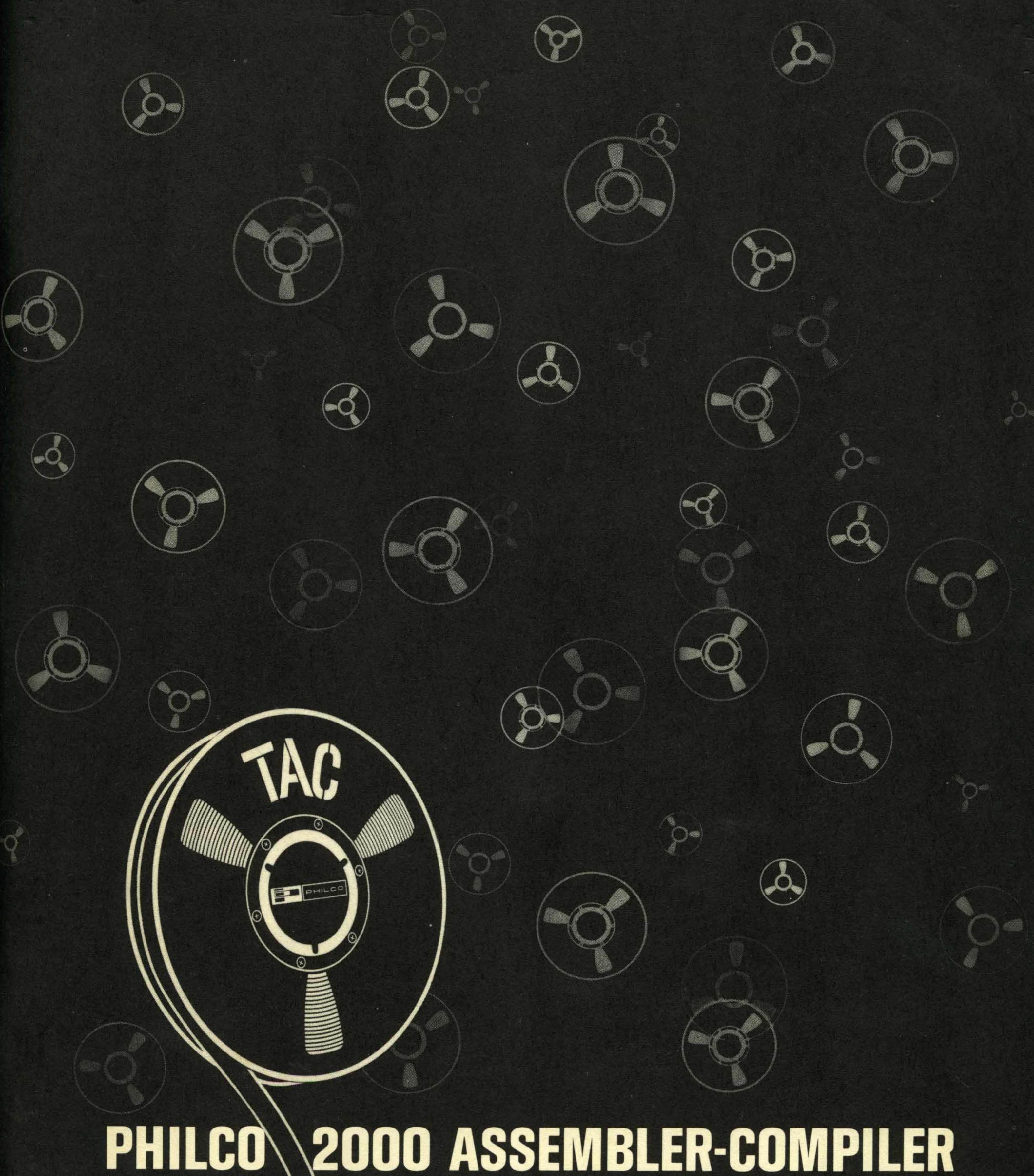


PHILCO 2000 ASSEMBLER-COMPILER

PHILCO ELECTRONIC DATA PROCESSING SYSTEMS

Preface

This manual discusses the Philco 2000 Assembler-Compiler, TAC. It defines the language of TAC, states the rules which must be followed when writing programs in this language, and describes the output produced.

The topics include TAC assembly-control instructions, i.e., control instructions to the TAC Assembly Program, as distinct from TAC Mnemonic instructions discussed in the Philco 210/211 and the Philco 212 Programming Manuals. Other topics include TAC Constants, Source and Object Program Formats, and information necessary for preparing a TAC language program to be run on a Philco 2000 computer.

No previous computer experience, other than a knowledge of the information presented in the above-mentioned programming manuals, is required for an understanding of the information presented herein.

Contents

		Page
Preface		iii
Introduction		xi
Chapter		
1	ELEMENTS OF THE SOURCE PROGRAM	1
	The Philco Coding Form	1
	The Identity and Sequence Field	3
	The Label Field	3
	The Location Field	5
	The Command Field	5
	The Address and Remarks Field	5
	Absolute Quantities	6
	Symbols	6
	Special Symbols	7
	Compound Symbols	7
	Address Arithmetic	7
	Index Register Notations	9
	Address Field Termination	9
	Remarks	9
2	CONTROL INSTRUCTIONS	11
	NAME	11
	IDENTIFY	12
	ASTOR	13
	AFEND	14
	Nullification of the AFEND Instruction	14
	SET	15
	SETSMALL and SETLARGE	16
	ASGN and SAME	17
	DEFINE	17
	SYMBOUT	19
	REFOUT	20
	COMSTOR	21
	SPACE	23
	PAGE	23
	SUBR	23
	END	24
3	COMMON SYMBOLS	25
	The C Label	25
	The Name COMMON	26

Contents (Continued)

Chapter		Page
4	CONSTANTS	27
	Pool and Non-Pool Constants	27
	Full-Word and Field Constants	28
	Full-Word Constants	28
	Fixed-Point Decimal Constants	28
	Floating-Point Decimal Constants	29
	Word Constants	29
	Location Constants	30
	Field Constants	30
	Alphanumeric Constants	31
	Octal Constants	32
	Hexadecimal Constants	32
	Numeric Constants	33
	Binary Constants	33
	Parameter Constants	34
	Command Constants	34
	Groups of Field Constants	35
5	LIBRARY ROUTINES	37
	General Description	37
	Subroutines	37
	Calling A Subroutine	37
	Writing A Subroutine	39
	Adding A Subroutine To The Library Tape	41
	Generators	41
	Calling A Generator	41
	Writing A Generator	43
	Symbols Permitting Communication Between TAC and Generators	45
	Adding A Generator To The Library Tape	47
	Macros	47
	Calling A Macro	47
	Skeleton Coding	48
	Adding Skeleton Coding to the Library Tape	49
6	OBJECT PROGRAM FORMATS	51
	Binary Object Program Cards	51
	A Relocatable Binary Deck	51
	The PMAX Card	52
	Symbol Definition Cards	53

Contents (Continued)

Chapter		Page
6 (Continued)	Relocatable Binary Instruction Cards	56
	The Relocatable End-Program Card	58
	An Absolute Binary Deck	60
	Absolute Binary Instruction Cards	61
	The Absolute End-Program Card	62
	Binary Object Program Tape	63
	RPL Object Programs	63
	The PROGRAM IDENTITY Control Word	64
	The LOAD Control Word	64
	The TRANSFER Control Word	65
7	MIXED INPUT DECKS	67
	The BITS Input Control Card	68
	The TACL Input Control Card	68
8	THE CODE-EDIT	69
	Contents of the Code-Edit	69
	Error Indications	74
	Serious Errors	74
	Possible Errors	75
	Generated Remarks	75
Appendix		
A	CONSOLE TYPEWRITER TYPE-OUTS	77
B	LOADING OBJECT PROGRAMS	79
C	CALLS ON FORTRAN SUBROUTINES	87
D	TABLE OF PHILCO CHARACTERS	89
E	TAC MNEMONICS	91
F	SUMMARY LIST OF TAC CONTROL INSTRUCTIONS	93

Figures

	Page
A TAC COMPILATION	xii
A PROGRAM RUN	xii
A TAC SOURCE PROGRAM	2
PHILCO 2000 CARD	3
A RELOCATABLE BINARY DECK	51
FORMAT OF THE PMAX CARD	52
FORMAT OF A SYMBOL DEFINITION CARD	54
FORMAT OF A RELOCATABLE BINARY INSTRUCTIONS CARD	56
FORMAT OF THE RELOCATABLE END-PROGRAM CARD	58
AN ABSOLUTE BINARY DECK	60
FORMAT OF AN ABSOLUTE BINARY INSTRUCTIONS CARD	61
FORMAT OF THE ABSOLUTE END-PROGRAM CARD	62
RPL COMPILATION OUTPUT	63
FORMAT OF AN RPL OBJECT PROGRAM	63
A MIXED INPUT DECK	67
LOADING OF RELOCATABLE OBJECT PROGRAMS	80
LOADING OF ABSOLUTE OBJECT PROGRAMS	85

Introduction

A TAC Compilation

The TAC Assembler Program is one of many automatic programming systems that are available with Philco 2000 computers. TAC is the basic system; most other programming systems translate into the language of TAC.

The TAC-language program which defines the operations to be performed by the computer is the *source* program. In a TAC compilation, the TAC Assembler:

- assemblies an *object* program in Philco 2000 machine language from the source program,
- compiles library routines into the assembled program, if desired,
- produces a Code-Edit on tape, listing both the source program and the compiled object program (see Chapter VIII),
- records the object program on tape in the object format specified.

The object format specified may be:

- Relocatable Binary Card Format (REL)
- Absolute Binary Card Format (ABS)
- Absolute Binary Tape Format (RPL)

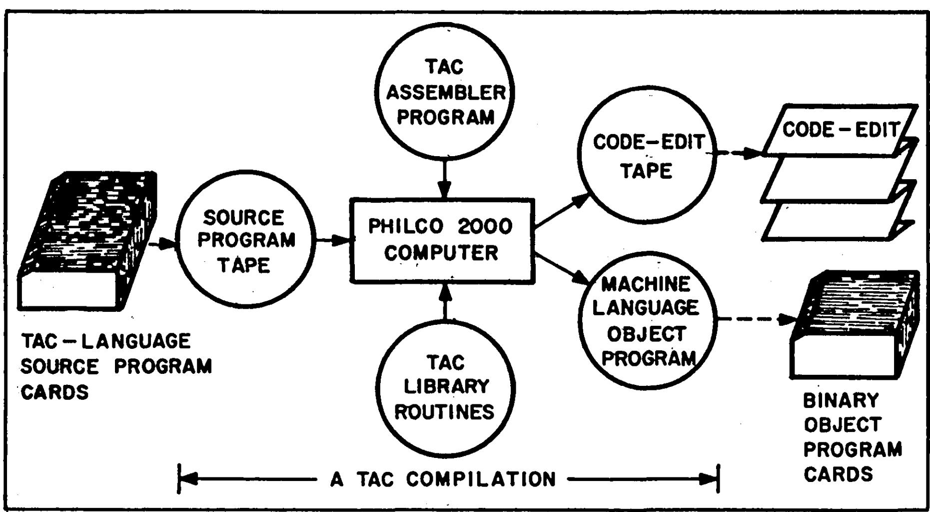
In an RPL or ABS compilation, all parts of the program (i.e., subroutines, separate logical sections, subprograms) are included in the compilation.

In an REL compilation, it is not required that all parts of the program be included in the compilation. Separately compiled program parts such as binary library subroutines or previously compiled subprograms can be included at load time.

Each object format reflects the use of a particular Loader (see Appendix B, *Loading Object Programs)*.

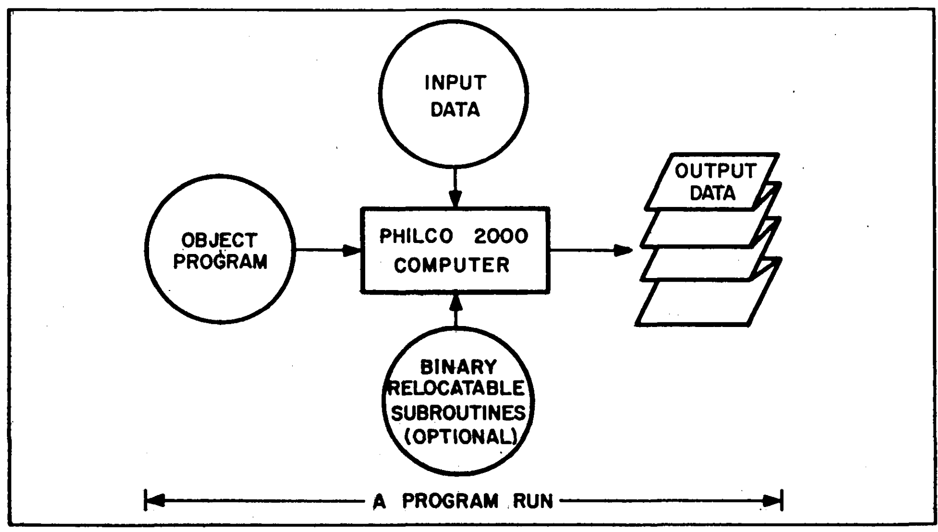
Subsequent to the compilation, the Code-Exit is printed off-line; binary cards of REL and ABS programs may be punched off-line.

The following diagram shows the relationship between source and object programs, and the TAC Assembler. Solid arrows (——→) denote on-line, continuous operation; broken arrows (---→) denote off-line operation.



A Program Run

In a program run, the object program on tape is loaded into memory, then executed. The following diagram depicts this process:



Chapter 1

ELEMENTS OF THE SOURCE PROGRAM

Philco Coding Form and Card Fields. Field Elements. Absolute Quantities. Symbols. Remarks.

A TAC source program consists of a series of instructions written in TAC-language format. Each instruction is written on a line of the Philco coding form (see Figure 1). An instruction or statement that is too long to fit on one line may be continued on succeeding lines, starting after column 24.

THE PHILCO CODING FORM

The columns of the coding form are divided into fields, as follows, corresponding to the fields on the Philco 2000 card (see Figure 2).

Identity and Sequence	L	Location	Command	Address and Remarks
1 8	9	10 16	17 24	25 80

PHILCO CODING FORM

Page 1 of 1

Program:			Programmer:	Date:

IDENTITY AND SEQUENCE (1–8)	L (9)	LOCATION (10–16)	COMMAND (17–24)	ADDRESS AND REMARKS (25–80)
			•	
			•	
SAMP0010	I		SAMPLE P	RØGRAM
SAMP0020	*		PRØGRAM	TØ EVALUATE A PØLYNØMIAL
SAMP0030			NAME	SAMPLE$
SAMP0040		CØEFF	ASTØR	N+1$ CØEFF IS STARTING LØCATIØN FØR CØEFFICIENTS
SAMP0050			ASGN	N,100$ DEFINE DEGREE ØF PØLYNØMIAL
SAMP0060		EXECUTE	TMD	L/CØEFF$
SAMP0070			TDXLC	,5$
SAMP0080	R		RPTNA	N+1$
SAMP0090			TMQ	0,5$
SAMP0100			FMMRS	1,5$
SAMP0110			SIXØ	N+1,5$ } CALCULATE CØEFFICIENTS (SAMP0060–SAMP0150)
SAMP0120	S	LØØP	FLØG10X	,5$
SAMP0130			TAM	,5$
SAMP0140			TMD	C/HLT,CØEFF+N+1;C/HLT,LØØP$
SAMP0150			AIXJ	1,5$
SAMP0160			PØLYVAL	ARG;N;CØEFF$ WITH ARGUMENT IN ARG,EVALUATE PØLYNØMIAL
SAMP0170			TAM	VALUE $
SAMP0180			PRT	F,VALUE $ PRINT VALUE ØF PØLYNØMIAL
SAMP0190		F	FØRMAT	(9H1VALUE = ,E17.5)$
SAMP0200			PØSITIØN	$
SAMP0210			JMP	M/5$
SAMP0220			END	EXECUTE $
			•	

TF-25

Figure 1 – A TAC Source Program

After the program is written, it is punched on cards, each line of the coding form corresponding to one card. The program is then transferred from card to magnetic tape in an off-line operation, before being read into the computer for compilation. During compilation, the TAC Assembler assigns a location to each mnemonic command, and computes its corresponding address field.

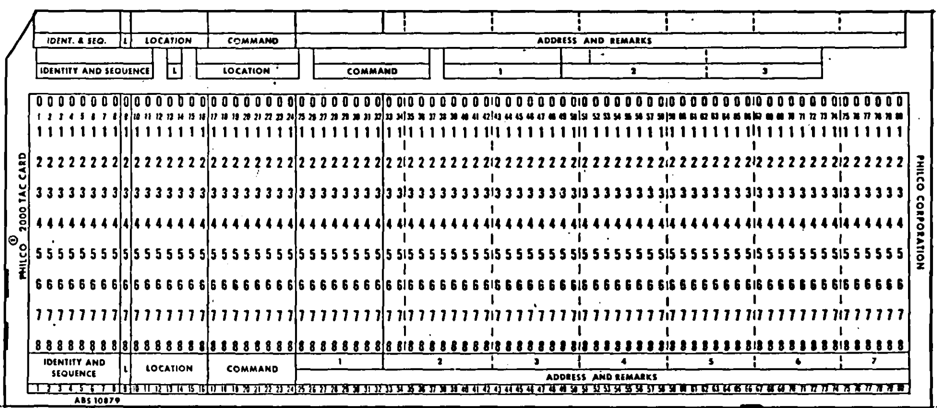


Figure 2 – Philco 2000 Card

THE IDENTITY AND SEQUENCE FIELD (Columns 1-8)

The identity and sequence field may be:

- Blank (spaces), or contain
- Alphanumeric† identity and sequence information, used to indicate to the programmer or operator the sequence of the source program cards, and to identify the program to which these cards belong.

Information in the field has no effect on the compilation.

THE LABEL FIELD (L) (Column 9)

The label field may contain:

- A blank, or
- Any of eleven TAC label characters: B, C, D, E, F, I, L, P, R, S, or * (asterisk).

† Any combination of alphabetic and/or numeric characters.

Each of these eleven label characters performs a special control function, as described below.

LABEL CHARACTER	FUNCTION
I	Indicates that the program's identity (an alphanumeric name identifying the program) is specified in columns 17-32. The identity is comprised of all sixteen characters (spaces included) in these columns. The "I-Card" must be the first card of the program.
L	Indicates that the instruction written in the command and address fields is to be placed in the *left* half of an instruction word. If the previous instruction already occupied the left half of the word, a NOP instruction will be inserted in the right half of the word, and this instruction placed in the *left* half of the succeeding word.
R	Indicates that the instruction is to be placed in the *right* half of an instruction word. If the previous instruction already occupied the right half of the word, a NOP instruction will be inserted in the left half of the next word, and the instruction placed in the *right* half of this word.
C	Indicates that the symbol in the *location field* of the instruction, or in the *address field* of an ASGN, SAME, SYMBOUT, or REFOUT instruction, is a *common symbol* (see page 28). A *common symbol* is a symbol which has a *singular* definition throughout the program.
B	Performs the functions of both C and L labels.
D	Performs the functions of both C and R labels.
E	Indicates the *end* of the effect of a previous AFEND control instruction (see page 16). This character is written in the label field of an AFEND instruction only.
P	Indicates that the *constant* beginning in the command field is a Pool Constant (see page 30).
S	Indicates that the command and address fields contain a TAC subroutine call (see page 41).
F	Indicates a FORTRAN subroutine call. See Appendix C.
*	Indicates that this line (card) contains remarks only, and has no effect on the compilation. (See page 12).

† For a review of the format of an instruction word, see the Philco 210/211 Programming Manual (TM-10), or the Philco 212 Reference Manual (TM-29).

THE LOCATION FIELD (Columns 10-16)

The location field is used to tag (assign a symbolic name to) an instruction, as a means of referencing that instruction. This field may be:

- Blank, or may contain
- A symbol* of from 1 to 7 characters.

If the location field contains a blank, and the command field contains a TAC mnemonic, the resulting instruction will be assigned the next consecutive available memory location.

If the location field contains a symbol (tag), all references to this symbol will be linked to the corresponding memory location assigned to the tagged instruction.

When used in the location field of a mnemonic, or in the location field of a SET, SETLARGE, SETSMALL, ASTOR, or COMSTOR control instruction, the symbol is considered *defined*, and it must not appear again in the location field, or appear as a symbol *(symb)* to be defined in the address and remarks field of an ASGN or SAME instruction (see page 17), in the same program section. If a symbol is doubly defined, the first definition is used and an error indication is printed on the Code-Edit.

THE COMMAND FIELD (Columns 17-24)

The command field is used to specify:

- A Philco 2000 Mnemonic
- The command portion of any TAC Control Instruction (see Chapter 2)
- A Constant, as described in Chapter 4
- Subroutine Calls, Generator Calls, or Macro Calls, as described in Chapter 5.

THE ADDRESS AND REMARKS FIELD (Columns 25-80)

The address and remarks field is used to specify address field elements, pool constants (see Chapter 4), and/or remarks.

An address field element may be:

- An Absolute Quantity
- A Symbol, of from 1 to 23 characters
- A Special Symbol
- A Compound Symbol
- An Index-Register Notation

TAC computes the resultant instruction address from the address field elements specified.

*A symbol is any group of alphanumeric characters with at least one character alphabetic.

