to modify assembler offsets.

3. Program Labels - program labels which begin with the same characters as new CPLII keywords will no longer be legal. These keywords include ADRLST, CURSOR, ENDDO, ENDLOOP, EQUATE, LOOP, LOOPWHILE, ORIGIN, RETRIEVE.

4. EXP=A, EXP=B, EXP=C, EXP=D - both the old CPL and new CPLII complier are accessible at the present time. Programs using EXP=A or EXP=B are handled by the old CPL compiler; additionally, if the EXP option is not used as part of the SYSTEM statement, the program is handled by the old CPL compiler.

   Programs compiled by the CPLII compiler must specify EXP=C (CPU-5 code or CPU-5 code with CPU-5 compatible offsets) or EXP=D (CPU-6 code) in the SYSTEM statement.

   NOTE: CPL will be supported until January 1, 1984, to allow for conversion of existing programs. All new programs, however, should be handled by CPLII using EXP=C or EXP=D.

CPU-5/CPU-6
CPL

SAMPLE PROGRAM

```
PAGE   1   ERRORS   0   ASSM 6.04   11/17/81   10:08:14     MASTER FILE UPDATE

0000                    *SYSTEM(EXP=D)
011D                    *!
011D                    *! NOTES:
011D                    *!                    SAMPLE PROGRAM
011D                    *!
011D                    *!*****************************************************
011D                    *!                    PROGRAM LINKAGE
011D                    *!*****************************************************
011D                    *!
011D                    *EXTERNAL GETK,GETR,PUTR
011D                    *!
011D                    *!*****************************************************
011D                    *!                    FILES
011D                    *!*****************************************************
011D                    *!
011D                    *FILE CRT:SYSIPT
013B                    *!
013B                    *FILE MASTER:SYS0,IND,CLASS=2,KEY=MKEY
04BF                    *SET MKEY:0
04C3                    *!
04C3                    *FILE TRANS:SYS1,RND,CLASS=2,RECSIZ=19,KEY=TKEY,BUFFER=400,*
0674                    *SET TKEY:0
0678                    *!
0678                    *!*****************************************************
0678                    *!                    RECORDS
0678                    *!*****************************************************
0678                    *!
0678                    *RECORD MREC(12)
067C                    * INTEGER MCLASS
0680                    * INTEGER SALES
0684                    * INTEGER INDEX
0688                    *ENDREC
0689                    *!
0689                    *RECORD TREC(19)
068D                    * STRING PRODUCT(10)
0698                    * INTEGER TCLASS
069C                    * INTEGER QUANTITY
06A0                    *ENDREC
06A1                    *!
06A1                    *!*****************************************************
06A1                    *!                    LOGIC
06A1                    *!*****************************************************
06A1                    *!
06A1                    *ENTRY
06BC                    *!
06BC                    *! INITIALIZATION
06BC                    *!
06BC                    *OPEN IO (CRT,MASTER,TRANS)
06C8                    *!
06C8                    *'MSG'="'MASTER RECORD UPDATED'"
06D0                    *TBLPUT MSG(1)
```

```
PAGE   2   ERRORS   0   ASSM 6.04  11/17/81  10:08:14    MASTER FILE UPDATE

06D8                    *'MSG'='NO UPDATE - CLASS NOT EQUAL'"
06E0                    *TBLPUT MSG(2)
06E8                    *'MSG'='NO UPDATE - INDEX NOT EQUAL'"
06F0                    *TBLPUT MSG (3)
06F8                    *!
06F8                    *LOOP MCLASS(1,9)
0709                    *WRITE(CRT,F01)'ENTER CURRENT PRICE FOR CLASS',MCLASS
0715                    *READ(CRT,F02)PRICE
071F                    *PRICE(MCLASS)=PRICE
0749                    *END LOOP
074C                    *!
074C                    *WRITE(CRT,F01)'ENTER INDEX FOR THIS UPDATE
0756                    *READ(CRT,F02)TINDEX
0760                    *!
0760                    *! UPDATING
0760                    *!
0760                    *READB (TRANS,TREC)
0766                    *LOOP WHILE (TCLASS.GT.0)
0771                    * CALL GETK(MASTER,PRODUCT)
0778                    * IF (STATUS) GO TO ELOOP
0786                    * CALL GETR(MASTER,MREC)
078D                    * IF (STATUS) GO TO ELOOP
0798                    * IF (TINDEX.EQ.INDEX) IF (MCLASS.EQ.TCLASS) DO
07B3                    *    SALES=SALES+(QUANTITY*PRICE(MCLASS))
07EA                    *    CALL GETK(MASTER,PRODUCT)
07F1                    *    IF (STATUS) GO TO ERROR
07FF                    *    CALL PUTR(MASTER,MREC)
080e                    *    IF (STATUS) GO TO ERROR
0814                    *    CALL MESSAGE(1)
0819                    *    COUNT= COUNT+1
0833                    *    END DO
0833                    *   ELSE CALL MESSAGE(2)
083B                    * ELSE CALL MESSAGE(3)
0843                    *GO TO ELOOP
0846                    *ERROR:
0846                    * WRITE(CRT,F03)'ERROR ON UPDATING PRODUCT:    PRODUCT
0852                    *ELOOP:
0852                    *INCR TKEY
0858                    *READB(TRANS,TREC)
085E                    *END LOOP
0861                    *!
0861                    *! TERMINATION
0861                    *!
0861                    *DONE:
0861                    * WRITE(CRT,F03) UPDATE COMPLETED:',TKEY-1,'TRANSACTION RECORDS READ'
0899                    * WRITE(CRT,F04)COUNT,'MASTER RECORDS UPDATED'
08A5                    *STOP
08A8                    *!
08A8                    *!*************************************************************
08A8                    *!                  SUBROUTINES
08A8                    *!*************************************************************
```

```
PAGE   3 , ERRORS   0   ASSM 6.04  11/17/81  10:08:14    MASTER FILE UPDATE

0898                    *!
0898                    *SUBROUTINE MESSAGE
0898                    *RETRIEVE (NUMBER,S)
08B1                    *TBLGET MSG(S)
08B5                    *WRITE(CRT,F01)MSG
08C2                    *RETURN
08C3                    *!
08C3                    *!*************************************************************************
08C3                    *!                    FORMATS AND STORAGE
08C3                    *!*************************************************************************
08C3                    *!
08C3                    *FORMAT F01:C29,N4
08CA                    *FORMAT F02:N4
08CE                    *FORMAT F03:C17,X2,D4,X2,C24
08DE                    *FORMAT F04:X19,N4,X2,C22
08EE                    *FORMAT F05:C28,C10
08F2                    *!
08F2                    *TABLE PRICE(8)
0918                    *TABLE MSG(27,81)
0987                    *!
0987                    *INTEGER TINDEX,S
098F                    *SET COUNT:0
0993                    *!
0993                    *END
0ABD                            PRINT  ON
0ABD          06A3              END    @30
```

Chapter One - INTRODUCTION

I.1 CPL

CPL (Centurion Programming Language) is an English-orientated, complex computer language. While CPL bears similarities to COBOL, FORTRAN and PL1, it is a unique language with distinct characteristics.

I.2 CPL Jobstreams

The CPL compiler is invoked by one of the following:

1. P.CPL/P.SCPL - run on the CPU-6 for the CPU-6
2. P.CPL5/P.SCPL5 - run on the CPU-6 for the CPU-4, CPU-5 or MICRO PLUS
3. S.CPL/S.SCPL - run on the CPU-5 or MICRO PLUS for the CPU-5 or MICRO PLUS

P.CPL (P.CPL5 or S.CPL), which links from source through relocatable to executable, compiles a main program; P.SCPL (P.SCPL5 or S.SCPL), which terminates after the compiler process leaving a relocatable file, compiles a subroutine.

The structure of the P.CPL/P.SCPL jobstream (CPU-6) is outlined in the following:

[P.CPL] [P.SCPL] file d prt [LIB] [XREF]

file - name of the source file without the leading "Z";
Note: The jobstream generates the initial "Z"; it should not be added by the programmer. The jobstream also generates an initial "X" to indicate an executable file and an initial "R" to indicate a relocatable file.
d - number of the disk which contains the file;
prt - name of the device (ie., printer, spooler, CRT or DUMMY) to be utilized during printing.

Note: If DUMMY is specified, no hard-copy program listing will be generated after a compile. However, run-time on the compiler will be much faster than that generated by any print device.

The keyword LST may also be used causing the jobstream to use file @LST#I ON #S. Listings generated by LST are placed into a Type "A" file which may be sent to a printer by the XLST utility.

Note: If errors exist within a program, the LST option results in the PSCAN utility scanning the listing for errors. Additionally, LST causes an automatic transfer to text upon exit from PSCAN.

LIB - optional keyword which indicates the presence of a copy library;

XREF - optional keyword which indicates that a cross-reference table will be output listing program labels, locations and a listing of all program references.

Note: If XREF is not stated, a sorted reference of all program labels will be output at the end of a compile.

I.3 CPL Processing

The CPL compiler is a single-pass facility. As such it operates by reading one line of source file (i.e. Type "A" file) and generating that line in assembler code. This process is repeated for each line until the compilation process is complete.

NOTE: This single-pass feature results in limited error detection. Syntax errors in the CPL command, undefined labels and and multiply-defined labels are among those errors detectable by CPL. Illegal use of data items is not detected.

An option which may occur at this point in the processing of CPL is the addition of copy libraries to the compiler. While any number of copy libraries may exist, only a maximum of 13 may be included at one time.

After compilation, the source file is read by assembler program XASSM and converted into binary instructions which are passed to the linker. During this process a listing of this program is output to the device specified in the CPL jobstream.

External subroutines may also be joined to the program at this time. The utility program XLINK performs this function by linking sections of code stored outside the program in subroutine libraries (i.e. OSLIB and APLIB) to the main program. Subroutine libraries function as a means to reduce program size.

NOTE: APLIB routines are called directly by the programmer with a CALL statement. OSLIB routines are called indirectly with the CALL being generated by the compiler.

I.4 CPL Execution

Two JCL commands (.USE and .RUN) are used in the execution process. .USE assigns the device or file to a logical unit. Once assigned, the device/file may be accessed either by a program or by the operating system.

.RUN passes control of a partition to an executable file. This sets the step flag and prevents the partition from being stopped from outside while the program is running. In addition, this step flag also causes end-of-step processing at the termination of the program. For additional information on .USE and .RUN see the Job Control Language Manual.

Chapter Two - CPL CONCEPTS

OVERVIEW

Only certain types of data may be used in a computer program. Literals, variables and expressions are all means of structuring data for use in a CPL program.

II.1 Literal

II.1.1 Usage

A literal is a quantity which remains constant throughout the execution of a program. A literal may be used in place of a variable name or expression.

> NOTE: For further information on the specific use of literals, consult individual command formats.

Literal types include 1) string literals and 2) integer literals.

String Literal
A string literal is an ASCII character string enclosed by single quotation marks (').

> Note: To represent a single quotation mark within a character string, two single quotation marks are used. For example, the display TWO 'TOO' would result from 'TWO ''TOO''' being entered.

Integer Literal
An integer literal is a sequence of digits which may be preceded by a leading positive (+) or negative (-) sign. Quotation marks and decimal notation are not allowed.

The size of the integer literals is determined by its value.

Integer Literal	Positive Range	Negative Range
1-byte integer literal	+127	-128
4-byte integer literal	+128 to +2,147,483,647	-129 to -2,147,483,648
6-byte integer literal	+2,147,483,648 to +140,737,488,355,327	-2,147,483,649 to -140,737,488,355,328

Note: Commas are included in this instance only for purposes of clarification. The only punctuation marks allowed are leading "+" or "-" signs.

A 1- or 4-byte literal may be stored as a 6-byte literal by adding an initial "?"; addition of "?" does not alter value. For example, a literal 0 (zero) would generate a 1-byte literal. A literal ?0 (zero) would generate a 6-byte literal containing 0 (zero).

Note: A 4- or 6-byte literal or string literal is stored in the object program one time only regardless of the number of times it may be referenced. One-byte literals are stored in-line and do not require such optimization.

II.2 Variable

II.2.1 Usage

A variable is the label of a data area whose value may change during the execution of a program. A variable may be used in place of a literal or expression. Variable names begin with an alphabetical character or a "?"; the remaining characters in the name can be alphanumeric, "?" or "@".

Variable types include 1) string variables and 2) integer variables.

String Variable
A string variable is the name of a non-numeric quantity made up of ASCII characters.

Integer Variable
An integer variable is the name of a numeric quantity.

II.3 Expression

II.3.1 Usage

An expression allows multiple operations to be combined within a single statement. In most instances an expression may be used in place of a literal or variable.

Expression types include 1) string expressions, 2) integer expressions and 3) assembler expressions.

String Expression
A string expression is a string variable name or group of string variable names combined by the concatenation operator (+).

> Note: String expressions are allowed only in string assignment statements.

If a string literal is present in an expression, that expression must be in double quotation marks ("). If string variables only are present in the expression, that expression must be in single quotation marks ('). An example of an expression containing a string literal would be:

```
"A + 'LIT'"
```

whereas an expression containing only string variables would be:

```
'A + B'
```

Integer Expression
An integer expression may be composed of 1) variable names, 2) integer literals, (3) the addition, subtraction, multiplication or division operators, (4) parenthetical expressions (to a maximum of 16 levels) and 5) built-in functions (i.e. ABS, LEN, MAX, MIN, MOD, ROUND, SGN).

> Note: Integer expressions are evaluated from left-to-right except where altered by parentheses and/or by built-in functions. Integer math is performed in all instances; no remainders are carried nor are decimals processed.

Assembler Expression

An assembler expression is any sequence of symbols which is passed unmodified to the assembler. Binary and hexidecimal expressions are allowed; for example, [X'FF83'] is treated as a literal expression.

Note: Assembler expressions must be enclosed in brackets ([]) unless otherwise specified.

Chapter Three - PROGRAM CONTROL

OVERVIEW

Certain commands control the set up of a program in addition to entry and exit. Code generated by these commands is mandatory and generally must occur in predetermined positions within the program.

NOTE: The commands included in Chapter Three are the only mandatory CPL commands. Every other statement in CPL is optional.

III.1 SYSTEM

III.1.1 Usage

The SYSTEM command marks the beginning of a CPL program or subprogram. In addition to starting the program, SYSTEM defines the CPL program or subprogram, generates the overhead data areas that CPL uses (i.e. working storage) and sets up the default names.

> NOTE: SYSTEM is a declarative command and must not occur within the logic section of the program.

The SYSTEM command is normally the first statement in a CPL program or subprogram. Every CPL program/subprogram must contain a SYSTEM statement; however, only one statement per program/subprogram is allowed.

Only CPL commands which directly affect the assembler, and do not generate object code, may precede SYSTEM. These commands include TITLE, PRINT ON/PRINT OFF,COM/PRINT OFF, DIRECT, CPL, COPY, SPACE, and PAGE EJECT/EJECT. TITLE, however, is normally the only statement to precede SYSTEM.

A SYSTEM command which contains the MAIN keyword adds 13 bytes to the length of a program; additional bytes are added equal to the amount included in the line length section (i.e. LL=n). A SYSTEM command which contains the SUBPGM keyword adds does not add to program length.

III.1.2 Command Format

The format of the SYSTEM command differs between the CPU-5/CPU-6 systems. Refer to the appropriate command format and the explanation included in this section before implementing SYSTEM.

> CPU-5
>
> SYSTEM [name] [(parameters)]
>
> NOTE: Parameters for the CPU-5 SYSTEM command include type, LL=n, EXP=A, EXP=B, EXP=C, EXP=D.

CPU-6

SYSTEM [name] [(parameters)]

NOTE: Parameters for the CPU-6 SYSTEM command include: type, LL=n, EXP=A, EXP=B, EXP=C, EXP=D, STACK=nnn.

- name - external name of the program/subprogram;
- type - keyword MAIN or SUBPGM;
- n - maximum number of characters to be stored in the program line buffer;
- A - uses OPSYS service calls and insures compatibility with CPU-5 source code (CPL compiler);
- B - generates CPU-6 assembler code (CPL compiler);
- C - uses OPSYS service calls and insures compatibility with CPU-5 source code (CPLII compiler);
- D - generates CPU-6 assembler code (CPLII compiler);
- nnn - number of bytes in the stack.

If any of the optional forms other than "name" are included, they must be enclosed in parentheses. If more than one optional form is included within the SYSTEM command, commas must be used to separate the entries contained within the parentheses. For example:

```
SYSTEM MOD (SUBPGM, EXP=A)
```

Program Name

A program "name" should be specified; however, if none is present, @CPL is the default.

Note: Although a default name is provided, a name should be specified to avoid multiple relocatable or executable programs named @CPL. Also, it is important to note that the SYSTEM "name" does not need to be the the same as the JCL filename for the source program. The SYSTEM "name" is an internal, logical label for the program.

Program Type

Program "type" may be either MAIN or SUBPGM. MAIN, which is the default, indicates a main program. If specified, the SYSTEM command defines the program line buffer, provides the external name of the program and creates ZERO and STATUS with initial values of zero (0).

> Note: Execution of a MAIN program begins with an ENTRY command. For additional information see ENTRY (III.2).

If the keyword SUBPGM is used, one of two subprogram types (i.e. an overlay program or an external subroutine) is specified. SUBPGM generates an external reference to ZERO and STATUS, but does not define them.

> Note: Since a subprogram contains no ENTRY commands, execution begins at the SYSTEM statement. A subprogram may be loaded by the CPL LOAD command or linked into a program from a binary library file. For additional information see LOAD (XIV.3).

Program Line Buffer

LL=n sets up the program line buffer used for formatted input/output. Default line buffer size is 132 characters; output/input greater than 132 will be truncated at 132. However, the line buffer may be set larger or smaller to accomodate individual needs.

> Note: It may be necessary to state LL=n with those printers which operate on compressed print capabilities and with formatted output to a Type "A" file.

