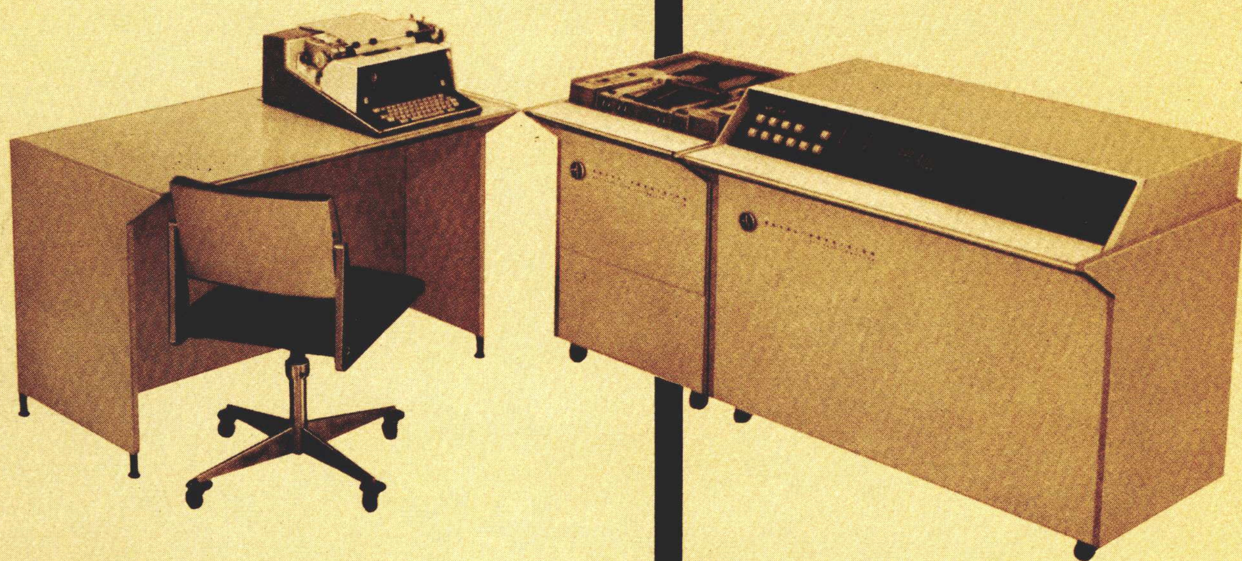
LGP 4000

H3-01.0



COMPACT Reference Manual

GENERAL PRECISION, INC.
Commercial Computer Division

COMPACT

REFERENCE MANUAL

for the **RPC 4000** General Precision Electronic Computer

PROGRAM NO. H3-01.0

© 1963, General Precision, Inc.

CONTENTS

	INTRODUCTION	1
	PUNCHING THE SOURCE PROGRAM	1
	COMPILATION	3
	Loading The Program	3
	Sense Switch Options	4
	Operation of The Compiler	5
	Error Recovery	5
	Initial Instruction	5
	Terminating Compilation	7
	THE COMPILED PROGRAM	7
	Symbols	7
	Roar Pseudo-Instructions	9
	Floating-Point Arithmetic	11
	"Undefined" Functions	13
	Summary of Function-Naming Conventions	14
	ASSEMBLY PROCEDURE	15
	Loading the Assembler	15
	Roar Sense Switch Options	15
	Operation of the Assembler	15
	COMPACT SUBROUTINE LIBRARY	17
	Arithmetic Subroutines	17
	Input/Output Subroutines	20
	Function Subroutines	21
	Assembling the Subroutine Library	23
APPENDIX	**SUMMARY OF COMPACT STATEMENTS**	25
	Arithmetic Statement	25
	Unconditional GO TO	25

Assigned GO TO	25
Assign	26
Computed GO TO	26
If	26
Sense Light	26
If Sense Light	26
If Sense Switch	27
If Accumulator Overflow	27
If Quotient Overflow	27
If Divide Check	27
Pause	28
Stop	28
Do	28
Continue	29
End	29
Call	29
Subroutine	30
Function	30
Return	30
Read	30
Punch	31
Print	31
Read Input Tape	31
Write Output Tape	31
Format	31
Dimension	32
Equivalence	32
Common	33

ILLUSTRATIONS

FIGURE 1 — Compact coating sheet	1
FIGURE 2 — Punching unnamed statements	2
FIGURE 3 — Punching named statements	2
FIGURE 4 — An assembled comment	11
FIGURE 5 — Range of exponents	12

INTRODUCTION

This manual is intended as a programming and operating aid for the user of General Precision's RPC-4000. It presupposes familiarity with the COMPACT Programming Manual and the ROAR III Programming Manual.

COMPACT is a FORTRAN-derived compiler which produces a ROAR-language program. This program, in turn, must be assembled by ROAR III so that it can be processed by the RPC-4000 directly.

The sole concern of this manual is the compilation and assembly of a source program. A discussion of the COMPACT subroutine library and its assembly concludes the subject area treated in this context. Operation of the RPC-4000 and its input and output devices is merely touched upon where necessary, as it is properly the subject of the RPC-4000 Programming Manual.

PUNCHING THE SOURCE PROGRAM

The COMPACT Coding Sheet, illustrated below, is divided into three fields: Comment, Statement Number or Name, and COMPACT Statement.

COMMENT	STATEMENT NUMBER OR NAME	COMPACT STATEMENT
T	h i s p	rogram is to be used only as an example of
c	o d i n g	and is not intended for actual compilation.
		DIMENSION AR1 [10], BC [5,5]
	B E G	POLTR=0.0
		READ FMT1, BC, NY, NZ
		IF [NY-NZ] 30,31,32
	3 2	POLTR = POLTR + COEFA + COEFB* [TEMP = TRIGF = =
		[ARG [IND]]] + COEFC* TEMP* TEMP

FIGURE 1 — Compact coating sheet

The first field is one character wide. If it contains any alphanumeric or typewriter control character other than zero, space, or tab, the line will be treated as a comment. A comment may be several lines long, but is terminated at every carriage return. Thus, the one-sentence example above is interpreted by COMPACT as two comments, each line constituting one comment.

If the Comment field contains a zero, space, or tab, the subsequent characters on that line are not treated as a comment. In addition, if a tab is the first character punched on tape for a given line, it is also assumed that the statement is not named. Figure 2 illustrates how line 3, above, may be punched on tape.

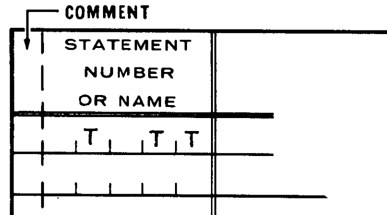
```
        (tab)  DIMENSION AR1 [10] , BC [5, 5]

   or as

        (sp)(sp)(sp)(sp)(sp)(sp) DIMENSION AR1  [10]  , BC [5,5]
```

FIGURE 2 — Punching unnamed statements

The second field is 5 characters wide and contains the statement name or number, if there is one. Spaces are counted in this field, but are otherwise ignored. For example, if a statement name is written on the coding sheet as



and punched on tape as

sp sp T sp TT

COMPACT would count 6 characters: 3 spaces and 3 letters. However, this statement name would be compiled as TTT and so appear on the ROAR-language* listing. Statement names or numbers may consist of up to 5 characters. If the first character is numeric, the field will be treated as a statement number; if it is alphabetic, as a statement name. The field may be terminated by a tab. For example, line 4 in Figure 1 could be punched as shown in Figure 3.

	Comment Field	Statement Name	Statement
	(sp)	(sp)(sp) BEG	POLTR=0. 0
	(sp)	(sp) BEG (sp)	POLTR=0. 0
	(sp)	BEG (sp)(sp)	POLTR=0. 0
or as:			
	(sp)	BEG (tab)	POLTR=0. 0

FIGURE 3 — Punching named statements

In the first method, the statement name BEG preceded by two spaces provides the required 5 characters for this field. A tab following BEG would be permissible but unnecessary. In the second method, the tab terminates the field although only 3 characters were entered.

*All references to "ROAR" in this document are to "ROAR III."

The third field is of indeterminate length and contains the COMPACT statement. Each carriage return indicates the end of a statement. However, if a statement has to be continued to the next line, this must be indicated by entering two consecutive equal signs (==) immediately preceding the carriage return. (See statement 32 in the example.) Note that the (==) convention cannot divide an entity like a variable name, constant, or operator or a Hollerith field. For example, the name SUMM cannot be written

SU==
MM

and the number 186.325 cannot be written

186.==
325

(The "continuation column" convention of FORTRAN is not allowed in COMPACT.)

All typewriter controls except carriage return are ignored in the third field, unless the statement contains a Hollerith field, within which all codes except blank and delete are recognized.