Absolute Quantities

An absolute quantity is a decimal or octal integer value written as follows:

GENERAL FORMS	EXAMPLES
xxxxx	12345
M/*yyyyy*	M/33333
where *xxxxx* is a decimal integer of up to five digits, and M/ indicates that *yyyyy* is an octal integer of up to five digits. Neither *xxxxx* nor *yyyyy* exceeds 32,767 decimal.	

Symbols

A symbol appearing in the address and remarks field may be written in either of two forms:

GENERAL FORMS	EXAMPLES
Symb	ALPHA
Name.Symb	PROGRAM.ALPHA
where *Symb* represents a symbol of up to 23 characters in length, and *Name* is the name of a program section (see page 13).	

During compilation, these symbols will be defined (assigned a value) by virtue of their appearance in:

- the location field of a mnemonic, or SET, SETLARGE, SETSMALL, ASTOR, or COMSTOR control instruction, or in
- the address and remarks field of an ASGN or SAME instruction (see page 19)

Once a symbol is thus defined, it cannot be redefined. Any attempt at redefinition will result in the symbol being doubly defined; in which case, the first definition is used, and an error indication is printed on the Code-Edit.

Undefined symbols (symbols not defined as above) in the program are defined by the TAC Assembler. These undefined symbols are called TEMPORARIES, and are so designated on the Code-Edit.

Preset symbols such as index-register designations (1X, 2X, etc., see page 11) are predefined in the symbol table of TAC, and cannot be redefined during a compilation. They can only be changed by modifying their assignment in TAC itself.

A detailed discussion of the *Name.Symbol* form is discussed under the NAME control instruction, page 13.

Special Symbols

The notations (P), (PMAX), and *n*H are special TAC symbols, each with a special meaning. These notations are written as follows:

GENERAL FORMS	EXAMPLES
(P)	(P)
(PMAX)	(PMAX)
*n*H	7H
where *n* is a decimal integer other than zero.	

The notation (P) refers to the current contents of the Program Counter, and represents the location of the current instruction (i.e., the location of the instruction in which the notation (P) appears).

The notation (PMAX) represents the left address immediately following the largest memory address occupied by the program.

The notation *n*H refers to *half* of a location word. It represents the nth half-word relative to (following or preceding) another half-word.

Compound Symbols

A Compound Symbol is an address field notation that consists of two or more absolute quantities, and/or symbols, and/or special symbols, separated by arithmetic operators. These operators are + (plus), - (minus), * (asterisk), and : (colon), denoting addition, subtraction, multiplication, and division, respectively.

Address Arithmetic

To determine the resultant instruction address that the Compound Symbol represents, address arithmetic is performed from left to right of the Compound Symbol as follows: multiplications and divisions first, then additions and subtractions. If the final result of the address arithmetic performed is negative, the two's complement of the result is used as the resultant address.

Example

Assume the symbols ALPHA and BETA have been assigned the values 4000 and 9000 respectively, and the Program Counter currently contains the value 7000. Then, the Compound Symbols:

```
ALPHA+50$
BETA-ALPHA:2$
ALPHA*M/10$
(P)+5H$
```

represents the resultant addresses (memory locations) 4050, 7000, 32,000, and the right half of location 7002, respectively.

The resultant address computed depends on the object program format specified (REL, ABS, or RPL, see pages 51, 60 and 63), and must be one of the following:

1. Absolute
2. Relative to program origin (the location of the first instruction of the program)
3. Relative to the common storage area (see page 83)
4. Symbolic with an increment*

A symbol is considered *absolute* when it is assigned a numeric value in an ASGN or SAME instruction (see page 19); it is considered *relative* when defined by TAC in terms of the program counter during compilation.

When REL object format is specified, the following restrictions apply:

- If *additions only* are to be performed, only one of the operands may be relative; all other operands must be absolute.
- If *subtractions only* are to be performed, the operands can be absolute or relative; however, if relative, they must be relative to either the *common origin* or to the *program origin.*
- If additions and subtractions are to be performed, there must be at least *n-1* relative operands involved in the subtraction process for the *n* relative operands involved in the addition process.
- Multiplication and division are permitted between two operands only if both are absolute.
- No more than one of the operands involved in address arithmetic may be a REFOUT symbol (see page 23).
- The result of address arithmetic performed on the address in an ASTOR or COMSTOR instruction (see pages 15 and 25) must be absolute.

There are no address arithmetic restrictions with ABS or RPL format.

*Resulting from a REFOUT instruction, see page 23.

Index Register Notations

An Index Register Notation is a notation, in the address and remarks field, that contains a reference to an Index Register. The general forms of Index Register Notations are:

GENERAL FORMS	EXAMPLES
,i or *val,i* where *i* is a decimal integer (0-8) or a symbol, and the characters *,i* represent Index Register *i*. When *i* is a decimal integer, an X may be written following it as an alternate way of specifying the Index Register. The symbol *val*, if present, represents a value in the form of an absolute quantity, symbol, special symbol, or compound symbol.	,5 100,5X ,BETA M/1000,BETA

The address formed by TAC has *V* and *N* fields of *val* and *i* respectively. During program execution, the resultant address is the sum of the contents of Index Register *i* and the value *val*.

Assuming that Index Register 5 contains the value 2000, and that the symbol BETA has been assigned the value 5, the above examples refer to memory locations 2000, 2100, 2000, and 2512, respectively.

Address Field Termination

Address field elements are terminated by a $ character or by means of a previous AFEND instruction (see page 16). Where necessary, these address elements may be continued into succeeding address and remarks fields until terminated by a $ character or by means of a previous AFEND instruction.

When address elements continue into the address and remarks field of succeeding (continuation) cards, and label, location, and command fields must be blank.

Remarks

Remarks may be written after the $ character terminating an address field element, or after the column specified by a previous AFEND instruction. (The * label discussed on page 5 is also used as an alternate way of writing remarks.) Remarks have no effect on the compilation.

Chapter 2

CONTROL INSTRUCTIONS

Control Instructions To The TAC Assembler Program.

Control instructions provide the TAC Assembler with information necessary for performing the following control functions:

- Identify a program or program section (*NAME* Instruction)
- Specify the memory size and number of index registers of the source computer (*IDENTIFY* Instruction)
- Reserve storage locations (*ASTOR* Instruction)
- Define the length of the address fields of instructions (*AFEND* Instruction)
- Alter the program counter (*SET*, *SETSMALL*, and *SETLARGE* Instructions)
- Define symbols (*ASGN* or *SAME* Instruction)
- Define new instructions in terms of current acceptable ones (*DEFINE* Instruction)
- Permit intercommunication between separately compiled relocatable object programs (*SYMBOUT* and *REFOUT* Instructions)
- Reserve common storage area for relocatable object programs (*COMSTOR* Instruction)
- Control line and page spacing on the High-Speed Printer (*SPACE* and *PAGE* Instructions)
- Reference a subroutine (*SUBR* Instruction)
- Indicate the end of a program (*END* Instruction)

The function of each control instruction is discussed in detail below. The instructions do not affect the flow of operation in the program, nor do they (except the SUBR Instruction) introduce any additional coding in the object program.

NAME

The NAME instruction is used to identify a program or program section (subprogram). This instruction permits TAC to distinguish between different program sections making up the complete

program, and between identical symbols used in the different program sections. The general form of the NAME instruction is:

GENERAL FORM		EXAMPLES	
Command	Address and Remarks	Command	Address and Remarks
NAME	*p*	NAME	ALPHA
where *p* is the name of a program or program section (subprogram). Program and subprogram names may be one to eight alphanumeric characters long, the first character of which must be alphabetic.		NAME	BETA101

All symbols following a NAME instruction are identifiable by the name *(p)* appearing in that NAME instruction. If the NAME instruction is omitted from a program, TAC assumes the name *NONAME* for that program.

The NAME instruction *must* be used in cases where a program comprises two or more sections. As many as 256 different NAME sections may occur in a program. If this amount is exceeded, an appropriate error indication is printed on the Code-Edit.

When an instruction of a subprogram is to be referenced from *outside* the subprogram, the name of the subprogram must be prefixed to the location of the instruction referenced. A period is used to separate the prefixed subprogram name from the location referenced. For example, address DELTA in subprogram B must be referred to as address B.DELTA when referenced from outside subprogram B.

The name COMMON has special meaning. Use of this name is discussed under *Common Symbols,* page 28.

IDENTIFY

The IDENTIFY instruction is used to indicate to TAC that the object computer (the computer on which the object program is to be run) differs in memory size and/or in the number of index registers from the source computer (the computer on which the source program is compiled).

The general form of this instruction is:

GENERAL FORM		EXAMPLES	
Command	Address and Remarks	Command	Address and Remarks
IDENTIFY	*m*K,*n*X	IDENTIFY	32K,8X
		IDENTIFY	16K
where *m* is 8, 16, or 32, indicating an 8192, 16,384, or 32,768 word object computer, respectively; and *n* is an integer* indicating the number of index registers of the object computer.		IDENTIFY	,8X

The parameters *m*K and *n*X are optional; either one may be omitted from the IDENTIFY instruction. The parameter *m*K may be omitted if source and object computers have the same size memory; the parameter *n*X may be omitted if both computers have the same number of index registers. Where source and object computers do *not* differ in memory size and number of index registers, the entire IDENTIFY instruction may be omitted from the program.

ASTOR

The ASTOR instruction is used to reserve a specified number of storage locations in memory, outside the area occupied by the program. The general form of this instruction is:

GENERAL FORM			EXAMPLES		
Location	Command	Address	Location	Command	Address
Symb	ASTOR	*n*	ARRAY	ASTOR	128
			BETA	ASTOR	M/7447
where *Symb* is a symbol, and *n* is an absolute quantity, or symbol or compound symbol defined in the program			KAPPA	ASTOR	SIZE
Symb represents the symbolic *address* of the first location of the storage area reserved; *n* represents the *number* of storage locations reserved.			DELTA	ASTOR	N*128-128

*If the value of *n* exceeds the computer memory size, the value is reduced modulo that size. If address arithmetic is indicated by *n*, the result of the address arithmetic must be absolute.

There can be as many ASTOR instructions in a program as memory will permit. The storage locations are reserved contiguously, in the order of appearance of the ASTOR instructions in the program.

AFEND

The AFEND (*A*ddress *F*ield *END*) instruction permits a programmer to terminate address fields without having to write a $ character after each address field. The AFEND specifies where the address field of individual instructions end. The general form of this instruction is:

GENERAL FORM		EXAMPLE	
Command	Address and Remarks	Command	Address and Remarks
AFEND	*n*$	AFEND	42$
where *n* is any decimal integer 25–80, indicating the column of the coding form or card where each subsequent address field is to be assumed terminated.			

The AFEND instruction causes TAC to assume a dollar sign in column *n* of each subsequent instruction, until the AFEND is nullified (see below). Remarks may be written after column *n* of the instructions.

Nullification of the AFEND Instruction

The effect of an AFEND instruction may be temporarily or permanently nullified at a subsequent point in the program.

Temporary nullification occurs for:

- Any subsequent instruction whose address is terminated with a $ character prior to column *n* of the AFEND.
- Any subsequent instruction whose address field contains a non-space character in column *n*. Such instructions *must* therefore be terminated by a $ character.
- Macro or Generator Calls. If remarks are associated with a Call, a $ character must precede the remark.
- Instructions which are included in the program as a result of a Subroutine, Macro, or Generator call. An AFEND in the main program does not affect such inserted coding. The inserted coding may contain their own AFEND instructions.

Permanent nullification occurs when:

- A subsequent AFEND instruction specifies a new value for n.
- A subsequent AFEND instruction with an E in its label field is encountered. The E indicates the End of the effect of the previous AFEND instruction. The format of this AFEND is:

L	Location	Command	Address and Remarks
E		AFEND	$

The effect of the AFEND instruction on alphanumeric constants is discussed on page 34.

SET

The SET instruction is used to set the program counter. By means of this instruction the programmer can specify the location of any instruction in his program, and can reserve memory locations *within* his program. The general form of this instruction is:

GENERAL FORM		EXAMPLES	
Command	Address and Remarks	Command	Address and Remarks
SET	*addr*	SET	512
where *addr* is any address field element except the Symbol (PMAX) and Index Register Notations.		SET	M/1000
		SET	BETA
		SET	(P)+50
		SET	(P)+BETA

For REL compilations, the compilation base (initial program counter setting) is zero; all relocatable instruction addresses are compiled with zero as the base. In ABS and RPL compilations, TAC assumes the compilation base preset by the installation. This base depends on the size of the operation system used, and is fixed by the installation*.

The SET instruction causes the program counter to be reset to the address element specified in its address and remarks field, thus changing the address at which the next and succeeding instructions are to be placed in the program. For example,

*If SYS (the Philco 2000 Operating System) is the operating system used, the compilation base is 1000 octal. If 32K SYS is the operating system, the compilation base is 10,000 octal.

if the following coding,

L	Location	Command	Address and Remarks
		.	
		.	
		NAME	TRACK$
	START	TMD	ALPHA$
		.	
		.	
		.	
		SET	2000$
	RHO	TMA	BETA$
		.	
		.	

appeared in a program, location RHO would be assigned the address 2000*. If the program counter read 1500 previously, 500 locations (words) would be skipped over when the SET is executed.

All symbolic address field elements must be defined prior to their appearance in a SET instruction. However, symbols defined in ASTOR or COMSTOR control instructions must not appear as an address field element of a SET instruction.

SETSMALL and SETLARGE

These two control instructions perform basically the same function as the SET instruction. The SETSMALL instruction causes the program counter to be reset to the *smaller* of the two address elements appearing in its address and remarks field; the SETLARGE instruction causes the program counter to be reset to the *larger* of the two address elements appearing in its address and remarks field. In all other respects, these two instructions are similar to the SET instruction. All rules concerning the SET apply.

The general forms of these two instructions are:

GENERAL FORMS		EXAMPLES	
Command	Address and Remarks	Command	Address and Remarks
SETSMALL	$addr_1$,$addr_2$	SETSMALL	A + 1, B + 2
SETLARGE	$addr_1$,$addr_2$	SETLARGE	C, D
where $addr_1$ and $addr_2$ are address field elements except the symbol (PMAX) and Index Register Notations.			

*In relocatable programs this address is relative to the program's loading origin.

ASGN and SAME

The ASGN and SAME instructions perform the same function. Either ASGN or SAME is used to assign a value to a symbol. The value assigned may be absolute or symbolic. The general forms of these two instructions are:

GENERAL FORMS		EXAMPLES	
Command	Address and Remarks	Command	Address and Remarks
ASGN	*symb,n*	ASGN	ALPHA, 2000
SAME	*symb,n*	ASGN	ALPHA, M/3720
where *n* is a value in the form of an address field element (other than an Index Register Notation), and *symb* is a symbol to which the value *n* is assigned.		SAME	BETA, TAU
Several symbols (*symb*) may be defined by a single ASGN or SAME instruction. In this case, each "*symb,n*" combination must be separated by semicolons, as follows:		SAME	BETA, EPSILON+50
ASGN	$symb_1,n_1;symb_2,n_2;...$	ASGN	A,2000;B,POS;C,M/1500
SAME	$symb_1,n_1;symb_2,n_2;...$	SAME	A, D-40; B, (P)+50
		ASGN	NP, M/1051+1H

As indicated in the examples, the parameter *n* may specify address arithmetic; the parameter *symb* may not.

A "C" in the label field of an ASGN or SAME instruction makes the symbols (*symb*) in the address and remarks field of the instruction *common* throughout the program. (See page 28.)

DEFINE

The DEFINE instruction is used to define or redefine a command. The general form of this instruction is:

GENERAL FORM		EXAMPLES	
Command	Address and Remarks	Command	Address and Remarks
DEFINE	*symb,c*	DEFINE	LLD, TMD
where *symb* is a symbol 1-8 characters long, and *c* is a command, or control word written as a 16-digit octal constant.		DEFINE	JBT, NOP
		DEFINE	PAUSE, O/0000000401150001

This instruction causes all *subsequent* symbols *symb* in the command field to be interpreted as the command or control word *c*. (The repeat mnemonic RPT is a special case, and must *not* appear as either *symb* or *c* in a DEFINE instruction.) The machine coding produced for *symb* upon compilation is that normally produced for *c*.

As indicated in the above examples, the DEFINE instruction permits a programmer to:

1. Define a new command in terms of an existing TAC command.
2. Redefine one TAC command in terms of another.
3. Define a computer half-word in terms of a control word of the following format:

BITS	CONTENTS
0-16	A 17-bit *mask* which is used to insert the S-bit, address field, and F-bit into the instruction. If the F-bit is a function of the address representation, bit 16 must be one. If the F-bit is determined by the mnemonic in bits 21-28, bit 16 must be zero.
17-20	Zeroes.
21-28	The command portion of the half-word being defined.
29	One or zero: A one causes a *possible error indication* to appear on the Code-Edit if the address field of the instruction refers to a pool constant.
30	One or zero: A one causes a *possible error indication* to appear on the Code-Edit if the address field of the instruction refers to an index register.
31	One or zero: A one causes a *possible error indication* to appear on the Code-Edit if the address field does *not* refer to an index register.
32-33	Address F-bit indicator. 00: Indicates that the address of the instruction may be a left or right address. 01: Indicates that the address of the instruction should be a left address. 11: Indicates that the address of the instruction should be a right address.
34	Zero.

BITS	CONTENTS
35	One or zero: A one causes a *possible error indication* to appear on the Code-Edit if the address field of the instruction refers to a *temporary* address, or to the location of an ASTOR instruction or a COMSTOR instruction.
36-37	00: Indicates that the instruction is either a left or right instruction. 01: Indicates that the instruction is a left instruction. 11: Indicates that the instruction is a right instruction.
38-46	Zeros.
47	One.

SYMBOUT

When relocatable programs are loaded together, symbols which are defined in one program may reference, or be referenced by, instructions in the other programs. To permit such inter-program referencing, SYMBOUT instructions (and REFOUT instructions, see next page) identifying the symbols must be included in the programs.

The SYMBOUT instruction permits TAC to supply the relocatable program loader with the definitions of the symbols which appear in its address and remarks field.

The general form of the SYMBOUT instruction is:

GENERAL FORM		EXAMPLES	
Command	Address and Remarks	Command	Address and Remarks
SYMBOUT	*addr*	**SYMBOUT**	LOOP
		SYMBOUT	ALPHA.BETA
		SYMBOUT	RHO; TAU; BETA

where *addr* is a symbol (*symb* or *name.symb*) that is defined in the program containing the SYMBOUT and that may be referenced by another program. (The element *symb* may be composed of from 1 to 8 characters.)

Any number of symbols may be specified in the address and remarks field of a SYMBOUT instruction. If several symbols are specified, they must be separated by semicolons, as follows:

SYMBOUT | $addr_1$;$addr_2$; ....

The symbol specified in the address and remarks field of the SYMBOUT instruction is defined in the program containing the SYMBOUT, and both the symbol and its definition are forwarded to the relocatable program loader on SYMBOL DEFINITION CARDS, produced during compilation from the SYMBOUT cards (see page 58). The REL loader applies the definition to the symbol wherever it appears in another program, provided that the symbol also appeared in a REFOUT instruction in that program.

A "C" in the label field of a SYMBOUT instruction causes the symbol(s) in the address and remarks field of the instruction to be made *common* throughout the program (see page 28).

REFOUT

The REFOUT instruction is used in conjunction with the SYMBOUT instruction to permit inter-program referencing of symbolic locations.

The REFOUT instruction indicates to TAC that the symbols appearing in its address and remarks field are defined in another program, and that the definitions of these symbols will be available to the loader at load time.

The general form of the REFOUT instruction is:

GENERAL FORM		EXAMPLES	
Command	Address and Remarks	Command	Address and Remarks
REFOUT	*addr*	REFOUT	LOOP
		REFOUT	ALPHA.SPOT
where *addr* is a symbol that is defined in another program, and appears in a SYMBOUT instruction in that program. Any number of symbols may be specified in the address and remarks field of a REFOUT instruction. If several symbols are specified, each symbol must be separated by a semicolon as follows:		REFOUT	A.RHO;B.TAU; G.BETA
REFOUT	$addr_1$; $addr_2$;		

This instruction causes all references to these symbols to be indicated *symbolically* in the relocatable binary deck produced from the compilation (see page 55). During loading, the relocatable program loader obtains from SYMBOL DEFINITION CARDS the definitions of these symbols.

REFOUTs are produced for subroutine calls automatically by TAC. (See pages 41 and 43). Any symbol in the program with the same name as the called subroutine will automatically be considered a REFOUT symbol.

A "C" in the label field of a REFOUT instruction causes the symbols in the address and remarks field of the instruction to be made *common* throughout the program (see page 28).

The following example shows the use of the SYMBOUT and REFOUT instructions in the two separately compiled programs PROGA and PROGB:

L	Location	Command	Address and Remarks
		.	
		.	
		NAME	PROGA
		.	
		SYMBOUT	LOOP $
		.	
	LOOP	TMA	ALPHA $
		.	
		.	
		.	
		END	$

L	Location	Command	Address and Remarks
		.	
		.	
		NAME	PROGB
		.	
		REFOUT	PROGA.LOOP $
		.	
		.	
		JMP	PROGA.LOOP $
		.	
		.	
		END	$

Because the symbol LOOP is defined in program PROGA and not in program PROGB in which it is referenced, an appropriate SYMBOUT instruction specifying the symbol is included in program PROGA, and an appropriate REFOUT in program PROGB.