Expansion C/ Expansion D

The Expansion C (EXP=C) and Expansion D (EXP=D) options are used with the CPU-6 SYSTEM command. EXP=C generates assembler code so that label or data offsets will be the same for both CPU-5 and CPU-6. This guarantees source code compatibility between the 5 and 6 operating systems.

EXP=D generates assembler code without guaranteeing compatibility between labels or data offsets. If a program

Expansion C/Expansion D (cont.)
contains embedded assembler, it may not be source code compatible with the CPU-5 when EXP=D is used. However, an EXP=D program will operate 50% to 90% faster than an EXP=C program.

The EXP=C and EXP=D options may also be used with the CPU-5 SYSTEM command. On the CPU-5, however, EXP=C and EXP=D will compile identically.

In all cases, EXP=C or EXP=D indicate use of the most current CPLII compiler. EXP=A and EXP=B are obsolete and are supported only for purposes of compatibility with existing CPL programs. EXP=C and EXP=D should always be used for new development.

Note: EXP=A is the default if no option is given.

STACK
The "stack" option is operative only with the CPU-6 SYSTEM command. STACK=nnn defines the size of the stack; 140 is the default.

The ENTRY command sets up the stack pointer on both the CPU-5 and CPU-6 systems. On the CPU-6 this stack area is at the lower end of the partition (i.e. the front of the program). Its size is defined by the STACK=nnn option on the SYSTEM command.

On the CPU-5 the stack area is at the high end of the partition and grows towards the end of the program. Consequently its size depends upon the size of the partition and the size of the program. For additional information see ENTRY (III.2).

Note: The ENTRY command sets the stack pointer for both CPU-5 and CPU-6 systems.

III.1.3 Cautions

1. Neither ZERO nor STATUS should be used for temporary storage. If ZERO is assigned any other value but 0, the value of STATUS may be affected.

III.2 ENTRY

III.2.1 Usage

The ENTRY command indicates the master program starting point. It also sets up communication between the program and operating system.

> NOTE: For information on how the ENTRY command is used to establish "stack" area, refer to section III.1.2 (STACK) of the SYSTEM command.

ENTRY may appear anywhere in the program; the statement immediately following the command word, however, must be executable. Additional code may precede the ENTRY command in the source program as long as control is properly transferred between sections of the program.

> NOTE: The JCL .RUN command passes control to the ENTRY statement. Program execution begins with the next statement following ENTRY.

The ENTRY command is required within a MAIN program; it may be used only once. The ENTRY command may not be used within a SUBPGM. The starting point of a SUBPGM is the statement immediately following the SYSTEM statement. For additional information on MAIN and SUBPGM, see SYSTEM (III.1.2, Program Type).

The ENTRY command adds 27 bytes to the length of a program.

III.2.2 Command Format

```
ENTRY
```

III.3 STOP

III.3.1 Usage

The STOP command terminates the CPL program and indicates a return to JCL. STOP returns control to the operating system for end-of-step processing. The value of the Completion Code (CC) may also be set by STOP.

> NOTE: STOP is an executable command and must occur within the logic section of the program. There is no limit on the number which may be used within a given program.

If a literal is used for the code, the STOP command adds 4 bytes to a program. If an expression is used, the number of bytes added to the program varies according to the complexity of the expression.

Program Logical Units
End-of-job processing turns off the step flag before checking all files assigned to Program Logical Units. Those files with PASS parameters remain assigned; those files with DELT parameters are deleted. All other Logical Unit Blocks are released and control is returned to SYSRDR.

III.3.2 Command Format

STOP [code]

code - integer variable, integer literal or integer expression whose value is to be assigned to the parameter CC.

Completion Code
The integer parameter Completion Code (CC) has a range from 0 to 255. Standard CC values include:

0	Normal termination
1-16	Values set by STAT (an APLIB subroutine)
100	I/O or program-controlled error

Note: If the integer variable, integer literal or expression is not within the 0-255 range, binary truncation will occur.

If a value of less than 0 is specified by the STOP command, the Completion Code remains unchanged. If no value is specified, the Completion Code remains unchanged.

III.4 END

III.4.1 Usage

The END command marks the end of a CPL program. It also defines the character strings EJECT, VTAB and BEEP.

END is a directive to the compiler indicating the end of the compilation process. Every CPL program must contain an END which must be the last statement in the program.

> NOTE: A syntax error will be generated if the END statement is omitted.

END also generates an "execution record" which indicates that the program is completely loaded and that execution should begin. The command also specifies the address of the ENTRY statement. If any storage areas were created by the compiler for subtotals in integer expressions, these areas are defined as part of the code generated by the END statement. Additionally, areas for all literals and remainders are defined.

EJECT - Top-of-Form Command
EJECT is defined as a 1-character string variable containing the ASCII form feed character (hexadecimal 8C). This form feed character causes top-of-form when sent to a printer; it causes the screen to be cleared when sent to the CRT and places the cursor at home position. EJECT is generally used with the WRITE command or with MSG, an external subroutine.

VTAB - Vertical Tab Control
VTAB is defined as a 1-character string variable containing the ASCII vertical tab character (hexadecimal 8B). This vertical tab character causes a printer to space to a predetermined position. This position is set by the printer's Vertical Format Unit.

> Note: Not all Centurion printers are equipped with V.F.U. The VTAB option is not operational in these instances.

BEEP - Bell Character
BEEP is defined as a 1-character string variable containing the ASCII bell character (hexadecimal 87). This character is used to create a sound by some CRTS/printers.

III.4.2 Command Format

END

III.4.3 Cautions

1. The END command, unlike the STOP command, will not terminate program execution. END is used to indicate termination of the compiler process. For additional information see STOP (III.3).

2. When EJECT is used to generate a form feed, it must be the only item on a line. Data following EJECT in this instance will be lost.

Chapter Four - COMPILER DIRECTIVES

OVERVIEW

Compiler directives are not part of the CPL program. They are commands to the compiler and generate no code.

IV.1 TITLE

IV.1.1 Usage

The TITLE command is used to print a title (or phrase) at the top of each page of the program listing.

The command issues a form-feed to the output listing before printing the title. This title is then printed at the top of every page until another TITLE command is found. If the first statement is not a TITLE command, a default title composed of blanks is used.

Although TITLE is usually the first command in a program, the command may appear anywhere in the program as many times as needed. In lengthy programs additional TITLE commands may be included to indicate subsections of the program.

IV.1.2 Command Format

```
TITLE 'title'
```

title - information contained in a 1-78 character string which is to appear at the top of each page.

IV.1.3 Cautions

1. TITLE does not place a title on the report generated by the program. It is used to place a title on a source listing.

IV.2 DIRECT/CPL

IV.2.1 Usage

The DIRECT command indicates that subsequent commands in a CPL program will be in assembly language. The CPL command indicates a return to CPL language.

DIRECT

DIRECT sets a flag in the compiler which causes subsequent commands to be passed directly to the compiler in assembler source code. While these commands are not translated, they are read by the compiler.

> Note: The DIRECT command also generates a PRINT ON command. For additional information see PRINT ON/PRINT OFF, COM/PRINT OFF (IV.3).

CPL

The CPL command turns off the flag in the compiler set by DIRECT and the compiler resumes operation.

> Note: The CPL command also generates a PRINT OFF, COM command. For additional information see PRINT ON/PRINT OFF, COM/PRINT OFF (IV.3).

IV.2.2 Command Format

```
DIRECT
  or
CPL
```

IV.2.3 Cautions

1. When programming in assembly language on the CPU-4 or CPU-5 errors may affect not only the surrounding program but all partitions on the system.

2. Assembler coding may be added to a CPL program using the DIRECT/CPL option. Inclusion of assembly language is not a recommended procedure, however.

3. ESP is an obsolete predecessor of the CPL command. While it is no longer used, it may be referenced in some programs.

IV.3 PRINT ON/PRINT OFF,COM/PRINT OFF

IV.3.1 Usage

The PRINT ON command notifies the compiler to display the generated assembler statement; the PRINT OFF, COM command allows the printing of CPL statements only (i.e. the normal print situation); the PRINT OFF command results in compilation with nothing being printed.

NOTE: CPL commands appear in the compiler listing preceded by an asterisk (*).

There is no limit to the number of times PRINT ON/PRINT OFF, COM/PRINT OFF may be used. These commands may occur anywhere in the program, as many times as needed.

IV.3.2 Command Format

```
PRINT ON
  or
PRINT OFF [,COM]
```

IV.3.3 Cautions

1. An average of ten or more lines of assembly language is generated for each CPL command. The use of PRINT ON can cause the compiler listing to be extremely lengthy.

IV.4 PAGE EJECT/EJECT

IV.4.1 Usage

The PAGE EJECT/EJECT command causes the compiler to issue a top-of-form command to the device/file to which the source listing is being output. PAGE EJECT/EJECT affects program listings only and is not a method to cause page ejection during program execution. For additional information on page ejection during program execution see WRITE (XI.3).

IV.4.2 Command Format

```
PAGE EJECT
  or
EJECT
```

IV.5 SPACE

IV.5.1 Usage

The SPACE command causes the compiler to insert blank lines in the program listing.

IV.5.2 Command Format

SPACE n

n - number of blank lines to be added to the program listing.

IV.6 COPY

IV.6.1 Usage

The COPY command allows specific portions of code to be joined into a source program from a file outside that program. The specified code is inserted into the program during compilation at the location of the COPY statement.

NOTE: On the CPU-5 copy member processing is done exclusively by CLPRP, a separate utility program. Execution of CLPRP is built into the jobstreams S.CPL/S.SCPL. On the CPU-6 copy member processing is done by the CPL compiler.

IV.6.2 Command Format

COPY label [SYSn]

label - name assigned to that portion of the library to be joined to the source program;

Note: The label must be 6 characters in length and may be composed of any ASCII character. Trailing blanks are allowed; however, the first character of the label may not be a blank.

SYSn - specified logical unit number to which the library is assigned.

Subfile

If a subfile is used, it may be assigned to any logical unit from SYS2 through SYS15; SYS2 is the default. Multiple copy libraries may be assigned by using more than one of the logical units mentioned above.

Note: If more than one copy library is included in a program, it must be preassigned before initiating P.CPL. Additionally, SYSn must be included. If only one copy library is included, SYS2 is the default; SYSn need not be included.

IV.6.3 Cautions

1. If COPY statements are part of an application, the appropriate copy libraries must be available.

2. A COPY statement is used for file layouts, record layouts and small subroutines only.

3. To insure CPU-5/CPU-6 compatibility when using COPY statements, use a discrete Type "A" file.

IV.7 Comment Line

IV.7.1 Usage

The semicolon (;) functions as the comment character. Inclusion of ";" as the first non-blank character on a line allows commentary and blank lines to be added to the source listing. Comments are output to the listing by the compiler, but are not translated.

The maximum size of a comment line depends upon the line in which it occurs. When ";" occurs after a CPL command, output will begin with column 38 on the compiled listing. If the first non-blank character on a line is a comment character, that line will be printed verbatim.

Comments do not add to the size of a program.

IV.7.2 Command Format

```
;[comment]
  or
[label:] [statement] ;[comment]
  or
[label:] ;[comment]
```

comment - text which will be output to the listing;
label - name assigned to any CPL statement;
statement - any CPL command.

IV.8 Continuation Line

IV.8.1 Usage

The hyphen (-) functions as the continuation character. Generally used with string data, inclusion of a "-" as the last item on a line of CPL code allows that line to be divided between two or more records of the source file.

When the CPL compiler reads from the source file, it transfers deblanked data into a 300 byte buffer area. If a "-" (followed by carriage return or comment character) is encountered, the next record from the source will be used as input to this buffer.

This transfer process continues until a normal terminator (i.e. a carriage return or comment character not preceded by a hyphen) is found or the 300 byte buffer area is filled. There is no limit on the number of continuation lines used within a program, as long as the 300 character maximum is not exceeded.

> NOTE: This limit does not include comments or blanks unless the blanks are included in a character string.

IV.9 Reverse Slash

IV.9.1 Usage

The reverse slash (\) allows multiple CPL statements to be coded on the same source code line by inserting a "\" between those statements.

IV.9.2 Cautions

1. ELSE must always be the first item on a line. It may never be used within a multiple statement unless it is the first item on the line.

Chapter Five - PROGRAM LINKAGE

OVERVIEW

Commonly used routines are often stored in a linker library. These external routines may be linked to other programs; key factors in this linkage process are "externals" and "entrypoints".

An external is an address in one program or external routine which is to be linked to another program. An entrypoint is an address logically in one program/external routine, but accessible from another program/external routine which has defined it as an external.

As long as they follow the SYSTEM statement, EXTERNAL and ENTRYPOINT commands may be located anywhere in a CPL program. Anything in CPL which has a label (i.e. data area, subroutine, I/O buffer, etc.) may be an external or entrypoint and there is no limit to the number that may be added to a program. Since both commands are actually compiler directives, they are not executable commands and do not reserve memory.

> NOTE: While EXTERNAL/ENTRYPOINT commands may be located at any point in the CPL program, readability is improved by grouping the commands.

Linkage of external routines to another program is accomplished by the linker-edit process. The utility program XLINK, run by CPL jobstreams S.CPL (CPU-5) or P.CPL (CPU-6), is utilized for this process.

V.1 EXTERNAL

V.1.1 Usage

The EXTERNAL command notifies the compiler that labels used within a program are defined as labels outside that program.

An EXTERNAL command does not increase program length.

V.1.2 Command Format

EXTERNAL label [,label, label...]

label - name of the external routine or entrypoint within an external routine.

Note: Although the label may be 1-255 characters in length, the linkage editor uses only the first 6. An EXTERNAL label may contain more than 6 characters, but these first 6 must be unique.

V.1.3 Cautions

1. The linkage editor will not link the label specified in an external command unless:

   a. the name of the label containing the entry-point is specified as an external;
   b. both the header and entrypoint are referenced within the program;
   c. the "name" on the SYSTEM statement (i.e. the external name of the program) is both externalized and referenced.

   Note: This last provision is valid only when a SYSTEM statement containing the MAIN keyword is to be linked to subroutines/subprograms.

   For additional information see ENTRYPOINT (V.2) and SYSTEM (III.1).

V.2 ENTRYPOINT

V.2.1 Usage

The ENTRYPOINT command notifies the compiler that labels which may be used as externals in other routines have been defined within the program.

The program name on the SYSTEM statement is automatically an entrypoint and does not have to be declared as such.

> NOTE: @CPL is the default if program name is not specified in a SYSTEM statement. If label names are not specified, the default will generate a variety of relocatable modules labeled ENTRYPOINT @CPL; this produces uncertain results in the assignment of externals and entrypoints. For additional information see SYSTEM (III.1).

An ENTRYPOINT command does not increase program length.

V.2.2 Command Format

ENTRYPOINT label [,label, label...]

label - specifies an address that may be referenced by an external program.

Note: Although the label may be 1-255 characters in length, the linkage editor uses only the first 6. An EXTERNAL label may contain more than 6 characters but these first 6 must be unique.

V.2.3 Cautions

1. The linkage editor will not link the label specified in an ENTRYPOINT command unless:

   a. the name of the label containing the entrypoint is specified as an external;
   b. both the header and entrypoint are referenced within the program;

c. the "name" on the SYSTEM statement (i.e. the external name of the program) is both externalized and referenced.

   Note: This last provision is valid only when a SYSTEM statement containing the MAIN keyword is to be linked to subroutines/subprograms.

For additional information see EXTERNAL (V.1) and SYSTEM (III.1).

Chapter Six - RESERVING MEMORY

OVERVIEW

An executable program occupies a block of system memory when that program is loaded. This block of memory contains statements which reserve data areas such as buffers, tables, integer strings and character strings. The executable form of program commands is also stored in this memory block.

Space must be reserved for all the areas mentioned above. Initial values may be assigned to integers and strings or a specific number of bytes may be reserved without assigning initial value.

VI.1 INTEGER

VI.1.1 Usage

The INTEGER command reserves an area of memory for a 4- or 6-byte integer variable. The command may also be used to pass data from one overlay to another. INTEGER, however, does not initialize the variable; it only reserves storage space and assigns a name.

> NOTE: INTEGER may be used on the CPU-5 to pass data from one program to the next since the partition stays in the same memory.

The INTEGER command may be located anywhere in a program so long as it follows SYSTEM. A program may contain as many INTEGER statements as necessary.

> NOTE: While INTEGER statements may occur at any point in a CPL program, it is best to locate them in one area separated from the logic section of a program.

Each INTEGER command increases program length. A 4-byte integer increases the program by 4 bytes; a 6-byte integer increases the program by 6 bytes.

VI.1.2 Command Format

```
INTEGER label [,label, label, ...]
```

label - name to be assigned to an integer variable.

The difference between a 4- or 6-byte integer is based on the name assigned to it. Integer names beginning with a "?", are treated as 6-byte integers. Integer names beginning with any other alphanumeric character are treated as 4-byte integers.

VI.1.3 Cautions

1. Whatever is in memory when a program loads is the value of the integer. INTEGER does not automatically clear a memory area to zero (0).

VI.2 SET

VI.2.1 Usage

The SET command reserves an area of memory for a 4- or 6-byte integer. SET also assigns an initial value to the specified integer.

The SET command may be located anywhere in a program so long as it follows SYSTEM. A program may contain as many SET statements as necessary.

> NOTE: While SET statements may occur at any point in a CPL program, it is best to locate them in one area separated from the logic section of a program.

Each SET command increases program length. A 4-byte integer increases the program by 4 bytes; a 6-byte integer increases the program by 6 bytes.

VI.2.2 Command Format

```
SET label : n [,label:n, label:n, ...]
```

label - name assigned to the integer;
n - initial literal value.
Note: The value assigned to a label in a SET statement must be a numeric value; expressions are not allowed. This value may be changed at any time during program execution.