COMPILATION

The compiler assumes a basic RPC-4000 System. The basic punch is selected by the compiler to output the ROAR-language program tape. The basic typewriter is used by the compiler to communicate with the operator whenever an error is encountered in the source program and to produce a hard-copy listing of the compilation. The input device is selected manually.

It is possible to alter the COMPACT program to allow output of the object program listing on some unit other than the RPC-4500 Typewriter. To accomplish this, make the following changes, replacing "XXX" with the desired PRD Select Code:

```
6563*   PRD*   XXX63*   9359*  *
7239*   PRD*   XXX39*   5243*  *
7255*   PRD*    6855*   9363*  *
9363*   PRD*   XXX63*   5309*  *
6460*   PRD*   XXX60*   7800*  *
5328*   PRD*   XXX28*   6132*  *
3753*   PRD*   XXX53*   3257*  *
```

With these changes, COMPACT will transfer all hard-copy output, except error comments, to the specified unit.

Loading The Program

The procedure for loading COMPACT into memory is as follows:

1. Depress MASTER RESET.
2. Place the COMPACT tape in the reader.

3. Select READER TO COMPUTER.

4. Depress ONE OPERATION.

5. Depress SET INPUT MODE.

6. Depress EXECUTE LOWER ACCUMULATOR.

7. Depress START COMPUTE. The computer will read one word and stop.

8. Depress START COMPUTE.

9. Raise EXECUTE LOWER ACCUMULATOR.

10. Depress SET INPUT MODE.

11. Raise ONE OPERATION.

12. Depress START COMPUTE. The program will be read and stored in Tracks 0 through 101. Tracks 102 through 122 are used by COMPACT for temporary storage during the compilation of a program. When input is completed, the computer will halt.

Sense Switch Options

Before control is transferred to COMPACT, Sense Switches must be set if any of the options are to be used. The Sense Switch settings are used to determine the form of the output. Depressed, they will cause the suppression of certain output as listed below. If none are depressed, the output would consist of the ROAR-language program tape (format-controlled with a tab after each field and containing all comments), and a typed listing of both the COMPACT-language source program as it is read and the resultant ROAR-language instructions.

Sense Switch	Function When Depressed
1	Causes the PRE tapes to be output on the RPC-4600 Punch.
2	Suppresses typing of COMPACT statements and comments.
4	Suppresses typing of the ROAR-language instructions.
8	Suppresses punching of COMPACT-language source statements.
16	Suppresses punching (and printing) of tabs after stop codes in the ROAR output tape.
32	No effect.

It is suggested that SENSE SWITCH 2 remain UP during compilation, so the operator can know which statement is currently being processed. With SENSE SWITCH 4 depressed (and 8 and 16, if desired), the compiler will run faster because it does not have to type the ROAR instructions. However, it does not save much time to suppress the typing of the original statements, and if an error should occur, it is difficult--although not impossible--to discover which statement was wrong. (See "ERROR RECOVERY.")

Operation of The Compiler

After the Sense Switches are set, place the COMPACT-language program in the appropriate input device and transfer control to Location 00000, the beginning location of COMPACT. When the program is initially stored in memory, depressing START COMPUTE will cause computer control to be transferred to the beginning location of the program; or the operator may transfer manually to Location 00000:

1. Depress MASTER RESET.
2. Depress ONE OPERATION.
3. Depress SET INPUT MODE.
4. Depress EXECUTE LOWER ACCUMULATOR.
5. Depress START COMPUTE.
6. Depress START COMPUTE.
7. Raise EXECUTE LOWER ACCUMULATOR.
8. Raise ONE OPERATION.
9. Depress START COMPUTE.

COMPACT will select the typewriter for output and print:

SELECT SINGLE CHARACTER MODE

(This printout will be bypassed if the SINGLE CHARACTER MODE switch is already down.) Depress this switch and START COMPUTE. Next, COMPACT will print:

SELECT INPUT DEVICE

COMPACT does not wait for the selection to be made, but immediately starts to punch a leader of blank tape. While this happens, depress the switch which selects the appropriate input device. Note: if an RPC-4510 Reader is to be used for input and a tape is not in the reader when the device is selected, an "F" will be loaded into the Accumulator.

Error Recovery

If an error is encountered in the source program during compilation, COMPACT will select the basic typewriter for output and print an indication of what is wrong with a particular statement. If COMPACT selects the typewriter for input after such a printout, it is usually possible to correct the error by retyping the last statement correctly. If the typewriter is not selected for input after an error printout, the error is not recoverable. In that case, depressing START COMPUTE will transfer control to the beginning of COMPACT. Correct the source program tape and begin again.

Initial Instruction

The compiler is stored in Tracks 0 through 101. The area above Track 101 is used for table storage. The entry location for the compiler is Location 00000, as it is most accessible. Unfortunately, this also makes the contents of that location most vulnerable to destruction by improper manual operations. If this happens, the initial instruction may be restored by the following procedure:

1. Depress MASTER RESET.
2. Select TYPEWRITER TO COMPUTER.
3. Depress ONE OPERATION.
4. Depress SET INPUT MODE.
5. Depress EXECUTE LOWER ACCUMULATOR.
6. Depress START COMPUTE. The light on the typewriter glows.
7. Type: 85F02A2EC0000000*
8. Raise EXECUTE LOWER ACCUMULATOR.
9. Raise ONE OPERATION.
10. Depress START COMPUTE.

The following procedure may be used to recover from an error halt during a blind assembly (i.e., an assembly with Sense Switches 2 and 4 depressed) if the typewriter light comes ON:

1. Note which on-line devices are selected when the error halt occurred.
2. Depress MASTER RESET.
3. Mark the character under the read head at the time of the halt.
4. Reposition the tape so that the first character of the erroneous statement is under the read head.
5. Depress READER SELECT and TYPEWRITER SELECT (off-line).
6. Depress START READ (off-line) and allow the erroneous statement to be typed.
7. Depress STOP READ (off-line).
8. Determine what corrections are necessary.
9. Position the tape so that the first character of the statement immediately following the erroneous statement is under the read head.
10. Raise READER SELECT and TYPEWRITER SELECT.
11. Select on-line all devices noted in step 1.
12. Depress START READ (on-line).
13. Type the statement that caused the error halt, making the necessary corrections. After that statement has been compiled, the typewrite light will glow, indicating that the next statement may be input.
14. Note which on-line devices are selected.

15. Depress MASTER RESET.

16. Re-select all on-line devices noted in step 14 except TYPEWRITER TO COMPUTER.

17. Depress READER TO COMPUTER.

18. Depress START READ (on-line) and compilation will continue normally.

Terminating Compilation

An END statement must be the last statement of every COMPACT-language program and subprogram. After the compiler reads the END statement followed by a carriage return, it outputs certain information which is necessary for the assembly and operation of the compiled program:

1. The instructions for the initialization of a subprogram (when applicable).

2. The array base addresses.

3. The RRS pseudo-instructions.

4. The PRE pseudo-instructions.

Items 1, 2, and 3 are punched on the ROAR-language tape and typed on the program listing; item 4 is only punched on tape.

THE COMPILED PROGRAM

Symbols

The output from the COMPACT compiler consists of a series of ROAR-language symbolic instructions, constants, and pseudo-instructions. It may be noted that many more names (symbolic locations) appear on the ROAR-language output listing than in the original COMPACT statements. There is a definite relationship between the form of such a name and its function. All names generated by the compiler are of the form X]XXX, where X is any alphabetic character and (]) the right bracket on the RPC-4480 Typewriter. The character to the left of the bracket designates the function of the name.

This character has a mnemonic relationship to the function it describes:

K]XXX -- name of a fixed-point constant

Q]XXX -- name of a floating-point constant

M]XXX -- name of a mask

X]XXX -- exit from a subroutine

W]XXX -- working storage location

L]XXX -- usually name of the location of an instruction

J]XXX -- name of a compiler-named Hollerith array.

Other initial letters signify the conversion of a COMPACT statement number to a statement name. This conversion is necessary since pure numeric names are interpreted by the ROAR assembly program as absolute addresses. The conversion is a straightforward process which changes decimal numbers to numbers base 26 and uses the 26 letters of the alphabet to represent them. Thus, statement number 5 would be converted to A]AAF; statement number 27, to A]ABB, etc. Any names in the compiled program with A], B], C], D], E], or F] as the leading characters indicate conversion from statement numbers.

In addition to individual names, the compiler must generate regional names to provide storage for arrays, format lists, etc. All regions designated by the compiler are tagged with a special character, thus leaving the alphabetic characters for use by the programmer. These region designations are as follows:

, Common region.

= Address table for subprogram parameters.

[Local array storage.

] Format and input/output address lists.

- Transfer vectors for Computed GO TO statements.

+ Parameter storage for arithmetic-type functions.

It is suggested that no regions tagged with a special character be used in programs written directly in ROAR language if there is any chance that they will be assembled together with a compiler-generated program.