COMSTOR

The COMSTOR instruction is used to reserve a specified number of words in memory as a common storage area, to be used by all relocatable programs loaded together.

The general form of this instruction is:

GENERAL FORM			EXAMPLES		
Location	Command	Address	Location	Command	Address
symb	**COMSTOR**	*n*	ALPHA	**COMSTOR**	80
where *symb* is a symbol representing the location of the first word of the area reserved, and *n* is any address field element (except a Special Symbol or an Index Register Notation) representing the *number* of memory locations that are reserved.			BETA	**COMSTOR**	500
			DELTA	**COMSTOR**	M/700

The common storage area is reserved outside the boundaries of the program containing the COMSTOR. The location of the area reserved is not defined at compilation time but at load time. At load time, the first word of common memory reserved is made identical for all relocatable programs loaded together.

There can be as many COMSTOR instructions in a program as memory will permit. The *total* amount of common storage specified by a program is equal to the sum of the amounts of storage specified by the individual COMSTORs in the program. The COMSTOR areas are reserved contiguously, in the order of appearance of the COMSTORs in the program.

When several relocatable programs are loaded together, the amount of common storage reserved will be the *largest* of the total amounts specified for the individual programs. For example, if the COMSTOR instructions in the above example represent the total amount of storage specified for three separately compiled relocatable programs loaded together, a single common storage area of 500 words would be reserved for use by all three programs.

For relocatable programs, the common storage area starts immediately after the last word occupied by the operating system. At load time all addresses in the common storage area are adjusted relative to common origin (the location of the first word in this area) by the relocatable program loader.

Although the COMSTOR instruction is used primarily in relocatable programs, it may also be used in absolute programs. When COMSTOR is used in an absolute program, the common storage area reserved starts after the last ASTOR area, and the overall program size indicated as (PMAX) will reflect the inclusion of this area.

SPACE

The SPACE instruction permits the programmer to control the vertical spacing of information on the Code-Edit. SPACE causes the High-Speed Printer to advance the Code-Edit a specified number of lines. The general form of this instruction is:

GENERAL FORM		EXAMPLE	
Command	Address and Remarks	Command	Address and Remarks
SPACE	*n*	SPACE	10
where *n* is any decimal integer from 1 to 32767, indicating the number of lines to be skipped.			

The value *n* does not include the margins at the top and bottom of each Code-Edit page.

PAGE

The PAGE instruction is used to control the amount of information to be printed on a page of the Code-Edit. PAGE causes the High Speed Printer to advance the Code-Edit to the top of the next page. The form of this instruction is:

GENERAL FORM		EXAMPLE	
Command	Address and Remarks	Command	Address and Remarks
PAGE		PAGE	

SUBR

The SUBR instruction is used to call a subroutine. (See also, the use of the S label, pages 5 and 41.) This instruction causes TAC to include the subroutine specified in the address and remarks field of the instruction in the compiled program. The general form of this instruction is:

GENERAL FORM		EXAMPLE	
Command	Address and Remarks	Command	Address and Remarks
SUBR	*name*	SUBR	XORD
where *name* is the name of the subroutine that is called.			

This instruction does not provide transfer of control to the subroutine. The programmer may do this by writing a jump instruction elsewhere in the program.

END

The END instruction is used to indicate the end of a program, and must be the last physical instruction of the program.

The general form of this instruction is:

GENERAL FORM		EXAMPLES	
Command	Address and Remarks	Command	Address and Remarks
END	*addr*	**END**	START
where *addr* is an address field element (except on Index Register Notation), which, if present*, represents the location to which control should be transferred after the program is loaded.		**END**	A·EXECUTE
		END	BEGIN + 1

*When compiling a subroutine for example, *addr* is omitted from the card. (See WRITING A SUBROUTINE, page 43.)

COMMON SYMBOLS

Chapter 3

Use of C Label and Name COMMON in Defining Common Symbols.

If a symbol that is used in different named sections of a program is to have the same definition in each section, this common definition may be specified by means of the C Label or by means of the name COMMON.

THE C LABEL

- A "C" in the label field of an instruction *other* than an ASGN, SAME, SYMBOUT or REFOUT instruction, causes the symbol specified in the *location field* of that instruction to be made *common*, and to have the same definition wherever it appears through the program, regardless of the NAME originally associated with it.

- A "C" in the label field of an ASGN or SAME instruction causes each symbol *symb* in the *address field* of that instruction (see page 19) to be defined as a *common symbol*, regardless of the NAME originally associated with it.

- A "C" in the label fields of SYMBOUT and REFOUT instructions causes the symbol(s) appearing in *both* of these instructions to be defined as *common* throughout the program, regardless of the NAME originally associated with it.

- A "C" in the label field of a NAME instruction causes each symbol *defined* in that named section to have the same definition wherever it appears *undefined* in the other named sections of the program. For example, according to the coding:

L	Location	Command	Address and Remarks
		.	
C		NAME	A
		.	
		TMD	ALPHA$
		.	
	BETA	D/7.5B3$	
		.	
		ASGN	DELTA, 100$
		.	
		.	
		.	
		NAME	B
		.	
		TMA	BETA$
		.	
		TMD	DELTA$
		.	
		.	

the symbols BETA and DELTA appearing in NAME sections A and B will be defined only once. The symbol ALPHA is *not* made common.

THE NAME COMMON

- The name COMMON given to a section of a program causes each symbol in that section that is not prefixed with a program name to be defined as a common symbol.
- The name COMMON prefixed to a symbol causes that symbol to be made common throughout the program.

Chapter 4

CONSTANTS

Pool and Non-Pool Constants. Full-Word Constants. Field Constants. Fixed-Point and Floating-Point Decimal Constants. Word and Location Constants. Alphanumeric, Octal, Hexadecimal, Numeric, Binary, Parameter, and Command Constants.

A constant is any full word of data, entered with the program, that does not vary from compilation to compilation. There are eleven different types of constants in TAC language. An alphabetic or numeric character written preceding the constant and separated from it by a slash, specifies the type of constant. During compilation, TAC converts each constant to its binary form and provides a storage location for it in memory.

With the exception of alphanumeric non-pool constants (see page 34), no more than one word of constants may be written per line of the coding form.

POOL AND NON-POOL CONSTANTS

Pool constants are constants that are placed by TAC in a separate section of the program called the constant pool; non-pool constants are constants that occupy the memory locations assigned to the positions where they appear in the program.

A constant is interpreted as a *pool constant* if it is written:

1. as the address of a mnemonic, or
2. beginning in the command field, with a P in the label field.

Non-pool constants are written starting in the command field, with *no* P in the label field.

Some pool constants are conservable (that is, if their *binary* configuration already exists in a word in the pool, it is not duplicated), some are not.

Constants written as the address of a mnemonic are conservable, except:

a. Location Constants
b. Field constants containing a Command Constant field or a Parameter Constant field.

A conservable pool constant occurring after another pool constant with similar binary configuration will be conserved on the basis of the former constant, provided the former constant is also of the conservable type, or is of the P-label type other than type (a) or (b), above.

In conserving pool constants, the TAC Assembler searches the constant pool to determine whether a binary configuration of the constant already exists in the pool. If one exists, its location is used and the constant is not repeated. If none exists, TAC inserts the constant in the next available word in the pool, and places the location of this word in the address field of the mnemonic.

If the constant is written as a P-label pool constant or as a non-pool constant, it may be assigned a symbolic location by the programmer. *Note, however, that because of the possibility of conservation occurring for a succeeding constant, a "P-label" constant with a symbolic location field should not be changed during program execution. P-label pool constants consecutively grouped will be stored in the pool exactly as grouped.*

FULL-WORD AND FIELD CONSTANTS

Some constants occupy a full word of memory, other constants may occupy fields or parts of a word. Constants that occupy a full word are called *full-word constants;* constants that occupy fields are called *field constants*. If combined field constants do not completely fill a word, the unused bit positions are filled with zeros or space symbols, depending upon the type of constant that is written.

FULL-WORD CONSTANTS

Full-word and field constants may be of the pool type or of the non-pool type.

There are four types of full-word constants in TAC language:

1. Fixed-point decimal constants
2. Floating-point decimal constants
3. Word Constants
4. Location Constants

Fixed-Point Constants (D/.....)

A fixed-point decimal constant is represented by the characters D/ followed by any combination of decimal digits, with or without a decimal point, not exceeding 140,737,488,355,327.

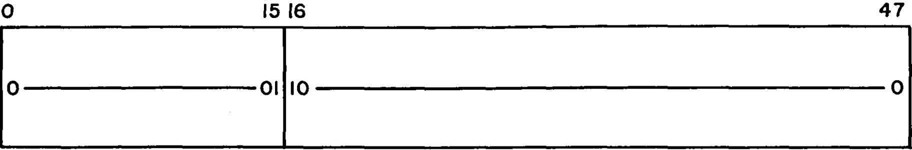
A fixed-point constant may be signed (+ or -) or unsigned. If unsigned, it is interpreted as positive. A negative fixed-point constant is stored in memory in two's complement form.

A binary position factor is used to indicate where the binary point should be placed in the computer word. The binary position factor is written immediately after the constant, and is designated by the letter B followed by a number (0-47) representing

the least significant bit position of the integral part of the decimal constant. If no binary position factor is written, the integral part of the constant is scaled B47 and the fractional part (if any) is lost.

Example

The constant D/1.5B15 would be stored in a word in memory as follows:



Floating-Point Decimal Constants

Floating-point decimal constants are represented by the characters F/ followed by any combination of decimal digits, with or without the decimal point. A *decimal scale factor,* En, which means "x10^n", may also be written following the constant. The character *n* may be any decimal integer exponent -600 to 600. A binary position factor is not used with a floating-point constant.

As is the case with fixed-point constants, a positive quantity is indicated by no sign or by a plus sign (+), and a negative quantity by a minus sign (-).

Examples

F/.000024 F/.240E3 F/240.E-4

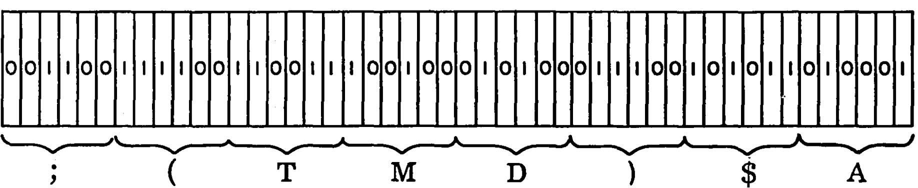
Word Constants (W/....)

Word constants are represented by the characters W/ followed by any eight Philco characters (see Appendix D), including the $ character. The constant is considered automatically terminated after the eighth character.

The eight characters in the word constant are stored in the order in which they are written. For example, the constant

W/;(TMD)$A

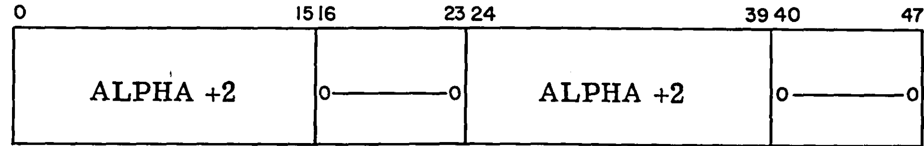
would be stored as follows:



Location Constants (L/.....)

A location constant is represented by the characters L/ followed by an address field element. The address field element is stored (in binary) in the address portion of *both* halves of the word containing the constant.

If the address field element ALPHA +2 represents a left address, the constant is stored as:



If the address field element represents a right address, an F bit of one will appear immediately to the right of the address in both halves of the word.

FIELD CONSTANTS

Field constants, as their name implies, occupy fields or parts of a computer word. A Position Factor, T*n*, written immediately following the constant, indicates where the constant (except *Alphanumeric* and *Command* constants, see pages 34 and 39) terminates in the word. The character *n* is a decimal integer (0-47) specifying the position of the least significant bit of the constant.

When several field constants are made to occupy a word, the individual constant designations are written separated by semicolons, and the individual position factors, if any, must be in ascending order. If a field reserved for a constant is such that the high order bits of the constant extends into a preceding constant, or beyond bit position zero, the high order bits are lost.

If position factors are not specified with the constants, the fields fill the word from left to right, in the order in which they are written.

There are seven types of field constants in TAC language:

- Alphanumeric Constants (*No* position factor is used.)
- Octal Constants (A position factor *may* be used.)
- Hexadecimal Constants (A position factor *may* be used.)
- Numeric Constants (A position factor *must* be used.)
- Binary Constants (A position factor *may* be used.)
- Parameter Constants (A position factor *must* be used.)
- Command Constants (*No* position factor is used.)

Alphanumeric Constants (A/.....)

Alphanumeric constants are represented by the characters A/ followed by any number of the sixty-four Philco characters listed in Appendix D, except the semicolon, dollar sign, and the right parenthesis. A semicolon, dollar sign, and in some instances a right parenthesis, automatically terminates the field. Each character is converted to a unique six-bit code; eight such characters occupy a full computer word.

Alphanumeric constants fill a word from left to right starting with the six high-order bits. A termination indicator *cannot* be used to position an alphanumeric field within a word; leading zeros may be used for this purpose, if necessary.

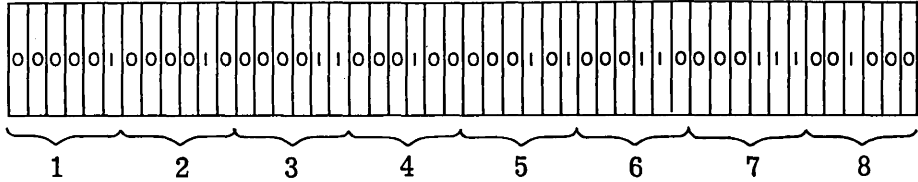
An alphanumeric constant that is written in the address and remarks field, may contain eight or less characters, but not more than eight. If the constant contains less than eight characters, and no other field constants are specified to fill the remainder of the word, a $ character must immediately follow the last character of the constant. The remainder of the word is filled with zeros.

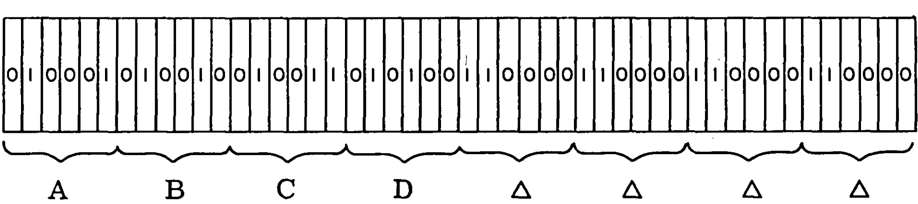
An alphanumeric constant that is written in the command field may contain any number of characters. These characters, if necessary, may be continued on succeeding lines of the coding form, beginning in the address and remarks field. Each eight-character group is placed in a consecutive memory location. If the number of characters written is not a multiple of eight, and no other field constants are specified to fill the remainder of the word, a $ character must immediately follow the last specified character. The remainder of the last word is filled with spaces.

Thus, the alphanumeric constant,

A/12345678ABCD$

written starting in the command field, would occupy two consecutive memory locations as follows:





If an AFEND instruction (see page 16) is used in the program, and the alphanumeric constant is *not* terminated by a $, the contents of all columns up to and including the column referred to by the AFEND are included in the constant. If the constant contains a non-space character in the column (n) referred to by the AFEND, the constant must be terminated by a $ character.

Octal Constants (O/.....)

Octal constants are represented by the characters 0/ followed by as many as sixteen octal digits (0-7). Each octal digit of the constant is converted to three binary bits. A position factor may be used to terminate the constant. When no position factor is used, the octal constant fills the constant word from left to right, starting at the 0*th* bit position. Unused bit positions are filled with zeros.

Example

The octal constant O/16T5, O7T4, O/1600000000000000, or O 16 would appear as follows:

```
0 1 2 3 4 5                                                   47
001110 ------------------------------------------------------- 0
```

Hexadecimal Constants (H/.....)

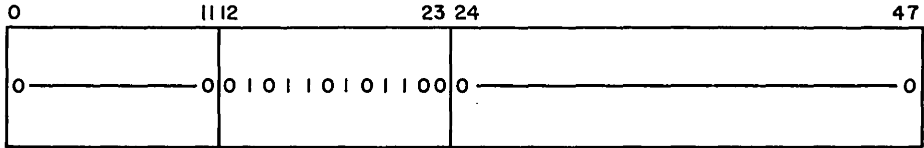
Hexadecimal constants are represented by the characters H/ followed by as many as 12 hexadecimal characters (0-9 and A-F). Each character of the constant is converted to four binary bits, as shown below:

HEXADECIMAL CHARACTER	BINARY EQUIVALENT
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

A position factor may be used with the constant. When no position factor is used, the hexadecimal constant fills the constant word from left to right, starting at the *0th* bit position. Unused bit positions are filled with zeros.

Example

The constant H/5AC T23 would be stored as follows:

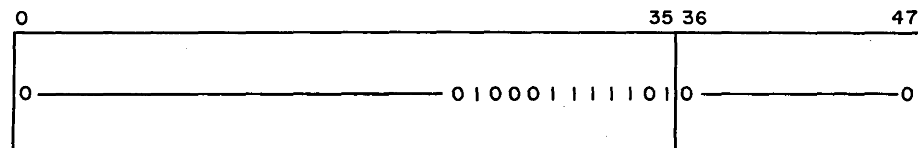


Numeric Constants (N/.....)

Numeric constants are represented by the characters N/ followed by an unsigned decimal integer. A position factor *must* be written following the decimal integer or an error will be indicated.

Example

The numeric constant N/1149T35 would be stored as follows:



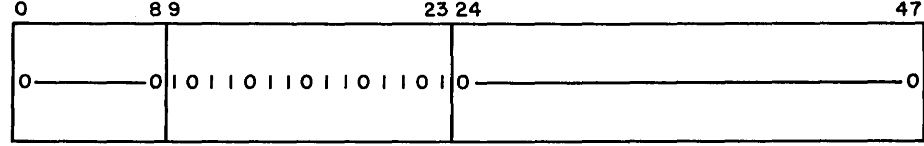
Binary Constants (n/.....)

Binary constants are represented by the characters *n*/ followed by some binary configuration. The character *n* is any decimal integer 1 to 48, indicating that the binary configuration specified in actually a pattern that is repeated *n* times in the constant.

A position factor may be used with the constant. When no position factor is used, the binary constant fills the constant word from left to right, starting at the *0th* bit position. Unused bit positions are filled with zeros.

Example

The binary constant 5/101T23 would be stored as follows:



Parameter Constants (P/.....)

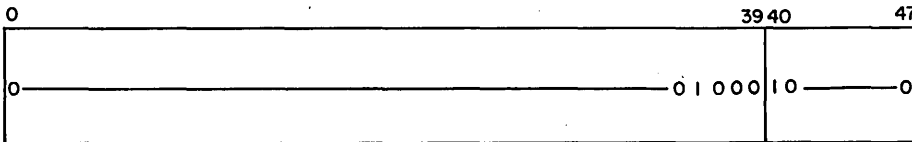
Parameter constants are represented by the characters P/ followed by an address field element. W position factor Tn or Fn, must be used with this constant, and is separated from the address field element specified by a comma.

When tn is used, the address element without its F-bit is stored in the constant word at the position specified by the termination indicator n; and unused bit positions are filled with zeros.

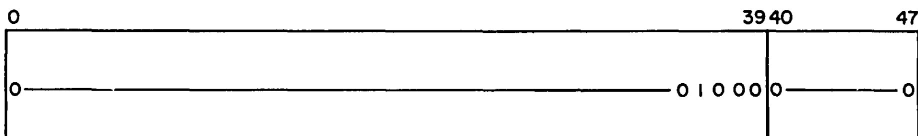
When F*n* is used, the address element is terminated at the (n-1)*th* bit, and the *nth* bit will be one or zero, depending on whether the address element represents a right or left address, respectively. (In relocatable programs, a relocatable address element in a Parameter constant *must* be scaled T15, F16, T39 or F40.)

Example

Assume that KAPPA represents a *right* address that has been assigned the value 8, then, the constant *P*/KAPPA, T39 would be stored as:



The same constant positioned F40 would be stored as follows:



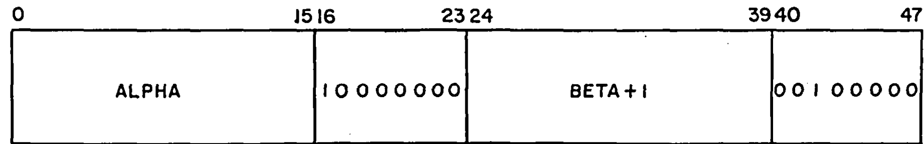
Command Constants (C/.....)

The command constant describes half-words of information in instruction format. Command constants are represented by the characters C/ followed by a Mnemonic and an address field element. The address field element is separated from the command by a comma. No position factor is used with the constant.

Two command constants, written separated by a semicolon, occupy an entire constant word. If only one command constant is written, the constant will occupy the left half of the word; the right half will be filled with zeros. If another type of field constant (positioned before T24) is written preceding the command constant, the command constant will occupy the right half of the constant word.