The difference between a 4- or 6-byte integer is based on the name assigned to it. Integer names beginning with a "?", are treated as 6-byte integers. Integer names beginning with any other alphanumeric character are treated as 4-byte integers.

VI.2.3 Cautions

1. Unlike INTEGER, the SET command does not allow a value to be left in memory from one overlay to the next. For additional information see INTEGER (VI.1).

VI.3 STRING

VI.3.1 Usage

The STRING command reserves an area of memory for character strings of a specified length. STRING does not assign a specific initial value.

The STRING command may be located anywhere in a program so long as it follows SYSTEM. A program may contain as many STRING statements as necessary.

> NOTE: While STRING statements may occur at any point in a CPL program, it is best to locate them in one area separated from the logic section of a program.

Variable Length Strings

All strings in CPL are "variable length strings" consisting of characters terminated by one byte of binary zero (i.e. the string terminator). A string may be defined according to its "memory length" (i.e. the maximum number of characters assigned to a string) or according to its "true length" (i.e. the actual number of character used up to the string terminator). CPL utilizes "true length" by outputting the actual number of characters used, regardless of the size of the assigned memory length.

Each STRING statement increases program length by the total number of characters in a string, plus an additional 1 byte for the terminator of each string.

VI.3.2 Command Format

```
STRING label (n) [,label (n), label (n), ...]
```

label - name of the character string to be reserved;

(n) - number of character in the character string.

Note: A null string of zero characters may be established. It is used to clear other strings and output blank lines.

VI.3.3 Cautions

1. The length of a string is not checked before a new value is assigned. If the new value exceeds the length at which the string was established, the string will be extended into the next area of memory.

2. Whatever is in memory when a program loads is the value of the string. STRING does not automatically clear a memory area to 0 (zero).

VI.4 DEFINE

VI.4.1 Usage

The DEFINE command reserves an area of memory for a character string. The command also assigns an initial value to that character string.

The DEFINE command may be located anywhere in a program so long as it follows SYSTEM. A program may contain as many DEFINE statements as necessary.

> NOTE: While DEFINE statements may occur at any point in a CPL program, it is best to locate them in one area separated from the logic section of a program.

Each DEFINE command increases program length by the total length of all strings, plus an additional 1-byte for the terminator of each string.

VI.4.2 Command Format

```
DEFINE label:'string' [,label:'string', label:
  'string', ...]
```

label - name of the character string to be defined;

'string' - initial value to be assigned to the label.

All alphabetical characters in the string are output as upper case letters. Spaces and special characters are allowed. Single quotation marks are not permitted, however, since they are used as a delimiter for the string. For additional information on the use of quotation marks within a string, see section II.1 (String Literal).

VI.4.3 Cautions

1. The length of a string is not checked before a new value is assigned. If the new value exceeds the length at which the string was established, the string will be extended into the next area of memory.

VI.5 BUFFER

VI.5.1 Usage

The BUFFER command reserves an area of memory to be used as a temporary holding area during input or output. An area defined by a BUFFER statement can only be used as an I/O buffer; it must not be used as a string by the program.

The BUFFER command may be located anywhere in a program so long as it follows SYSTEM. A program may contain as many BUFFER statements as necessary.

> NOTE: While BUFFER statements may occur at any point in a CPL program, it is best to locate them in one area separated from the logic section of a program.

The minimum byte size for a disk file buffer is 1 sector (400 bytes). Two bytes added to the beginning of the buffer contain the buffer length; a binary zero terminator adds an additional byte.

VI.5.2 Command Format

```
BUFFER label (n) [,label (n), label (n), ...]
```

label - name assigned to the area of memory to be reserved;
n - number of bytes in the buffer.
Note: Although a larger or smaller buffer size may be used, "n" should always be stated and equal 400.

VI.6 TABLE

VI.6.1 Usage

The TABLE command reserves an area of memory for a table. It establishes a work area through which information is passed to and from a table.

The TABLE command may be located anywhere in a program so long as it follows SYSTEM. A program may contain as many TABLE statements as necessary.

> NOTE: While TABLE statements may occur at any point in a CPL program, it is best to locate them in one area separated from the logic section of a program.

Program length of a table composed of 4-byte integers is (n + 1) * 4 + 2; program length of a 6-byte integer table is (n + 1) * 6 + 2; program length of a table composed of strings is n + len + 3.

VI.6.2 Command Format

Integer Table

```
TABLE name (n) [,name (n), name (n), ...]
```

name - area through which an entire table of integers is accessed;
n - total number of integers in a table.

String Table

```
TABLE name (len,n) [,name (len,n), name (len,n)...]
```

name - area through which an entire table of strings is accessed;
len - length of a string in the table;
n - total number of characters in a table.

The difference between a 4- or 6-byte table name composed of integers is based on the name assigned to it. A name beginning with a "?", is treated as a table of 6-byte integers. A name beginning with any other alphanumeric character is treated as a table of 4-byte integers. For additional information on tables see Subscripted Variables (XIV.8), TBLGET (XIV.9), and TBLPUT (XIV.10).

VI.6.3 Cautions

1. Whatever is in memory when a program loads is the value of the table. A table will normally need to be initialized prior to execution.

Chapter Seven - CPL ASSIGNMENT STATEMENTS

OVERVIEW

Utilizing the equal sign (=) as a substitution command, assignment statements are used to insert the value of an expression into a variable. The target of the substitution command may be an integer, multiple integers or a string.

VII.1 Integer Assignment Statement

VII.1.1 Usage

An integer assignment statement is used to change the value of an integer during execution of a CPL program. Any complexity of expression is allowed with CPL evaluating the expression and storing it in the variable name.

Program size depends upon the complexity of the integer assignment statement. To determine program size, the following should be noted:

Mathematical Processing
Mathematical operations are performed from left to right. If parentheses are present, resolution of the equation begins with the innermost pair and moves outward.

If the first character inside a left-hand parenthesis is a minus (-), the remaining characters of the operand must be numeric. Similarily, if the first character after the last equal sign is a minus (-), the remaining characters of the next operand must be numeric.

If more than one integer is to be assigned a new value by a single integer assignment statement, no operators other than the equal sign (=) may be used to the left of the last equal sign.

If an integer variable appears on both sides of the equal sign, the equation is calculated using the current value of the integer variable. The value of the equation is then stored in that integer variable.

On the CPU-6 system, the remainder of the last division performed is saved. If the divisor of that division is a 4-byte integer, then the remainder is a 4-byte integer and that remainder is stored in @REM. If the divisor is a 6- or 8-byte integer, the remainder is a 6-byte integer stored in ?@REM.

VII.1.2 Command Format

[...variable =] variable = expression

variable - name of the integer variable (defined in a CPL INTEGER or SET statement) to be assigned a new value;

Note: There is no limit to the number of variables that may be included to the left of the equal sign (=). Additionally, the variable may be subscripted. For additional information see Subscripted Variables (XIV.8).

expression - name of an expression made up of integers, literals, and/or operators.

The target of an integer assignment statement may be a 4- or 6-byte integer variable.

The "expression" in an integer assignment statement may be composed of 1) variable names, 2) integer literals, 3) the addition, subtraction, multiplication or division operators, 4) parenthetical expressions (to a maximum of 16 levels) and 5) built-in functions (i.e. ABS, LEN, MAX, MIN, MOD, ROUND, SGN).

NOTE: Integer math is performed in all instances; no remainders are carried, nor are decimals processed.

STATUS

When working with an integer assignment statement, STATUS=0 indicates no overflow/underflow; STATUS=2 indicates overflow/underflow.

Note: Division by 0 is considered a STATUS=2.

If STATUS=2 occurs, nothing has been stored in the specified integer variable and its contents remain unchanged. STATUS=2 may be caused by the expression becoming too large to process during evaluation or by the expression becoming to large to store in the specified integer after processing.

VII.2 String Assignment Statement

VII.2.1 Usage

A string assignment statement is used to change the value of a character string during execution of a CPL program.

String assignment statements increase the size of a program by 7 bytes. An additional 2 bytes must be added for each variable or literal present on the right side of the equation.

VII.2.2 Command Format

```
'string' = 'value a'
  or
'string' = "value b"
```

string - name of the string variable (defined in a CPL STRING or DEFINE statement) to which a new value is to be assigned;

'value a' - name of string or variable separated by the plus sign (+);

"value b" - name of string, variable and/or literal(s) separated by the plus sign (+).

Note: The plus sign (+) indicates string concatenation.

VII.2.3 Cautions

1. The length of a string is not checked before a new value is assigned during string concatenation. If a new value exceeds the length at which the string was established in the STRING or DEFINE statement, the string will extend into the next area of memory. For additional information see STRING (VI.3) and DEFINE (VI.4).

2. The CPU-6 system provides partition protection and keeps the program from reading or writing outside the partition. The CPU-5 system, how-

2. (cont.)
ever, offers no such protection. An undetected overflow on the CPU-5 can destroy the partition and eventually disrupt the OPSYS.

3. If the same string variable is included on both sides of an equation, it must be the first item in the string equation.

4. String concatenation does not allow multiple equates. Only one equal sign (=) may appear in each string equation.

5. The string equation must be enclosed in either single (') or double (") quotation marks. If quotation marks are omitted, the operating system assumes both terms to be integers. For additional information see Literal (II.1) and Expression (II.3).

VII.3 INCREMENT/DECREMENT

VII.3.1 Usage

The INCREMENT (INCR) command increases the value of a specified integer; DECREMENT (DECR) decreases the value of a specified integer.

Either command increases program size by 15 bytes. This increase is constant if either 4- or 6-byte integers are used or if the abbreviated form of the commands are used.

VII.3.2 Command Format

```
INCR[EMENT] integer [,n]
  or
DECR[EMENT] integer [,n]
```

integer - name of a variable (defined in a CPL INTEGER or SET statement) whose value is to be changed;

n - literal to be added or subtracted.

Note: A literal specifying the number to be added or subtracted may be part of the command format. The default value is 1.

Chapter Eight - FUNCTIONS

OVERVIEW

Built-in functions are subroutines which may be used in place of an integer variable in an integer expression. A built-in function reference consists of the function-name followed by a left parenthesis, one or more arguments and a right parenthesis. Each argument may be an integer expression containing built-in functions.

VIII.1 ABS

VIII.1.1 Usage

The ABS command evaluates the expression and returns the absolute value of that expression.

An example of the ABS command would be:

```
A = B + ABS(c)
```

Each ABS command adds 8 bytes to program length.

VIII.1.2 Command Format

```
ABS (arg)
```

arg - integer expression/integer literal.

VIII.2 LEN

VIII.2.1 Usage

The LEN command returns the true length of data contained in a string (in characters or bytes), not including the terminator.

An example of the LEN command would be:

```
IF (LEN(STR1).EQ.0) 'STR1' = 'STR2'
```

Each LEN command adds 8 bytes to program length.

VIII.2.2 Command Format

```
LEN (arg)
```

arg - string variable name.

VIII.3 MAX

VIII.3.1 Usage

The MAX command returns the highest value in a specified set.

An example of the MAX command would be:

```
A = MAX (A, B, C + D)
```

Each MAX command adds 1 byte to program length, plus an additional 10 bytes per "argn".

VIII.3.2 Command Format

```
MAX (arg1, arg2, ...)
```

argn - integer expression variable or literal.

VIII.4 MIN

VIII.4.1 Usage

The MIN command returns the lowest value in a specified set.

An example of the MIN command would be:

```
IF (MIN (A*B,C).GT.10) GO TO NEXT
```

Each MIN command adds 1 byte to program length, plus an additional 10 bytes per "argn".

VIII.4.2 Command Format

```
MIN (arg1, arg2, ...)
```

"argn" - integer expression variable or literal.

VIII.5 MOD

VIII.5.1 Usage

The MOD command returns the remainder after a division operation in integer math.

An example of the MOD command would be:

```
A = MOD (B+1,C)
```

Each MOD command adds 20 bytes to program length.

VIII.5.2 Command Format

```
MOD (arg1, arg2)
```

arg1 - integer expression variable or literal to be divided by arg2;
arg2 - integer, expression variable or literal to be divided into arg1.

VIII.6 ROUND

VIII.6.1 Usage

The ROUND command eliminates the designated number of decimal places by dividing "arg1" by 10**arg2. This rounding process is performed by truncation of the low order decimal places. If the highest order digit truncated was 5 or greater, then the lowest remaining digit is increased by 1. For example:

```
A=ROUND  (123456,2)  = 1235

or

A=ROUND  (1234,1) = 123
```

Each ROUND command adds 20 bytes to program length.

VIII.6.2 Command Format

```
ROUND  (arg1, arg2)
```

arg1 - expression to be rounded;
arg2 - expression representing the number of decimal places to be eliminated.

VIII.6.3 Cautions

1. The "rounding" which occurs with ROUND is not similar to that involved in truncation or with other forms of rounding.

VIII.7 SGN

VIII.7.1 Usage

The SGN command is used to determine the sign of the expression/literal. When SGN is used, a value of +1, 0 or -1 is returned depending upon whether the value of the expression/literal is positive, zero or negative.

An example of the SGN command would be:

```
IF (SGN (A-B+C).EQ.1) D=0
```

VIII.7.2 Command Format

```
SGN (arg)
```

arg - integer expression variable or literals.

Chapter Nine - TRANSFER OF CONTROL

OVERVIEW

The operating system normally executes CPL commands in sequential order. There are times, however, when this is not desirable and it is necessary to transfer control from one section of a program to another.

Program control can generally be transferred only between points which have previously been identified by a program label. Program labels either identify a paragraph of code (i.e. label followed by a colon) or identify the beginning of a subroutine (i.e. label preceded by a SUBROUTINE command). In the case of the various IF commands, program control may be transferred within the IF statement without benefit of labels.

> NOTE: A label may be the sole item on a line of CPL. Inclusion of labels in this manner is a means of organizing a program and increasing readability.

A subroutine is an isolated section of code used repeatedly within the same program. It may also be an external subroutine which is located outside the program and linked to it after assembly.

IX.1 Labels

IX.1.1 Usage

Program labels (or program connection points) provide a means of transferring control between various portions of a program.

The compiler creates program labels which begin with "@". These program labels should not be used by the CPL programmer.

IX.1.2 Command Format

label:

label - program label, 1-255 characters in length.

Label Name Formation Rules governing the formation of label names are as follows:

1. The initial character in a program label must be an alphabetical character, "?" or "@".

   Note: An initial "?" indicates a 6-byte integer name; an initial "@" usually indicates a label generated by the compiler.

2. The remaining characters in the label may be alphanumeric, "?" or "@".
3. The final character in the label must be followed by a colon (:).

IX.1.3 Cautions

1. A label name may not begin with the characters of a CPL command. The label SETUP: would be illegal since it incorporates the SET command.

IX.2 GO TO

IX.2.1 Usage

The GO TO command specifies a label within the program to which control will be transferred. The specified label may be located within or outside of a program. When a GO TO occurs, the logic flow will proceed to the specified label. Execution begins either with the statement on that label or, if the label contains no statement, with the statement following that label.

The GO TO command may be conditional or unconditional. In a conditional GO TO an expression is evaluated and, depending upon the value of the expression, is transferred to one of the specified labels.

> NOTE: If the value of the specified expression is 1 at the time the statement is executed, control will be transferred to the first of the labels. If the value is 2, the second label will be used, etc. If the value of the expression is negative, zero (0) or greater than the total number of labels, execution will continue with the next sequential statement.

In an unconditional GO TO program control is transferred to a specified label and no evaluation takes place.

The number of bytes added to program length by a conditional GO TO depends upon the complexity of the expression contained within the command format. An unconditional GO TO adds 3 bytes to program length.

IX.2.2 Command Format

Conditional GO TO

GO TO (label, label, ...) ON expression

Unconditional GO TO

GO TO label

label - program label to which control is to be transferred;
expression - an integer expression.

IX.3 LOOP

IX.3.1 Usage

The LOOP command permits multiple executions of a group of statements.

The LOOP command adds 17 bytes to program length.

IX.3.2 Command Format

LOOP j (a, b, c)

- j - 4-byte integer or literally-subscripted 4-byte table name;
- a - 1- or 4-byte literal, 4-byte integer or a literally-subscripted 4-byte table name;
- b - 1- or 4-byte literal, 4-byte integer or a literally-subscripted 4-byte table name;
- c - 1- or 4-byte literal, 4-byte integer or a literally-subscripted 4-byte table name.

The LOOP command functions in the following manner:

1. The "j" value is set equal to the "a" value.
2. The body of the LOOP is executed repeatedly.

   Note: The "body" is the group of CPL statements immediately following the LOOP statement with END LOOP being the last statement in the body.

3. After each execution, "j" is incremented by the value of "c" and compared to the value of "b".
4. When "j" is greater than/equal to "b", the loop is terminated by END LOOP. For additional information see END LOOP (IX.5).
5. Execution resumes with the statement following END LOOP.

IX.3.3 Cautions

1. Although the procedure is not recommended, a GO TO may be used to exit a LOOP statement. For additional information see GO TO (IX.2).

2. LOOP and LOOP WHILE may be embedded to 16 levels within a CPL program; however for proper processing; each LOOP/LOOP WHILE command must be terminated with its own END LOOP. Failure to include an END LOOP will produce unpredictable program results.

IX.4 LOOP WHILE

IX.4.1 Usage

The LOOP WHILE command allows for repeated execution of a group of statements as long as a specified comparison is true.

The number of bytes added to program length by the LOOP WHILE command depends upon the complexity of the expressions contained within the command format.

IX.4.2 Command Format

LOOP WHILE (value operator value)

value - integer, literal or integer expression;
operator - condition of the comparison (i.e. .EQ., .NE., .LT., .LE., .GT. or .GE.).
Note: Periods are required punctuation and no space is allowed between period and operator.

The LOOP WHILE command functions in the following manner:

1. The body of LOOP WHILE is executed repeatedly.

   Note: The "body" is the group of CPL statements immediately following the LOOP WHILE statement with END LOOP being the last statement in the body.