All sequential storage described by the compiler will be assigned to the low end of memory, beginning in Track 0 Sector 01. The regions will be located in the sequence listed above. Thus, the common region (,) with base address 00000 is the first region, and its first word "00001" is in the absolute Location 00001. (Location 00000 is reserved for the beginning of an error print routine in the Subroutine Library. *)

Sequential storage assignment is made as follows: the common region is assigned the required number of locations extending from Location 00001 upwards; the last location of the common region becomes the base address of the next region; i. e., region(=), the address table region.

When several compiler-generated programs are assembled together, ROAR adds the region reservations for each region, and makes the region large enough to handle all the required entries for the programs. For example, if the first program required 12 locations in the (=) region and the second required 10, the total number of locations set aside for the (=) region would be 22. This is true

---

*Location 11519 contains a self-addressed halt which is executed after the error printout "TRANSFER ERROR" (at run time, not compile time). Location 00000 contains the instruction *PRD*9500*11119**which is the first print instruction for the error printout; the next is in 11119; etc.; the last Print instruction is in 11959 (*PRD*159*11519**). If the programmer does not want the computer to halt after the error printout or wants to be able to continue after such a halt, the Next-address in Location 11959 or 11519 (respectively) should be changed accordingly.

of all except the common (,) region, which is chosen to accomodate the requirements of the program in the group which needs the largest number of common region locations. To illustrate, assume that three compiled programs are being assembled together and that the first requires 14 sectors in the common (,) region, the second 10 sectors in the (,) region, and the third 22 sectors in the (,) region. When the three PRE tapes are read by ROAR, it will assign 22 sectors to the (,) region. Note, however, that of the 22 symbols which will be assigned storage in the (,) region, only 10 can actually be used by all three programs. Furthermore, in order for ROAR to assign the correct region locations to these common symbols, they must have been listed in the same sequence in the COMMON statement in each program. The order in which the other symbols were listed in the COMMON statement is immaterial. (See COMPACT PROGRAMMING MANUAL, "The COMMON Statement," Chapter 5.)

The compiler-generated programs make use of the Recirculating Track and of the Double-access Tracks, and the floating-point and function routines also use the Recirculating Track. All of Track 127 is used. Track 124 (and its associated Track 126) is used to store parameters for hand-coded function routines with more than one parameter. Sectors 63, 0, 1, 2, 3, and 7 of Track 123 (125) are used to simulate sense lights and should not be used for any other purpose if the statements SENSE LIGHT n or IF SENSE LIGHT n are used in the COMPACT-language source program. Furthermore, Sectors 32 through 63 on Track 123 (125) and all of Track 124 (126) are used for regional storage by the Input/Output subroutine in the COMPACT Subroutine Library.

Roar Pseudo-Instructions

Five pseudo-commands have been included in the assembly program, ROAR III, to allow it to process compiler-generated programs. Four of these are logically required, the fifth is merely handy. These pseudo-instructions are as follows:

1. PRE - Prepare ROAR. This pseudo-instruction is generated by the compiler and is the last item output after the program compilation is complete. It contains the name of the subprogram, if any, and the information concerning the amount of storage required by that routine for each of the six special-character regions. The PRE pseudo-instruction must be input to ROAR before the COMPACT object program.

   The information output as the PRE pseudo-instruction is in hexadecimal notation. The first word is the name of the subprogram if any. It is in 6-bit binary-coded decimal. If there is no name, the word will be all zeros. Any subprogram name introduced by a PRE pseudo-instruction is defined as a global symbol by ROAR. The second and subsequent words of the PRE pseudo-instruction are made up of two parts: the first 4 hexadecimal characters give the name of the region; the last 4 indicate (at a "q" of 30) how many memory locations are required for that region. For example, the pseudo-instruction

   PRE*229E8BAD*34DC000E*36DC003E*35DC0000*3ADC0000*3BDC0000*
   37DC01DC*

   informs ROAR that subroutine INOUT requires

   7 locations for region ,

   31 locations for region [

   0 locations for region =

0 locations for region +

0 locations for region –

238 locations for region]

2. RRS - Relocate Regional Storage. Every compiler-generated program which requires regional storage has an RRS pseudo-instruction at the end of the program. With the exception of the common region, regional storage from one program must not overlap that of another. Since the COMPACT compiler can not determine during compilation of a program whether it is to be assembled with another, the RRS pseudo-instruction assures assignment of unique sequential storage regions by informing ROAR how many locations in each region have been used in the program under consideration.

   When ROAR encounters an RRS pseudo-instruction, it relocates the base address of the specified region upward by the number of locations given in the Data-address field. Thus, when the next program is assembled which uses the same region, storage will commence from that point. For example, if two separately compiled programs were to be assembled together and each used 10 locations in the (=) region, the two PRE pseudo-instructions would so inform ROAR, and 20 locations would be reserved for this region. After assembly of the first program, this pseudo-instruction would follow:

   RRS=00010***

   causing ROAR to relocate the base address of the (=) region upward by 10 locations. Then, when the second program refers to =00001, ROAR will assign the 11th location in the region to that symbol.

3. SET - Establish Global Symbols. This pseudo-instruction is used to set up the global symbols, which are symbols common to all programs assembled as a single operating package. They are used for communication between sections of a program or between subprograms and a program. SET is not generated by the COMPACT compiler, but is required if any symbols--other than a subprogram name established by a PRE pseudo-instruction--are to be global. A SET and EQR Tape is provided to all COMPACT users. This tape consists of a SET pseudo-instruction containing the names of all the built-in and library subroutines COMPACT uses and a list of EQR pseudo-instructions designating the location of the initial instruction of each subroutine. It also provides the region reservations which are required for the Input/Output routines when the symbolic Subroutine Library is assembled. By using the SET and EQR Tape and including SET and EQR pseudo-instructions to provide global symbols for any hand-coded subroutines, the programmer provides all the global symbols required by ROAR to assemble his program properly.

   When the SET pseudo-instruction is executed, ROAR places in the Set Table all the symbols which follow it. Up to 256 symbols may be entered with a single SET pseudo-instruction. Each symbol, consisting of no more than 5 characters, is followed by a stop code. When a global symbol is encountered by ROAR for the first time during the assembly, the symbol is placed in the Symbol Table and is assigned an absolute address. Thereafter that absolute address is used by ROAR whenever it sees that symbol.

When assembling a COMPACT-compiled program, the SET pseudo-instruction must precede all program instructions.* Names defined as global in either a PRE or SET pseudo-instruction are preserved in the Set Table as long as the ROAR program is not initialized.

4. RST - Restore Symbol Table. The first instruction of every compiler-generated program is an RST pseudo-instruction. When ROAR encounters this pseudo-instruction, it will clear all non-global symbols from the Symbol Table. This does not affect the Set Table. ROAR examines the list of global symbols established by SET and PRE pseudo-instructions to determine whether any of these symbols have been assigned absolute addresses in the object program. If they have, the symbols are re-established in the Symbol Table, and their absolute addresses are retained. All other symbols are cleared from the Symbol Table. In this way two different programs which are assembled together may use the same non-global symbols to mean different things without conflict.

5. (56)(57)(56) - COMPACT-Generated Comments. This pseudo-command consists of the three non-printing characters (56)(57)(56) and instructs ROAR to ignore all characters which follow, including stop codes (*) unless the stop code is preceded by the character (56). This allows the original COMPACT-language statement to be included in the ROAR symbolic output as a comment and helps to make the final program listing more comprehensible. Each COMPACT statement is compiled as a block of instructions, the constants (if any) appearing at the bottom of the block. The block is preceded by the original COMPACT statement which caused the generation of the block of coding. The statement is preceded by the comment pseudo-instruction (56)(57)(56) and is followed by (56)*. (See "SENSE SWITCH OPTIONS.")

   Since codes 56 and 57 have no representation on the keyboard, this comment pseudo-command cannot be typed. However, it can be punched on tape (either directly by the computer or by overpunching with the typewriter) and will enter the computer like any other character. When the program containing this comment pseudo-instruction is assembled by ROAR, the typewriter output will show only two stop codes, a carriage return and the remarks provided, and finally a carriage return and the last stop code. For example, the comments shown in Figure 1 appear on the assembly listing as:

```
**

This program is to be used only as an example of

*

**

coding and is not intended for actual compilation.

*
```

FIGURE 4 — An assembled comment

Floating-Point Arithmetic

Floating-point arithmetic operations are performed by means of subroutines included in the Subroutine Library package. Entrances to these and other special routines are named by the compiler by means of the convention [XXXX where, as before, XXXX represents alphabetic characters and [is the left bracket on the typewriter. Again some mnemonic relationship exists; [FMP is the entrance to

* In previous publications it has been stated that the PRE pseudo-instruction must be the first input to ROAR and must be followed by the SET and EQR Tape. However, the order of the PRE and the SET and EQR Tapes is inconsequential except that they must both precede the symbolic program which is to be assembled.

the floating multiply routine, [FDV is floating divide, etc. The calling sequence generated by COMPACT for these subroutines varies, depending on the particular subroutine to be used. The calling sequences for all the subroutines are explained under "COMPACT SUBROUTINE LIBRARY."