Examples

The constant C/HLTR,ALPHA; C/JMPL, BETA+1 would be stored as:



The constant 2/101T23; C/HLTL,KAPPA would be stored as:

0 — 23	24 — 39	40 — 47
0———————0101101	KAPPA	0———0

Groups of Field Constants

Some typical field constant groups could be:

N/5T15;C/HLT,ALPHA+5H$

1/1T0;O/7T23;A/ABCD$

P/ALPHA-BETA, T15; C/HLT,M/5000$

H/ABCD;8/1T23;O/7;N/1T47$

O/32;A/TYPEOUT$

48/1$

Chapter 5

LIBRARY ROUTINES

Subroutines. Generators. Macros. Calling and Writing Library Routines. Adding Library Routines to the Library Tape. TAC Generator Symbols. Skeleton Coding.

GENERAL DESCRIPTION

Library routines are Subroutines, Generators, and Macros on the TAC Library tape. Each routine is an individual program, or set of instructions, designed to perform a frequently required operation. If the operation to be performed at a point in the program can be accomplished by one of these routines, the programmer need only reference (call on) the respective routine at that point in his program, thus saving valuable coding time and effort by not having to code the routine himself.

SUBROUTINES

A subroutine is a program that operates under control of a calling program, and that is included only once in the calling program regardless of the number of times referenced.

Library subroutines are subroutines on the TAC library tape. These may be in TAC-language or in relocatable binary form.

Calling a Subroutine

A subroutine call is used to reference a subroutine. This call may be written in either of two forms:

GENERAL FORMS				EXAMPLES			
L	Location	Command	Address	L	Location	Command	Address
S		*entrance*	*param*	S		FLEJ	ADR1;ADR2;ADR3
		SUBR	*entrance*			SUBR	FLEJ

where *entrance* is an entrance to the subroutine being referenced, and *param* represents the parameters of the subroutine.

Parameters are separated by semicolons. If a parameter is represented by a group of field constants, the parameter must be enclosed in parentheses to prevent ambiguity in the meaning of the semicolons.

During compilation, TAC replaces the subroutine call with:

1. A TMA instruction, to place the first parameter in the Call in the A Register.
2. A TMQ instruction, to place the second parameter in the Call in the Q Register.
3. TMD and TDM instructions, to place the third and succeeding parameters in consecutive memory locations immediately following the entrance word. (See *Writing A Subroutine*, page .)
4. A JMP instruction, to transfer control to the subroutine. The SUBR call does not provide for parameter setup as in (1), (2) or (3) above, nor for transfer of control to the subroutine. The programmer may transfer control to the subroutine elsewhere in his program.

During its execution, the subroutine obtains the first parameter in the call from the A Register, the second parameter from the Q Register, and the third and succeeding parameters from consecutive memory locations immediately following the entrance word.

If a parameter will already be in its proper register or memory location when the subroutine obtains control, this parameter may be omitted from the subroutine call. For example, the following calls:

L	Location	Command	Address and Remarks
S		ENTRAN1	;ADR2 $
S	DELTA	ENTRAN2	ADR1;;ADR3;;ADR5 $
S		ENTRAN3	$

indicate that the first parameter of subroutine ENTRAN1 is already in the A Register; the second and fourth parameters of subroutine ENTRAN2 are in the Q Register and in memory location ENTRAN2. ENTRAN2+2, respectively; and, all parameters of subroutine ENTRAN3 are in their proper locations.

When a parameter is thus omitted from a call, the corresponding TMA, TMQ, or TMD and TDM instructions are not generated.

Subroutines generally store their output values in the same order and locations as their parameters (i.e., the first output value in the A Register, the second in Q, etc.). Individual output procedures are indicated in the respective subroutine descriptions.

Subroutines may call on generators, macros, or other subroutines, which in turn may call on still other subroutines. There is no repetition of subroutines in a program; TAC always checks to see if a subroutine was previously requested, in which case it would already be scheduled for incorporation into the program.

If a section of the calling program has the same name as a subroutine being called, the call is assumed satisfied, and the library tape is not searched for that subroutine.

If a called subroutine is not included in the compilation, the notice "THE FOLLOWING SUBROUTINES NOT INCLUDED" is printed on the Code-Edit. In an RPL or ABS compilation, this is a *serious error* condition.

If subroutines are *not* to be compiled in the program, but are to be brought in at load time (in which case both the subroutine and the calling program must be in relocatable binary form), references to the entrances of the subroutines are automatically compiled as REFOUT symbols.

Most subroutines use Index Registers 1 and 2. The previous contents of these registers are not restored by the subroutines; the contents of all other Index Registers are saved.

Writing a Subroutine

A subroutine may have several entrances. Each entrance permits access to a specific group of instructions that perform a particular operation.

The first word of each group of instructions is the entrance word. The left instruction in this word establishes the address for returning control to the calling program. The right instruction in this word provides a transfer of control to the instruction immediately following the last of the words reserved for storing the third and succeeding parameters (if any) of the call. The last instruction of the group provides an exit from the subroutine by transferring control to the half-word immediately following the subroutine call.

An entrance may be left or right; if it involves more than two parameters, it must be left.

---

*For calls on FORTRAN type subroutines, refer to Appendix C.

The following example illustrates the format of a standard TAC subroutine:

L	Location	Command	Address and Remarks	
		NAME	ENTRAN1$	
L	ENTRAN1	TJM	EXIT$	First entrance
		JMP	ALPHA $	
		. . .		Locations reserved for parameters
	ALPHA	. . .		Beginning instruction of group
		JMP	EXIT$	Last instruction of group
L	ENTRAN2	TJM	EXIT$	Second entrance
		JMP	BETA $	
		. . .		Locations reserved for parameters
	BETA	. . .		Beginning instruction of group
		JMP	EXIT $	Last instruction of group
		. . .		Other entrances and groups of instructions
	EXIT	JMP	1SUBERR$	Exit from subroutine
		ENDSUB	$	

If the subroutine is to be a TAC-language library subroutine, it is added to the library tape as it appears above. If it is to be added to the library as a relocatable binary subroutine:

- the ENDSUB $ instruction is replaced an END $ instruction,
- the entrances and other symbols of the subroutine that may be referred to from outside the subroutine must be defined as SYMBOUT symbols and appear in the address and remarks field of SYMBOUT cards in the subroutine. Generally, a SYMBOUT symbol represents a *left* address, as this prevents the instruction in the calling program that refers to this address from being altered at load time. (Refer to *F-Bit Modifications at Load Time,* page 88.)
- the subroutine is compiled in REL format, and the resultant relocatable program deck is added to the library.

Adding a Subroutine to the Library Tape

For a description of how to add subroutine to the TAC library, see Program Report 13, PLUM (*P*rogram for *L*ibrary *U*pdate and *M*aintenance).

GENERATORS

A Generator is an RPL program on the TAC library tape which, when called upon, generates coding to perform a specific operation. The coding generated depends on the type of generator call, and the parameters contained in the call.

Calling on a Generator

A generator call is used to call on a Generator. The general form of this call is:

GENERAL FORM		EXAMPLES	
Command	**Address and Remarks**	**Command**	**Address and Remarks**
Cmnd	$p_1, \ldots, p_n$	PRT	15;A;B;C$
where *Cmnd* is a generator command (1 to 8 alphanumeric characters long) representing an operation to be performed, and each *p* is a symbol representing a parameter of the generator.		FORMAT	(5E8.2,2F6.3)$
The number, order, and format of the parameters are established by the generator.		RIT	10,F200,A(10)$

With some generator commands, such as in the second example, the complete set of parameters must be enclosed in parentheses. Parameters too many to fit on *one* card may be continued in the address and remarks field of succeeding (continuation) cards.

TAC reserves processing of generator calls until all regular TAC instructions (control instructions, mnemonics, etc.) in the program are processed. TAC then determines the specific generator that is associated with each call, supplies the generator with information about the call, and transfers control to the generator. The generator processes the call, generates TAC-language coding compatible with the type of call and the parameters specified, and returns control to TAC. TAC then compiles the generated coding into the calling program.

TAC reserves a *unique* 8-character *symbol* (in location EG2.1GNEW, see page 49) as the address to be assigned to the coding generated. If the generator call requires a transfer of control to the generated coding, TAC replaces the call with a

JMP *Unique Symbol*

instruction. Whatever was specified in the label and location fields of the call would be indicated in similar fields of the JMP instruction substituted. The generator provides the appropriate linkage between the JMP and the generated coding by generating a definition for the *unique symbol*, and by generating an appropriate NAME instruction. (The program name that is to appear in the generated NAME instruction is supplied to the generator by TAC, and is the name of the program section containing the call.)

If the call does *not* require transfer of control to the generated coding, it is deleted from the program after being processed; no JMP instruction is generated.

The calls are processed in the order indicated by sort numbers assigned to the generator commands *(Cmnd)* when the generator is added to the library tape. This permits all the calls of one type to be processed before calls of another type.

After all generators calls in the program have been processed by their respective generators, TAC inserts an E AFEND and an ENDGEN instruction immediately after the last instruction of the generated coding.

The generated coding appears after the calling program and before the area reserved for pool constants on the Code-Edit (see page 75).

Writing a Generator

The following coding illustrates the general form of a Generator:

Location	Command	Address and Remarks	
	. . . . .		ASGN or SAME instructions permitting communication between TAC and the Generator. (See page 49.)
	SET	1GENADD$	Establish entry point*
	TJM	EXIT$	Establish exit
	. .		
	JMP	EG2.1GNC$	Obtain parameters
	. . .		Generate coding
	JMP	EG2.1DUMP$	Write out generated coding
	. . .		
	ASGN	EXIT,(P)$	
	JMP	(P)$	Return control to TAC
	END	EG2.ENDCARD$	

After the generator is selected from tape, it is loaded into memory, initialized, then executed. Initialization is performed upon a transfer of control to location EG2.ENDCARD (specified in the END card), the starting address of a generator-initializing routine in TAC. After initialization, TAC supplies the generator with the following information about the call, and transfers control to location 1GENADD of the generator:

- The generator command *(Cmnd)* left justified and with trailing spaces, in the A Register.
- The sort number of the generator command scaled T15 in the Q Register. (See *Adding A Generator To The Library,* page 51.)

---

*Address fields of subsequent SET instructions, if any, must always contain an address greater than 1GENADD.

- A unique 8-character symbol in location EG2.1GNEW, to be used as the entrance location of the generator coding if, during program execution, control is to be transferred to the generated coding.
- The label field of the generator call scaled T5 in location EG2.1GLABEL.
- The location field of the generator call, in location EG2.1GFLAD.
- The program name in effect at the time the call was encountered, in location EG2.1GNAME.
- The P-count in binary, scaled T40 in location EG2.1GPCTR, corresponding to the point in the program where the call occurred.

The generator processes the above information and:

- Obtains the contents of the address and remarks field of the call from TAC one character* at a time, by transferring control each time to location EG2.1GNC. (The character obtained appears scaled T5 in both the A and Q Registers, after which, control is returned to the generator at the instruction following the last executed JMP EG2.1GNC. The generator should test for a $ character to determine when no more characters in the call remain to be processed.)
- Generates coding compatible with the type of call and the parameters specified.
- Writes-out the generated coding on tape by transferring control to location EG2.1DUMP, with the number of cards to be output scaled T15 in the A Register and the starting location of the generated coding scaled T29 in the Q Register. (After the output cards are written, control is returned to the generator at the instruction following the last executed JMP EG2.1DUMP.)
- Returns control to TAC at the instruction immediately following the instruction that transferred control to location 1GENADD of the generator.
- Either obtains and processes (in the manner indicated above) the next call on the generator when control is returned to location 1GENADD, or generates and writes close-out coding if no other call remains to be processed by the generator. (The condition of no calls remaining to be processed is indicated by all ones in the A Register at the time TAC returns control to location 1GENADD.) After producing close-out coding, if desired, the generator returns control to TAC at the instruction following the last executed JMP 1GENADD.

*Space characters included. If control is transferred to location EG2.2GNC instead of EG2.1GNC, space characters are ignored and not transferred to the generator.

Symbols Permitting Communication Between TAC and Generators

To permit communication between TAC and the generator, ASGN or SAME cards defining the following symbols must be included in the generator. In the event of modification of TAC, the symbol TACBASE must be redefined.

SYMBOL	DEFINITION	EXPLANATION
TACBASE	NTLOAD1.TACBASE	The lowest memory location used by TAC itself (i.e., the origin of TAC in memory).
1GENADD	TACBASE+M/2000	Location of first executable instruction of generator.
ENDCARD	TACBASE+M/1016	Generator end card address.
1GNAME	TACBASE+M/1003	Location containing alphanumeric program name in effect when call occurred.
1GNEW	TACBASE+M/1004	Location containing TAC-created, alphanumeric symbol for generated coding.
1GPCTR	TACBASE+M/1006	Location containing, in binary form at T40, the TAC *program* (P) *count* corresponding to the point where the call occurred.
1GFLAD	TACBASE+M/1007	Location containing, in alphanumeric form, the location field of the call.
1GLABEL	TACBASE+M/1010	Location containing the label field of call, scaled T5.
1GNC	TACBASE+M/1011	Location of a *parameter-input* subrouting in TAC that is available to generators.
2GNC	TACBASE+M/1012	Location of a second *parameter-input* subroutine in TAC that is available to generators. Unlike the preceding subroutine, this subroutine ignores space (blank) characters.

SYMBOL	DEFINITION	EXPLANATION
1DUMP	TACBASE+M/1013	Location of an *output* subroutine in TAC that is available to generators. When this subroutine is entered, the A Register must contain the number of cards to be output, scaled T15, and the Q Register, the starting location of the generated coding to be output, scaled T39.
OVERLAY	TACBASE+M/1014	Location of a *get-next-block-on-library* subroutine in TAC that is available to generators. This subroutine reads the next block from the library tape containing the generator. The starting address of the 128-word area into which the block is to be read must be scaled T15 in the A Register at the time this subroutine is entered.
RPLOVER	TACBASE+M/1015	Location of a *get-next-RPL-program-on-library* subroutine in TAC that is available to generators. This subroutine reads the next RPL program on the library tape containing the generator, and transfers control to the instruction immediately following the JMP EG2.RPLOVER Also, the end card of this "next RPL" must be END EG2.ENDCARD to permit TAC to return control to the generator properly.

Adding a Generator to the Library Tape

After the generator program is written, it is compiled in RPL format, then added to the TAC library tape. At the time of its addition to the library, information about the location and size of the generator on tape, the calls it will accept, the order in which the calls are to be processed, and which calls require transfer of control to the generated coding, must be supplied.

Procedures for supplying this information when adding a generator to the library tape are discussed in detail in Program Report 13, PLUM (*P*rogram for *L*ibrary *U*pdate and *M*aintenance).

MACROS

A Macro-generator is a program, supplied with the TAC library tape, which when called upon by means of a macro-call (macro-instruction), inserts coding into the calling program.

The coding inserted consists of a *fixed* number of instructions, and replaces the macro-call in the program.

Calling a Macro

A macro-call (macro-instruction) is used to call on the Macro-generator. The general form of this call is:

GENERAL FORM		EXAMPLES	
Command	**Address and Remarks**	**Command**	**Address and Remarks**
Cmnd	$p_1;\ldots;p_n$	**RDMTF** **CHKMT**	UNIT;CSA;NBP;NBS$ ORDER;INCOMPLETE; ERROR;B$
where *Cmnd* is a symbol (1 to 8 alphanumeric characters long) specifying an operation to be performed for which coding must be inserted into the program, and each *p* is a numeric or alphanumeric symbol representing a parameter to be used in the inserted coding. Parameters are separated by semicolons.			

When TAC encounters a macro-call in a program during compilation, it determines from the Table of Contents on the library (see PLUM, Program Report 13) the amount of coding that will be inserted for the call, and reserves the appropriate space in the program for the coding.

The coding to be inserted exists in skeleton form (see below) following the Macro-generator on tape. TAC transfers control to the Macro-generator, which expands the skeleton coding by placing the parameters of the macro-call in their proper places in the skeleton coding, and inserts the expanded coding into the calling program.

As many parameters may be specified in a macro-call as are provided for in the corresponding skeleton coding. Parameters too many to fit on one card may be continued in the address and remarks field of succeeding continuation cards.

Skeleton Coding

Numbers and letters assigned to the parameters in the skeleton coding indicate where each parameter is to be inserted into the coding. The parameters are labeled 1 through 9 and A through Z, with each number or letter preceded by an equal (=) sign. The first parameter is inserted at the point where the =1 characters appear in the skeleton coding, the second parameter is inserted where the =2 characters appear in the skeleton coding, the tenth parameter where the =A characters appear, and so on, until the skeleton coding is completely expanded and all parameters are included.

The following example is an illustration of a macro-expansion. Consider the Macro TLUEQ (Table Look-up for Equality), the skeleton coding of this Macro is:

L	Location	Command	Address and Remarks
		TMD	L/=3$
		TDX	,1X$
		TMQ	O/=2$
		ETA	=1$
R		RPTAN	=4$
		ETD	1,1X$
		JAED	(P)+2H$
		JMP	=5$
		SIXO	1,1X$

For a macro-call of the form:

L	Location	Command	Address and Remarks
		TLUEQ	KEY;7777;TABLE;100;ALPHA $

the expanded coding will be:

L	Location	Command	Address and Remarks
		TMD	L/TABLE$
		TDX	,1X$
		TMQ	0/7777$
		ETA	KEY$
R		RPTAN	100$
		ETD	1,1X$
		JAED	(P)+2H$
		JMP	ALPHA$
		SIXO	1,1X$

This expanded coding is inserted in-line into the calling program, replacing the macro-call. On the Code-Edit, the expanded coding is located after the calling program.

Adding Skeleton Coding to the Library Tape

At the time the TAC-language skeleton coding is added to the library tape, information about its Marco-generator, the number of locations it will occupy in the program after compilation, and whether its first instruction is a left or right instruction, must be supplied.

Procedures for supplying this information when adding skeleton coding to the library are presented in Program Report 13, PLUM (*P*rogram for *L*ibrary *U*pdate and *M*aintenance).

Chapter 6

OBJECT PROGRAM FORMATS

Binary Object Program Cards and Tape. A Relocatable Binary Deck. An Absolute Binary Deck. An RPL Object Program Tape.

As indicated earlier in the Introduction, TAC object programs may be compiled in any of three formats: REL, ABS, or RPL. Consistent with the format specified, a RELocatable Binary Card Deck, an ABSolute Binary Card Deck, or an RPL Absolute Binary Program on tape, is produced.

BINARY OBJECT PROGRAM CARDS

The format and contents of the object program cards and the RPL object program tape are discussed in this chapter. The manner in which a loader processes these cards is discussed in Appendix B.

A RELOCATABLE BINARY DECK

The binary deck produced on a REL compilation may be depicted as follows:

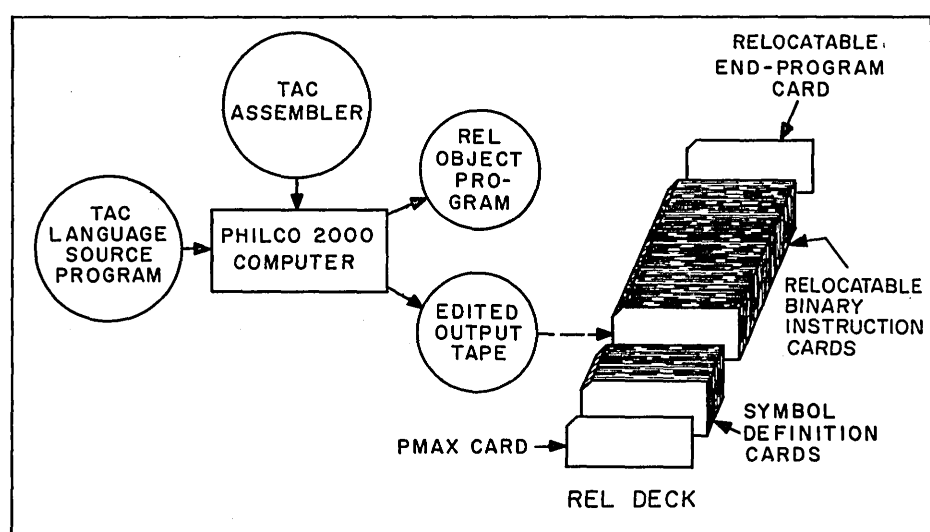


Figure 3 – A Relocatable Binary Deck

As shown in the preceding figure, each relocatable object deck contains the following cards:

1. PMAX CARD
2. SYMBOL DEFINITION CARDS
3. RELOCATABLE BINARY INSTRUCTION CARDS
4. RELOCATABLE END-PROGRAM CARD

The formats and functions of these cards are discussed below. A 1 indicates a punch, a 0 indicates a blank or non-punch.

Except for identify and sequence information in columns 1-8, which is in Hollerith code, all other information on the cards is in binary.

The PMAX Card

The PMAX card is the first card of the object program. It contains the program identifier, information as to the length (size) of the program, and the amount of common storage required by the program. (Program length is the address of the first available memory location relative to program origin. It includes the amount of storage required by the program, temporaries, ASTOR areas, and areas reserved for pool constants.)