2. After each execution, the two values are compared according to the specified operator.
3. When the specified comparison is false, the loop is terminated by END LOOP. For additional information see END LOOP (IX.5).
4. Execution resumes with the statement following END LOOP.

IX.4.3 Cautions

1. Although the procedure is not recommended, a GO TO may be used to enter/exit a LOOP WHILE. For additional information see GO TO (IX.2).

2. LOOP and LOOP WHILE may be embedded to 16 levels within a CPL program; however for proper processing, each LOOP/LOOP WHILE command must terminate with its own END LOOP. Failure to include an END LOOP will produce unpredictable program results.

IX.5 END LOOP

IX.5.1 Usage

The END LOOP command terminates the body of the LOOP or LOOP WHILE command and indicates that all specified comparisons are concluded.

> NOTE: The "body" of the command is the group of statements immediately following LOOP or LOOP WHILE. For additional information see LOOP (IX.3) and LOOP WHILE (IX.4).

The END LOOP command adds 3 bytes to program length.

IX.5.2 Command Format

```
END LOOP
```

IX.5.3 Cautions

1. LOOP and LOOP WHILE may be embedded to 16 level within a CPL program; however for proper processing, each LOOP/LOOP WHILE command must terminate with its own END LOOP. Failure to include an END LOOP will produce unpredictable program results.

IX.6 CALL

IX.6.1 Usage

The CALL command transfers control to a subroutine which may be located within or outside the program.

> NOTE: If located outside the program, the subroutine must be declared in an EXTERNAL statement. Additionally, if parameters are passed in a CALL statement, they must be retrieved. For additional information see EXTERNAL (V.1) and RETRIEVE (IX.9).

CALL adds 3 bytes to a program. Each parameter passed to the subroutine adds an additional 2 bytes.

IX.6.2 Command Format

CALL subroutine [(name, name, ...)]

subroutine - label identifying the subroutine to which control is to be passed;
name - parameter(s) to be passed;
Note: If an offset is desired, the name plus the number of bytes to be added or subtracted must be included in brackets. For example: [name + n] or [name - n].

IX.6.3 Cautions

1. The system does not differentiate between the subroutine name identified in the CALL statement and the program label (i.e. illegal label usage). Therefore, the subroutine name should not be used with a GO TO statement since this does not store the proper address for the return. For additional information see GO TO (IX.2).

IX.7 SUBROUTINE

IX.7.1 Usage

The SUBROUTINE command sets the beginning of a subroutine contained within the CPL program. In addition, a subroutine may be located within an external module. Such a subroutine, which may be called by a main program, is considered an external subroutine.

> NOTE: In this instance, the subroutine label would need to be referenced by ENTRYPOINT and externalized by the main program. For additional information see EXTERNAL (V.1) and ENTRYPOINT (V.2).

SUBROUTINE defines the label to which control may be transferred by a CALL command. The logic path of the subroutine must begin with the point at which the routine is entered and must terminate with a RETURN /RETURN TO statement. For additional information see CALL (IX.6) and RETURN /RETURN TO (IX.8).

> NOTE: A subroutine is fully recursive and may call itself.

A SUBROUTINE command is used to increase readibility in program documentation. It does not generate code and does not add to program length.

IX.7.2 Command Format

SUBROUTINE label

label - name of the subroutine within the CPL program.

Name Formation

Rules govering the formation of subroutine names are as follows:

1. The initial character in a subroutine name must be an alphabetical character, "?" or "@".

   Note: An initial "@" usually indicates a label generated by the compiler.

2. The remaining characters in the name may be alphanumeric, "?" or "@".

IX.8 RETURN/RETURN TO

IX.8.1 Usage

The RETURN command returns control to the statement following the CALL statement of the subroutine. The RETURN TO command allows a subroutine to return to a point in the program other than to the CALL statement which initiated the subroutine.

Multiple RETURN commands may exist within a subroutine; however, all exits from the subroutine must be through a RETURN command. In addition, one subroutine may call another with RETURN transferring control to the point from which the last subroutine was called.

The RETURN command adds 1 byte to program length; the RETURN TO command adds 4 bytes to program length.

IX.8.2 Command Format

RETURN [TO label]

label - point in the program to which control is to be transferred.

IX.8.3 Cautions

1. Control may be transferred to a completely random address if the RETURN command is ordered from within the main routine rather than from a subroutine.

2. A GO TO should not be used to return a subroutine to a main program. Use of GO TO rather than CALL alters the "program stack" and may result in eventual destruction of the program (CPU-5) or a program abort (CPU-6). For additional information see GO TO (IX.2) and CALL (IX.6).

IX.9 RETRIEVE

IX.9.1 Usage

The RETRIEVE command is used to retrieve parameters (i.e. data) out of the CALL argument list. For additional information see CALL (IX.6).

One RETRIEVE statement must be used for each parameter passed; the statements must be in the same order in which the parameters are passed.

The number of bytes the RETRIEVE command adds to program length is determined by which keyword is used. For additional information see Type Designation (IX.9.2) below.

IX.9.2 Command Format

RETRIEVE (type, location [,location, ...])

type - keyword identifying data being retrieved;
Note: Type includes FILE, FORMAT, CONTROL, EXT, NUMBER, NUM48, STRING and ADDRESS.
location - label within subroutine where parmeter is to be placed.

Type Designation

1. FILE - the parameter being passed is a label which represents the address of a CPL FILE statement. The label of the statement into which the parameter is inserted is the "location". The statement may be READ, WRITE, REWRITE, READB, WRITEB, WRITEN/WRITN, CURP, CURS or CURB. The address will always be inserted into the first file name within the statement.

Note: This form is used with I/O operations.

The file keyword adds 2 bytes to the length of the program, plus an additional 3 bytes for each location.

2. FORMAT - the parameter being passed is a label which represents the address of a FORMAT statement. The label of the statement into which the parameter is being inserted is the "location". The statement may be READ, WRITE,

2. (cont.)
REWRITE, WRITEN/WRITN, DECODE or ENCODE. The address then becomes the FORMAT within the statement.

The FORMAT keyword adds 2 bytes to the length of the program, plus an additional 3 bytes for each location.

3. CONTROL - the parameter being passed is a label which represents the address of a CPL FILE statement. The label of the statement into which the parameter is being passed is the "location". The statement may be HOLD, FREE, OPEN, CLOSE or REWIND. The address will always be inserted into the first file name within the statement.

Note: This form is used with control operations.

The CONTROL keyword adds 2 bytes to the length of the program, plus an additional 3 bytes for each location.

4. EXT - any 2-byte parameter may be passed to the subroutine. The label of the statement into which the parameter is being inserted is the "location". The statement may be ENDFILE, CURSOR or CALL.

Note: In the case of ENDFILE and CURSOR, the address becomes the file name within the statement; with CALL it becomes the first parameter in the list.

Each EXT keyword adds 2 bytes to the length of the program, plus and additional 3 bytes for each location.

5. NUMBER - the parameter being passed is either:

A. the address of a 4-byte integer which contains the positive or negative value to be passed.

B. a 1 or 4-byte positive literal value. "Location" is the name of the integer which will receive the value.

The NUMBER keyword adds 9 bytes to the length of the program, plus and additional 5 bytes for each location.

6. NUM48 - the parameter being passed is either:

   A. the address of a 6-byte integer which contains the positive or negative value to be passed.

   B. a 1 to 6-byte positive literal value. "Location" is the name of the integer which will receive the value.

The NUM48 keyword adds 9 bytes to the length of the program, plus an additional 5 bytes for each location.

7. STRING - the parameter being passed is a label which represents the address of a value. The value may be a string variable or a string literal. The name of the string variable into which the value is being inserted is the "location".

The STRING keyword adds 2 bytes to the length of the program, plus an additional 10 bytes for each location.

8. ADDRESS - any 2-byte parameter may be passed to the subroutine. The value of the parameter is stored in the 2 bytes beginning at "location".

The ADDRESS keyword adds 2 bytes to the length of the program, plus an additional 3 bytes for each location.

Note: The ADDRESS keyword is used when assembler code is contained within the subroutine.

IX.9.3. Cautions

1. RETRIEVE may only be used in a subroutine and not in a main routine. For additional information see CALL (IX.6).

IX.10 IF

IX.10.1 Usage

The IF command will perform a function provided a specified comparison is true. For example:

```
IF (A.EQ.B) D = 100
```

If the specified condition is true, the remainder of the line following the right parenthesis is executed.

If the specified condition is false, the program will continue with the next sequential statement.

> NOTE: If a CALL is specified within an IF statement, program execution (upon return from the subroutine) continues with the statement following IF (or ELSE or END DO in the case of more complex IF statements).
>
> If a GO TO is specified within an IF statement, program control is transferred to that label; there is not necessarily a return to the initiating IF statement.

The number of bytes added to a program by an IF statement depends upon the complexity of the expression(s) contained within the command format. Each ELSE adds an additional 3 bytes to program length.

IF-ELSE

An optional form of the IF command incorporates ELSE followed by a statement. For example:

```
IF (A.LT.C) X = 0
ELSE X = 1
```

If the specified condition in an IF statement containing the ELSE option is true, the remainder of the line following the right parenthesis is executed. Program execution continues with the statement following ELSE.

If the specified condition in an IF statement is false, the remainder of the line containing ELSE is executed.

IF-ELSE DO

An optional form of the IF command incorporates ELSE DO. For example:

```
IF (A.GT.B) A = D
ELSE DO
A = C
GO TO L4
END DO
```

If the specified condition in an IF statement containing the ELSE DO option is true, the remainder of the line following the right parenthesis is executed. Program execution continues with the statement following END DO. For additional information see END DO (IX.12).

If the specified condition in an IF statement containing the ELSE DO option is false, execution continues with the statement following ELSE DO.

IF-DO

The IF-DO option provides for the execution of a group of statements if, and only if, a certain condition is true. For example:

```
IF (K.LT.L) DO
WRITE (CRT, F00)  'DONE'
STOP 0
END DO
```

The group of statements to be executed is terminated by an END DO statement (IX.12).

If the specified condition in an IF statement containing the DO option is TRUE, the group of statements following IF-DO is executed.

If the specified condition in an IF statement containing the DO option is FALSE, execution continues with the statement following END DO (IX.12).

IF-DO-ELSE

An optional form of the IF-DO command incorporates ELSE followed by a statement. For example:

```
IF (D.EQ.F) DO
A = B
GO TO L4
END DO
ELSE A = 0
```

If the specified condition in an IF-DO statement containing the ELSE option is true, execution continues from the statement following IF-DO to the END DO (IX.12). Control is then transferred to the statement following the ELSE statement.

If the specified condition in an IF-DO statement containing the ELSE option is false, the remainder of the line containing the ELSE is executed.

IF-DO-ELSE DO

An optional form of the IF-DO command incorporates ELSE-DO. For example:

```
IF (D.EQ.F) DO
A = B
GO TO L4
END DO
ELSE DO
A = C
GO TO L5
END DO
```

If the specified condition in an IF-DO statement containing the ELSE-DO option is true, execution continues from the statement following IF-DO to the END DO associated with that IF-DO. Control is then transferred to the statement following the END DO (IX.12) associated with the ELSE DO.

If the specified condition in an IF-DO statement containing the ELSE-DO option is false, execution continues with the statement following ELSE-DO.

IF-Null-ELSE

An optional form of the IF command incorporates null ELSE. The null ELSE option does not contain a statement. For example:

```
IF (A.EQ.B) X = 0
ELSE
```

If the specified condition in an IF statement containing the null ELSE option is true, the remainder of the line following the right parenthesis is executed.

If the specified condition in an IF statement containing the null ELSE option is false, execution continues with the line below the ELSE statement.

Note: The null ELSE form only has meaning in a "nested" (or complex) IF statement.

IF-DO-Null-ELSE

An optional form of the IF-DO command incorporates a null ELSE. The null ELSE option does not contain a statement. For example:

```
IF (E.GT.F) DO
A = B
GO TO L4
END DO
ELSE
```

If the specified condition in an IF-DO statement containing the null ELSE option is true, execution continues from the statement following IF-DO to the END DO (IX.12). Control is then transferred to the statement following ELSE.

If the specified condition in an IF-DO statement containing the null ELSE option is false, execution continues with the statement following ELSE.

Note: The null ELSE form only has meaning in a "nested" (or complex) IF statement.

IF(x)
All of the preceding IF forms may be used with the IF(x) form utilizing the specifications previously cited.

IF(x) is modification of the IF command. The IF(x) form is equivalent to IF(x.NE.0) and eliminates the need to code the operator and the value contained in the body of the IF statement.

Note: IF(x) is valid only when x is a 4-byte integer variable.

IX.10.2 Command Format

IF
IF (value operator value) action

IF-ELSE
IF (value operator value) action
ELSE action

IF-ELSE DO
IF (value operator value) action
ELSE DO
actions
END DO

IF-DO
IF (value operator value) DO
actions
END DO

IF-DO-ELSE
IF (value operator value) DO
actions
END DO
ELSE action

IF-DO-ELSE DO
IF (value operator value) DO
actions
END DO
ELSE DO
actions
END DO

IF-Null-ELSE
IF (value operator value) action
ELSE

IF-DO-Null-ELSE
IF (value operator value) DO
actions
END DO
ELSE

IF(x)
IF (value) action

value - 4- or 6-byte integer variable, literal or integer expression;
Note: In the IF(x) form, "value" must be a 4-byte integer variable.

operator - condition of the comparison (i.e. .EQ., .NE., .LT., .LE., .GT., or .GE.);
Note: Periods are required punctuation and no space is allowed between period and operator.

action - any executable CPL statement(s) on a single line;
Note: The reverse slash (\) may be used if multiple statements are to be coded on the same line. For additional information see Reverse Slash, (IV.9).

actions - any executable CPL statement(s) on one or more lines.

IX.10.3 Cautions

1. The "action" portion of an ELSE may not be an IF or IFSTRING command. For additional information see Command Format, IX.10.2.

IX.11 IFSTRING/IFS

IX.11.1 Usage

The IFSTRING/IFS command will perform a function provided a specified comparison is true. For example:

```
IFSTRING (S.EQ.T) D = 100
IFS (S.EQ.T) D = 100
```

NOTE: IFS is the abbreviated form of IFSTRING. Usage does not increase or decrease program length.

If the specified condition is true, the remainder of the line following the right parenthesis is executed.

If the specified condition is false, the program will continue with the next sequential statement.

NOTE: If a CALL is specified within an IFSTRING/IFS statement, program execution (upon return from the subroutine) continues with the statement following IFSTRING/IFS (or ELSE or END DO in the case of more complex IFSTRING/IFS statements).

If a GO TO is specified within an IFSTRING/IFS, program control is transferred to that label; there is not necessarily a return to the initiating IFSTRING/IFS statement.

The IFSTRING/IFS command adds 10 bytes to the length of a program. Each ELSE adds an additional 3 bytes to program length.

IFSTRING-ELSE
An optional form of the IF command incorporates ELSE followed by a statement. For example:

```
IFSTRING (S.LT.T) X = 0
ELSE X = 1
```

If the specified condition in an IFSTRING statement containing the ELSE option is true, the remainder of the line following the right parenthesis is executed. Program execution continues with the statement following ELSE.

If the specified condition in an IFSTRING statement is false, the remainder of the line containing ELSE is executed.

IFSTRING-ELSE DO

An optional form of the IFSTRING command incorporates ELSE DO. For example:

```
IFSTRING (S.GT.T) A = D
ELSE DO
A = C
GO TO L4
END DO
```

If the specified condition in an IFSTRING statement containing the ELSE DO option is true, the remainder of the line following the right parenthesis is executed. Program execution continues with the statement following END DO. For additional information see END DO (IX.12).

If the specified condition in an IFSTRING statement containing the ELSE DO option is false, execution continues with the statement following ELSE DO.

IFSTRING-DO

The IFSTRING-DO option provides for the execution of a group of statements if, and only if, a certain condition is true. For example:

```
IFSTRING (S.LT.T) DO
WRITE (CRT, F00) 'DONE'
STOP 0
END DO
```

The group of statements to be executed is terminated by an END DO statement (IX.12).

If the specified condition in an IFSTRING statement containing the DO option is true, the group of statements following IF-DO is executed.

If the specified condition in an IFSTRING statement containing the DO option is false, execution continues with the statement following END DO (IX.12).

IFSTRING-DO-ELSE

An optional form of the IFSTRING-DO command incorporates ELSE followed by a statement. For example:

```
IFSTRING (S.EQ.T) DO
A = B
GO TO L4
END DO
ELSE A = 0
```

If the specified condition in an IFSTRING-DO statement containing the ELSE option is true, execution continues from the statement following IFSTRING-DO to the END DO (IX.12). Control is transferred to the statement following the ELSE statement.

If the specified condition of an IFSTRING-DO statement containing the ELSE option is false, the remainder of the line containing the ELSE is executed.

IFSTRING-DO-ELSE DO

An optional form of the IFSTRING-DO command incorporates ELSE-DO. For example:

```
IFSTRING (S.EQ.T) DO
A = B
GO TO L4
END DO
ELSE DO
A = C
GO TO L5
END DO
```

If the specified condition in an IFSTRING-DO statement containing the ELSE-DO option is true, execution continues from the statement following IFSTRING-DO to the END DO (IX.12) associated with that IFSTRING-DO. Control is then transferred to the statement following the END DO associated with the ELSE DO.

If the specified condition of an IFSTRING-DO containing the ELSE-DO option is false, execution continues with the statement following ELSE-DO.

IFSTRING-Null-ELSE

An optional form of the IFSTRING command incorporates a null ELSE. The null ELSE option does not contain a statement. For example:

```
IFSTRING (S.GT.T) A=D
ELSE
```

If the specified condition in an IFSTRING statement containing the null ELSE option is true, the remainder of the line following the right parenthesis is executed.