The floating-point word, as stored in memory, consists of a sign, a mantissa (24 bits), and an exponent carried excess 128 (8 bits), for a total of 33 bits. This is accomplished in the following way:

1. All floating-point numbers are normalized.

2. All floating-point numbers are positive. Negative numbers are represented by a positive mantissa and a "1" in the sign bit; in other words, a detached sign convention is used. Complement numbers are not allowed as the mantissa.

3. Since bit position one of a normalized number must always contain a "1," except when a true zero is to be represented, this bit is redundant and is discarded.

4. A floating-point zero (as well as the fixed-point zero) is represented by a machine word containing all zeros; i.e., both mantissa and exponent are zero.

5. All floating-point routines insert the "dropped bit" as required, and then discard it after the designated operation.

A "floating accumulator" is simulated in memory, using the Recirculating Sectors 0, 1, and 3. It is loaded by means of the <u>Floating Bring Routine</u>, which separates a floating-point word into its three components and stores them in the three recirculating locations: exponent in Sector 00, mantissa in Sector 01, and sign in Sector 03. The dropped bit is inserted into the mantissa at this time. The entrance to the Floating Bring Routine is named [FBR.

The floating-point exponent, which consists of the last eight bits of the word and is always positive and greater than zero, represents the power of 2 by which the mantissa is to be multiplied. The convention adopted is called "excess 128," since the high-order position of the eight bits contains a "1" if the represented power of 2 is positive, and a "0" if it is negative. For this reason it is sometimes called the "reversed sign convention." Figure 5 illustrates in chart form how the exponents are represented.

10000000 = 2^0	
10000001 = 2^1	01111111 = 2^{-1}
10000010 = 2^2	01111110 = 2^{-2}
10000011 = 2^3	01111101 = 2^{-3}
10000100 = 2^4	01111100 = 2^{-4}
.	.
.	.
.	.
.	.
11111101 = 2^{125}	00000011 = 2^{-125}
11111110 = 2^{126}	00000010 = 2^{-126}
11111111 = 2^{127}	00000001 = 2^{-127}

FIGURE 5 — Range of exponents

This chart shows that the largest possible exponent is 11111111, representing 2^{127}, and the smallest permissible exponent for a non-zero number is 00000001, representing 2^{-127}. If the number to be represented is smaller than $.5 \times 2^{-127}$ (which is 1×2^{-128}), it is considered to be zero. That is, exponent underflow (2^{-128} = 00000000) generates a zero number.

In terms of decimal notation, numbers between the approximate limits of 10^{38} and 10^{-38} can be represented. The mantissa contains a sufficient number of binary bits to represent seven decimal digits of significance.

"Undefined" Functions

Two different types of subroutine calling sequences are generated by the compiler, corresponding to the "arithmetic-type" functions and the "subprogram-type" functions. Both types of subroutines may be written in COMPACT source language and in ROAR language and provision has been made for their assembly with compiler-generated programs. A few precautions must be observed.

ARITHMETIC-TYPE FUNCTIONS This type of function is called by its use in an arithmetic statement. For example, the statement Y = SAMF(X) will cause the compiler to generate the necessary instructions to bring X into the Upper Accumulator and transfer control to the routine called SAMF. To insure that it is available, the name SAMF must have been entered into the global list by means of a SET pseudo-instruction. (If SAMF is being assembled as a part of the program under consideration, this latter step is not required.)

The compiler provides the arithmetic-type functions with the actual values of the parameters, not their addresses. The first parameter listed will be in the Upper Accumulator (the exit instruction is in the Lower Accumulator and must be stored by the subroutine for later execution). If more than one parameter is required, the others will be stored in Track 126--the trailing head of the Double-access Track--in the order of their appearance in the argument list. The last parameter in the list will always be stored in absolute Location 12663, the next-to-last in 12662, etc. For example, assume a subroutine is written in ROAR language and is to be used with a COMPACT-compiled program. It is named BILLF (the name must end with an F and be 4 or 5 characters in length), and it requires four arguments. In the COMPACT source program the subroutine is called by the statement

```
Y = ANY + BILLF (ONE, TWO, THREE, FOUR)
```

The subroutine must store the exit instruction so that control may be transferred to the correct instruction in the main program when the subroutine has finished. The following ROAR instruction would accomplish this:

```
BILLF* CLL* EXIT***
```

It is desirable to use the first parameter immediately, if possible, since it is in the Upper Accumulator. Otherwise, it must be stored for future use. If the second parameter is to be used, the contents of Location 12661 must be brought to the Upper Accumulator. The third parameter will be in Location 12662, and the fourth in Location 12663. At the end of the subroutine, the instruction that was stored in EXIT must be executed.

When the subroutine is assembled, the name BILLF must be included in a SET pseudo-instruction, and the subroutine should begin with the pseudo-instruction RST to avoid conflicts with COMPACT-generated programs.

For an arithmetic function of a single argument (which is the most common form) the compiler generates the following instructions:

from Y = ANYF [QP]

will come

```
L]XYZ*  RAU*   QP*          **
     *  RAL*     *   ANYF**
     *  STU*    Y*  L]XZA**
```

Thus, if anything is to be stored in Y, it must be in the Upper Accumulato when control is transferred from the subroutine to the main program, and ı the floating accumulator if the output is a floating-point number.

SUBPROGRAM-TYPE FUNCTIONS This type of function can be hand-coded in ROAR language (or in absolute machine language) and may be used with a compiler-generated program, providing the calling sequence is understood. In this type of function call, the subroutine is provided with a base address of a list of parameter addresses (not with the actual parameter values, as in the arithmetic-type functions).

Upon entry, the Lower Accumulator contains the exit instruction, as before. The Index Register contains an address that is 1 less than the location in which the address of the first parameter is stored. Therefore, when the instruction

XRAU 00001

is executed, the Upper Accumulator contains the address of the first parameter (at a "q" of 17). XRAU 00003 would bring to the Upper the address of the third parameter, etc. These may then be handled in any suitable manner.

If the routine were called by means of a CALL statement, no value need be in the Upper Accumulator at exit. If it were called by its appearance in an arithmetic statement, then the Upper must contain the output value, as must the floating accumulator if the output is a floating-point number.

The subroutine name must be rendered global by use of the SET pseudo-instruction, and the routine should be preceded by an RST to avoid conflict.

Summary of Function-Naming Conventions

The naming conventions for these different types of functions are summarized below for easy reference.

An arithmetic-type function must have a name consisting of 4 or 5 characters, the last of which must be "F". If the value of the function (the output from the subroutine) is to be treated as fixed-point, the first character of the name must be "X". If the name begins with any other letter, the value of the function will be treated as floating-point.

The subprogram type of function may have a name consisting of 1 to 5 characters. If the name is 4 or 5 characters in length, the last character must not be "F". The value of the function will be treated as fixed-point if the first character of the name is I, J, K, L, M, or N, and will be treated as floating-point if the first character is any other letter. For example

Name	Designates
JSINF	Arithmetic Function, floating-point
XSINF	Arithmetic Function, fixed-point
XSING	Subprogram Function, floating-point
JSING	Subprogram Function, fixed-point
ERF	Subprogram Function, floating-point

When hand-coding either type of routine, the programmer should remember that no parameters may be called by name unless the names are global, and that all items with global names are available to all programs which are assembled together. For example, if there is a subprogram named OUT in the assembly, any reference to OUT can only be to the entrance of this subprogram.

ASSEMBLY PROCEDURE

Complete operating procedures for the assembly program may be found in the ROAR III manual. However, as a matter of convenience, a condensation of the procedures is given here.

Loading the Assembler

After compilation is completed, the symbolic-language tape that was produced must be assembled by ROAR to obtain a machine-language program. To load ROAR in the computer, use the same procedure as for loading COMPACT.

Roar Sense Switch Options

Various Sense Switch options control the input to and output from ROAR;

Sense Switch	Function When Depressed
1	Functions only after an error halt. Each time START COMPUTE is depressed, ROAR will read from the symbolic input tape until a stop code is sensed and list that information on the RPC-4500 Typewriter, but will not assemble it. After SENSE SWITCH 1 is raised, depressing START COMPUTE prepares ROAR for an input to allow error recovery.
2	Bypass listing of symbolic input.
4	Use High-Speed Reader for input.
8	Do not output a bootstrap.
16	List decimal output on RPC-4500 Typewriter in addition to the selected output device.
32	Bypass decimal output.