The format of the PMAX card is:

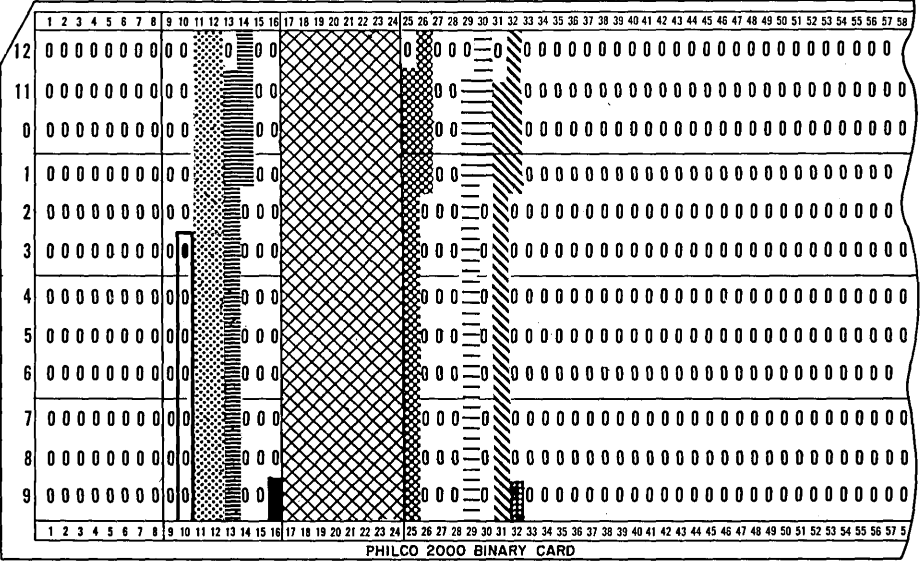


Figure 4 – Format of the PMAX Card

EXPLANATION OF PMAX CARD FORMAT			
LEGEND	COLUMNS	ROWS	CONTENTS
	1-8	12-9	Identity and Sequence information, in Hollerith.
	9-12	12-9	Punches, or Zero. If zero, card is ignored. If one or more of these columns contain punches, card is processed.
	10	3-9	1000000 (PMAX Card identification)
	11-12	12-9	Checksum* or Zero
	13 14	11-9 12-1	A binary value. This value is interpreted according to the contents of row 9 of column 16.
	16	9	An indicator bit: 0 indicates that columns 13-14 specify the *length* of the object program. 1 indicates that columns 13-14 specify the *loading origin* of the object program.
	17-24	12-9	Program Identification (96 bits). These bits affect checksum only.
	25 26	11-9 12-1	Number of words of COMMON storage required by the program about to be loaded.
	29 30	11-2 12-1	Number of TEMPORARY and ASTOR locations used by the program about to be loaded.
	31 32	11-9 12-1	Starting address, relative to program origin, of TEMPORARY/ASTOR area.
	32	9	1 or 0. If 1, record is not replaced by record of subsequent TEMPORARY/ASTOR area. Next card is processed.

Columns not mentioned above have no effect on loader processing other than their effect on the checksum.

*A *sum* of all relevant bits on the card, used to *check* accurate transfer of the information on the card. The actual checksum technique is discussed on page 88.

Symbol Definition Cards

Symbol Definition Cards provide a loader with the definitions of certain symbols used in the object program. These cards are produced during compilation from SYMBOUT source cards, and are of the following format:

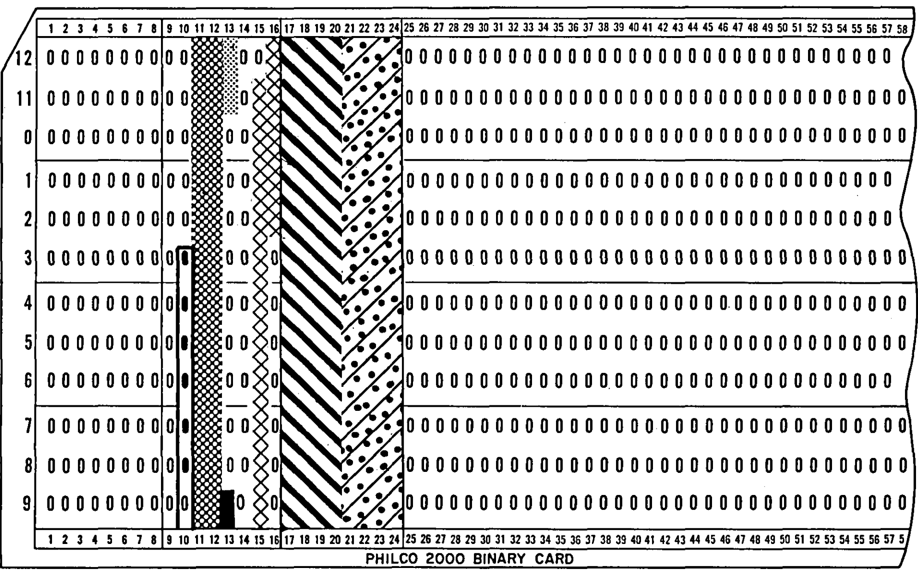


Figure 5 – Format of a Symbol Definition Card

	EXPLANATION OF SYMBOL DEFINITION CARDS FORMAT			
	LEGEND	COLUMNS	ROWS	CONTENTS
		10	3-9	1111110 (Symbol Definition Card identification)
		1-8	12-9	Identity and Sequence information, in Hollerith
		9-12	12-9	Punches, or zero. If zero, card is ignored. If one or more of these columns contain punches, the card is processed.
		11-12	12-9	Checksum or zero
First Symbol-Definition Field (Columns 13-24)		13	12-11	*Type* of address defining the NAME.SYMBOL which appears in column 15: 00 indicates the address is relative to program origin 10 indicates the address is absolute 11 indicates the address is relative to common origin
		13	9	An indicator bit: 0 indicates that the information contained in this symbol-definition field is to be processed 1 indicates that the information contained in this symbol-definition field is to be ignored*
		15 16	11-9 12-2	Address definition of NAME.SYMBOL. (Row 2 of column 16 contains the F-bit.)
		17-20	12-9	Subprogram NAME associated with SYMBOL. (If these columns are blank, this symbol-definition field is ignored, and the next symbol-definition field is processed.)
		21-24	12-9	SYMBOL
Additional Symbol-Definition Fields		25-36		Second symbol-definition field.**
		37-48		Third symbol-definition field.**
		49-60		Fourth symbol-definition field.**
		61-72		Fifth symbol-definition field.**

Columns not mentioned above have no effect on loader processing, other than their effect on the checksum.

* Note the possible use of this feature to have a double definition of a symbol ignored.

**For second, third, fourth, and fifth symbol-definition fields, the format is the same as for the first symbol-definition field shown above.

Relocatable Binary Instruction Cards

Relocatable Binary Instruction Cards contain the symbols whose definitions are to be found on Symbol Definition Cards, source program instructions in relocatable binary form, and certain control information for a loader.

The format of these cards is:

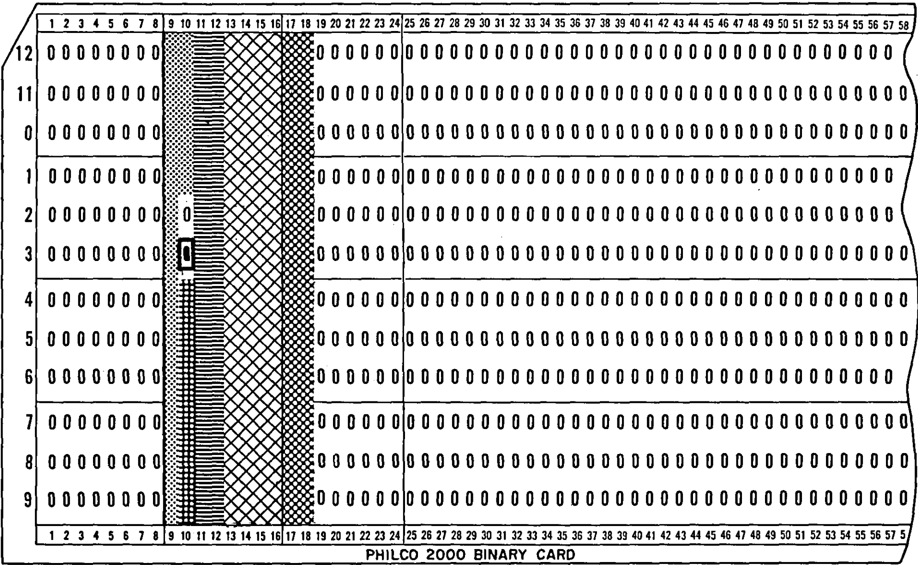


Figure 6 – Format of a Relocatable Binary Instructions Card

EXPLANATION OF RELOCATABLE BINARY INSTRUCTIONS CARD FORMAT			
LEGEND	COLUMNS	ROWS	CONTENTS
	1-8	12-9	Identity and sequence information, in Hollerith.
	9-12	12-9	Punches, or zero. If zero, card is ignored. If one or more of these columns contain punches, the card is processed.
	9	12-9	Core Starting Address of first instruction on card.
	10	12-1	Row 1 of column 10 is interpreted as the F-bit: 0 indicates that the first instruction starts in column 17. 1 indicates that the first instruction starts in column 19.
	10	3	1 (Relocatable Binary Instructions Card Identification)
	10	4-9	Number (1-32) of instructions on card.
	11-12	12-9	Checksum or zero
	13-16	12-9	A series of variable length (1-4 bits) indicators, which specify the type of address in each instruction: 0 indicates the address is relative to program origin 10 indicates the address is absolute 110 indicates the address is relative to common origin 1110 indicates the address references a REFOUT symbol
	17-18 or 17-26	12-9	First instruction on card. An instruction usually occupies two columns on the card; however, if the address specified in the instruction references a REFOUT symbol this instruction will occupy ten columns on the card. (The last eight columns will contain the REFOUT symbol.)
	19-80 or 27-80	12-9	Additional instructions on card.

The Relocatable End-Program Card

The Relocatable End-Program Card is produced during compilation from the END card, and is the last card of the relocatable binary program deck. This card causes a loader to:

1. transfer control to the address specified in the END card in order to start execution of program, or

2. commence loading the next relocatable object program.

The format of this card is:

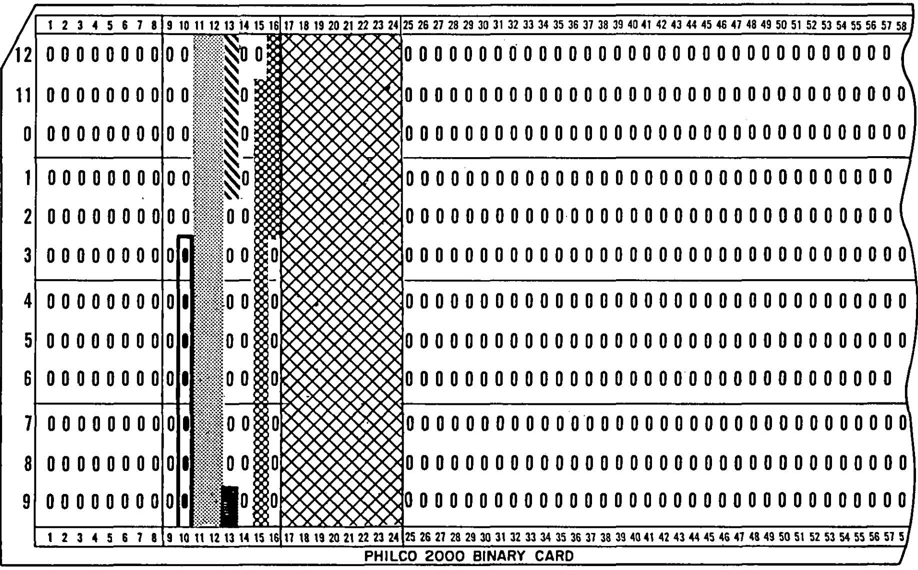


Figure 7 – Format of the Relocatable End-Program Card

EXPLANATION OF RELOCATABLE END-PROGRAM CARD FORMAT			
LEGEND	COLUMNS	ROWS	CONTENTS
	1-8	12-9	Identity and sequence information, in Hollerith.
	9-12	12-9	Punches, or zero. If zero, card is ignored. If one or more of these columns contain punches, the card is processed.
	10	3-9	1111111 (Relocatable End-Program Card identification)
	11-12	12-9	Checksum or zero
	13	12-1	Indicator bits, specifying the type of address in columns 15 and 16: 0000 indicates that the address is relative to program origin 1000 indicates that the address is absolute 1100 indicates that the address is relative to common origin 1110 indicates that the address references a REFOUT "NAME.SYMBOL"
	13	9	An indicator bit, which is interpreted follows: 0 indicates that next program is to be loaded 1 indicates that after this program is loaded, control is to be transferred to the jump address specified in columns 15 and 16
	15 16	11-9 12-2	An address in the program to which the loader transfers control after loading the program. (Row 9 of column 13 must have been 1.)
	17-24	12-9	A NAME.SYMBOL. (Rows 12-1 of column 13 must have been 1110)

Columns not mentioned above have no effect on loader processing other than their effect on the checksum.

AN ABSOLUTE BINARY DECK

The binary deck produced on an ABS compilation may be depicted as follows:

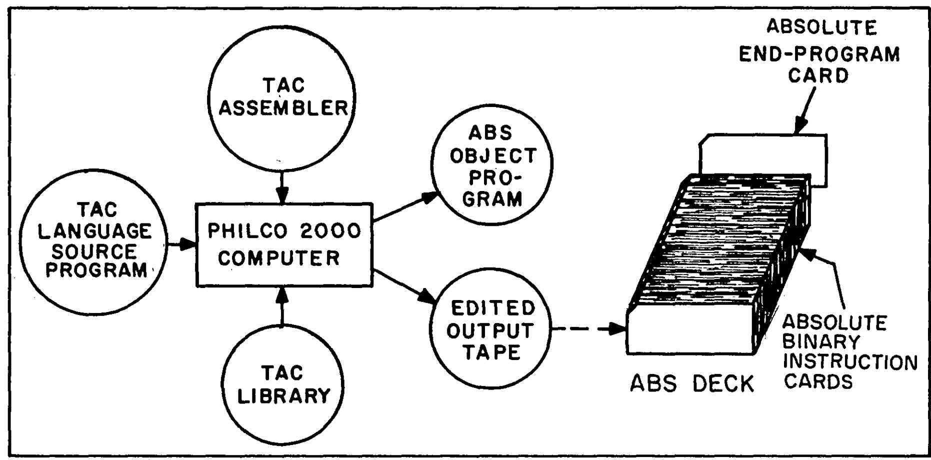


Figure 8 – An Absolute Binary Deck

As shown in the above figure, each absolute object deck contains the following cards:

1. ABSOLUTE BINARY INSTRUCTION CARDS

2. ABSOLUTE END-PROGRAM CARD*

Except for identity and sequence information in Hollerith in columns 1-8, all other information on the cards is in binary.

*Also referred to as BINARY JUMP CARD

Absolute Binary Instruction Cards

Absolute Binary Instruction Cards contain the source program instructions in absolute binary form, and certain control information for a loader. The format of this card is:

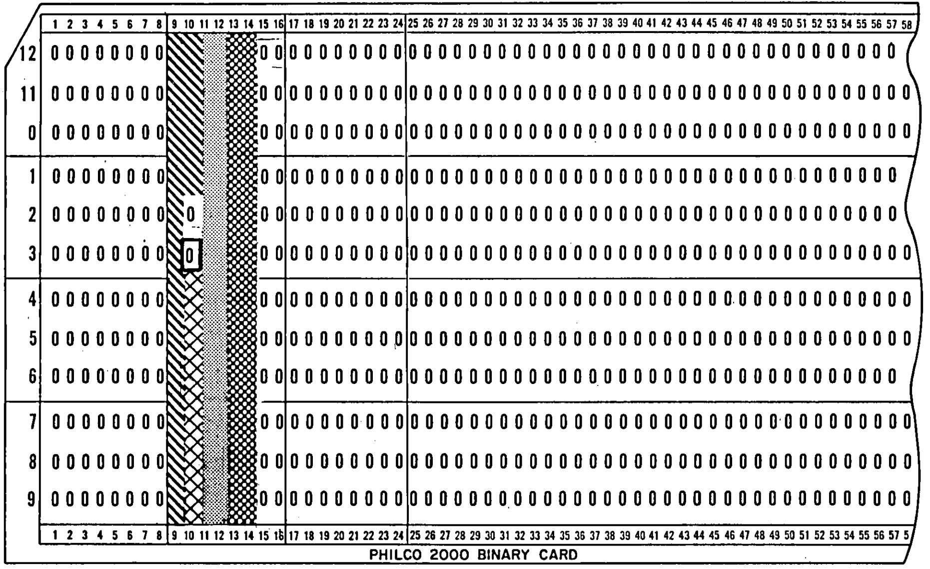


Figure 9 – Format of an Absolute Binary Instructions Card

EXPLANATION OF ABSOLUTE BINARY INSTRUCTIONS CARD FORMAT			
LEGEND	COLUMNS	ROWS	CONTENTS
	1-8	12-9	Identity and sequence information, in Hollerith
	9-12	12-9	Punches, or zero. If zero, card is ignored. If one or more of these columns contain punches, the card is processed.
	9 10	12-9 12-1	Core Starting Address of first instruction on card. Row 1 of column 10 is interpreted as the F bit: 0 indicates that the first instruction is in columns 13 and 14 1 indicates that the first instruction is in columns 15 and 16
	10	3	0 (Absolute Binary Instructions Card identification)
	10	4-9	Number (1-34) of instructions on card
	11-12	12-9	Checksum or zero
	13-14 or 15-16	12-9	First instruction on card
	15-80 or 17-80	12-9	Additional instructions on card. (Two columns per instruction.)

The Absolute End-Program Card

The Absolute End-Program Card is produced during compilation from the END card, and is the last card of an absolute binary program deck. This card causes a loader to transfer control to an address in the program, prior to the execution of the program.

The format of the Absolute End-Program Card is:

Figure 10 – Format of the Absolute End-Program Card

EXPLANATION OF ABSOLUTE END-PROGRAM CARD FORMAT			
LEGEND	COLUMNS	ROWS	CONTENTS
	1-8	12-9	Identity and sequence information, in Hollerith
	9-12	12-9	Punches, or zero. If zero, card is ignored. If one or more of these columns contain punches, the card is processed.
	9 10	12-9 12-1	The address of the first executable instruction in the program to which a loader transfers control after loading the program. Row 1 of column 10 is interpreted as the F-bit.
	10	3-9	0111111 (Absolute End-Program Card identification)
	11-12	12-9	Checksum or zero

Columns not mentioned above have no effect on loader processing other than their effect on the checksum.

BINARY OBJECT PROGRAM TAPE

All object programs (REL, ABS, and RPL) are recorded on the Object Program Tape. In the case of REL and ABS programs, object decks are subsequently produced; in the case of RPL programs, no object decks are produced. Except for this latter fact, and the fact that no checksum is calculated for RPL programs, the RPL program is the same as the ABS program.

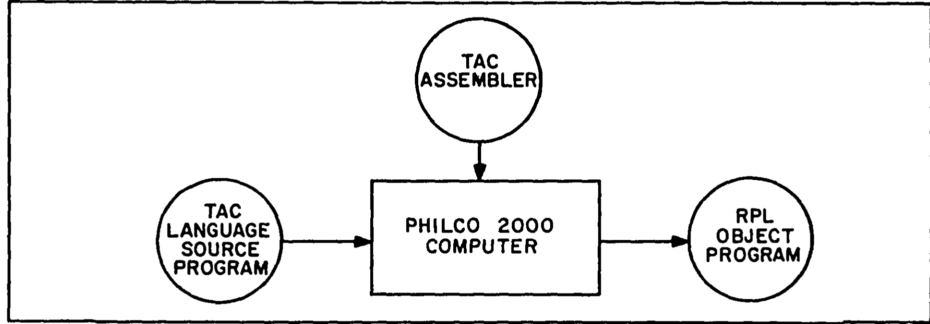


Figure 11 – RPL Compilation Output

Evidence of the similarity between ABS and RPL programs may be found, for example, in the *number* (34) *and arrangement* of the instructions on the cards and on tape. Also, the RPL instructions on tape occur in groups, with as many as 34 instructions per group; and, preceding each group is appropriate load information (see Figure 12, below).

RPL OBJECT PROGRAMS

The RPL object program on tape includes three types of control words produced during compilation:

1. A PROGRAM IDENTITY control word
2. A LOAD control word
3. A TRANSFER control word

These control words permit a loader to *locate* the object program on the Object Program Tape, to *load* this object program, and to *start execution* of the program by transferring control to an instruction in the program.

The following figure shows how an RPL program is arranged on tape.

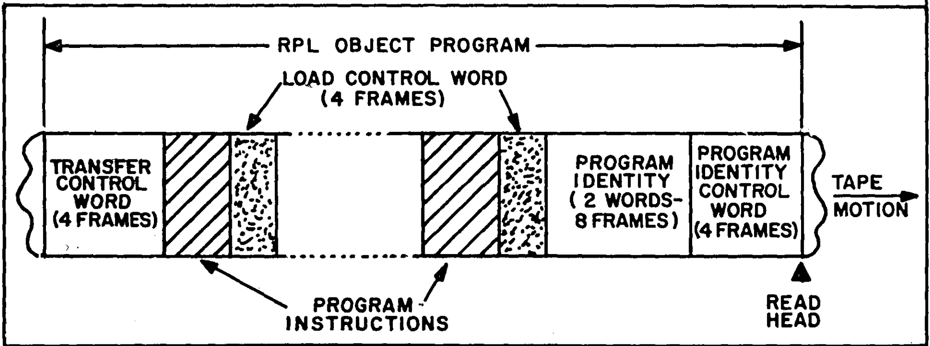
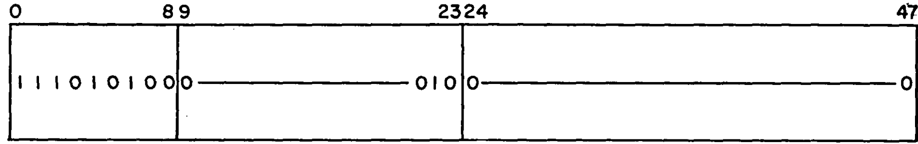


Figure 12 – An RPL Object Program

The PROGRAM IDENTITY Control Word

The RPL object program starts at the beginning of a block. The first word of the first block is the Program Identity Control Word. This control word indicates the number of words following it that contain the program's identity.