If the specified condition in an IFSTRING statement containing the null ELSE option is false, execution continues with the line following the ELSE statement.

Note: The null ELSE form only has meaning in a "nested" (or complex IF) statement.

IFSTRING-DO-Null-ELSE

An optional form of the IFSTRING-DO command incorporates a null ELSE. The null ELSE option does not contain a statement. For example:

```
IFSTRING (S.GT.T) DO
A = B
GO TO L4
END DO
ELSE
```

If the specified condition in a null IFSTRING-DO statement containing the null ELSE option is true, execution continues from the statement following IFSTRING-DO to the END DO (IX.12). Control is then transferred to the statement following the ELSE.

If the specified condition in an IFSTRING-DO statement containing the null ELSE option is false, execution continues with the statement following ELSE.

Note: The null ELSE form only has meaning in a "nested" (or complex) IFSTRING statement.

IX.11.2 Command Format

IFSTRING
IFSTRING (string operator string) action

IFSTRING-ELSE
IFSTRING (string operator string) action
ELSE action

IFSTRING-ELSE DO
IFSTRING (string operator string) action
ELSE DO
actions
END DO

IFSTRING-DO
IFSTRING (string operator string) DO
actions
END DO

IFSTRING-DO-ELSE
IFSTRING (string operator string) DO
actions
END DO
ELSE action

IFSTRING-DO-ELSE DO
IFSTRING (string operator string) DO
actions
END DO
ELSE DO
actions
END DO

IFSTRING-Null-ELSE
IFSTRING (string operator string) action
ELSE

IFSTRING-DO-Null-ELSE
IFSTRING (string operator string) DO
actions
END DO
ELSE

NOTE: IFS may replace IFSTRING in each of the commands listed previously.

string - name of a string variable, string literal or string expression previously defined in a CPL STRING (VI.3) or DEFINE (VI.4) statement;

operator - condition of the comparison;

Note: Although .EQ. and .NE. are normally the operators employed in an IFSTRING/IFS comparison, .LE., .LT., .GE. and .GT. may also be used.

A string comparison may use .HEQ., .HNE., .HLE., .HLT, .HGE. or .HGT. With these operators trailing blanks are not dropped and lower case characters are not converted to upper case.

action - any executable CPL statement(s) on a single line;

Note: The reverse slash (\) may be used if multiple statements are to be coded on the same line. For additional information see Reverse Slash, (IV.9).

actions - any executable CPL statement(s) on one or more lines.

IX.11.3 Cautions

1. The "action" portion of an ELSE may not be an IFSTRING or IF command. For additional information see Command Format, IX.11.2.

IX.12 END DO

IX.12.1 Usage

The END DO command terminates any of the forms of the IF or IFSTRING/IFS statements which contain the DO option. For additional information see IF (IX.10) and IFSTRING/IFS (IX.11).

IX.12.2 Command Format

```
END DO
```

Chapter Ten - FILE DEFINITION AND CONTROL

OVERVIEW

All forms of program input and output in CPL are controlled through the System and Program Logical Units. No physical device control is allowed or necessary with CPL.

X.1 FILE

X.1.1 Usage

The FILE command assigns labels to all System and Program Logical Units used within the program and reserves space used by the operating system to communicate with the device/file. FILE also establishes a Record Control Block (RCB) in memory. This command can only be used with expansion D.

The FILE statement is the link between the logical unit, the CPL program and the device/file. Considered a piece of data rather than logic, FILE (and additional CPL statements) enables the program to perform I/O.

A FILE command normally adds 30 bytes to a program. If FILE is used in conjunction with a spanned-sector file, 46 bytes are added to a program. For additional information on spanned-sector files see Chapter 13 - SPANNED-SECTOR I/O.

X.1.2 Command Format

```
FILE file: SYSccc, [access], [CLASS=n], [BUFFER=n,
  buffer], [RECSIZ=n], [KEY=integer], [FILTYP=c],
  [LSR=routine], [BLOCKING=n]
```

file	- name by which the device/file will be known inside the program;
ccc	- suffix of a System Logical Unit or the number of a Program Logical Unit;
access	- keyword SEQUENTIAL (SEQ), RANDOM (RND) or INDEXED (IND);
buffer	- name of a buffer (defined in a CPL BUFFER statement);
integer	- name of a 4-byte integer which contains the location of the record to be accessed;
routine	- number of the Logical Service Routine or the name of a non-resident routine.

In actual use the FILE statement does not extend beyond one line. All data beyond "FILE", "file name", ":" and "SYSccc" is optional and may appear in any order. Commas are required to separate optional fields, however.

Keyword Designation

1. SYSccc - the first keyword after the colon specifies the logical unit to which the internal file name is assigned. SYS units include the following: SYSRDR, SYSLOG SYSLST, SYSIPT, SYS000, SYS001, etc. Tape files cannot be assigned to SYSRDR or SYSIPT. All other SYS assignments are valid for tape files.

2. Access - keywords SEQUENTIAL (SEQ), RANDOM (RND) and INDEXED (IND) are used for access. SEQUENTIAL (SEQ) is used for Type "A" or Type "B" files, a CRT, a printer, a spooler, or tape; RANDOM (RND) is used for Type "C" files; and INDEXED (IND) is used for spanned-sector files. If no keyword is given, access defaults to SEQ. However, access should always be specified for internal documentation purposes.

3. CLASS=n - keyword insures that only a specific type of device or file may be assigned to the specified logical unit. The allowable values are:

CLASS=0	console device only
CLASS=1	device independent, unbuffered
CLASS=2	buffered disk file only
CLASS=4	tape files

If CLASS=n is not specified, CLASS=0 is the default.

Note: A mismatch between the assigned file and the specified class will abort the program when the file is opened.

4. BUFFER=n, label - keyword specifies an area of memory for file input/output. Whenever a sector of a specified file is accessed, it is read into the temporary holding area created by the buffer.

The BUFFER= keyword is followed by two data items separated by a comma. The first item is the size (in bytes) of the buffer reserved for the file; minimum buffer length for disk files is 400 bytes. The second item is the name of the buffer to be used in the FILE statement. This label name must be defined in a CPL BUFFER statement. The name of a buffer may also be represented by an asterisk (*). When BUFFER=n, * is used, the compiler automatically generates a label name and no BUFFER statement is necessary.

Note: All Type "B" files must be buffered. A buffer must also be included with all CLASS=2 designations. This keyword is not valid with tape files.

5. RECSIZ=n - keyword establishes a true record length for Type "C" random-access files. It may also be used to establish a maximum record length for Type "A" and Type "B" sequential files. While the actual records may be shorter than the space reserved for them, no record may exceed the established maximum length.

Note: Statement of RECSIZ=n is optional for all sequential (Type "A" or Type "B") files. RECSIZ=n is required for all random (Type "C") and indexed (spanned-sector/VSI) files. There is no default for RECSIZ=n on Type "A", "B" and "C" files (CPU-5). Default for Type "I" files is 400 (CPU-6). For blocked tape files, RECSIZ is used to calculate the buffer size. If it is not stated, RECSIZ defaults to 512. For unblocked tape files, RECSIZ is not used.

6. KEY=label - keyword specifies the name of the 4-byte integer which is the key to the file when reading or writing a Type "C" random access file.

Note: A key must always be specified for a random or indexed file; a sequential file does not contain a key. This keyword is not used with tape files.

7. FILTYP=c - although FILTYP=c is not normally stated, the keyword specifies whether a Type "A", "B", or "C" file is being used. If the keyword is omitted (and the file is sequential), a Type "A" file is assumed; if the keyword is omitted and the file is random or indexed, a Type "C" file is assumed.

Note: If Type "B" is required, FILTYP=B must be stated. This keyword is not used with tape files.

8. LSR=routine - although the compiler usually calculates the correct Logical Service Routine, the keyword specifies the routine performed in processing the file. The following is a list for standard system LSR routines.

0	Type "A" unbuffered device or file
1	Type "A" buffered file
2	Type "C" random unbuffered file
3	Type "C" random buffered file
4	Type "B" buffered file
5	Relative sector I/O
8	Tape files

Note: LSR=5 is never calculated by the compiler and must always be stated when used.

Optionally, a Logical Service Routine other than a standard system routine may be used. In this case LSR=label would be assigned by the programmer; "label" would state the name of the non-resident routine.

9. BLOCKING=n - number of logical records in a tape If the number of records is not stated, BLOCKING defaults to 1. Blocked tape files are accessed through a buffer with size = (RECSIZ x BLOCKING) bytes. BLOCKING=n is used only with tape files (CLASS=4).

X.1.3 Caution

Failure to initialize a Type "C" indexed file through RECSIZ=n or GETK, NEWK, DELK will cause the record length to retain the default value of zero.

TYPE	DEVICE	Type 'A' FILE	Type 'B' FILE	Type 'C' FILE (RND)	Type 'C' FILE (RND SPN)	Type 'C' FILE * (4-BYTE IND) * OBSOLETE	Type 'C' FILE (6-BYTE IND) * OBSOLETE	Type 'C' FILE VSI (CPU 5)	Type 'C' FILE VSI (CPU-6)	REL SEC	PGM LSR	TAPE
ACCESS	SEQ	SEQ	SEQ	RND	RND	RND	RND	IND	IND	RND	RND	SEQ
VAR LEN	YES	YES	YES	NO	NO	NO	NO	NO	NO	NO	USER DEFINED	YES
MAX RECSIZ	DEVICE	193 (CPU-5) 198 (CPU-6)	193 (CPU-5) 198 (CPU-6)	195 (CPU-5) 400 (CPU-6)	2048 2048	195 (CPU-5) 400 (CPU-6)	395 (CPU-5) 400 (CPU-6)	2048	2048	400	USER DEFINED	DEVICE DEPENDENT
INPUT	READ READB	READ	READB	READB	GETR READB (CPU-6)	READB	READB	GETR	READB (GETR)	READB	READ READB	READB
OUTPUT	WRITE WRITEB	WRITE	WRITEB	WRITEB	PUTR WRITEB (CPU 6)	WRITEB	WRITEB	PUTR	WRITEB (PUTR)	WRITEB	WRITE WRITEB	WRITEB
BUFFER	NO	OPTIONAL	REQUIRED	OPTIONAL	OPTIONAL (CPU 5) REQUIRED (CPU-6)	OPTIONAL	OPTIONAL	OPTIONAL	REQUIRED	NO	USER DEFINED	OPTIONAL
SPECIAL	CURP WRITEN CURS REWIND CURB CURSOR ENDFILE	ENDFILE REWIND REWRITE	ENDFILE REWIND	NONE	NONE	NONE	NONE	NONE	NONE	NONE	USER DEFINED	RESET RSKIP
HOLD/ FREE	NO	NO	NO	YES	HLDR/FREE (CPU-5) HOLD/FREE (CPU-6)	YES	YES	HLDR FREE	HOLD/FREE HLDR/FREE	YES	USER DEFINED	NO
KEY ROUTINES	NONE	NOTE POINT	NOTE POINT	NONE	NONE	GETKEY NEWKEY	?GKEY ?NKEY	GETK NEWK DELK	GETK NEWK DELK NEXK	NONE	USER DEFINED	NONE
INIT	NONE	NONE	NONE	NONE	NONE	XHINT	X?RINT	XVHINT	NEW	NONE	USER DEFINED	NONE
UTILITIES	NUMEROUS BASIC (CPU-6) SMART	NUMEROUS SORT BASIC (CPU-6) SMART	SORT BASIC (CPU 6) SMART	SORT SMART BASIC (CPU-6)	SORT (CPU-6) BASIC (CPU-6) SMART	XWTAG	X?WTAG	XVCOPY XTRACT SMART	XVCOPY SMART XTRACT XXTEST SORT BASIC	N/A SMART	USER DEFINED	TDSP TCOP COPT

FILE STATEMENTS
CPU-5/CPU-6

TYPE	STATEMENT
CRT	FILE name: SYSccc, - Used by CRT only; Class=0 (also the default)
PRINTER/ SPOOLER	FILE name: SYSccc, CLASS=1 - includes CRT, printer, spooler DUMMY, BLIND and Type "A" file.
DEVICE INDEPENDENT	FILE name: SYSccc, CLASS=1 - includes CRT, printer, spooler, DUMMY, BLIND and Type "A" file.
TYPE "A" UNBUFFERED FILE	FILE name: SYSccc, CLASS=1 - includes CRT, printer, spooler, DUMMY, BLIND and Type "A" file.
TYPE "A" BUFFERED	FILE name: SYSccc, CLASS=2, BUFFER=400, label
TYPE "B" BUFFERED	FILE name: SYSccc, CLASS=2, BUFFER=400, label, FILTYP=B
TYPE "C" RANDOM UNBUFFERED	FILE name: SYSccc, RND, CLASS=1, RECSIZ=n, KEY=integer
TYPE "C" RANDOM-BUFFERED	FILE name: SYSccc, RND, CLASS=2, RECSIZ=n, KEY=integer BUFFER=400, label NOTE: All remaining Type "C" buffered/unbuffered files are structured like the two cited above. Buffered files are CLASS=2 (with a buffer statement); unbuffered files are CLASS=1 (no buffer statement).
TYPE "C" RANDOM-SPANNDED (CPU-5)	FILE name: SYSccc, IND, CLASS=1[2], RECSIZ=n, KEY=integer, BUFFER=400, label NOTE: A BUFFER=n statement may or may not be present depending on CLASS.

FILE STATEMENTS CONTINUED

TYPE	STATEMENT
TYPE "C" 4-BYTE INDEXED	FILE name: SYSccc, RND, CLASS=1[2], RECSIZ=n, KEY=integer NOTE: A BUFFER=n statement may or may not be present depending on CLASS.
TYPE "C" 6-BYTE INDEXED	FILE name: SYSccc, RND, CLASS=1[2], RECSIZ=n, KEY=integer
TYPE "C" VSI (CPU-5)	FILE name: SYSccc, IND, CLASS=1[2], [RECSIZ=n], KEY=integer
TYPE "C" RANDOM-SPANNED (CPU-6)	FILE name: SYSccc, SPANNED [SPN], CLASS=2, RECSIZ=n, KEY=integer NOTE: BUFFER=n cannot be stated since it is generated automatically by RECSIZ=n.
TYPE "I" VSI (CPU-6)	FILE name: SYSccc, IND, CLASS=2, RECSIZ=n, KEY=integer NOTE: BUFFER=n cannot be stated since it is generated automatically by RECSIZ=n.
RELATIVE SECTOR I/O	FILE name: SYSccc, RND, CLASS=1, RECSIZ=400, KEY=integer, FILTYP=t, LSR=5
PROGRAMMER WRITTEN LSR	FILE name: SYSccc.....LSR=label NOTE: Since the programmer creates this logical Service Routine (LSR) the body of the statement is left to the discretion of the programmer and the needs of the LSR.
TAPE UNBLOCKED	FILE name: SYSccc, SEQ, CLASS=4
TAPE BLOCKED	FILE name: SYSccc, SEQ, CLASS=4, RECSIZ=n, BLOCKING=n

X.2 OPEN

X.2.1 Usage

The OPEN command allows a CPL program to access a file. In the case of a buffered file, OPEN causes the first sector of that file to be read into the buffer area.

The command compares the parameter(s) specified in the FILE statement to those of the actual file assigned to the appropriate logical unit. A mismatch in the comparison causes an abort at program execution time.

OPEN, as a piece of logic, may be located anywhere in the program so long as it follows SYSTEM. As many OPEN statements as needed may be used.

Each OPEN command adds 4 bytes to a program, plus an additional 3 bytes per file to be opened.

X.2.2 Command Format

```
OPEN access file
  or
OPEN access (file, ...) [,access (file, ...) ,...]
```

access - INPUT, OUTPUT or IO;
file - label of a file (defined in a CPL FILE statement) to be opened.

Note: Multiple file names may be specified for each type of access. They must be enclosed in parentheses and separated by commas. Commas may also be used to separate multiple types of access on a single line.

X.2.3 Cautions

1. No file may be accessed until it is opened.

2. On a CPU-5 system, a file may either be opened for INPUT and written to immediately or opened for OUTPUT and read immediately. On a CPU-6 system this is not allowed.

3. Any indexed file (i.e. 4-byte, 6-byte or VSI) must always be opened with access type IO.

X.3 CLOSE

X.3.1 Usage

The CLOSE command prohibits further access to a file within a CPL program. The command is generally used to prevent jeapordizing data by accessing a file unintentionally.

> NOTE: Usage of CLOSE is optional. The STOP command should be used when program termination is required. At that time end-of-step processing automatically closes all open files. For additional information see STOP (III.3).

A CLOSE command adds 6 bytes to a program for each file to be closed.

X.3.2 Command Format

CLOSE file [,file, file, ...]

file - label of file (defined in a CPL FILE statment) to be closed.

X.3.3 Cautions

1. If a file is assigned a buffer, the contents of that buffer are not automatically written out when the file is closed. ENDFILE must be used for this purpose. For additional information see ENDFILE (X.5).

2. End-of-file records are not automatically written to sequential files when the file is closed. ENDFILE must be used for this purpose. For additional information see ENDFILE (X.5).

3. All files must be closed prior to executing a full program overlay. For additional informa- see LOAD (XIV.3).

X.4 ENDFILE

X.4.1 Usage

The ENDFILE command writes an end-of-file marker for a specified sequential (i.e. Type "A" or Type "B") file. If the file is buffered, the command writes the last buffer into the file.

An end-of-file marker is meaningful only when files are accessed sequentially. The principal function of the ENDFILE command is to insure that a buffer containing new data is written back to a disk file before a program is terminated. For additional information also see WRITE (XI.2).

ENDFILE adds 5 bytes to a program for each file listed in the ENDFILE command.

X.4.2 Command Format

```
ENDFILE file [,file, file, ...]
```

file - label of a file (defined in a CPL FILE statement) to which the end-of-file marker is to be written.