NOTE: When both SENSE SWITCH 2 and 32 are depressed, the hexadecimal program tape is the only output. This shortens the assembly time for COMPACT-generated programs.

Operation of the Assembler

The operator can select the input/output devices ROAR will use during an assembly. The selections are the first input to ROAR and are made in response to the printout I/O SELECTIONS.

If no change from the normal selections is to be made, type only a stop code. ROAR will then proceed to the next preliminary question, i.e., SUBROUTINE TAPE REGION STORAGE. Listed below are the selections ROAR will make if no changes are indicated.

1. RPC-4500 Reader (Input symbolic tape) code 64

2. RPC-4500 Punch (Output hexadecimal tape) code 97

3. RPC-4500 Typewriter (Output decimal listing) code 98
4. RPC-4500 Typewriter (Output preliminary questions) code 98
5. RPC-4500 Typewriter (Input responses to preliminary questions) code 68

Changes are made by entering the numeric input/output selection codes in thi. order:

1. The input device for entering the symbolic program.
2. The output device for the hexadecimal output.
3. The output device for the decimal listing.
4. The output device for the remaining preliminary questions.
5. The input device for answering preliminary questions.

If any device is to be changed, all preceding fields must be entered. Once the changes have been made, the devices remain selected until changed again. This may be done either by transferring to the entry point of ROAR and entering new selection codes (typing only a stop code leaves previously selected devices still selected) or by reloading ROAR.

If tabbed output was obtained from COMPACT and a typed listing is to be produced during the assembly, the tab settings should be placed from the left-hand margin in the following increments: 9, 7, 9, 9, 25, 2.

After all preliminary questions have been answered, the first inputs to ROAR will be the SET and EQR Tape for the Subroutine Library Package and the PRE pseudo-instruction output by COMPACT at the end of the symbolic program tape. If several compiled programs are to be assembled together, enter all the PRE tapes. The SET and EQR Tape for the Subroutine Library Package contains an RES to protect the package. Any desired RES, REG, etc., pseudo-instructions may be entered after the PRE and SET and EQR Tapes.

After the SET and EQR Tape and the PRE tapes have been input and the necessary memory allocations have been made, enter the symbolic program tape output by COMPACT. Availability of storage space on the drum determines how many programs may be assembled together. This may be in any order desired. Following assembly of the last program, special input/output selection codes may be entered in the input/output selection table if they are required. (See "INPUT/OUTPUT SELECTION TABLE.")

Each program should be concluded with a NIX or END pseudo-instruction. A NIX pseudo-instruction causes ROAR to print the number of memory locations used since the beginning of the assembly if this is the first NIX, or since the previous NIX if there are more than one. Then ROAR will halt. When START COMPUTE is depressed, ROAR will continue assembling as if the interruption had not occurred. This pseudo-instruction is convenient when several programs are being assembled together, since it gives the operator time to change tapes, etc. The END pseudo-instruction causes ROAR (1) to punch in the output tape a transfer instruction to a specified location and a final checksum; (2) to print the total number of locations that were assigned during the assembly; and (3) to halt. Since COMPACT cannot know the order in which the programs will be assembled, it does not generate a NIX or END pseudo-instruction. If these pseudo-instructions are used, they must be entered manually.

To save time when loading several tapes for assembly, the operator may want to produce a ROAR tape which includes the SET and EQR Tape information. To prepare such a tape, the following procedure is recommended:

1. Load ROAR.

2. Enter the SET and EQR Tape.

3. Load Hexadecimal Output 5, program J4-06.0, beginning in Track 60.

4. Punch Tracks 0 through 59.

5. Load ROAR.

6. Load Hexadecimal Output 5, beginning in Track 0.

7. Punch Tracks 60 through 122.

8. Enter Location 11654 as the Transfer Location. (The entry GOTOROAR -- Location 11321 -- cannot be used since it allows initialization of ROAR's internal tables; NOWBEGIN -- Location 6419 -- cannot be used because it does not provide a bootstrap.)

The bootstrap produced by Hexadecimal Output 5 at step 7 must be eliminated. This is easily done if two punch units are available: the second bootstrap can be output on a different device from the one used to punch the program. The same result is achieved by removing the bootstrap from the second tape and then reproducing the two remaining portions as a single tape.

The combined ROAR/SET and EQR Tape will be only slightly longer than the standard ROAR tape, require little additional load time, and eliminate the need to input the SET and EQR Tape for the first assembly after ROAR is loaded.

COMPACT SUBROUTINE LIBRARY

After assembly is completed, the object program can be stored in memory for execution. First, however, the Subroutine Library Package must also be in memory. This package consists of a group of routines which are necessary to perform floating-point arithmetic operations, decimal data input and output, trigonometric functions, and built-in functions.

The COMPACT Subroutine Library is available in two forms: a pre-assembled package and a symbolic package. The pre-assembled package consists of a single hexadecimal tape which is self-loading and stores the subroutines in Tracks 79 through 122. The symbolic package consists of three tapes: Arithmetic Subroutines, Input-Output Subroutines, and Function Subroutines.

When a Subroutine Library is being assembled, any or all subroutines in the Function package may be omitted if they are not called by the COMPACT-language source program. However, it is inadvisable to omit subroutines from the other two packages, since those subroutines may be called by COMPACT even though their names are not used in the source program. The storage requirements and calling sequences for the subroutines are explained on the following pages for programmers who wish to assemble their own subroutine library or to use these subroutines without the compiler. The subroutines in each group are listed in the order of their appearance on the symbolic tape.

Because of the similarity between the letter "O" and the digit "0", the letter O will be written with a slash, Ø, in the following discussions when it is part of a symbolic name.

Arithmetic Subroutines

The symbolic Arithmetic Subroutines Tape contains three groups of programs: the floating-point arithmetic group, the float-fix subroutines, and the repeated store subroutine.

The floating-point arithmetic subroutines are all interdependent and require 3 tracks, 51 sectors of storage. The Upper Accumulator must contain a floating-point quantity before entry to any of the routines, and each will exit to the instruction which is brought to the Lower Accumulator by the calling sequence.

Subroutine	Calling Sequence	Function
[FBR Floating Bring	* RAL* * [FBR* *	Copy the contents of the Upper into the floating accumulator.
[FAD Floating Add	* RAL* * [FAD* *	Add the contents of the Upper to that of the floating accumulator and leave the sum in both the Upper and the floating accumulator.
[FSB Floating Subtract	* RAL* * [FSB* *	Subtract the contents of the Upper from that of the floating accumulator and leave the difference in both the Upper and the floating accumulator.
[FSBI Inverse Floating Subtract	* RAL* * [FSBI* *	Subtract the contents of the floating accumulator from that of the Upper and leave the difference in both the Upper and the floating accumulator.
[FDV Floating Divide	* RAL* * [FDV* *	Divide the contents of the floating accumulator by that of the Upper and leave the quotient in both the Upper and the floating accumulator.
[FDVI Inverse Floating Divide	* RAL* * [FDVI* *	Divide the contents of the Upper by that of the floating accumulator and leave the product in both the Upper and the floating accumulator.
[FMP Floating Multiply	* RAL* * [FMP* *	Multiply the contents of the Upper by that of the floating accumulator and leave the product in both the Upper and the floating accumulator.
[CHSG Change Sign	* RAL* * [CHSG* *	Multiply the contents of the Upper by -1 and store this value in the floating accumulator.

The Normalize routine is included in the coding of the arithmetic routines. Although a calling sequence to this subroutine will not appear in a COMPACT-compiled program, it is given here as it may be useful to programmers who do hand-coding.

Subroutine	Calling Sequence	Function
[NØRM Normalize	* RAL* EXIT* ** * STL* RECRC2* ** * RAL* EXIT* ** * STL* RECRC6*]NØRM**	Normalize the numbers which is in the Upper and the floating accumulator. The normalized value is left in the Upper and the floating accumulator.

The following symbolic statement names are special entry locations to the arithmetic subroutines; they are not used by or output from the compiler and are of no special interest to the COMPACT programmer except that they appear on the SET and EQR Tape.

[A = 0

[= = =

[= = +

[MERG

[A = ØV

[DIL2

[N2FW

The float-fix subroutines require 1 track, 22 sectors of storage. Subroutine names beginning with [are composed by the compiler in the process of compilation; the other names may be used by the COMPACT programmer, e.g., M = XFIXF(B).

Subroutine	Calling Sequence	Function
FLØATF [FLØT FLØTF	* RAL* * FLØTF* * * RAL* * [FLØT* * * RAL* * FLØTF* *	Convert the fixed-point quantity contained in the Upper to a floating-point quantity and leave the result in both the Upper and the floating accumulator.
[FLUP	* RAL* * [FLUP* *	Convert the fixed-point quantity contained in the Upper to a floating-point quantity and leave the result in the Upper; do not disturb the floating accumulator.
XINTF XFIXF [FIX	* RAL* * XINTF* * * RAL* * XFIXF* * * RAL* * [FIX* *	Convert the floating-point argument in the Upper to a fixed-point integer and leave the result in the Upper. The argument is truncated to the largest integer less than or equal to itself.