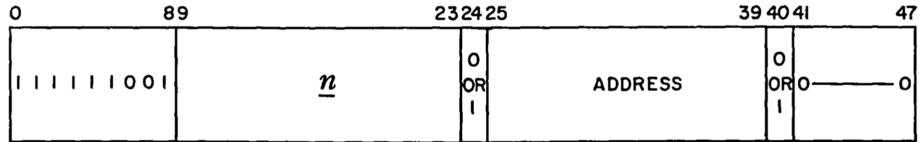
Bits 0-8 identify this word as the Program Identity Control Word.

Bits 9-23 specify the number (2) of words, following the Program Identity Control Word, that contain the RPL program's identity.

Bits 24-47 are ignored.

The LOAD Control Word

A LOAD Control Word precedes each group of instructions of the program. This control word indicates the number of full words of instructions following it that are to be loaded into memory. The format of this word is:



Bits 0-8 identify this word as the Load Control Word.

Bits 9-24 specify the number (*n*) of full words of instructions, following this load control word, that are to be loaded into memory. (*n* represents any number 1-34.)

Bit 24 specifies whether the number of instructions to be loaded is even or odd:

- 0 indicates an even number of instructions to be loaded
- 1 indicates an odd number of instructions to be loaded

Bits 25-39 indicate the starting address where the instructions in the group following are to be loaded sequentially into memory.

Bit 40 specifies which half-word is to contain the first instruction of the group:

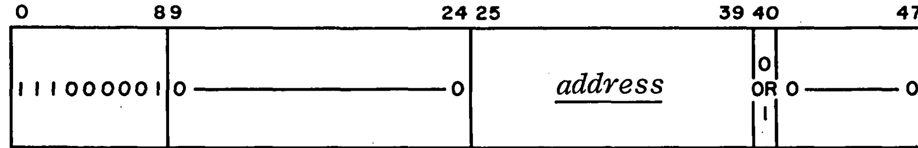
- 0 indicates that the first instruction of the group is to be loaded into the *left-half* of the memory location specified in bits 25-39.
- 1 indicates that the first instruction of the group is to be loaded into the *right-half* of the memory location specified in bits 25-39.

Bits 41-47 are ignored.

The TRANSFER Control Word

The last word of an RPL object program is the Transfer Control Word. This word causes a loader to transfer control to an instruction in the program after loading.

The format of the Transfer Control Word is:



Bits 0-8 identify this word as the Transfer Control Word.

Bits 9-24 are ignored.

Bits 25-39 indicate the *address* of the instruction to which control is to be transferred after the loading function is completed.

Bit 40 is an F-bit specifying the instruction (0: left, 1: right) to which control is to be transferred.

Bits 41-47 are ignored.

Chapter 7

MIXED INPUT DECKS

Mixed Input. The BITS Input Control Card. The TACL Input Control Card.

RELocatable binary decks may be combined with TAC-language source decks to form a mixed input deck. (See Figure 13, below). This mixed deck may then be compiled in one of the three object formats: REL, ABS, or RPL.

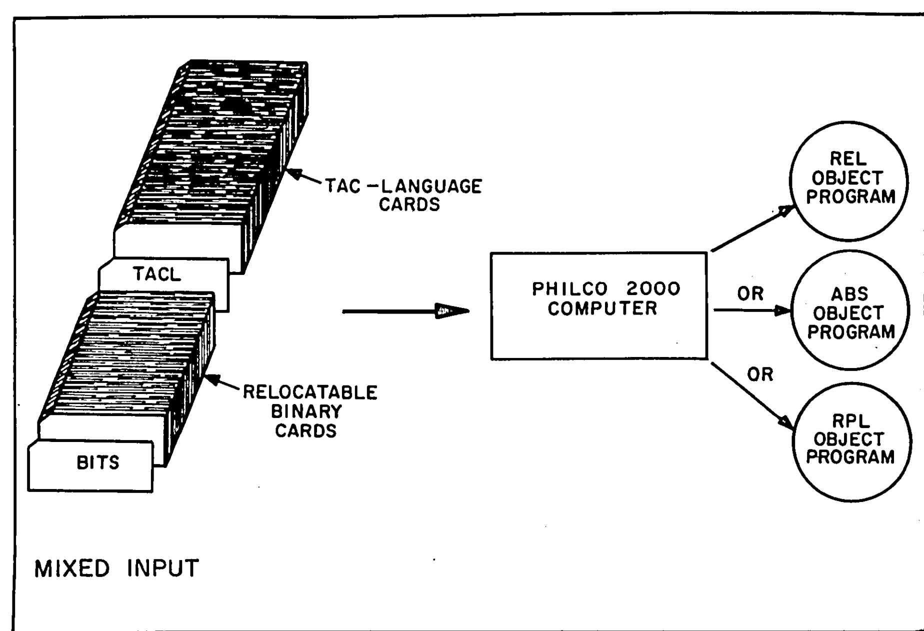


Figure 13 – A Mixed Input Deck

To specify the forms (TAC-language or binary) of the respective decks, the input control cards TACL and BITS (TACL before the TAC-language decks, and BITS before the relocatable binary decks) must be included by the programmer. Also, the pre-compilation card-to-tape operation must be performed in image-mode.*

*See the Philco 2000 Operating System Manual, TM-23, for a definition of image mode.

THE BITS INPUT CONTROL CARD

The BITS input control card is inserted preceding the relocatable binary decks of the mixed deck, and it indicates that relocatable binary cards follow. The characters BITS are punched in Hollerith in columns 9-16 of the card.

THE TACL INPUT CONTROL CARD

The TACL input control card is inserted preceding the TAC-language decks of the mixed deck, and it indicates that TAC-language cards follow. The characters TACL are punched in Hollerith in columns 9-16 of the card.

Chapter 8

THE CODE-EDIT

Contents of the Code-Edit. Source and Object Program Listing. Error Indications. Generated Remarks. Miscellaneous Information.

CONTENTS OF THE CODE-EDIT

The Code-Edit is a printed list of the source program, the object program, and other information relevant to the compilation process. The source program is shown from I card to END card exactly as written by the programmer; the corresponding object program is printed in octal adjacent to and following the source program.

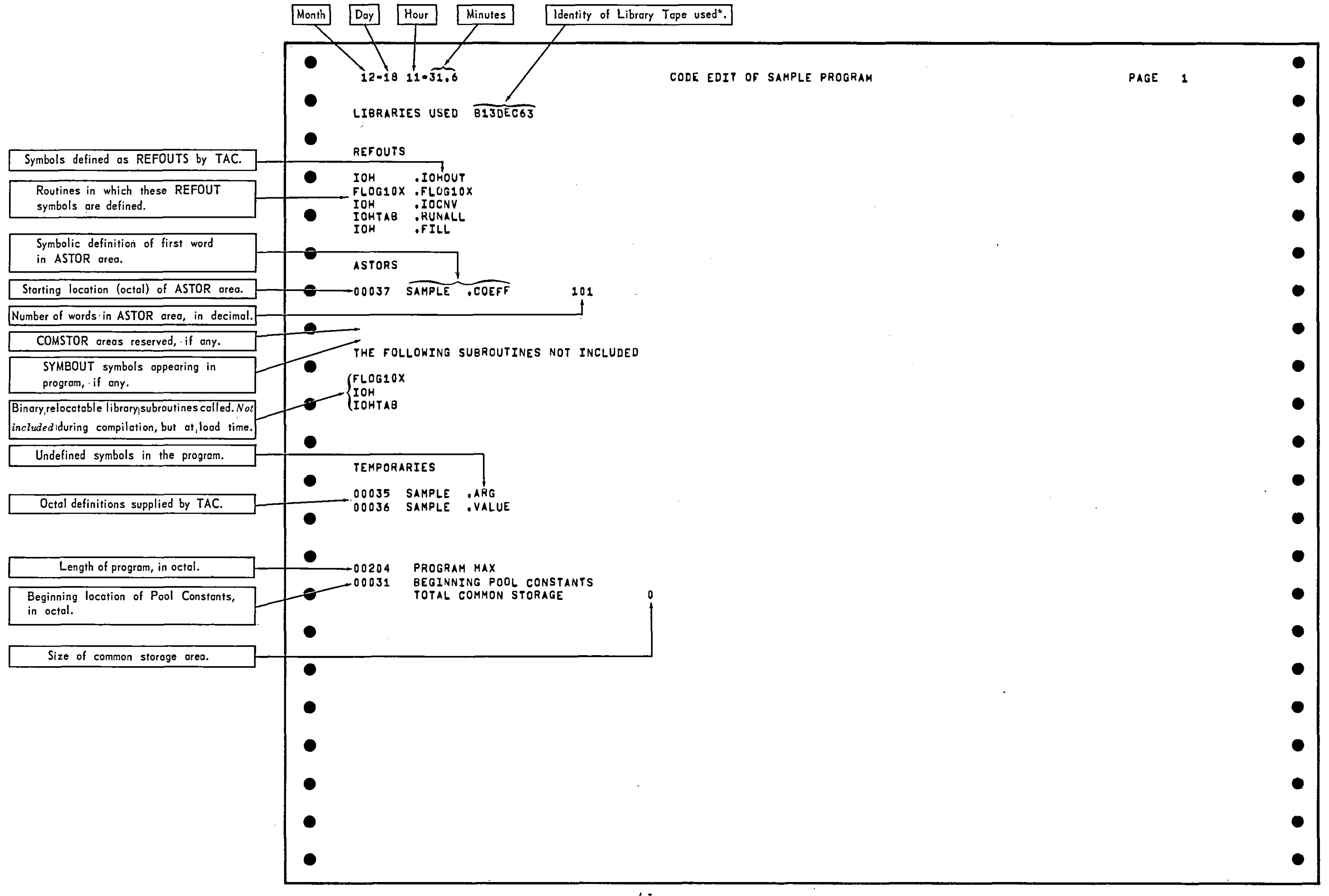
The Code-Edit is produced by TAC during compilation, and is written on the Edited Output Tape*, then printed off-line. In addition to *remarks* and *error notices* which may be included to aid the programmer in debugging an unsuccessfully compiled program, the following information is also printed:

- Library tapes used
- REFOUT symbols
- ASTOR symbols
- COMSTOR symbols
- SYMBOUT symbols
- Called library subroutines that are not included in the compilation
- TEMPORARY symbols
- Program length
- Starting location of Pool Constants
- Amount of common storage
- Symbol Table
- Source and Object Programs
- The coding of called subroutines (if compiled with the program)
- Pool Constants

A Code-Edit of the sample program in Figure 1 is shown below. This program was compiled in RELocatable format; if it were compiled in ABS or RPL format, information such as SYMBOUTS and REFOUTS would be irrelevant, and the notice SUBROUTINES NOT INCLUDED would constitute a serious error condition.

---

*Also referred to as the Code-Edit Tape.



*If no libraries are used in the compilation process, the words "NO LIBRARIES USED" are printed.

The Symbol Table lists the definitions of all symbols (alphanumerically sorted) encountered during the compilation process.

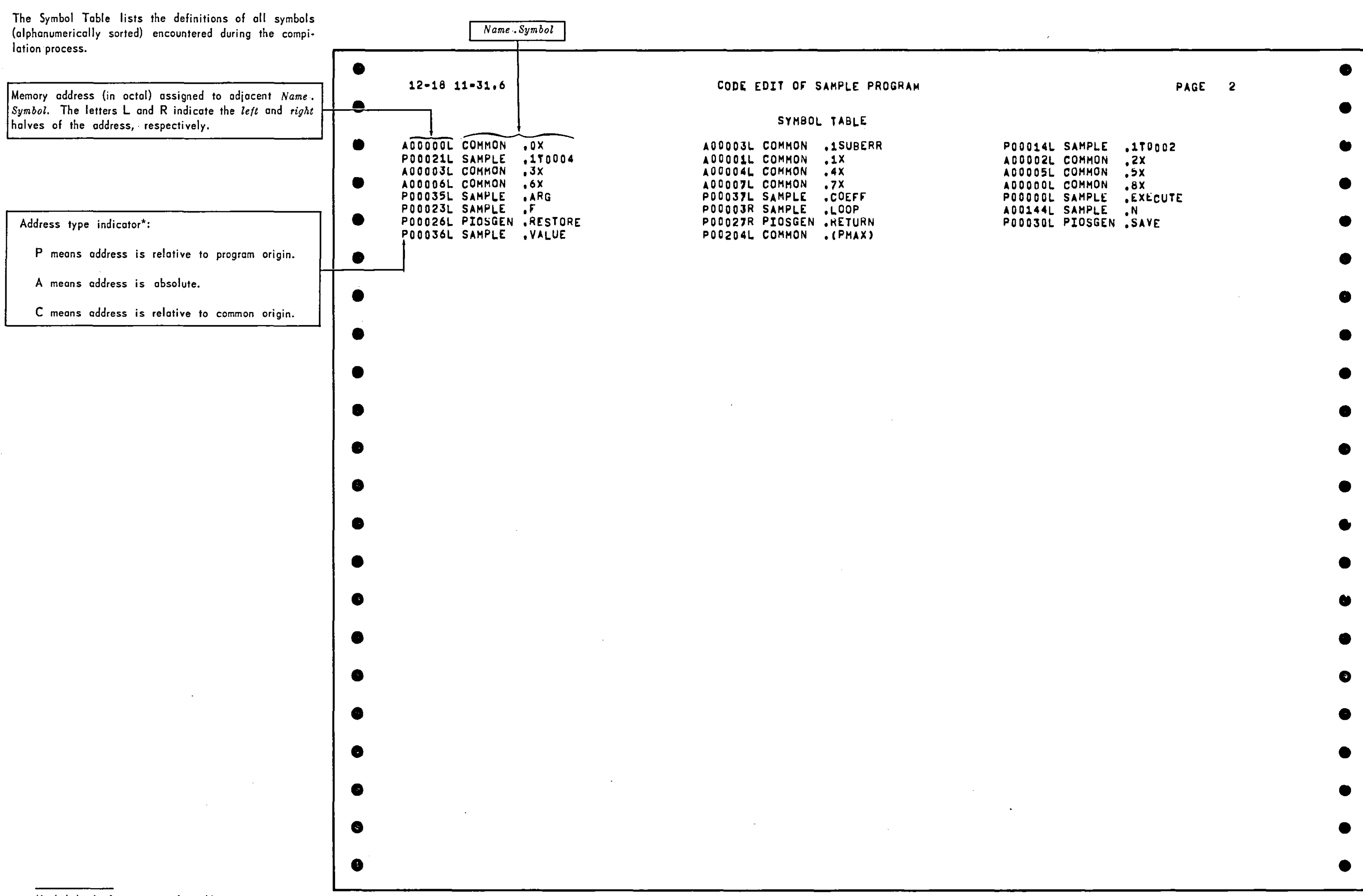
Memory address (in octal) assigned to adjacent *Name. Symbol*. The letters L and R indicate the *left* and *right* halves of the address, respectively.

Address type indicator*:

P means address is relative to program origin.

A means address is absolute.

C means address is relative to common origin.

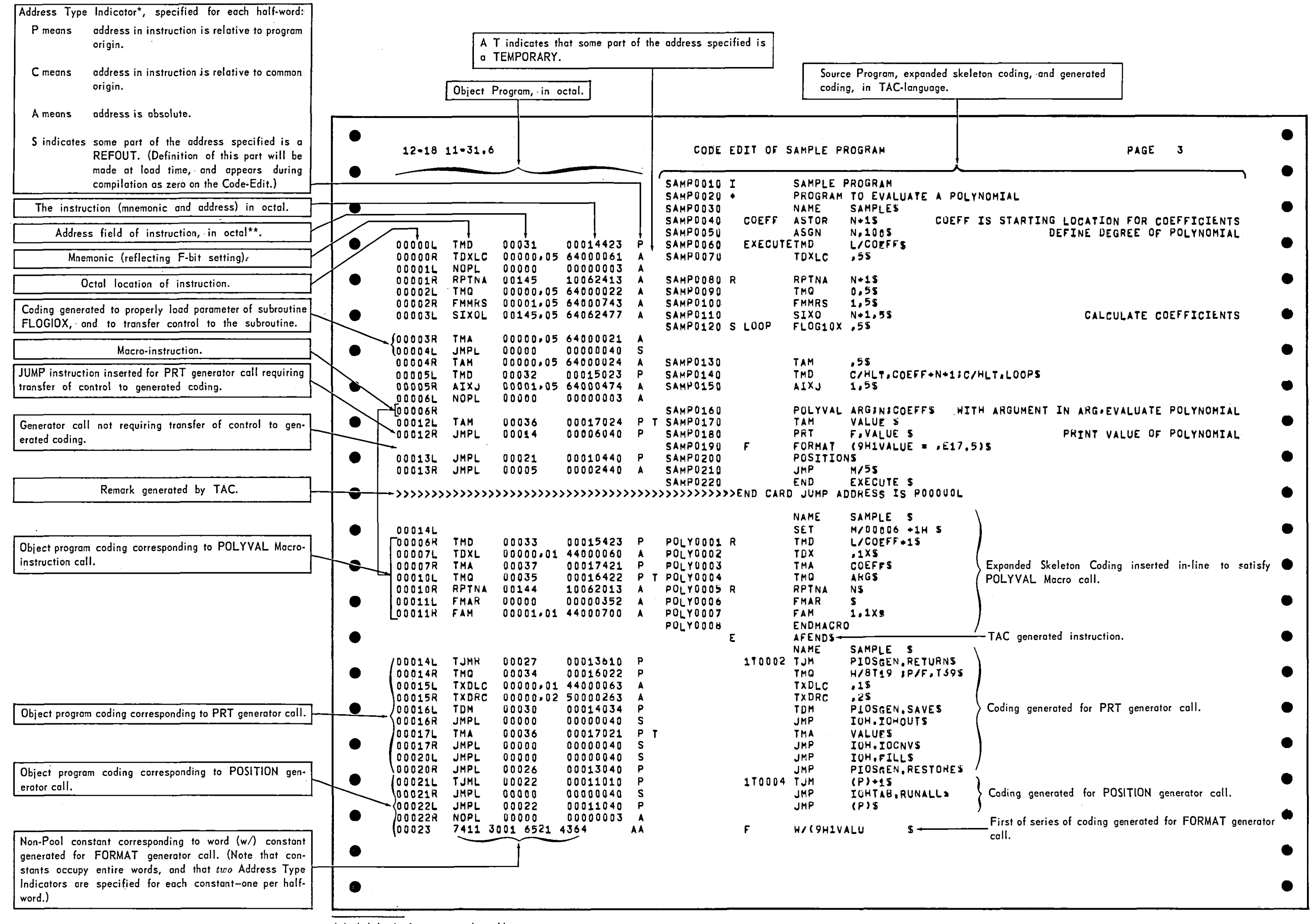



```
12-18 11-31.6                    CODE EDIT OF SAMPLE PROGRAM                         PAGE  2

                                        SYMBOL TABLE

A00000L COMMON  .0X          A00003L COMMON  .1SUBERR       P00014L SAMPLE  .1T0002
P00021L SAMPLE  .1T0004      A00001L COMMON  .1X            A00002L COMMON  .2X
A00003L COMMON  .3X          A00004L COMMON  .4X            A00005L COMMON  .5X
A00006L COMMON  .6X          A00007L COMMON  .7X            A00000L COMMON  .8X
P00035L SAMPLE  .ARG         P00037L SAMPLE  .COEFF         P00000L SAMPLE  .EXECUTE
P00023L SAMPLE  .F           P00003R SAMPLE  .LOOP          A00144L SAMPLE  .N
P00026L PIOSGEN .RESTORE     P00027R PIOSGEN .RETURN        P00030L PIOSGEN .SAVE
P00036L SAMPLE  .VALUE       P00204L COMMON  .(PMAX)
```

*Included only if program is relocatable.