X.4.3 Cautions

1. Normally the ENDFILE command is only used to terminate output to a sequential file. If used with sequential input files, ENDFILE will truncate that file after the last record which was read.

   For a summary of ENDFILE usage, consult the following:

   (1) Use of ENDFILE is optional for devices and is ignored.
   (2) Use of ENDFILE is optional for "A" and "B" Type files (input).
   (3) Use of ENDFILE is required for "A" and "B" Type files (output).
   (4) Use of ENDFILE is illegal for all other forms.

X.5 REWIND

X.5.1 Usage

The REWIND command allows the first record of a sequential file (i.e. Type "A" or Type "B") to be accessed. REWIND basically reopens a sequential file and resets the address of the current record in the file. On input, if the file is buffered, the first sector of that file is read into the appropriate buffer.

> NOTE: REWIND may not be used with random "C" Type files or indexed files. It may be used with devices; however, in this case REWIND would be ignored.

REWIND adds 6 bytes to a program for each file rewound.

X.5.2 Command Format

REWIND file [,file, file, ...]

file - label of file (defined in a CPL FILE statement) to be rewound.

X.5.3 Cautions

1. The contents of the buffer in a buffered file are not written out before the file is rewound and the first sector read into the buffer. For additional information see ENDFILE (X.4).

X.6 SETFORM

X.6.1 Usage

The SETFORM command is only valid on the CPU-6 system. This command will allow the form number and/or HOLD/FREE status of a job going to spooler to be modified.

NOTE: If multiple SETFORM commands are used, the last SETFORM issued prior to CLOSE will be the one that will affect the spooler.

X.6.2 Command Format

```
                                [,HOLD]
SETFORM  (file [,] [n]  [,FREE])
```

file - name of file to be set;

n - form number, if omitted, the current form number will be unchanged;
NOTE: Valid form numbers are 0 thru 32,767.

HOLD - causes the job to be held within the spooler;

FREE - causes the job to be freed (or released to the printer).
NOTE: FREE is assumed, if no options is specified.

X.6.3 Examples

```
SETFORM  (PRT,, FREE); form# unchanged , FREE

SETFORM  (PRT)       ; form# unchanged , FREE

SETFORM  (PRT,18)    ; form#=18 , FREE

SETFORM  (PRT, HOLD) ; form# unchanged, HOLD

SETFORM  (PRT,18,HOLD); form#=18, HOLD
```

NOTE: The first and second examples will yield the results, the second example is merely a simpler method.

X.7 SKIP

X.7.1 Usage

The SKIP command advances the tape by the number of records specified when the command is entered. This command is used only with tape files (CLASS=4) and can be used only with expansion D.

The SKIP command adds six bytes to a program.

X.7.2 Command Format

SKIP (file,n)

file - label of a file (defined in a CPL statement) in which records are to be skipped;

n - the number of records to skip

X.8 RESET

X.8.1 Usage

The RESET command is used to rewind a tape to the beginning of the current file. This command is used only with tape files (CLASS=4) and can be used only with expansion D.

This command adds five bytes to a program.

X.8.2 Command Format

```
RESET file [,file,file,...]

     file - label of a file (defined in
            a CPL statement) to be reset
```

Chapter Eleven - FORMATTED INPUT/OUTPUT

OVERVIEW

Formatted Input/Output, which differentiates between binary (integer) data and character data, is used in transferring ASCII data. Since numeric ASCII data must be converted to binary before arithmetic can be performed, formatted I/O commands are generally used on all buffered/unbuffered Type "A" files, printers and console devices.

XI.1 FORMAT

XI.1.1 Usage

The FORMAT command reserves a string in memory. This string is read to determine the form data will take when converted by a READ, WRITE, WRITEN/WRITN, REWRITE, ENCODE or DECODE command and may be considered a type of record layout. Since formats are not assigned on a one-to-one basis to individual statements, the same format may be used many times within the same program.

All input or output between a CPL program and an ASCII device or Type "A" file requires the use of field specifications set up in FORMAT statements. Input commands convert ASCII characters from a device/file (READ) or from a character string (DECODE) into one of the following:

```
Character String(s) - "C" Field Specification
4-Byte Integer(s)   - "N" Field Specification
6-Byte Integer(s)   - "D" Field Specification
No Input            - "X" Field Specification
```

Output commands (i.e. WRITE, WRITEN/WRITN, REWRITE, ENCODE) reverse the process outlined in the preceding section. Therefore, if a 6-byte integer is to be converted into character form, a "D" Field Specification must be used on output. Accordingly, a "N" Field Specification must be used to convert a 4-byte integer to character form, a "C" Field Specification must be used to copy character for character and a "X" Field Specification must be used to insert blanks into the output string.

As a piece of data, FORMAT statements may occur anywhere in a program following SYSTEM. As many FORMAT statements as are necessary may be used within a CPL program.

> NOTE: For purposes of readibility it is suggested that all FORMAT statements be located in one section of the program.

Each FORMAT command adds 1 byte to program length. An additional 3 bytes is added for each format specification in the list.

XI.1.2 Command Format

FORMAT format: specification, specification, ...

format - label to be associated with field specifications following the colon (:);

specifications - characters "N", "D", "C" or "X" followed by the 4-byte integer or literal representing the number of characters in the field.

Note: If a 4-byte integer is used, it must be enclosed in parentheses.

The FORMAT statement must be associated with a READ, WRITE, WRITEN/WRITN, REWRITE, ENCODE or DECODE command for I/O to occur. In the following "C" field specification, the READ and FORMAT statements are used to create a record layout.

```
READ (FILEA, FRED) STR1, STR2, STR3
FORMAT FRED: C6, C(LEN), C10
```

The label (i.e. FRED) assigned to the "format" section of the READ statement must be the same as that assigned to the FORMAT statement; additionally, each variable in the READ statement (i.e. STR1, STR2, STR3) is associated with a specification in the FORMAT statement (i.e. C6, C(LEN), C10).

NOTE: Each label in the list of the specified I/O is expected to have a field specification in the assigned format. If the number of labels in the list exceeds the number of field specifications, those specifications are repeated. If the number of field specifications exceeds the number of labels, the excess specifications are ignored.

"N", "D", "C" and "X" field specifications may be combined, separated by commas and/or spaces. This combination process allows several types of information to be stored in a single record or string before being input into a list of strings and/or integers. During output a list of strings and/or integers may be combined into a single record or string.

NOTE: If a record is too long for the specified line length, truncation of data begins with the first specification to exceed the limit (not with the first character to exceed the limit). The remainder of the line following the truncation would remain blank.

"N" and "D" Field Specification (Input)

"N" or "D" is followed by the number of characters from the source device/file/string to be converted to the binary integer format.

Note: A "N" field specification is used with 4-byte integers; a "D" field specification is used with 6-byte integers.

The source is read from left to right until the specified number of ASCII numeric characters (or the end of the record as indicated by the ASCII carriage return symbol) is reached. If the number of characters in the source exceeds the number in the field specification, the low order digits will be truncated.

A positive (+) or negative (-) sign may either precede or follow the input string. If no sign is specified, the integer will be assumed to be positive. The number of characters specified must include one character for the sign; if a trailing sign is truncated, the integer will be converted as postive.

The following conversions would take place under a "N4" specification:

String	Integer
12345	1234
+12345	123
12345+	1234
-12345	-123
12345-	1234

While leading blanks are allowed, trailing blanks brought from the input record/string under the specified format will not allow the conversion to take place and the target integer will retin its original value. STATUS will be set to 2 indicating a a format error.

A decimal point in the field specification does not cause a decimal point to to included in the converted integer. It merely insures that if a decimal point is found in the source string, exactly that number of decimal places are output. If the string contains a decimal point, but fewer than the specified number of , zeroes will be added. If the string contains too many places places the extra places will be truncated.

Note: The number before the decimal point specifies the total number of characters (including decimal points and signs) which will be read.

The following conversions would take place under a "N5.2" specification:

String	Integer
123	123
1.2	120
1.234	123
1.	100
123-	-123
123.45	12340
-123.45	-12300

"N" and "D" Field Specifications (Output)
"N" or "D" is followed by the number of ASCII numeric characters the target string is to contain. The target string is filled from right to left. If the number is negative, the "-" sign will normally be output last at the far left end of the string. If the number is positive , the "+" is omitted. No blanks are output during this process. If the field is too small for all digits to be output, the high-order digits are truncated.

The length of the data stored in the string is dependent upon the field specification, not on the size of the source integer. The string must be large enough to contain the length of the field specified. A right-hand sign is obtained by including a "-" to the right of the "N" or "D" character in the field specification. Although a left-hand sign is assumed, it may be indicated by a negative sign to the left of the "N" or "D".

In the following examples the integer shown is encoded into a 6-character string which originally contained the characters XXXXXX:

Integer	Field Specification	String
12345	N6	X12345
12345	N3	345XXX
123-	N6	-XX123
-12345	N-6	12345-

Note: A "X" is output only with the ENCODE command; with all other commands "X" becomes a space.

If the output string is to contain a decimal point, the number of decimal places preceded by the decimal point must be called for in the field specification. If a right-hand sign is to be included when decimals are used, decimal places are counted from the second position to the right.

Assuming a pre-blanked string, the following examples illustrates the conversion process:

Integer	Field Specification	String
-12345	N10.3	12.345
-12345	N-10.3	12.345-
12345	N-10.3	12.345
12345	N5.3	2.345
-12345	N-5.3	.345-
-12345	N5.3	-.345

Note: In the last two examples, the sign is retained even though the first two characters are truncated.

"C" Field Specification
"C" (character) field specifications make a literal transposition of ASCII data from source to target. This transposition occurs when data is being both input and output. Character data is stored from left to right until the specified number of characters or the end of the record, indicated by the ASCII carriage return symbol, is reached.

If the terminator is reached before all characters called for in the field specification have been used, the difference will be made up with blanks. If the string is longer than the field specification, the remaining characters in the target field will be unchanged. Also, blanks are provided on output to make up the difference between a string and a longer field specification.

In the following examples, the data shown is encoded into a 6-character string which originally contained the characters XXXXXX:

Source	Field Specificati n	String
ABC	C6	ABC
ABC	C5	ABC X
ABCDEF	C4	ABCDXX

Note: A "X" is output only with the ENCODE command; with all other commands "X" becomes a space.

"X" Field Specification

The "X" Field Specification during input causes the specified number of characters to be read or decoded; these characters, however, are not transferred to memory. During output, blank characters are generated which may be inserted between other output.

Note: "X" Field Specifications are used almost exclusively in combination with other specifications.

XI.1.3 Cautions

1. The total length of the record specified in a FORMAT statement must be less than that of the system line buffer. Fields which exceed this length are lost.

   Note: The length of the system line buffer is normally 132 characters.

2. Any attempt to input or output a non-numeric character under a numeric field specification will result in a format error.

3. Leading blank(s) may precede a numeric string being converted to an integer. Trailing blanks, are not permitted, however.

4. If a format error should occur while a list of labels is being processed, subsequent labels in the list will not be converted and will retain their current values.

XI.2 READ

XI.2.1 Usage

The READ command - or "formatted" read statement - transfers ASCII data from a device/file to one or more integers or strings. Each READ command accesses an entire record, terminated by an ASCII carriage return character. The next READ command accesses the next sequential record of the device/file.

NOTE: READ is an executable command and should be located within the logic section of the program.

Each READ statement adds 9 bytes to the length of a program. An additional 2 bytes is added for each integer or string in the list.

XI.2.2 Command Format

READ (file, format) variable, variable, ...

file - label of the device/file (defined in a CPL FILE statement) from which the record is to be read;
format - label of a CPL FORMAT statement;
variable - name(s) of the string variable, integer variable or literally-subscripted table-name in which data is to be stored.

Data read by the READ command may be stored in character or in binary integer form depending on the format used. For additional information see FORMAT (XI.1).

Sectors
The sector containing the record to be read is copied from a disk into a buffer. If no specified buffer is assigned to the file, the system buffer is used. Individual records are then read sequentially from this buffer. For additional information see FILE (X.1).

STATUS
The value of STATUS is affected by each READ command. STATUS values include 0 (Normal Completion), 1 (End-of-File) and 2 (I/O Error or Format Error).

Note: For additional information on the causes of format errors during a READ statement, see FORMAT (IX.1)

XI.2.3 Cautions

1. STATUS should be checked after each READ command from a disk file. STATUS is set to 1 only by that READ command which encounters an end-of-file mark; subsequent READ commands return STATUS to 0.

2. A READ should not be performed after a WRITE since READ does not write a completed sector back to disk. The changes made by the WRITE command inside the buffer are not transferred to the disk before the READ command brings in the next sector. REWRITE should be used when it is necessary to change a record in the middle of a Type "A" file.

XI.3 WRITE

XI.3.1 Usage

The WRITE command - or the "formatted" write statement - outputs data stored in one or more integers or strings to a specified ASCII device/file. Each WRITE command outputs one record which is terminated by an ASCII carriage return character.

NOTE: WRITE is an executable command and should be located within the logic section of a program.

Each WRITE command adds 9 bytes to the length of the program. An additional 2 bytes is added for each integer, string or literal in the list. Arithmetic expressions also add 5 bytes to the program plus an additional 5 bytes for each variable or literal in the expression.

XI.3.2 Command Format

WRITE (file, format) variable, variable, ...

file - label of the device/file (defined in a CPL FILE statement) into which the data is to be written;
format - label of a CPL FORMAT statement;
variable - name of the string variable, literal or expression and/or integer variable, literal or expression.

NOTE: An integer literal or integer variable must use an "N" or "D" field specification depending on the type of the integer. An integer expression must use a "D" field specification. A string must use a "C" field specification. For additional information see FORMAT (XI.1).

Sectors

If no specific buffer is assigned to a file, the system buffer is used. For additional information see FILE (X.1).

At the end of each record the WRITE command outputs an end-of-sector mark to insure the integrity of the data. A subseguent WRITE command in the same sector converts the end-of-sector mark to a carriage return character indicating an end-of-record. When the buffer no longer has enough room for the next record, the sector is copied to disk.

STATUS

The value of STATUS is affected by each WRITE command. Possible STATUS values are 0 (Normal Completion), 2 (I/O Error or Format Error) and 3 (End-of-Medium on Output).

Note: In the case of a Type "A" file which expands automatically, end-of-medium is an indication that the disk is full. For additional information see FORMAT (XI.1).

XI.3.3 Cautions

1. A entire buffer of data must be written before that data will be transferred back to disk. Therefore, writing to a disk file should be followed by an ENDFILE command. For additional information see ENDFILE (X.5).

2. Literals in the range of -127 to +127 used in a WRITE statement will generate a 4-byte literal.

XI.4 WRITEN/WRITN

XI.4.1 Usage

The WRITEN/WRITN command is used to transfer ASCII data stored in one or more integers or strings to a console-type device without including a carriage return at the end of the line. Except for the fact that no carriage return is output and that the cursor is positioned immediately following the data output, WRITEN/WRITN functions identically to WRITE.

NOTE: WRITEN/WRITN are executable commands and should be located within the logic section of the program.

Each WRITEN/WRITN command adds 9 bytes to the length of the program. An additional two bytes is added for each integer or string in the list. Arithmetic expressions add 6 bytes to the program.

XI.4.2 Command Format

```
WRITEN (file, format) variable, variable, ...
  or
WRITN (file, format) variable, variable, ...
```

file - label of the console-type device (defined in a CPL FILE statement) to which data is to be written;
format - label of a CPL FORMAT statement;
variable - name(s) of the string variable, literal or expression and/or integer variable, literal or expression.

NOTE: An integer literal or integer variable must use an "N" or "D" field specification depending on the type of the integer. An integer expression must use a "D" field specification. A string must use a "C" field specification. For additional information see FORMAT (XI.1).

STATUS
The value of STATUS is affected by each WRITEN/WRITN command. Possible STATUS values are 0 (Normal Completion, 2 (I/O Error or Format Error) and 3 (End-of-Medium on Output).

XI.4.3 Cautions

1. Do not use WRITEN/WRITN with an "A" Type disk file.

XI.5 DECODE

XI.5.1 Usage

The DECODE command is used to transfer ASCII string data from a source string to one or more strings or integers or to translate ASCII numerics into binary.

NOTE: DECODE is an executable command and should be located within the logic section of the program.

Functioning to a large extent like the READ command, DECODE performs no I/O and takes its data from a character string stored in memory. If the string is less than 132 characters in length, trailing spaces are added to form a 132 character input record. For additional information see READ (XI.2).

Each DECODE command adds 9 bytes to the length of a program. An additional 2 bytes is added for each integer or string in the list.

XI.5.2 Command Format

DECODE (string, format) variable, variable, ...

string - label of an ASCII character string (defined in a CPL STRING statement) from which data is to be read;
Note: If an offset is desired, the string label plus the number of bytes to be added/subtracted must be included in brackets. For example, [string + n] or [string - n].

format - label of a CPL FORMAT statement;

variable- name(s) of the string variable, integer variable or literally-subscripted table-name in which data is to be stored.

STATUS
The value of STATUS is affected by each DECODE command. STATUS values include 0 (Normal Completion) and 2 (Format Error).

XI.5.3 Cautions

1. Since spaces are added to the end of the source string, care must be taken in decoding under a numeric "N" or "D" field specification. If a space is encountered to the right of the number during the conversion process, the conversion will not take place. For additional information see FORMAT (XI.1).

XI.6 ENCODE

XI.6.1 Usage

The ENCODE command transfers data stored in character string(s) or integer(s) and/or calculated from arithmetic expressions, to a single character string.

> NOTE: ENCODE is an executable command and should be located within the logic section of the program.

Functioning to a large extent like the WRITE command, ENCODE performs no I/O and retains the terminator of the original target string ignoring any terminators supplied in the source strings or integers. This allows designated integers (rather than blanks) to be expressed in character form while retaining dollar signs, commas and/or other characters present in the target string.