The repeated store subroutine is used to set all elements of an array to the same value. It requires 57 sectors of storage.

Subroutine	Calling Sequence	Function
[RPTS Repeated Store	*RAL* K]XXX * * * *CLL* RECRC4 * * *	Where K]XXX is a constant indicating the number of locations to be filled.
	RAU K]XXZ * * *	Where K]XXZ is the quantity with which to fill them.
	LDX [YYYY * * *	Where [YYYY is the beginning location to be filled.
	RAL * [RPTS* *	Go to the repeated store subroutine and return to the location contained in the Lower Accumulator.

Input/Output Subroutines

The Input/Output control subroutine, [IØUT, is the only one of these subroutines called upon by any COMPACT object program. [IØUT, however, calls upon two other programs: the data input subroutine, [INPT, and the data output subroutine, [ØUTP. The symbols [P... and [CLEX , found on the SET and EQR Tape, are special communication symbols used by these three routines* The store requirements are as follows:

[IØUT	9 tracks	54 sectors
Region A		50 sectors
Region B		32 sectors
Region C		32 sectors
Region D	1 track	00 sectors
[INPT	4 tracks	23 sectors
[ØUTP	7 tracks	01 sectors
Region /		39 sectors
Region R		09 sectors
Region M		06 sectors

The reservations for these seven regions are on the SET and EQR Tape, separated from the other information by a length of blank tape. (They are also listed in the example under "ASSEMBLING THE SUBROUTINE LIBRARY.")

The calling sequence for [IØUT consists of the following four instructions:

* RAU* K]XXX* * * * EXC* 498* * *		Where K]XXX contains the beginning location of the I/O address list, and is negative (order = 16) only for Input Statements.
* RAU* K]XXZ* * *		If the code word K]XXZ is negative, it is the PRD select code for the desired input or output unit; if K]XXZ is positive, it is the symbolic name which will be searched for in the input/output selection table (Region C, Locations 11800 through 11831) to determine the I/O selection.
* RAL* *[IØUT* *		Go to Input/Output control subroutine, then exit to the instruction contained in the Lower Accumulator.

INPUT/OUTPUT SELECTION TABLE The Subroutine Library contains a table to which the input/output subroutine refers when the routine has been called by a READ INPUT TAPE or WRITE OUTPUT TAPE statement in the COMPACT source-language program. These two statements permit any desired input/output selection at object program time by means of the proper entries in this double-entry table. The input/output statements with set selections are

READ selects the RPC-4500 Reader for input.

PRINT selects the RPC-4500 Typewriter for output only.

PUNCH selects the RPC-4500 Punch for output.

* [P...contains the scaling factor controlled by the FORMAT symbol nP. [CLEX contains the exit instruction for both [INPT and [ØUTP; i.e., both subroutines return to [IØUT, which in turn exits to the source program.

Since these statements are fairly limited, the more general statements are used most often. They cause the input/output routine to search the selection table in Track 118 for the proper input/output selections. These table entries may be made at assembly time (input to ROAR as ALF pseudo-instructions at the end of the program and preceding END) or may be included in the Subroutine Library Package. For example, if one writes

READ INPUT TAPE BETA, FMT2, X, Y, Z

and at assembly time makes the table entries

11800*	ALF*	BETA*	*	*
11801*	00*	6800*	0*	*

then the variables X, Y, and Z will be called for on the typewriter and read according to format statement FMT2.

If one writes

WRITE ØUTPUT TAPE ØUT3, 3, A, B, C

and makes the table entries

11802*	ALF*	ØUT3*	*	*
11803*	16*	9800*	*	*
11804*	00*	10600*	*	*

then the variables A, B, and C will be output on both the typewriter and High-Speed Punch, according to Format Statement 3. Note the order "16" in Location 11803, which tells the input/output routine that another selection follows, and 11804 must contain that next selection. The order "00" in Location 11804, and in Location 11801 in the previous example, indicates that no more select codes follow. As many such entries for input and output selections as desired may be stored in the first 32 locations of Track 118. If the input/output routine completes the search of the table without finding the specified code, an error halt will occur. The correct entries in Track 118 will have to be made before the program can be executed properly.

Certain entries have been made in the input/output selection table of the Subroutine Library Package which may be used or changed, as desired. These entries occupy Locations 11800 through 11813. The code names for these entires and the devices that will be selected are listed below:

Name	Device
I68	selects the RPC-4500 Typewriter for input.
I74	selects the High-Speed Reader, forward.
I75	selects the High-Speed Reader, reverse.
Ø99	selects the RPC-4500 Punch and Typewriter for output.
Ø106	selects the High-Speed Punch for output.
I6498	selects the RPC-4500 Reader for input and the RPC-4000 Typewriter for output, and turns ON Copy Mode.

Function Subroutines

The symbolic Function Subroutines Tape contains the trigonometric functions, exponential routines, and the functions referred to in FORTRAN as "built-in" functions. All these subroutines require that the routines on the Arithmetic

Subroutine Tape be in memory at execution time. The trigonometric functions may be used without the exponential routines or built-in functions, but the converse is not true; i.e., the exponential routines [PRXX, [PRLX, [PRLI, and [PRLL require LØGF and EXPF.

The calling sequence for each of the trigonometric functions requires the argument in the Upper Accumulator and the exit instruction in the Lower. The storage requirements are as follows:

Subroutine	Type of Function	Storage Required
SINF CØSF	Sine Cosine	1 track, 38 sectors
ATANF	Arctangent	1 track, 51 sectors
TANHF	Hyperbolic Tangent	14 sectors
LØGF FLØG*	Natural Logarithm	60 sectors
EXPF	Exponential	1 track, 36 sectors
SQRTF	Square Root	56 sectors
Total Storage		6 tracks, 63 sectors

The calling sequences for the four exponential routines vary, depending on the placement of the arguments, which must be as follows:

Subroutine	Function	Storage of Argument
[PRLL	Raise a floating-point number to a floating-point power: A^B	A is in floating accumulator B is in Upper Accumulator
[PRLI	Inversely raise a floating-point number to a floating-point power: B^A	B is in Upper Accumulator A is in floating accumulator
[PRXX	Raise a fixed-point number to a fixed-point power: K^J	K is in RECRC6 J is in Upper Accumulator
[PRLX	Raise a floating-point number to a fixed-point power: A^J	A is in floating accumulator J is in Upper Accumulator

All the calling sequences must have the exit instruction in the Lower Accumulator. This group of subroutines requires 3 tracks, 03 sectors of storage.

The built-in functions comprise the last group on the Function Tape. This group requires 3 tracks, 13 sectors of memory and is assembled as a unit. The calling sequence must position the base address of the argument list in the Index Register, the first argument in the Upper Accumulator, and the

* This name is an entry to the LØGF subroutine which does not call upon FBR. It is not used by or output from the compiler, nor should it be used by the programmer. It is noted here only because it appears on the SET and EQR Tape.

exit instruction in the Lower Accumulator. The first entry in the argument list must specify the number of entries in the list, and the list must follow in sequential locations, beginning with the second argument.

Name	Type of Function	No. of Args.	Mode of Argument	Mode of Function
INTF	Truncation	1	Floating	Floating
MØDF	Remaindering	2	Floating	Floating
XMØDF			Fixed	Fixed
ABSF	Absolute value	1	Floating	Floating
XABSF			Fixed	Fixed
SIGNF	Transfer of sign	2	Floating	Floating
XSIGNF*			Fixed	Fixed
MAX1F	Choosing the largest value	≥ 2	Floating	Floating
XMAX1F*			Floating	Fixed
XMAX0F*			Fixed	Fixed
MAX0F			Fixed	Floating
MIN1F	Choosing the smallest value	≥ 2	Floating	Floating
XMIN1F*			Floating	Fixed
XMIN0F*			Fixed	Fixed
MIN0F			Fixed	Floating
DIMF	Positive difference	2	Floating	Floating
XDIMF			Fixed	Fixed

Assembling the Subroutine Library

When assembling a Subroutine Library, the following procedure is recommended:

1. Assemble the SET and EQR Tape.
2. Make available the number of locations required for the Subroutines to be assembled and establish the necessary regions. For example, to assemble the Arithmetic and Input-Output Subroutines, make the following memory allocations:

```
*AVL*      9200*   12163**
*REG*/ 12043*   12117**
*REG*R 12002*   12010**
*REG*M 12037*   12042**
*REG*A 11900*   11949**
*REG*B 12332*   12363**
*REG*C 11800*   11831**
*REG*D 12600*   12663**
```

* These six-character symbols are acceptable to COMPACT but not to ROAR; therefore, the compiler outputs the following equivalent symbols, respectively: XSGNF, XMX1F, XMX0F, XMN1F, and XMN0F.