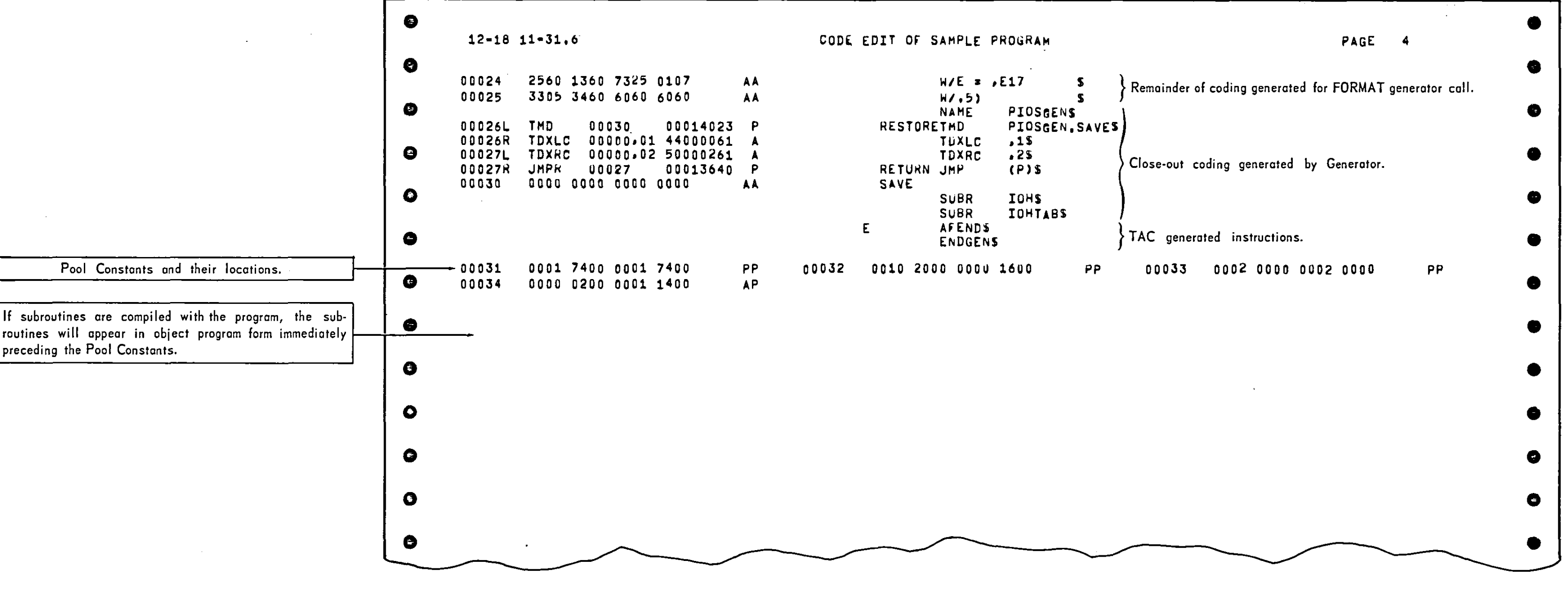

```
12-18 11-31.6                          CODE EDIT OF SAMPLE PROGRAM                                   PAGE   3

                                         SAMP0010 I        SAMPLE PROGRAM
                                         SAMP0020 *        PROGRAM TO EVALUATE A POLYNOMIAL
                                         SAMP0030          NAME    SAMPLES
                                         SAMP0040  COEFF   ASTOR   N+1$       COEFF IS STARTING LOCATION FOR COEFFICIENTS
                                         SAMP0050          ASGN    N,100$                        DEFINE DEGREE OF POLYNOMIAL
00000L TMD    00031    00014423  P       SAMP0060  EXECUTETMD      L/COEFF$
00000R TDXLC  00000,05 64000061  A       SAMP0070          TDXLC   ,5$
00001L NOPL   00000    00000003  A
00001R RPTNA  00145    10062413  A       SAMP0080 R        RPTNA   N+1$
00002L TMQ    00000,05 64000022  A       SAMP0090          TMQ     0,5$
00002R FMMRS  00001,05 64000743  A       SAMP0100          FMMRS   1,5$
00003L SIXOL  00145,05 64062477  A       SAMP0110          SIXO    N+1,5$                          CALCULATE COEFFICIENTS
                                         SAMP0120 S LOOP   FLOG10X ,5$
00003R TMA    00000,05 64000021  A
00004L JMPL   00000    00000040  S
00004R TAM    00000,05 64000024  A       SAMP0130          TAM     ,5$
00005L TMD    00032    00015023  P       SAMP0140          TMD     C/HLT,COEFF+N+1;C/HLT,LOOP$
00005R AIXJ   00001,05 64000474  A       SAMP0150          AIXJ    1,5$
00006L NOPL   00000    00000003  A
00006R                                   SAMP0160          POLYVAL ARG;N;COEFF$  WITH ARGUMENT IN ARG,EVALUATE POLYNOMIAL
00012L TAM    00036    00017024  P T     SAMP0170          TAM     VALUE $
00012R JMPL   00014    00006040  P       SAMP0180          PRT     F,VALUE $                   PRINT VALUE OF POLYNOMIAL
                                         SAMP0190   F      FORMAT  (9H1VALUE = ,E17.5)$
00013L JMPL   00021    00010440  P       SAMP0200          POSITION$
00013R JMPL   00005    00002440  A       SAMP0210          JMP     M/5$
                                         SAMP0220          END     EXECUTE $
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>END CARD JUMP ADDRESS IS P0000L

                                                           NAME    SAMPLE  $
00014L                                                     SET     M/00006 +1H $
00006R TMD    00033    00015423  P       POLY0001 R        TMD     L/COEFF+1$
00007L TDXL   00000,01 44000060  A       POLY0002          TDX     ,1X$
00007R TMA    00037    00017421  P       POLY0003          TMA     COEFF$
00010L TMQ    00035    00016422  P T     POLY0004          TMQ     ARG$
00010R RPTNA  00144    10062013  A       POLY0005 R        RPTNA   N$
00011L FMAR   00000    00000352  A       POLY0006          FMAR    $
00011R FAM    00001,01 44000700  A       POLY0007          FAM     1,1X$
                                         POLY0008          ENDMACRO
                                                  E        AFENDS
                                                           NAME    SAMPLE  $
00014L TJMR   00027    00013610  P               1T0002    TJM     PIOSGEN,RETURN$
00014R TMQ    00034    00016022  P                         TMQ     H/8T19 ;P/F,T39$
00015L TXDLC  00000,01 44000063  A                         TXDLC   ,1$
00015R TXDRC  00000,02 50000263  A                         TXDRC   ,2$
00016L TDM    00030    00014034  P                         TDM     PIOSGEN,SAVE$
00016R JMPL   00000    00000040  S                         JMP     IOH,IOHOUT$
00017L TMA    00036    00017021  P T                       TMA     VALUE$
00017R JMPL   00000    00000040  S                         JMP     IOH,IOCNV$
00020L JMPL   00000    00000040  S                         JMP     IOH,FILL$
00020R JMPL   00026    00013040  P                         JMP     PIOSGEN,RESTORE$
00021L TJML   00022    00011010  P               1T0004    TJM     (P)+1$
00021R JMPL   00000    00000040  S                         JMP     IOHTAB,RUNALL$
00022L JMPL   00022    00011040  P                         JMP     (P)$
00022R NOPL   00000    00000003  A
00023  7411 3001 6521 4364       AA                 F      W/(9H1VALU      $
```

* Included only if program is relocatable.

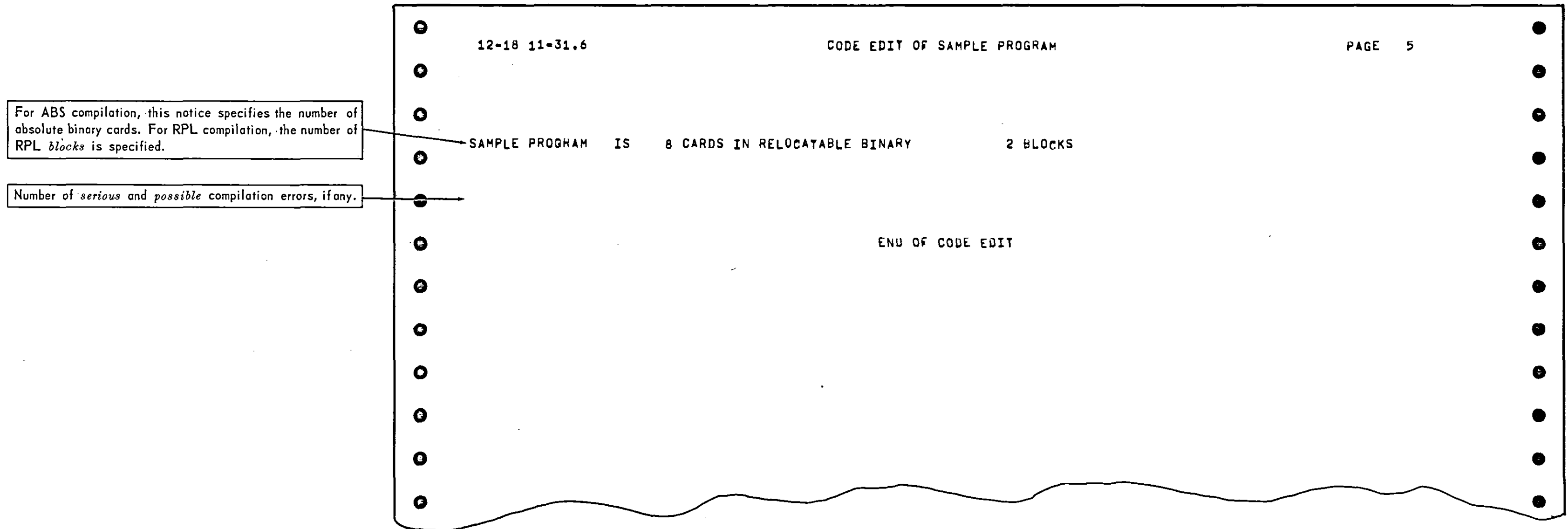
**Both V and N fields, if an index register is specified.



```
12-18 11-31.6                        CODE EDIT OF SAMPLE PROGRAM                          PAGE   4

00024    2560 1360 7325 0107       AA                      W/E = ,E17     $
00025    3305 3460 6060 6060       AA                      W/,5)          $
                                                           NAME     PIOSGEN$
00026L   TMD     00030    00014023  P            RESTORETMD     PIOSGEN,SAVE$
00026R   TDXLC   00000,01 44000061  A                      TDXLC    ,1$
00027L   TDXRC   00000,02 50000261  A                      TDXRC    ,2$
00027R   JMPR    00027    00013640  P            RETURN JMP     (P)$
00030    0000 0000 0000 0000       AA            SAVE
                                                           SUBR     IOH$
                                                           SUBR     IOHTABS
                                              E            AFENDS
                                                           ENDGENS

00031    0001 7400 0001 7400       PP     00032    0010 2000 0000 1600     PP     00033    0002 0000 0002 0000     PP
00034    0000 0200 0001 1400       AP
```

```
12-18 11-31.6                        CODE EDIT OF SAMPLE PROGRAM                          PAGE   5


SAMPLE PROGRAM    IS    8 CARDS IN RELOCATABLE BINARY          2 BLOCKS



                                END OF CODE EDIT
```

ERROR INDICATIONS

If an error is encountered in the source program during compilation, an appropriate error indication is printed following the erroneous instructions or constant on the code-edit, and compilation continues. Serious error indications are preceded by minus signs; possible error indications* are preceded by asterisks.

Serious Errors

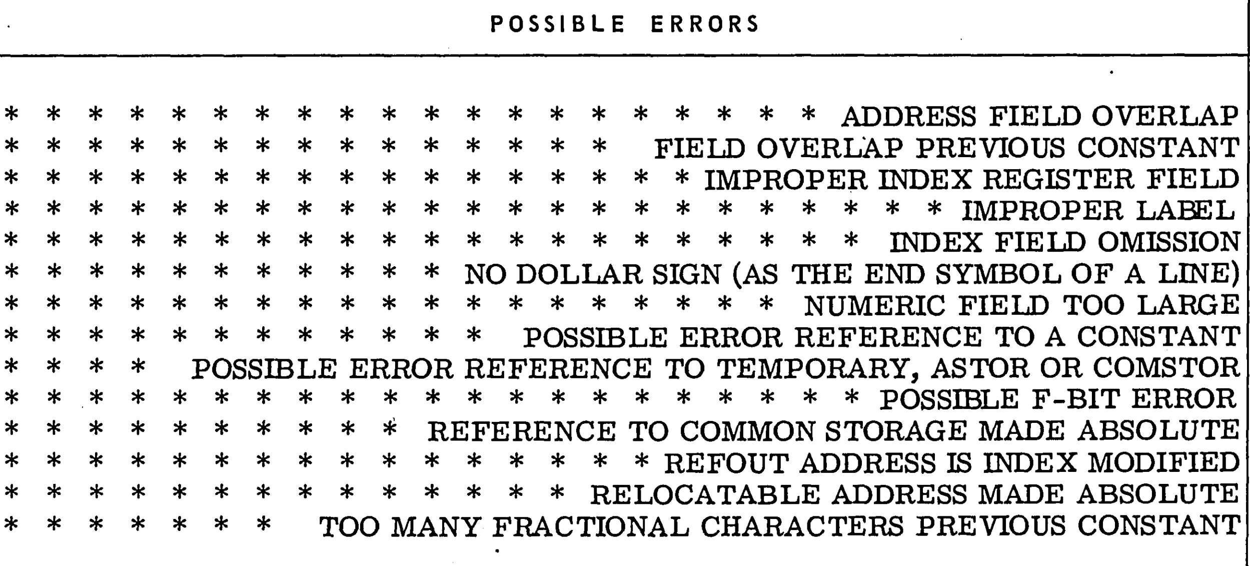
The following is a list of *serious* error indications that may be printed on the code-edit.

SERIOUS ERRORS
- ADDRESS FIELD ERROR
- - - - - - - - - - - - - - - - ADDRESS OF NEXT INSTRUCTION CYCLES MEMORY
- AMBIGUOUS OR CONFLICTING F BITS
- BINARY CARD "nnnnnnnn" BAD
- COMMAND FIELD ERROR
- CONTROL CARD ADDRESS FIELD ERROR
- - - - - - - - - - - - - - - - - - CSA OF BINARY CARD IMPROPER, USED ZERO
- DOUBLE ASSIGNMENT
- - - - - - - - DOUBLE ASSIGNMENT WOULD OCCUR IF SYMBOL WERE COMMON
- END CARD ADDRESS ERROR
- ILLEGAL CONSTANT ON PREVIOUS CARD
- ILLEGAL CONTINUATION CARD
- - - - - - - - - - - - - - - - - - IMPROPER REFERENCE TO AN INDEX REGISTER
- LABEL FIELD ERROR
- LOCATION FIELD ERROR
- NAME FIELD ERROR, NONAME USED
- P/ CONSTANT NOT PROPERLY POSITIONED
- PROGRAM CYCLES MEMORY
- - - - - - - - - - - - - - - - - - SET PARAMETER NOT PREVIOUSLY DEFINED
- SYMBOL TABLE OVERFLOW
- TOO MANY NAMES, NONAME USED
- "xxxxxxxx" NOT DEFINED
- 212 COMMAND IN 211 PROGRAM

*Indications of minor errors which are due to unusual coding techniques, but may have been intended by the programmer.

Possible Errors

The following is a list of possible error indications that may be printed on the code-edit:



POSSIBLE ERRORS

* ADDRESS FIELD OVERLAP
* * * * * * * * * * * * * * * FIELD OVERLAP PREVIOUS CONSTANT
* * * * * * * * * * * * * * * * * IMPROPER INDEX REGISTER FIELD
* IMPROPER LABEL
* INDEX FIELD OMISSION
* * * * * * * * * * * NO DOLLAR SIGN (AS THE END SYMBOL OF A LINE)
* * * * * * * * * * * * * * * * * * * NUMERIC FIELD TOO LARGE
* * * * * * * * * * * * POSSIBLE ERROR REFERENCE TO A CONSTANT
* * * * POSSIBLE ERROR REFERENCE TO TEMPORARY, ASTOR OR COMSTOR
* POSSIBLE F-BIT ERROR
* * * * * * * * * * REFERENCE TO COMMON STORAGE MADE ABSOLUTE
* * * * * * * * * * * * * * * * REFOUT ADDRESS IS INDEX MODIFIED
* * * * * * * * * * * * * * RELOCATABLE ADDRESS MADE ABSOLUTE
* * * * * * * TOO MANY FRACTIONAL CHARACTERS PREVIOUS CONSTANT

GENERATED REMARKS

In addition to the error indications discussed above, any of the following three remarks about the END card may also be included on the code-edit by TAC:

> > > > > > > > > > > > NO ADDRESS IN END CARD
> > > > > > > > > > > > > > REFOUT IN END CARD
> > > > > > > > END CARD JUMP ADDRESS IS xxxxxxxx

Appendix A lists other comments, accompanying the Code-Edit, that are typed out on the Console Typewriter by TAC during compilation.

Appendix A

CONSOLE TYPEWRITER TYPE-OUTS

Operating System Typeouts of Source Program Errors, Tape and Other Errors Encountered During Compilation.

During the compilation process, one or more of the following notices are typed out on the Console Typewriter by the TAC Assembler.

TYPEOUT	MEANING	SUGGESTED RECOVERY ACTION
ID IS *xx*. . .*xx*	Program Identity is *xx*. . .*xx*	
TAPE *n* NOT AV	Tape unit identified as *n* is not available. Computer halts with M/11111 displayed in its Program Register.	Make tape available and press ADVANCE to continue.
TAPE *n* NO LIB	The tape mounted on Tape Unit *n* was not a library tape.	Mount correct library tape and press ADVANCE to continue.
BAD LIB TAPE *n* IGNORED	An illegal control word was encountered in the Table of Contents of the library tape now on tape *n*.	None. TAC continues assembling, ignoring the rest of the Table of Contents of that library tape.
BAD RPL	A missing control word was detected in a called Generator or Macro-generator program. Computer halts with M/77777 displayed in the Program Register.	None.
TAPE *n* IN LOCAL	Tape unit *n* is in local status. Computer halts with M/11111 displayed in the Program Register.	Make tape available and press ADVANCE to continue.

TYPEOUT	MEANING	SUGGESTED RECOVERY ACTION
TAPE *n* WR RING	Tape unit *n* is missing a write ring. Computer halts with M/ 11111 displayed in the Program Register.	Make tape available and press ADVANCE to continue.
TAPE *n* ROCKED 5	A Parity or Sprocket error was detected on tape *n*. Program tried to correct error five times and failed. Computer halts with M/11111 displayed in the Program Register.	Press ADVANCE to attempt to correct error five more times, or change tape and restart job.
TAPE *n* NO GOOD	Non-recoverable tape error detected on tape *n*. Program	Change tape *n* and restart job.
GEN ERR	A Generator requested more parameters than were supplied in its generator call by the programmer. TAC assumes the call to be satisfied and proceeds with the compilation.	Check generator or generator call.
1 DMP TAC or .2 DMP TAC	A TAC Assembler Program error. Computer halts with M/ 77777 displayed in the Program Register.	Get post-mortem dumps and call Philco's Programming Department.
T *n* *x* CARDS *y* BLOCKS	This typeout occurs at the end of compilation if the object program is in REL or ABS format. It tells the number (*x*) of cards of information transferred to the Object Program Tape, *n*, and the number of blocks on this tape that contains this transferred information. (Card output blocks are always in image mode, 20 words per card, 6 cards per block. The last block is filled with blanks.)	
RPL BLOCKS *y*	This typeout indicates the number of blocks (*y*) the RPL object program comprises on tape.	

Appendix B

LOADING OBJECT PROGRAMS

Typical REL, ABS, and RPL Program Loaders. Processing of Object Program Cards and Tape. Address Modifications. F-Bit Modifications. Checksum.

This appendix briefly describes the loading functions performed by typical TAC loader programs.

During compilation, a binary object program is produced on tape, together with information necessary for a loader to properly load this object program. The loader may be a RELocatable program loader, an ABSolute program loader, or an RPL program loader, depending on whether REL, ABS, or RPL object program format was specified.

THE RELOCATABLE PROGRAM LOADER

It is not necessary in a REL compilation for every individual program section, such as a binary library subroutine or a previously compiled subprogram, to be included in the compilation. At load time, the separately compiled programs or program sections can be loaded together by the REL loader to form an integrated program. All the necessary linkage or intercommunication between the respective programs is achieved by the loader from information on SYMBOL DEFINITION CARDS (see page 58), and RELOCATABLE INSTRUCTION CARDS (see page 60), which are produced during compilation from SYMBOUT and REFOUT source cards respectively (see pages 22 and 23).

The loading origin for relocatable programs is a variable which depends on the program size and the amount of available memory locations. The loading origin of a relocatable program is calculated by the Relocatable Program Loader, by subtracting the length of the program, which is supplied on a PMAX card by TAC (see page 56), from the address of the last available memory location. For example, if a REL program requires 4000 memory locations, and the amount of available memory is 32,768 locations, then, the program's loading origin would be location 28,768 (i.e., 32,768 - 4000).

If the address definition assigned to the symbol ALPHA during compilation is 2000, ALPHA will refer to location 30,768 (i.e., 2000 locations relative to 28,768).

REL object programs are loaded in a forward direction (in order of increasing location) at the *end* of memory. The first object program is loaded in such a way that the last location of the program occupies the last memory location; the second object program is loaded so that *its* last location occupies the location immediately preceding the location occupied by the first location of the first program, and so on. (Refer to Figure 14, below.)