The ENCODE command treats numeric and character data differently in some instances. Under a "N" field specification, the string is filled from right to left within the limits of the field specification. Under a "C" field specification, the string is filled from left to right. Blanks are used to fill out field lengths in character conversions. Leading zeroes, however, will not be added to numeric strings when integers are converted. Thus, if the numeric string does not fill all specified places, the leftmost characters will retain their original value. For additional information see FORMAT (XI.1).

Each ENCODE command adds 9 bytes to the length of the program. An additional 2 bytes is added for each integer or string in the list. Arithmetic expressions add 6 bytes to the program.

XI.6.2 Command Format

ENCODE (string, format) variable, variable, ...

string - label of ASCII character string (defined in a CPL STRING or DEFINE statement) into which data is to be written;

Note: If an offset is desired, the string plus the number of bytes to be added or subtracted must be included in brackets. For example, [string + n] or [string - n].

format - label of a CPL FORMAT statement;

variable- name(s) of the string variable, literal or expression and/or integer variable, literal or expression from which data is to be used.

STATUS

The value of STATUS is affected by each ENCODE command. STATUS values include 0 (Normal Completion) and 2 (Format Error).

XI.6.3 Cautions

1. Strings are not cleared when set up by a STRING statement. Random values found in that area of memory are retained in the bytes of that string not written over by the ENCODE command.

2. ENCODE does not add terminators during the transfer of data. Indeed, terminators may be lost during the process.

   When data in a string takes up fewer characters than the number called for in the STRING statement a terminator is used to separate the valid data from the random values which fill out the string. The terminator may be lost if a group of characters longer than the existing valid data is encoded into the string. This may result in the appearance of random values at the end of the string.

XI.7 NOTE

XI.7.1 Usage

The NOTE command allows the location of a specific record within a Type "A" or Type "B" file to be stored in an integer. Used with POINT, the command allows a limited form of random access to be performed on a sequential file.

NOTE may be located anywhere in a program so long as it follows SYSTEM. A program may contain as many NOTE statements as necessary.

Each NOTE command adds 8 bytes to the length of the program.

XI.7.2 Command Format

NOTE (file, integer)

file - label of the Type "A" or Type "B" file (defined in a CPL FILE statement) containing the record whose address is to be noted;

integer - label of the 4-byte integer (reserved in a CPL INTEGER or SET statement) in which the address is to be stored.

The operating system keeps track of the sector address and displacement of the last record read or written. NOTE causes the location of this previous record to be stored in the specified 4-byte integer. Therefore, it must follow the READ or READB command which accesses the desired record.

XI.7.3 Cautions

1. Use WRITE with NOTE and POINT commands when records being rewritten are of the same length; use REWRITE when records are being written to Type "A" files and are of unequal length. For additional information see POINT (XI.8) and REWRITE (XI.9).

XI.8 POINT

XI.8.1 Usage

The POINT command is used in conjunction with NOTE to locate a specific record within a sequential Type "A" or Type "B" file. POINT positions the file at the beginning of the record specified by a previous NOTE.

POINT may be located anywhere in a program so long as it follows SYSTEM. A program may contain as many POINT statements as necessary.

When used with a Type "A" file, REWRITE may be used to change a record in place. READ may also be used with NOTE and POINT to access a Type "A" file. When NOTE and POINT are used with a Type "B" file, READB and WRITEB may be used to access the file. For additional information see REWRITE (XI.9), READ (XI.2), Note (XI.7) and READB (XII.2) and WRITEB (XII.3).

Each POINT command adds 8 bytes to the length of a program.

XI.8.2 Command Format

POINT (file, integer)

file - label of the Type "A" or Type "B" file (defined in a CPL FILE statement) containing the record to be located;

integer - label of the 4-byte integer (reserved in a CPL INTEGER or SET statement) containing the location of the desired record.

Note: This integer must be set by the NOTE command. It contains the sector address and offset (i.e. number of bytes preceding the record) in that sector.

XI.8.3 Cautions

1. Use WRITE with NOTE and POINT commands when record being rewritten is of the same length; use REWRITE when records being written to Type "A" files are of unequal length. For additional information see NOTE (XI.7) and REWRITE (XI.9).

XI.9 REWRITE

XI.9.1 Usage

The REWRITE command allows a record in a Type "A" file to be updated in place. Each time a REWRITE command is completed, the sector containing that record is written back to the disk. This action prevents a subsequent READ command from overlaying the buffer in which the new data is stored.

NOTE and POINT must be used to locate the record to be written. Once the record has been located and the file positioned at the beginning of that record, the specified data will be written into the exact same space occupied by the record to be replaced.

> NOTE: This action means that a record longer than the original will be truncated; a record shorter than the original will be padded with blanks.

Each REWRITE command adds 9 bytes to the length of a program. An additional 2 bytes is added for each integer or string in the list. Arithmetic expressions add 6 bytes to the program.

XI.9.2 Command Format

REWRITE (file, format) variable

- file - label of the device/file (defined in a CPL FILE statement) into which the data is to be written;
- format - label of a CPL FORMAT statement which contains output field specifications;
- variable - name(s) of the string variable, literal or expression and/or integer variable, literal or expression.

STATUS

The value of STATUS is affected by each REWRITE command. Possible STATUS values are 0 (Normal Completion), 2 (I/O Error or Format Error) and 3 (End-of-Medium).

XI.9.3 Cautions

1. REWRITE is normally used in conjuction with the NOTE and POINT commands. For additional information see NOTE (XI.7) and POINT (XI.8).

CPU-5/CPU-6
CPL

Chapter Twelve - BINARY INPUT/OUTPUT

OVERVIEW

Binary Input/Output - unlike Formatted Input/Output - makes a byte-for-byte transfer of data without differentiating between binary and character data.

Binary I/O is primarily used with Type "B" or Type "C" disk files. Type "B" files are similar to Type "A" files and operate in almost an identical manner. In a Type "C" file the location of the next record to be read is stored in an integer identified as the KEY for the file. This integer may be set directly by the program by equating it with a literal or with another integer. It may also be calculated through a subroutine (e.g. GETK). For additional information see FILE (X.1).

Type "C" files are divided into spanned-sector files and discrete-sector files. Spanned-sector files are defined with the keyword IND in a FILE statement. Since these files may not be accessed with either a READB or WRITEB statement, special external subroutines must be used in these instances. Spanned-sector files are useful in that they provide program flexibility and economy of disk space. For additional information see Chapter 13 - SPANNED-SECTOR I/O and the APLIB Reference Manual.

Discrete-sector files are defined with the RND keyword in a FILE statement. They may be accessed with either a READB or WRITEB statement. Discrete-sector files are useful in that they provide program simplicity, as well as reduce the size of the program.

The Logical Service Routine (LSR) for Type "B" files accesses records sequentially, keeping track of the proper address in memory. For additional information see FILE (X.1).

The Logical Service Routine for Type "C" discrete-sector files uses a key to determine the location of the record to be read. This key may either be set directly with a literal or integer or it may be calculated within the program based some other piece of information usually known as the argument.

XII.1 RECORD/ENDREC

XII.1.1 Usage

The RECORD and ENDREC commands are used to reserve a binary record with RECORD defining the beginning address of a memory area and ENDREC defining the ending address.

NOTE: This memory area contains an entire record from a file with individually defined fields.

The RECORD statement is generally followed by INTEGER, STRING, DEFINE, SET or TABLE statement(s). These combine with RECORD and ENDREC to describe a record layout. For additional information see INTEGER (VI.1), SET (VI.2), STRING DEFINE (VI.4) and TABLE (VI.6).

Since RECORD is not associated with any given piece of I/O, it may be used by multiple I/O statements. As a piece of data, RECORD/ENDREC statements may occur anywhere in a program following SYSTEM. As many RECORD/ENDREC statements as are necessary may be used within a CPL program.

Each RECORD/ENDREC command increases the size of the program. RECORD adds 4 bytes to the program, while ENDREC adds 1 byte. In addition, the size of the program is increased by the record length (i.e. the number enclosed in parentheses in the command format).

XII.1.2 Command Format

```
RECORD record (n)
 ...
 ...
ENDREC
```

record - label of the record area;
n - literal value representing the size of the record area.

A record area must be used whenever Type "B" or Type "C" files input or output data. This record area may be divided into integers and character strings with INTEGER, SET, STRING, DEFINE and TABLE statements following the RECORD statement.

NOTE: Memory will be reserved for each integer and string beginning with the lowest address in the data area of the record.

The length of a record in a Type "B" file is determined by the associated record statement. Therefore, different records (and different record lengths) may be used with a single Type "B" file.

In a Type "C" file each record occupies the amount of space specified by the RECSIZ= keyword in the FILE statement for that file. This occurs even if the assigned record area may be smaller. For additional information see FILE (X.1).

XII.1.3 Cautions

1. If the integers and strings specified within the record is less than the total record length, the remaining bytes will be reused by the operating system.

2. An extra byte must be added to the length of each string when calculating a total record length. This allows for the terminator.

3. The CLREC subroutine may be used before data is first written into a record area. This removes random values from the unused portions of the record. For additional information see the APLIB Reference Manual (CLREC).

XII.2 READB

XII.2.1 Usage

The READB command - or "unformatted" read statement - makes a byte-for-byte transfer of data from a device/file to a record area in memory. The READB command is used with Type "B" or Type "C" files.

Each READB command adds 7 bytes to the program.

XII.2.2 Command Format

READB (file, record)

file - label of a file (defined in a CPL FILE statement) from which a record is read;

record - label of the record (defined in a CPL RECORD statement) into which data is read.

Note: The format of the record area specifies the number of bytes to be read through the RECORD statement, as well as the assignment of data to integers and strings within that record.

STATUS

The value of STATUS is affected by each READB command. Possible STATUS values are 0 (Normal Completion), 1 (End-of-File on Input - Type "B" files only or Key Out of Range - Type "C" files only) and 2 (I/O Error - Type "B" and Type "C" files or Invalid Key - Type "C" files only).

XII.2.3 Cautions

1. READB should not be used with Type "C" spanned-sector files. Since the record length in such files may exceed a single sector, use the external subroutine GETR with Type "C" spanned-sector files. For additional information on spanned-sector I/O, see Chapter 3 - SPANNED-SECTOR I/O and the APLIB Reference Manual (GETR).

2. The specified number of bytes give in the RECORD statement will be read by READB; no error checking is performed.

3. For CPU-5 compatibility READB should not be used after using GETK/NEWK, instead use GETR.

XII.3 WRITEB

XII.3.1 Usage

The WRITEB command - or "unformatted" write statement - makes a byte-for-byte transfer of data from a record in memory to a device/file. For additional information see RECORD/ENDREC (XII.1).

Each WRITEB command adds 7 bytes to the program.

XII.3.2 Command Format

WRITEB (file, record)

file - label of a file (defined in a CPL FILE statement) to which a record is written;
record - label of the record area (defined) in a CPL RECORD statement) from which data is written.

STATUS
The value of STATUS is affected by each WRITEB command. Possible STATUS values are 0 (Normal Completion), 2 (I/O Error - Type "B" and Type "C" files) AND "B" file or Invalid Key - Type "C" file).

Note: An attempt to write to a record number higher than the maximum number of records in the file will result in a STATUS of 3.

XII.3.3 Cautions

1. WRITEB should not be used with Type "C" spanned-sector files. Since the record length in such files may exceed a single sector, use the external subroutines PUTR with Type "C" spanned-sector files.

   For additional information on spanned-sector files, see Chapter 13 - SPANNED-SECTOR I/O and the APLIB REFERENCE MANUAL (PUTR).

2. For CPU-5 compability WRITEB should not be used after using GETR/NEWK, instead use PUTR.

XII.4 HOLD/FREE

XII.4.1 Usage

The HOLD command sets a flag in a shared, discrete-sector Type "C" file. This flag indicates that a record in the file is being accessed and that the sector containing that record should not be accessed by another partition. The FREE command turns off the flag set by HOLD.

The location of the next record to be accessed in a Type "C" file is stored in memory. Although this is similar to the procedure followed by Type "A" and Type "B" files, the record number in a Type "C" file may be changed directly by the program. If the program does not change the number of the record to be accessed, it will remain the same no matter the number of times it is read or written. The sector containing the next record to be accessed is noted in a table in the operating system when the HOLD command is given.

NOTE: On the CPU-5, the maximum number of entries in the table is equal to 6 times the number of partitions on the system. However, an individual partition may use more than 6 holds at a given time. On the CPU-6, there is a "sector hold table" within the system of a finite size and the maximum number of entries is system generated.

Each HOLD and FREE command adds 6 bytes to the program.

XII.4.2 Command Format

```
HOLD (file)
  or
FREE (file)
```

file - label of a file (defined in a CPL FILE statement) from which a sector is to be held or freed.

STATUS
The value of STATUS is affected by each HOLD command. Possible STATUS values after execution of a HOLD command are 0 (Hold

STATUS (cont.)
Sucessful), 1 (Required Sector Held by Another Partition) or 2 (Record Outside of File or Invalid Key). The HOLD command has an effect only on another HOLD. The value of STATUS after execution of a FREE command is always zero (0).

XII.4.3 Cautions

1. In order for HOLD to be effective, it must be used by all partitions with the ability to change the file. In addition, it must be used with the STATUS check mentioned above. The HOLD command does not prevent access to a sector by another partition.

2. On the CPU-5, HOLD/FREE may be used only with discrete-sector Type "C" files. On the CPU-6, HOLD/FREE may be used both with discrete-sector and spanned-sector Type "C" files.

3. For CPU-5 compatibility HOLD/FREE should not be used after using NEWK/GETK, instead use HLDR/FRER.

Chapter Thirteen - SPANNED-SECTOR INPUT/OUTPUT

OVERVIEW

No CPL commands exist for the processing of Type "C" spanned-sector files on the CPU-5. Special external subroutines located in APLIB must be used instead. While input and output functions performed by the operating system are limited by sector boundaries, these special subroutines perform their own I/O through IOC, a common subroutine.

> NOTE: Use of these subroutines on the CPU-6 is optional. However, to insure CPU-5/CPU-6 compatability, they should be used.

These subroutines are able to access records greater than a sector in length by using the extra bytes appended to the Record Control Block of a Type "C" spanned-sector file. Because the records may cross sector boundaries, the subroutines are also able to divide a record between two or more sectors. This utilizes otherwise wasted space at the end of a sector.

> NOTE: For additional information on the creation of Record Control Blocks see FILE (X.1.1).

Indexing routines are used on both the CPU-5 and CPU-6 to locate specific information. A file is prepared for these routines by the system utility VRINT. VRINT creates an indexing area at the beginning of the file. This indexing area is divided into index records which point to records in the prime data area of the file.

> NOTE: The VRINT utility is utilized on the CPU-5 system only.

Indexing routines employ an algorithm to convert the argument into the relative key of a record within the index area. Only integers and strings may be used as arguments; literals are not permitted.

NOTE: Integers are used if the argument is a 4- or 6-byte integer; strings are used if the argument is between 7 and 35 alpha-numeric characters in length.

Synonyms are different arguments which produce the same results. For this reason, each index record contains an index pointer which links all synonyms which are the product of a given algorithm. If the algorithm yields the relative key of an index record which does not have an identical argument, the chain of index records is read until the proper argument is located or read until it is determined that the record does not exist.

The subroutines GETK, NEWK, DELK and NEXK are used by the program to communicate with the indexing area. For additional information see the APLIB Reference Manual (GETK, NEWK, and DELK,).

GETK - sets the key integer of the file to the relative key for the specified argument.

NEWK - creates a new index record for a specified argument by reserving space for that record in the data area of the file. Since NEWK does not check for the prior existence of a record with the given argument, GETK must be performed first.

DELK - sets the prime data pointer in the index record to zero. This makes the index record available to the NEWK subroutine and removes the record from the chain of synonyms.

NEXK - used only on the CPU-6 system, NEXK searches forward in the key area for the next valid key. It sets the key integer to that key and the argument variable to the argument for that key.

NOTE: 1) No order of keys is guaranteed and the order that results is unpredictable.

2) If no additional keys exist, NEXK sets STATUS=1.

XIII.1 GETR

XIII.1.1 Usage

The GETR subroutine is used to transfer a record from a Type "C" spanned-sector file to a record area in memory.

NOTE: The APLIB GETR subroutine corresponds to the CPL READB command. Since it is an APLIB subroutine, it must be externalized. For additional information see EXTERNAL (V.1).

GETR may be used by both the CPU-5 and CPU-6 systems.

The complex of subroutines which includes GETR - along with the externals called by this routine - takes up approximately 950 bytes on the CPU-5.

XIII.1.2 Command Format

CALL GETR (file, record)

file - label of the Type "C" spanned-sector file (defined in a CPL FILE statement) to be read;
record - label of the record area into which the record is to be read.

NOTE: The CALL in CALL GETR is the CPL CALL command word. For additional information see CALL (IX.6).

Relative Key
The relative record number (i.e. relative key) must be given in the key integer assigned to the file.

STATUS
The value of STATUS is affected by each GETR subroutine call. Possible STATUS values include 0 (Normal Completion), 1 (Record Not Found) and 2 (I/O Error or Invalid Key).

XIII.2 PUTR

XIII.2.1 Usage

The PUTR subroutine transfers a record from a record area in memory to a Type "C" spanned-sector file.

NOTE: The APLIB PUTR subroutine corresponds to the CPL WRITEB command. Since it is an APLIB subroutine, it must be externalized. For additional information see EXTERNAL (V.1).

PUTR may be used by both the CPU-5 and CPU-6 systems.

The complex of subroutines which includes PUTR - along with the externals called by this subroutine - takes up approximately 950 bytes on the CPU-5.

XIII.2.2 Command Format

CALL PUTR (file, record)

file - label of the Type "C" spanned-sector file (defined in a CPL FILE statement) to which the record is to be written;
record - label of the record area from which the record is to be read.