3. Make the necessary memory reservations for the required subroutines. In the example stated above, use

   *RES*0* 9163**

4. Assemble the subroutines.

   NOTE: It is recommended that the Input/Output Subroutines be assembled first to insure the best optimization for these critical subroutines.

SUMMARY OF COMPACT STATEMENTS

This appendix summarizes the types of statements which are available in the COMPACT language. They include the (1) arithmetic, (2) control, (3) specification, (4) input and output, and (5) subprogram link and termination statements. Each statement type is represented by typical examples.

Arithmetic Statement

SUM = EXPR

Evaluate the expression EXPR, place the result in location SUM, and go to the next executable statement. EXPR may be in fixed-point, floating-point, or mixed mode and may be compounded; that is, contain other replacement operators. If the location in which the result is to be stored has a fixed-point name, EXPR is stored as a fixed-point quantity, regardless of the mode of EXPR, and vice versa.

Example
A = S + 3.648
J = C − JL + 2*D
B[J, 3, LL]= SINF[GØL]
B[J, 2*K, LL/3] = KMAX = SINF [GØL/4]

Unconditional GO TO

GO TO STAT

Go to, (transfer control to), the statement whose name or number is STAT.

Example
GØ TØ 45
GØ TØ START

Assigned GO TO

GO TO K, [$STAT_1$, $STAT_2$, ... $STAT_n$]

Go to "K", where "K" is some non-subscripted variable from a previously executed ASSIGN statement and the list ($STAT_i$) is "n" statement names or numbers, any of which may be assigned to "K". Either the comma following "K" or the list, or both, may be omitted. Control passes to that statement whose name or number is the last value assigned to "K" by an ASSIGN statement.

Example
GØ TØ K, [45, 3, 15, 90]
GØ TØ K [13, ALPHA, BETA]
GØ TØ K

Assign

ASSIGN STAT TO K

Assign the statement name or number STAT to "K", where "K" is some non-subscripted variable name. If STAT is a statement name, it must be delimited by brackets or commas. Note that values assigned through ASSIGN statements may be used only in conjunction with an ASSIGNED GO TO statement; that is, the statements ASSIGN 2 to K and K = 2 are not equivalent.

Example
ASSIGN 5 TØ JN
ASSIGN [JUMP] TØ KX
ASSIGN, TØP, TØ JIL

Computed GO TO

GO TO [$STAT_1$, $STAT_2$, ... $STAT_n$], EXPR

Evaluate the expression EXPR and transfer control to $STAT_1$ if the value of EXPR is 1, to $STAT_2$ if the value of EXPR is 2, to $STAT_3$ if the value of EXPR is 3, etc. The $STAT_n$ are statement names or numbers, and the comma following the list is optional. The value of EXPR is converted to fixed-point if EXPR is not a fixed-point expression.

Example
GØ TØ [10, 20, 3, 30], M
GØ TØ [9, ALPHA, 3, BETA] K + M/4

If

IF [EXPR] $STAT_1$, $STAT_2$, $STAT_3$

Evaluate the expression EXPR and if its value is less than zero, transfer control to $STAT_1$; if its value is exactly zero, transfer control to $STAT_2$; or if its value is greater than zero, transfer control to $STAT_3$. EXPR may be any expression, and the $STAT_i$ are statement names or numbers. The value of EXPR is converted to fixed-point if EXPR is not a fixed-point expression.

Example
IF [A [J, K] -B]4, 10, 12
IF [J–K = K + 1] ALPHA, 3, BETA

Sense Light

SENSE LIGHT I (for I = 1, 2, 3, 4, or 0)

Turn ON sense light "I", where "I" is 1, 2, 3, or 4; if "I" is 0, turn all four sense lights OFF.

Example
SENSE LIGHT 3
SENSE LIGHT 0

If Sense Light

IF [SENSE LIGHT I] $STAT_1$, $STAT_2$ (for I = 1, 2, 3, or 4)

If sense light "I" is ON, turn it OFF and transfer control to statement $STAT_1$;

if sense light "I" is OFF, transfer control to statement $STAT_2$. $STAT_1$ and $STAT_2$ may be either statement names or numbers. $STAT_2$ may be omitted, in which case control is transferred to the next executable statement when sense light "I" is OFF.

The brackets around "SENSE LIGHT I" may be omitted. However if they are, a delimiter (", ") must precede $STAT_1$.

Example
IF [SENSE LIGHT 2] 46, 44
IF SENSE LIGHT 3, ALPHA, 5
IF SENSE LIGHT 4, ALPHA

If Sense Switch

IF [SENSE SWITCH I] $STAT_1$, $STAT_2$ (for I = 1, 2, 3, 4, 5, or 6)

If sense switch "I" is depressed, transfer control to statement $STAT_1$, otherwise transfer control to statement $STAT_2$. The $STAT_1$ and $STAT_2$ may be either statement names or numbers. $STAT_2$ may be omitted, in which case control is transferred to the next executable statement when sense switch "I" is not depressed.

The brackets around "SENSE SWITCH I" may be omitted. However if they are, a delimiter (,) must precede $STAT_1$.

COMPACT sense switches correspond to the RPC-4000 console sense switches as follows:

Sense switch 1	=	console Sense Switch 1
Sense switch 2	=	console Sense Switch 2
Sense switch 3	=	console Sense Switch 4
Sense switch 4	=	console Sense Switch 8
Sense switch 5	=	console Sense Switch 16
Sense switch 6	=	console Sense Switch 32

Example
IF [SENSE SWITCH 5] 12, 11
IF SENSE SWITCH 4, BETA, 9
IF SENSE SWITCH 4, BETA

If Accumulator Overflow
If Quotient Overflow
If Divide Check

IF OVERFLOW

The IF ACCUMULATOR OVERFLOW, IF QUOTIENT OVERFLOW and IF DIVIDE CHECK statements function like the IF OVERFLOW statement and serve to interrogate the RPC-4000 Branch Control switch which is turned ON whenever overflow occurs.

IF OVERFLOW, $STAT_1$, $STAT_2$

If the overflow switch (Branch Control Switch) is ON, transfer control to statement $STAT_1$ and turn the overflow switch OFF, otherwise transfer control to statement $STAT_2$. The $STAT_1$ and $STAT_2$ may be either statement names or numbers. $STAT_2$ may be omitted, in which case control is transferred to the next executable statement when the overflow switch is OFF. The comma following "OVERFLOW" is also optional.

Note that the IF OVERFLOW statements should immediately follow the arithmetic statement which is being interrogated for overflow, because the Branch Control switch is also affected by IF statements and the loop tests in DO statements.

Example
IF ACCUMULATØR ØVERFLØW, 16, 17
IF QUØTIENT ØVERFLØW, 50, 51
IF DIVIDE CHECK, 3, 13
IF ØVERFLØW, 3, 8
IF ACCUMULATØR ØVERFLØW, FIND
IF DIVIDE CHECK REDØ, CØNT

Pause

PAUSE n (where n = 0 to 8 hexadecimal digits)

Stop computing and display "n" (right justified) in the Lower Accumulator on the computer console. Then continue on to the next executable statement when the operator depresses the START COMPUTE switch.

Example
PAUSE FFFFF
PAUSE

Stop

STOP n (where n = 0 to 8 hexadecimal digits)

Stop computing and display "n" (right justified) in the Lower Accumulator on the computer console. This statement causes the object program to stop on a self-addressed Halt instruction, and the machine will not continue computing when the START COMPUTE switch is depressed.

Example
STØP 80000000
STØP

Do

DO $STAT_1$, K = EXPR1, N, INCR

Execute repeatedly the statements which follow, up to and including statement $STAT_1$. The first time, execute the statements with K = EXPR1. Before each succeeding execution increase "K" by the value INCR and if "K" does not exceed the value of N, repeat the statements up to statement $STAT_1$. If "K" exceeds the value of N, transfer control to the first executable statement following $STAT_1$. $STAT_1$ may be any statement name or number and "K", any non-subscripted fixed-point variable name.

INCR may be omitted, in which case its value will be taken to be 1.

The expressions (EXPR1, N, and INCR) are all evaluated before beginning the first loop. If they are floating-point or mixed mode expressions, they are truncated to fixed-point before being used in the DO-loop testing.

The first instruction in the range of a DO, (instruction immediately following a DO statement), must be executable, that is, not a FORMAT statement. Statement $STAT_1$ must not be any kind of a transfer statement, like IF or GO TO. This restriction may be met by making $STAT_1$ a CONTINUE statement.