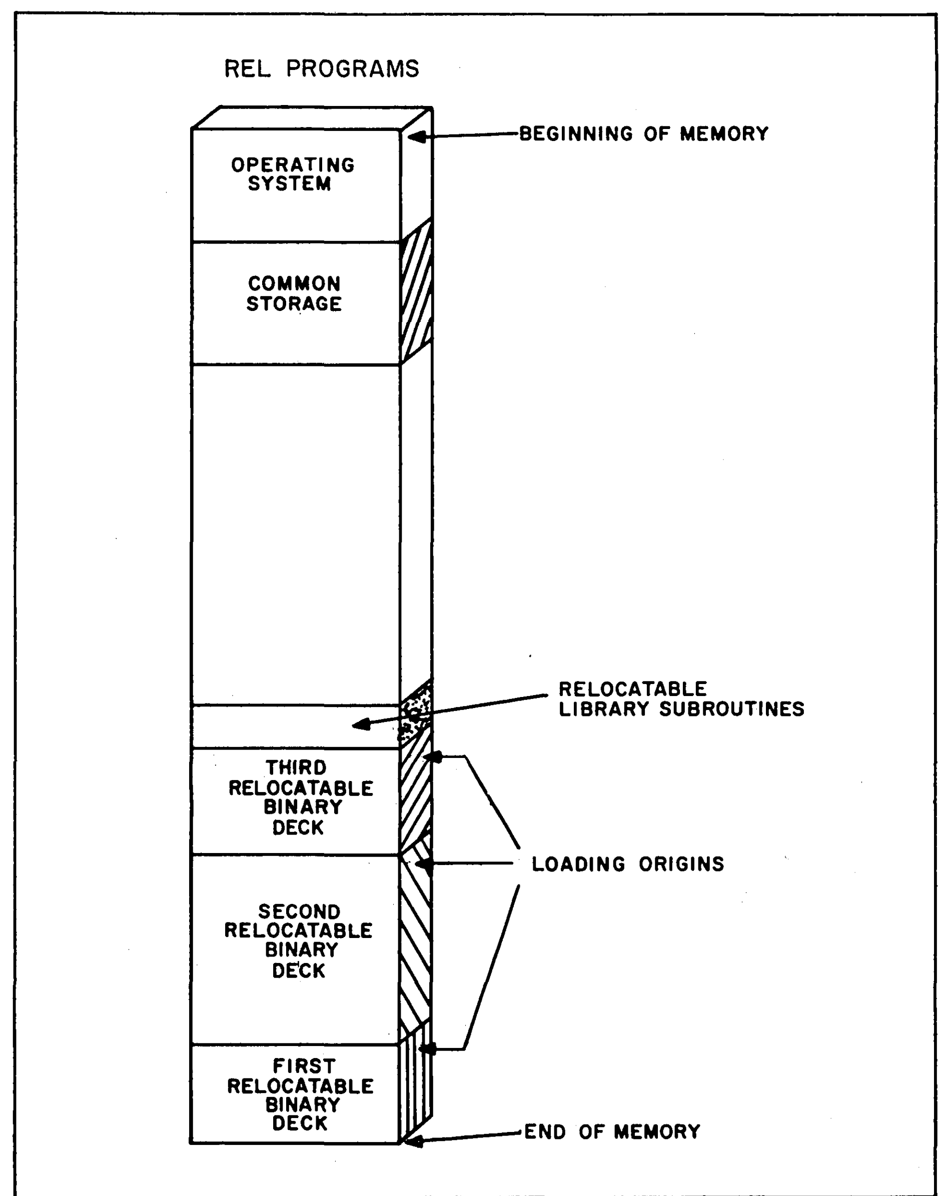


Figure 14 – Loading of Relocatable Object Programs

For REL programs, the common storage area starts immediately after the last memory location occupied by the operating

system.* The length of this area will be the largest of the common storage requirements indicated on the individual PMAX cards.

An error check is made by the REL loader to insure that no relocatable program overlaps the common storage area. It adjusts all addresses relative to the common origin or relative to a program origin.

After all the respective binary card decks representing the individually compiled program sections are loaded, the REL loader checks for remaining undefined symbols (REFOUTs). If undefined symbols exist, the loader searches the TAC binary relocatable library tape for the subroutines containing definitions on SYMBOL DEFINITION CARDS for these REFOUT symbols. These subroutines are then loaded. If undefined symbols still exist after the library search is completed, an appropriate error notice is given and the loader transfers control to the operating system.

REL LOADER PROCESSING OF OBJECT PROGRAM CARDS

The REL loader processes every object program card. The manner in which the information on these cards (shown in the format explanation charts in Chapter 6) is processed is described below.

Processing Done For All Cards

- Columns 1-8 of each card are ignored. These columns contain Identity and Sequence information in Hollerith, and have no effect on loader processing.
- Columns 9-12 of each card are checked to determine if they contain punches. If these columns contain no punches, the card is ignored.
- If one or more of columns 9-12 contain punches, row 3 or rows 3-9 of column 10 are checked for card identification. If these rows are not punched, loader processing halts and control is returned to an error routine in the operating system.
- Columns 11 and 12 of each card contain a checksum for that card. If this checksum and that computed by the loader do not agree, loader processing halts and control is returned to an error routine in the operating system.**

  If columns 11 and 12 are blank, no checksum is computed for the card.

Address Modifications

The Relocatable Program Loader adjusts all relative addresses appearing on the SYMBOL DEFINITION CARDS, the RELOCATABLE BINARY INSTRUCTION CARDS; and the RELOCATABLE END-PROGRAM CARD. Absolute addresses are not modified.

---

* If SYS (the Philco 2000 Operating System) is the operating system used, the last memory location occupied by the operating system is 777 octal. If 32K SYS is the operating system used, the last memory location occupied by the operating system is 7,777 octal.

** Refer to the checksum technique discussed on page 88.

Processing The PMAX Card

In addition to the processing indicated on page 81, which is done for all object program cards, the PMAX card is further processed by the loader as follows:

- The loader determines the loading origin for the object program from the contents of columns 13, 14, and 16:

  If row 9 of column 16 is punched (one), the address specified in columns 13 and 14 is the loading origin of the object program.

  If row 9 of column 16 is not punched (zero), the information contained in columns 13 and 14 is subtracted from the address of the *last available* memory location, and this difference becomes the loading origin of the object program.

- The loader establishes the origin of the common storage area as that location immediately following the last location occupied by the operating system, and adjusts all common addresses, specified by the contents of columns 25 and 26 of the PMAX card, relative to this common origin.

- After processing the information in the other columns of the PMAX card, the loader checks to insure that available memory is not exceeded, and no overlap of the common storage area occurs. If memory is exceeded or an overlap occurs, loading is terminated, an appropriate error indication is typed by the console typewriter, and a post-mortem dump is performed.

Processing Symbol Definition Cards

The loader constructs a list of the symbols whose definitions are found on SYMBOL DEFINITION CARDS. This symbol list occupies the area that was assigned to ASTORS, TEMPORARIES, and common storage at compile time. The loader inserts the definitions specified in these cards into the symbol list as follows:

- If the symbol is not already in the list, it is placed in the list with its definition.

- If the symbol is already in the list, and is undefined (that is, it is placed in the list because of a previously encountered REFOUT on a RELOCATABLE INSTRUCTIONS CARD OR RELOCATABLE END-PROGRAM CARD), its definition is placed in the list, and each address, on the previously encountered RELOCATABLE INSTRUCTIONS and RELOCATABLE END-PROGRAM cards, that contains a reference to the symbol is replaced with the *sum* of the definition of the symbol *and* the compiled definition of the address.

- If the symbol is already defined in the list, a "double definition" error notice is typed on the Console Typewriter, and the first definition specified is used.

Processing Relocatable Binary Instruction Cards

In processing a RELOCATABLE BINARY INSTRUCTION CARD, the Relocatable Program Loader adjusts the definitions of the addresses on the card relative to program origin or to common origin, according to the indicator bits in columns 13-16 of the card. No adjustment is made for absolute addresses.

If the address references a REFOUT symbol, the loader:

- Adds the REFOUT symbol to the symbol list if it is not already in the list, or

- If the symbol and its definition is already in the list (derived from the symbol's appearance on a previously encountered SYMBOL DEFINITION CARD), the loader obtains its definition from the list, and replaces the address referencing that symbol with the *sum* of the definition of the symbol *and* the compiled definition of the address.

Processing The Relocatable End-Program Card

If an address element was specified in the address and remarks field of the END source card, the resulting RELOCATABLE END-PROGRAM CARD will cause the loader to transfer control to this address after loading the program. Before transferring control, however, the loader:

- Adjusts the address in columns 15 and 16 of the card according to the indicator bits in column 13, in the same manner as described above in processing RELOCATABLE BINARY INSTRUCTION CARDS.

- Checks its list for any remaining undefined symbols (REFOUTs). If no undefined symbols remain, control is transferred to the address in the End-Program Card.

  If undefined symbols remain, the loader searches the TAC relocatable binary library tape for subroutines containing defintions on SYMBOL DEFINITION CARDS for these REFOUT symbols. The loader then loads these subroutines and transfers control to the address in the End-Program Card.

  If undefined symbols still exist after the library search is completed, an appropriate error notice is given, and the loader transfers control to the operating system.

If no address element was specified on the END source card, the loader will attempt to load the next program directly after loading this program.

F-Bit Modifications At Load Time

In order to have an instruction that contains a REFOUT symbol refer to the proper half of the address represented by that symbol, the relocatable program loader may modify the F-bit of the instruction by adding zero or one (corresponding to the address being left or right, respectively) to the F-bit. For example, if the symbol ENTRAN1 represents a right address in a called subroutine, and the instruction TDM in a calling program refers to this address, TDM will be changed to SCD at load time.

Checksum Performed by The Loader

The loader computes the checksum for an object program card by summing all the *relevant* data bits on the card, and compares this computed checksum with the checksum value existing in columns 11 and 12 of the card. If these two columns are blank, no checksum is computed.

The checksum values will agree when data on the card has been transferred accurately. If these values do not agree, loader processing halts, and control is returned to an error routine in the operating system.

The following coding shows the checksum procedure; assume that symbolic index register CARD is set to the address of a word containing columns 1-4 of the card being checksummed:

L	Location	Command	Address and Remarks
		TJM	CKSUMX $
		CD	$
		TXDLC	0,CARD $
		TDM	CARDLOC $
		TMQ	2,CARD $ TEST FOR COLUMNS
		ETA	24/1T47 $ 11-12 BLANK
		JAZ	CKSUMX $
		ETA	24/1T23 $ COLUMNS 9-10 TO A_L
		CQ	$
		SRAQ	1 $
		AIXOL	3,CARD $ SET TO COLUMN 13
R	VARRPT	RPTAN	(0) $ PRESET BY CARD TYPE
		AMA	1,CARD $
		SRAQ	1 $
		AQA	$
		TAD	$
		SCD	24 $
		AD	$
		SRA	24 $
		TMD	CARDLOC $
		TDXLC	0,CARD $ RESET TO COLUMN 1
		TMQ	24/1T47 $ COMPARE WITH
		ES	2,CARD $ CHECKSUM ON CARD
	CKSUMX	JAZ	CKSUMX $ CHECKSUMS AGREE
		JMP	ERRORB $ NO AGREEMENT

Because the number of columns to be summed depends on the card type, the symbol VARRPT in the above coding should be assigned:

- the value *17* for PMAX, SYMBOL DEFINITION, and RELOCATABLE END-PROGRAM cards
- the value *0* for ABSOLUTE END-PROGRAM cards
- the symbol *Y* for RELOCATABLE and ABSOLUTE BINARY INSTRUCTION CARDS. The symbol *Y* is defined as the integer part of the quantity $(x+k+1)$, where

$$x = \frac{\text{Number of instructions on card specified in rows 4-9 of column 10}}{2}$$

and k = 1/2 or 1, depending on whether the F-bit in row 1 - column 10 of the card is zero or one, respectively.

THE ABS AND RPL PROGRAM LOADERS

The loading origin for absolute programs is the compilation base. ABS and RPL programs are loaded in a forward direction, starting at the compilation base.

The common storage area may be located anywhere in memory. The ABS or RPL loader does not keep a record of common storage, not checks for memory overlap.

The following figure shows the loading scheme for ABS and RPL *programs*.

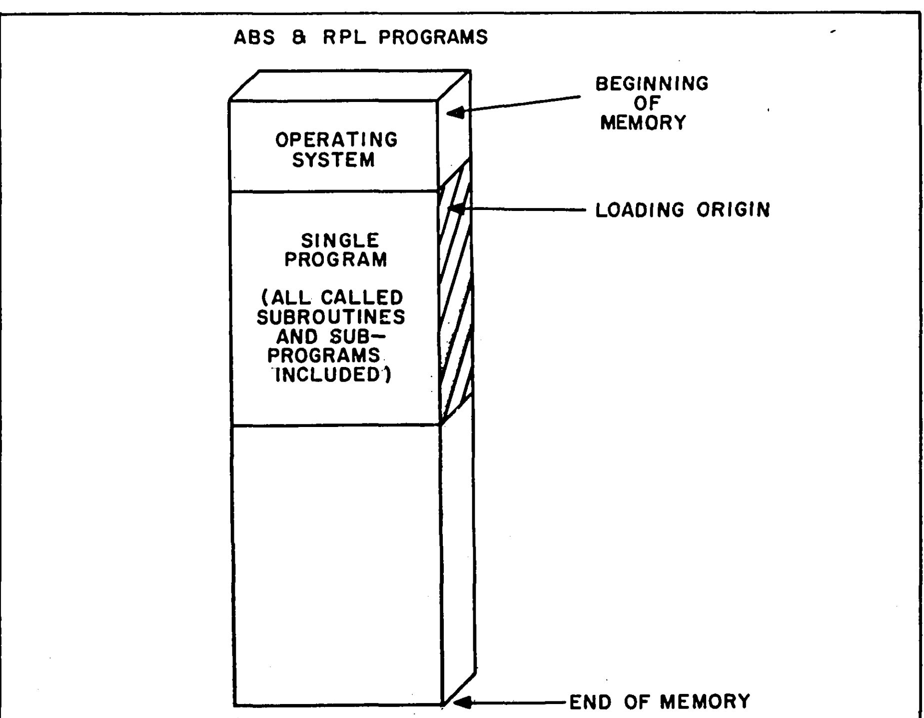


Figure 15 – Loading of Absolute Object Programs

ABS LOADER PROCESSING OF OBJECT PROGRAM CARDS

The ABS loader processes every object program card. The information in columns 1-12 of these cards are processed in the same manner as for Relocatable Program Cards, indicated on page 85. No address modifications are performed.

ABS Loader Processing of Absolute Binary Instruction Cards

In processing an Absolute Binary Instructions Card, the ABS loader loads the instructions appearing on the card consecutively into memory, according to the Core Starting Address specified in columns 9 and 10 of the card.

ABS Loader Processing of The Absolute End Program Card

The address indicated in columns 9 and 10 of the Absolute End-Program Card is the address originally specified in the address field of the END source card, and it is to this address that the loader transfers control subsequent to loading the program.

Appendix C

CALLS ON FORTRAN SUBROUTINES

The F label is used for calls on FORTRAN type subroutines. It is currently a feature of the 32KSYS version of TAC* only.

The general form of this call is:

L Location	Command	Addess and Remarks
F *symb*	*entrance*	$param_1; param_2; \ldots ; param_n\$$

where *symb* represents and optional symbol, and each *param* is a parameter of the subroutine.

The format of the call on a FORTRAN subroutine is similar to that on a standard TAC subroutine (see page 37) except that:

- F in the label field signifies a call on a FORTRAN subroutine.
- None of the parameters may be omitted from the call.

The resultant in-line coding generated for the FORTRAN subroutine call is:

JMP	*entrance.entrance*$
NOP	$param_1\$$
NOP	$param_2\$$
.	
.	
.	
.	
NOP	$param_n\$$

The symbol in the location field, if any, is assigned to the location of the generated JMP instruction. Because FORTRAN subroutines expect transfer of control to have come from a right hand instruction, the JP instruction is always made to occupy the right half of the word as if an R had been written in its label field.

A NOP instruction is generated for each parameter. The address field of the NOP is the address of the parameter.

---

*The version of TAC that is designed to operate under control of the Philco Operating System, 32KSYS.

Appendix D

TABLE OF PHILCO CHARACTERS

PHILCO CHARACTER	OCTAL CODE	HOLLERITH PUNCH
0	00	0
1	01	1
2	02	2
3	03	3
4	04	4
5	05	5
6	06	6
7	07	7
8	10	8
9	11	9
@	12	8-2 ①
=	13	8-3
;	14	8-4
≡	15	8-5 ①
&	16	8-6 ①
'	17	8-7
+	20	12
A	21	12-1
B	22	12-2
C	23	12-3
D	24	12-4
E	25	12-5
F	26	12-6
G	27	12-7
H	30	12-8
I	31	12-9
n ②	32	12-8-2 ①
.	33	12-8-3
)	34	12-8-4
%	35	12-8-5 ①
?	36	12-8-6 ①
"	37	12-8-7 ①

PHILCO CHARACTER	OCTAL CODE	HOLLERITH PUNCH
-	40	11 or 8-4 ①
J	41	11-1
K	42	11-2
L	43	11-3
M	44	11-4
N	45	11-5
O	46	11-6
P	47	11-7
Q	50	11-8
R	51	11-9
¬	52	11-8-2 ①
$	53	11-8-3
*	54	11-8-4
<	55	11-8-5 ①
#	56	11-8-6 ①
␣	57	11-8-7 ①
Blank (space)	60	Blank
/	61	0-1
S	62	0-2
T	63	0-3
U	64	0-4
V	65	0-5
W	66	0-6
X	67	0-7
Y	70	0-8
Z	71	0-9
|	72	0-8-2 ①
,	73	0-8-3
(	74	0-8-4
>	75	0-8-5 ①
:	76	0-8-6 ①
e ②	77	0-8-7 ①

① Multiple punched.

② These two characters are not acceptable TAC characters, and are included here only to show the complete character codes.

Appendix E

TAC MNEMONICS

The following TAC mnemonics are currently acceptable to the TAC Assembler.

MNEMONICS					
AD	DORMS	FMA	JAGQFR	MMAR	SRD
ADXL	DR	FMAA	JAGQL	MMARS	SRDN
ADXR		FMAAR	JAGQR	MMAS	SRQ
AIXJ	EA	FMAARS	JANL	MMR	SRQN
AIXJEG	EI	FMAAS	JANR	MMRS	SWD
AIXJS	EIS	FMAD	JAPL	MMS	
AIXOL	ENDDP	FMAR	JAPR	MSU	TAD
AIXOR	ES	FMARS	JAZL		TAM
AM	ETA	FMAS	JAZR	NOPL	TAQ
AMA	ETD	FMM	JBTL	NOPR	TCM
AMAS	ETX	FMMA	JBTR		TCXS
AMS		FMAAR	JDPL	RPT	TCXZ
AQ	FAD	FMAARS	JDPR		TDA
AQA	FAM	FMAAS	JL	SCD	TDC
AQAS	FAMA	FMMR	JMPL	SD	TDM
AQS	FAMAS	FMMRS	JMPR	SDXL	TDQ
AWCS	FAMS	FMMS	JNOL	SDXR	TDXL
	FAQ	FMSU	JNOR	SETDP	TDXLC
CA	FAQA	FSD	JOFL	SIXJ	TDXLY
CAM	FAQAS	FSM	JOFR	SIXJES	TDXR
CAMA	FAQS	FSMA	JQEL	SIXJG	TDXRC
CAMAS	FCAM	FSMAS	JQER	SIXOL	TDXRY
CAMS	FCAMA	FSMS	JQNL	SIXOR	TIJL
CAQ	FCAMAS	FSQ	JQNR	SKC	TIO
CAQA	FCAMS	FSQA	JQOL	SKF	TIXS
CAQAS	FCAQ	FSQAS	JQOR	SLA	TIXZ
CAQS	FCAQA	FSQS	JQPL	SLAN	TJML
CD	FCAQAS		JQPR	SLAQ	TJMR
CM	FCAQS	HLTL	JR	SLAQN	TMA
CQ	FCSM	HLTR		SLQ	TMD
CSM	FCSMA		LWD	SLQN	TMQ
CSMA	FCSMAS	ICOS		SM	TQA
CSMAS	FCSMS	ICOZ	MA	SMA	TQD
CSMS	FCSQ	INCAL	MAA	SMAS	TQM
CSQ	FCSQA	INCAR	MAAR	SMS	TTD
CSQA	FCSQAS		MAARS	SQ	TXDL
CSQAS	FCSQS	JAEDL	MAAS	SQA	TXDLC
CSQS	FDA	JAEDR	MAD	SQAS	TXDLY
	FDAQ	JAEQL	MAR	SQS	TXDR
DA	FDAQS	JAEQR	MARS	SRA	TXDRC
DAQ	FDAS	JAGDL	MAS	SRAN	TXDRY
DAQS	FEA	JAGDR	MM	SRAQ	TYXS
DAS	FES	JAGQFL	MMA	SRAQN	TYXZ

Appendix F

SUMMARY LIST OF TAC CONTROL INSTRUCTIONS

This appendix provides a convenient reference to all TAC control instructions discussed in the manual.

INSTRUCTIONS		Page Reference
Command	Address and Remarks	
AFEND	*n*	14
ASGN	*symb, n*	17
ASTOR	*n*	13
COMSTOR	*n*	21
DEFINE	*symb, c*	17
END	*addr*	24
IDENTIFY	*m*k,*nX*	12
NAME	*p*	11
PAGE		23
REFOUT	*addr*	20
SAME	*symb,n*	17
SET	*addr*	15
SETLARGE	$addr_1, addr_2$	16
SETSMALL	$addr_1$, $addr_2$	16
SPACE	*n*	23
SUBR	*name*	23
SYMBOUT	*addr*	19

TM-35 6-64 1M-1

PRINTED IN U.S.A.