NOTE: The CALL in CALL PUTR is the CPL CALL command word. For additional information see CALL (IX.6).

Relative Key
The relative key must be given in the key integer assigned to the file.

STATUS
The value of STATUS is affected by each PUTR subroutine call. Possible STATUS values include 0 (Normal Completion), 2 (I/O Error or Invalid Key) and 3 (End-of-Medium on Output).

XIII.3 HLDR/FRER

XIII.3.1 Usage

The HLDR subroutine is used to notify a program attempting to hold a record that that record is already being held. The FRER subroutine cancels that hold. Both HLDR and FRER are designed to deal with records that may cross sector boundaries.

> NOTE: The APLIB HLDR and FRER subroutines correspond to the CPL HOLD and FREE commands. HLDR/FRER may be used by both the CPU-5 and CPU-6 systems.

XIII.3.2 Command Format

```
CALL HLDR (file)
  or
CALL FRER (file)
```

file - label of the file containing the sector(s) to be held.

> NOTE: The CALL in CALL HLDR and CALL FRER is the CPL CALL command word. For additional information see CALL (IX.6).

All sectors containing any part of the record will be held by the HLDR command. A separate entry is made for each sector in the table of held sectors in the operating system. This causes the HLDR subroutine to fill a table more quickly than the HOLD command.

STATUS
The value of STATUS after a call to HLDR are (Hold Successful), 1 (Required Sector Held by Another Partition) or 2 (Record Outside of File or Invalid Key). The value of STATUS after a call to FRER is always equal to 0 (Hold Successful).

Chapter Fourteen - MISCELLANEOUS COMMANDS

XIV.1 CURP/CURSOR/CURB/CURS

XIV.1.1 Usage

The CURP, CURSOR, CURB and CURS commands are used to manipulate the display on the screen of a console device. Each command allows communication between the operator and the software.

Each CURP, CURSOR and CURS command adds 8 bytes to a program; each CRUB commands adds 6 bytes.

CURP

The CURP (cursor position) command determines the point at which the input/output of the next character will be displayed. CURP has no effect on data already displayed. Subsequent keyboard entries or program output will overlay existing data, however.

CURSOR

The CURSOR command is used to specify where a CRT cursor is to be placed. Although similar to the CURP command, CURSOR differs in the following ways:

1. The line (vertical) position is specified first.
2. The line (vertical) and column (horizontal) positions on a CRT begin with 0, not 1.
3. The line and column positions may be specified integer expressions. For example:

```
CURSOR (CRT, 0, K*2)
```

4. The column position is optional; if omitted there is a default to 0.

CURB

The CURB (cursor blank) command blanks a specified number of characters beginning with the present location of the cursor. CURB returns the cursor to its original position after output of the blank character(s).

CURS

The CURS (cursor substitute) command allows the program to display a character other than a blank. In all other respects it functions in the same manner as CURB.

XIV.1.2 Command Format

```
CURP (file, column, line)
  or
CURSOR (file, line, column)
  or
CURB (file, number)
  or
CURS (file, number, string)
```

file - label of the device (defined in a CPL FILE statement) where cursor manipulation will take place;

column - 4-byte integer or literal (or integer expression if dealing with the CURSOR command) where the cursor will be placed;

Note: The CURP command utilizes a column range of 1-80; CURSOR utilizes a range of 0-79.

line - 4-byte integer or literal (or integer expression if dealing with the CURSOR command) where the cursor will be placed;

Note: The CURP command utilizes a line range of 1-24; CURSOR utilizes a range of 0-23.

number - 4-byte integer or literal designaing the number of spaces to be filled;

Note: CURB fills designated spaces with blanks; CURS fills designated spaces with the specified character.

string - label of a 1-character string containing a character to be displayed by the CURS command.

XIV.1.3 Cautions

1. CURP, CURSOR, CURB and CURS may be used on CRT-type devices only.

2. CLASS=0 is used for CURP, CURSOR, CURB and CURS.

   Note: CLASS=0 is also the default.

XIV.2 DUMP

XIV.2.1 Usage

The DUMP command is used to display the contents of the "A", "B", "X", "Y", "Z" and "S" registers, the contents of the stack and the contents or a specified area of memory.

All displays are in hexidecimal notation. The memory dump is accompanied by an ASCII translation on the right side of the screen. The first line displays the contents of the registers and the first 18 bytes of the "S" stack. After the display is completed, control is returned to the next command following the DUMP command.

NOTE: The display will be terminated with an asterisk if more than 18 bytes are stored in the stack.

The DUMP command may be located anywhere in a program so long as it follows SYSTEM. A program may contain as many DUMP statements as necessary.

Each DUMP command adds 8 bytes to a program.

XIV.2.2 Command Format

DUMP (address1, address2)

address1 - specifies the address at which the memory dump begins;
address2 - specifies the address at which the memory dump ends.

XIV.2.3 Cautions

1. DUMP is to be used in program testing only; it should not be included in released software.

2. Only the simplest form of an offset label (i.e. a label plus a literal) may be used with the DUMP command.

   Note: Integers, variables, integer expressions, etc. may not be substituted for labels.

XIV.3 LOAD

XIV.3.1 Usage

The LOAD command allows one program to load another during execution. The second program may be loaded 1) at the same address as the first program, overlaying it completely, 2) above the first program and used like a subroutine or 3) at any point within the first program, overlaying the remainder of that program.

> NOTE: A full overlay must be a main program. All files must be closed prior to performing a full overlay; however, no end-of-step pro-processing is performed. A partial overlay must be a subprogram. This subprogram is used like a subroutine. It is not necessary to close the file involved unless a FILE statement is being overlain.

Each LOAD command adds 11 bytes to the program. An additional 2 bytes are added to program length for each item in the parameter list.

XIV.3.2 Command Format

LOAD (mask, label, option) [(name, name, ...)]

mask - label of string (defined in a CPL DEFINE statement) used to locate the program to be loaded;

label - location within the program at which the second program is to be loaded;

Note: If an offset is desired, the name plus the number of bytes to be added or subtracted must be included in brackets. For example, [label + n] or [label - n].

option - specifies where control is to be passed;

name - parameter to be passed to the overlay.

Note: This parameter functions in a manner similar to the "name" in a CALL statement (IX.6.2).

Mask
The mask contains the JCL name to be used for the full or partial overlay. The mask may contain the full name of the program to be called. If the overlay program has characters in common with the original .RUN file name, spaces may be used to represent these common characters. If the mask is less than 6 characters (CPU-5) or 10 characters (CPU-6), the unused characters of the mask are not masked by the .RUN filename.

Note: In a library file the mask is compared to the subfile name. An overlay for a library file must be a subfile in the same library as the .RUN file name.

Label
If a full overlay is specified, a literal 0 should be used for the label field. If the second program is to be loaded immediately above the calling program, HICORE should be specified.

HICORE, which is defined for every program, is a subroutine containing nothing but the label itself. It is composed of the address of the last byte of the program, plus 1. The link jobstream is designed so that HICORE will be linked last; this provides a label at the very end of the program.

Note: HICORE remains constant throughout a particular program. This allows multiple overlays to be loaded with each overlay replacing the previous one.

Option
Possible options include:

0 - full overlay

1 - subroutine partial overlay

Note: This allows the subroutine to be loaded and control to be passed immediately to the subroutine.

Options (cont.)
2 - partial overlay

Note: This allows the subroutine to be loaded only and control returned to the loading program. After loading, the entry address is located in the "B" register.

When option 1 or 2 is used, the parameter name(s) may be listed in parentheses. The overlay may then access this name(s) with a RETRIEVE statement(s). For additional information see RETRIEVE (IX.9).

Note: An overlay is a subprogram on the CPU-5 system, which must be linked by using S.SCPL rather than S.CPL. On a CPU-6, P.SCPL rather than P.CPL must be used.

STATUS
The value of STATUS is affected by each LOAD command. STATUS values include 0 (Overlay Loaded) or 4 (Overlay not Found).

XIV.3.3 Cautions

1. If programs are loaded at a label other than 0 or HICORE, necessary code could be overlain by the second program.

2. The CPU-5 does not provide automatic partition growth or shrinkage. If a partition is not made large enough initially, an ABORT 4 will result when attempting a LOAD. Since the operating system on the CPU-6 provides automatic partition adjustment, it is not necessary to monitor partition size.

Note: Maximum partition size on the CPU-6 is 32K.

XIV.4 ORIGIN

XIV.4.1 Usage

The ORIGIN command is used to change the location counter during compilation. ORIGIN is useful in redefining data areas within a program.

NOTE: Any piece of information in a program may be overlain by another piece of information by using ORIGIN. In this respect ORIGIN is the equivalent of the assembler directive ORG. For additional information on ORG, see the Assembler Language Manual.

XIV.4.2 Command Format

ORIGIN address

address - name of a program label, a data area or an assembler expression.
Note: The location counter is set to the 2-byte value of the address.

XIV.4.3 Cautions

1. When overlaying records, the last record defined should be as long as the longest record overlain.

2. Only the initial values - if any exist - in the final overlaying record should be considered valid. At program load time any initial values in the overlain record will be destroyed by any initial values in the overlaying record.

XIV.5 EQUATE

XIV.5.1 Usage

The EQUATE command is used to assign a label to a specific location. EQUATE equates an expression to a label. Anytime that label were used in a CPL program, the value of that expression would be used.

The following sequence of statements would allow an integer variable to be accessed either as a 4-byte (VAL) or 6-byte (?VAL) integer variable.

```
SET ?VAL:0
EQUATE VAL, [?VAL + 2]
```

XIV.5.2 Command Format

EQUATE label, expression

label - name to be assigned to a specific location;
expression - address.
Note: Only one EQUATE per name is allowed.

XIV.6 GTIME

XIV.6.1 Usage

The GTIME Command allows a CPL program to access the system time (if a clock exists) and store it in an integer/string. Primary use of GTIME is in the calculation of elapsed time.

> NOTE: The CPU-4 system stores the integer form of GTIME in 1/20,000 seconds of the day since 00:00:00. This integer must be divided by 20,000 to derive an even number of seconds. The CPU-5/CPU-6 system stores the integer form of the command in 1/10 seconds; must be divided by 10 for seconds.

The GTIME command adds 5 bytes to program length if an integer is used; 9 bytes are added if a string is used.

XIV.6.2 Command Format

```
GTIME (INTEGER, integer)
  or
GTIME (STRING, string)
```

integer - label of an integer (defined in a CPL INTEGER or SET statement) in which the system time (either in 1/20,000 seconds for the CPU-4 or 1/10 seconds for the CPU-5/CPU-6) is to be stored;

string - label of a character string (defined in a CPL DEFINE or STRING statement) in which the system time (hh:mm:ss) is to be stored.

XIV.6.3 Cautions

1. The string used to contain the STRING form of the time must be 8 characters in length.

XIV.7 LDATE/SDATE

XIV.7.1 Usage

The LDATE command is used to load a date into the "A" register. That date may then be stored either in integer or string form by SDATE.

To determine how each command (and keyword) affect program size, consult the following:

Command	Size in Bytes
LDATE (INTEGER, X)	3
LDATE (STRING, X)	6
LDATE (WORD, X)	3
LDATE (FILE, X)	6
LDATE (CURRENT)	3
LDATE (GRIN, X)	6
SDATE (INTEGER, X)	8
SDATE (STRING, X)	14
SDATE (WORD, X)	3
SDATE (FILE, X)	6
SDATE (GRIN, X)	10

XIV.7.2 Command Format

```
LDATE (form) [,label]
  or
SDATE (form, label)
```

form - type of date (indicated by the keywords STRING, INTEGER, WORD, CURRENT, FILE, GRIN) to be loaded or stored;

label - name of an integer, string or file.

Note: A "label" is not used with the keyword CURRENT.

Keyword Designation

1. STRING indicates that the date to be transferred is in 6-character or 8-character form with optional leading zeroes included. Input may be in the form mm/dd/yy or may have any non-numeric character(s) separating the components of the date. When used with SDATE, the date is output as mm/dd/yy.

The label used with the STRING keyword must specify a character string defined in a CPL STRING or DEFINE sta.ement.

2. INTEGER indicates that the date to be transferred is in the form of number of days since December 31, 1899. (For example, January 1, 1900 = 1.)

Note: INTEGER is useful in computing elapsed days.

The label used with INTEGER must specify a 4-byte integer defined in a CPL INTEGER or SET statement.

3. WORD functions in the same manner as INTEGER except it allows the integer form of a date to be stored in 2 bytes instead of 4.

Note: This option is seldom used since the date may not be manipulated in this form. In addition, offset labels must be used if this shorter form is to be used efficiently.

4. CURRENT is used only with the LDATE command. LDATE (CURRENT) causes the current system date to be loaded. It may then by output by SDATE in any form desired.

The CURRENT command requires no label.

5. FILE indicates that a date is to be transferred to or from the directory entry for a specified file.

The label used with FILE must specify a file defined in a in a CPL FILE statement.

6. GRIN indicates that the date is to be transferred in Gregorian Integer Form (i.e. mmddyy).

Note: GRIN is useful in computing elapsed months and years.

The label used with GRIN must specify a 4-byte integer defined in a CPL INTEGER or SET statement.

STATUS

With LDATE the value of STATUS is either 0 (Normal Completion) or 2 (Zero or other invalid date). With SDATE the value of STATUS is affected only if 0 or another invalid date is stored. In this case STATUS is set to 2. Otherwise, it remains unchanged (i.e. retains the value from LDATE).

Note: An invalid date causes a target integer to be set to 0 or a target string to be set to blanks.

XIV.7.3 Cautions

1. SDATE must immediately follow LDATE. Since the "A" register is used to hold the date, any command executed between LDATE and SDATE is likely to change the contents of this register. The only exceptions are the unconditinal GO TO (IX.2), CALL (IX.6) and RETURN (IX.8).

INTEGER DATE

JDATE = (IDATE/7*7)

0 = SAT
1 = SUN
2 = MON
3 = TUE
4 = WED
5 = THU
6 = FRI

XIV.8 Subscripted Variables

XIV.8.1 Usage

Elements in an integer table may be accessed by means of subscripted variables. The table name is subscripted with an integer literal (i.e. literally-subscripted) or an integer expression.

Changing a table element by means of a subscripted variable does not change the value of the work area. For example, the following input:

```
FORMAT  F01:N2
TABLE   A(2)
A=1
A(1)=2
A(2)=3
WRITE (CRT, F01) A, A(1), A(2)
```

will display -

```
1  2  3
```

NOTE: A(0) will not access the work area in table "A".

In integer assignment statements use of literally subscripted variables does not add to program length. Use of integer expressions as subsrcipts adds 5 bytes to the length of a program for each variable or literal contained in the expression.

Subscripted variables have replaced the TBLGET/TBLPUT commands in some systems. However, those programs employing TBLGET or TBLPUT may continue to use those forms. For additional information see TBLGET (XIV.9) and TBLPUT (XIV.10).

XIV.9 TBLGET

XIV.9.1 Usage

The TBLGET command transfers an entry from a string table to the work area assigned to that table. It may also be used with integer tables. Once that transfer is complete, that entry may be used like any other string or integer.

The compiler handles the TBLGET external subroutine like a CPL command. The size of the external subroutine is added to the program size when it is referenced within the program.

If the table-name is literally subscripted, the TBLGET command adds 7 bytes to the length of a program. Otherwise, the increase in length depends upon the complexity of the subscript.

XIV.9.2 Command Format

TBLGET table (integer)

table - name of a work area assigned to a table in a CPL TABLE statement;
integer - name of an integer or variable which specifies the number of the table entry to be accessed.

XIV.9.3 Cautions

1. Table handling routines require a service call each time a table entry is accessed. When a table must be searched an entry at a time, the operation of the program is slowed considerably.

2. Since no range checking is performed when a table is accessed, an entry outside the bounds of the table should not be accessed.

3. Although TBLGET is still operational for integer tables, subscripted variable integers provide a more efficient means of handling tables. For additional information see Subscripted Variables (XIV.8).

XIV.10 TBLPUT

XIV.10.1 Usage

The TBLPUT command transfers the contents of the work area assigned to a table to a specified position within that table.

The compiler handles the TBLPUT external subroutine like a CPL command . The size of the external subroutine is added to the program size when it is referenced within a program.

If a literal integer value is specified, the program size is increased by the length of the integer, plus 18 bytes. If an integer expression is used to specify the value or as the subscript for the table-name, the increase in program size depends on the complexity of the expression.

XIV.10.2 Command Format

TBLPUT table (integer) [: 'string'] or [:value]

table - name of a work area assigned to a table in a CPL TABLE statement;
integer - name of an integer or variable which specifies the number of the table entry into which the contents of the work area are to be transferred;
string - initial value being assigned to the specified entry in a string table;
value - initial value being assigned to the specified entry in an integer table.

Information resident in the work area at the time the command is given will be transferred to the specified entry. A new value may be assigned to this area before the transfer, if a colon (:) and a character string in single quotes (') or a numeric value follow the command.

XIV.10.3 Cautions

1. Table handling routines require a service call each time a table entry is accessed. When a table must be searched an entry at a time, the operation of the program is slowed considerably.

2. Since no range checking is performed when a table is accessed, an entry outside the bounds of the table should not be accessed.

3. Although TBLPUT is still operational for integer tables, subscripted variables provide a more efficient means of handling integer tables. For additional information see Subscripted Variables (XIV.8).

XIV.11 ADRLST

XIV.11.1 Usage

The ADRLST command allows a list of addresses to be coded. The object code generated for each address in the list is the 2 bytes of storage containing that address.

> NOTE: If a 1-byte literal is used in place of an address, that literal (in its 2-byte form) is generated rather than an address.

ADRLST is primarily used by assembler routines. As a piece of data, the command should not be located in the logic section of a CPL program.

XIV.11.2 Command Format

ADRLST (address, address, ...)

address - term used to designate a data item or a program label.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

XYZ