Example
DØ 12 IGH = 1, 8, 2
DØ K = 10, 20
DØ END, L = AJ + K, [3*K] + 3, M = K/4
DØ LØØP, IJ = M/2, K*M + 2

Continue

CONTINUE

Continue to the next executable statement. The CONTINUE statement is a dummy, usually the last statement in the range of a DO. It provides a transfer address for IF and GO TO statements which are intended to begin another repetition of the DO range.

Example	
5	DØ 9 L = 1, 35
	IF [ARGX - ARG [L]]9, 51, 9
9	CØNTINUE

End

END

Terminate compilation and output all program array and region storage data. The END statement must be the last statement in every source program, including FUNCTION subprograms and SUBROUTINE subprograms. That is to say, END is physically the last statement in any source program.

Example
END

Call

CALL MATAD [EXPR1, EXPR2, EXPR3, ... EXPRn]

Evaluate the "n" expressions, if given; call on the SUBROUTINE named MATAD; execute the SUBROUTINE using the "n" parameter values; then return to the next executable statement. MATAD must be a SUBROUTINE name, one to five characters in length and not ending in the letter "F" if more than three characters long. The EXPR's may contain any combination of fixed- and floating-point variables, constants, or expressions; but they must correspond in mode and in sequence to the "n" parameters defined by SUBROUTINE MATAD.

Example
CALL GEØGR
CALL MATPY [L, A1, GT, KM, ARRAY]
CALL INTRP [K-3, 6.78, F/T + 5, ARI]

Subroutine

SUBROUTINE MATMT [S, T, ... U_n]

The SUBROUTINE statement causes compilation of the program which follows as a SUBROUTINE subprogram. The argument list, the S, T, ... U_n, are all non-subscripted variable names and may include array names without subscripts. The list must contain the names of all input and output variables which the SUBROUTINE and the calling program have in common. If there are no common variables needed (that is, if the SUBROUTINE contains its own Input and Output statements), the argument list may be omitted. SUBROUTINE subprograms are entered from the calling program through CALL statements. The name, MATMT, can be one to five characters long but must not end in F if it is more than three characters long.

Example
SUBRØUTINE GEØGR
SUBRØUTINE MATPY [K, C, D, J, A]
SUBRØUTINE INTRP [IF, F, BAL, A]

Function

FUNCTION ERF [Z, TR, E, ... $_n$]

The FUNCTION statement causes compilation of the program which follows as a FUNCTION subprogram. The argument list, Z, TR, ... $_n$, are all non-subscripted variable names and may include array names without their subscripts. The list contains the names of all input variables which the FUNCTION and the main program have in common, and there must be at least one argument in the list. A FUNCTION subprogram exits to the main program with a single function value in the accumulator; for this reason, the FUNCTION name, ERF, must appear with its argument list in some arithmetic expression within the calling program. A FUNCTION name is any name, from one to five characters in length, the first letter of which is not "I", "J", "K", "L", or "M" unless it is a fixed-point function, and the last letter of which is not "F" unless it is fewer than four characters long.

Example
FUNCTIØN TAN [X]
FUNCTIØN FMEAN [A, B, C, D]
FUNCTIØN IF [K, KK, KKK]

Return

RETURN

The RETURN statement terminates any SUBROUTINE subprogram or FUNCTION subprogram and returns control to the calling program. Therefore, a RETURN statement must be the last executed statement in a subprogram. It need not be physically the last statement of a subprogram (the last statement physically is END); it can be at any terminal point reached by a path of control, and a single subprogram may have any number of RETURN statements.

Read

READ n, list

On the RPC-4000, read the following variables via the prime paper tape reader. See PRINT statement, for additional information.

Punch

PUNCH n, list

On the RPC-4000, punch the following variables via the prime paper tape punch. See PRINT statement, for additional information.

Print

PRINT n, list

On the RPC-4000, print the following variables via the prime typewriter. The following information applies equally to all READ, PUNCH, and PRINT statements.

Read, punch, or print the variables given in the list, using the Input or Output FORMAT statement named or numbered n. Then continue to the next executable statement. FORMAT statement n gives pertinent information on the field widths and conversion types of the listed variables and also all alphanumeric heading, output page or tape, and input tape format data. The list may contain any combination of fixed- and floating-point variable names and array names, either subscripted or non-subscripted.

Example
READ 100, 1, K, [[ARRAY [J, L] , J = 1,11, 2], L = 1, K]
PRINT 92, ARRAY
PUNCH FRMT1, A, B, ARRAY [4, K]

Read Input Tape

READ INPUT TAPE N, STAT, list

Write Output Tape

WRITE OUTPUT TAPE N, STAT, list

Input or output the variables in list via input or output unit N using FORMAT statement STAT. N may be any name or number fewer than six characters in length. The computer operator at each installation must enter the name or number, N, along with its appropriate input/output selection code or codes, into a special input/output selection table within the Subroutine Library Package.

Example
READ INPUT TAPE 6, 14, A, B, ARRAY
WRITE ØUTPUT TAPE HPNCH, FMT1, A, B
WRITE ØUTPUT TAPE 5, FMT2, K, ARRAY

Format

$STAT_n$ FORMAT $[f_1, f_2, f_3, \ldots f_n]$

FORMAT statements are used in conjunction with the five input/output statements and describe the information format to be used with the transmitted data. FORMAT statements also specify the type of conversion to be performed between machine-language data representation and the external input or output unit. FORMAT statements may be placed anywhere in a source program, except as the first statement in the range of a DO, since they are not executed but merely supply information to the object program. The statement name or number, $STAT_n$, corresponds to the FORMAT statement name or number in the input or output statement which refers to them.

Following is a brief list of conversion specifications and format codes, f's. The FORMAT f-list consists of the following format specifications, each separated by commas and all enclosed in brackets.

Specification Code	Definition
nIw	n integer conversions of field-width w
nFw.d	n fixed-point decimal conversions of field-width w and d fractional decimal places
nEw.d	n floating-point decimal conversions of field-width w and d fractional decimal places
nOw	n octal characters of field width w
nAw	n alphanumeric characters of field width w
nP	scale the following E and F conversions so that: (external number = 10^n x internal number)
nX	the next n characters of input or output are spaces
nH	the next n format-list characters are a Hollerith field
/	new line; may also replace the comma as a format-list specification delimiter

Dimension

DIMENSION sv_1, sv_2, sv_3, sv_4, . . .

The DIMENSION statement is a non-executable statement which provides the necessary information to allocate storage in the object program for arrays (subscripted variables). A DIMENSION statement must appear somewhere before the first executable statement or Arithmetic Subroutine statement in every source program and subprogram which contains subscripted variables (the sv's above). The DIMENSION statement must contain the name of every subscripted variable or array along with the maximum dimension of each subscript. There is no limit to the number of dimensions an array may have; however, these arrays must never exceed the specified dimensions.

Example
DIMENSIØN AR1 [3, 5] AR2 [4, 4, 4], K [10]
DIMENSIØN KRAY [3, 3, 3, 3]

Equivalence

EQUIVALENCE [v_1, v_2, v_3, . . .] , [v_a, v_b, v_c. . .] , . . .

The EQUIVALENCE statement is a non-executable statement which allows the programmer to control data storage allocations in the object program. If the program permits, the programmer may assign two or more variables or array elements to share the same memory location. Each series of variable names to be assigned a common location (equivalence group) must be enclosed in brackets and separated by a comma from the next equivalence group. See the general form below. COMPACT does not allow the same variable or array name to appear more than once in an EQUIVALENCE statement in any one program or subprogram. EQUIVALENCE statements must appear somewhere before the first executable statement or Arithmetic Subroutine statement in a program or subprogram.

Each of the group elements, v's, has the general form A(q). In general, A(q) is defined for q>0 to mean the (q-1)th location after A or after the beginning of array A; that is, the qth location in the array. If q is not specified, it is taken to mean the location named A or (if A is dimensioned) the first location in array A.

Example
EQUIVALENCE [A[5], B[1], ANY], [SOME, DRAY [10]]
EQUIVALENCE [A[1], C[9], KRAY], [A[45], AFIN]

Common

COMMON v_1, v_2, v_3, ...

The COMMON statement is a non-executable statement which enables the programmer to share data storage between programs and subprograms in much the same way as the EQUIVALENCE statement permits data storage sharing within a single program or subprogram. The variables (v's above), including non-subscripted array names appearing in COMMON statements, are assigned storage locations in common storage, a storage region completely separate from the program instructions, constants and other data. Common storage is assigned separately for each program or subprogram assembled together in memory. In this way, the locations which contain variables and arrays from one program, may be used to contain other variables and arrays from another program. Proper use of COMMON can result in a large saving of storage space.

In COMPACT, a COMMON statement must appear before the first executable statement or Arithmetic Statement subprogram in a program.

GENERAL PRECISION

COMMERCIAL COMPUTER DIVISION

PRINTED IN U.S.A.