FAST manual

GRI Computer Corporation

320 NEEDHAM STREET, NEWTON, MASSACHUSETTS 02164

TEL: (617) 969-0800

Price $1.75

GRI-909

FAST MANUAL

GRI Computer Corporation, 320 Needham Street, Newton, Massachusetts 02164

71-44-002-A
1170-700

TABLE OF CONTENTS

CHAPTER ONE

THE FAST ASSEMBLER

The FAST acronym stands for Functional Assembly System Technique. It is an aid in the preparation of programs for the GRI-909 computers.

1.1 Introduction:

An assembler is an indispensible aid to the process of preparing programs to run on a computer. All of these programs to be run on a computer are called object programs. They are often prepared in a punched paper tape format where the holes or absence of holes literally represent the 1's and 0's of the machine language instructions. A program called a loader is then used to load this object tape into the computer's memory where it can be executed by the computer.

An assembler literally does what its name implies; it assembles object programs from symbols and expressions that have one for one correspondence to binary portions of an instruction word, or in some cases, have a binary word equivalent. The use of symbolic statements or expressions permits the programmer to relate the problem and its parameters more closely to the language he must write in order to produce an object tape. This simplifies the task of programming without degrading program efficiency. It differs from a compiler language in that each compiler instruction might generate many equivalent machine language instructions, decreasing program efficiency and making it difficult to relate to time based operations.

The assembler, then, is a translator of source material (called the SOURCE PROGRAM) into an OBJECT PROGRAM which can then be loaded into the computer's memory and run. The assembler also provides other useful functions for the programmer. It furnishes a listing which can be used to debug the object program once it is stored in memory. This listing is also used to correlate the object program as it resides in memory to the source program as it was originially prepared. The preparation of source programs is relegated to a program called the SOURCE TEXT EDITOR (%STE), Manual M44-3.

FAST is a functional assembly technique that is very much related to the basic organization of the GRI-909 computer. It is a symbolic assembly language that deals at a high functional level with the components of the system and the computer. It also provides all of the symbolic aids that are normally provided by an assembly language.

The basic organizational concept of the GRI-909 treats all components in a computer system as equal members of a data system that either produce or receive data. Furthermore, the control structure of the GRI-909 permits the basic instruction; move data from Source A to Destination B to be programmed by the user where the source and destination of data are any two system devices. When these system devices are internal parts of a computer, they also perform processing functions on the data they receive. Often these processing functions are independent parallel processing functions that are going on while other devices are in operation. Thus, an arithmetic unit independently performs its function on data received. It makes a result, which is some function of the data received, available to the system for transfer to some other system device or function.

The GRI-909 architecture also provides a bus modifier which permits simple modifications to the data while it is being transmitted from source to destination. These modifications are incrementing the data, shifting and complementing (the latter being performed prior to incrementing or shifting so that combined operations are permissible). Thus, the basic format of the instruction now becomes: SOURCE A (mod) to DESTINATION B.

The functional components of the system are assigned mnemonics, and the user describes the data transfers that take place between these components in concise, highly functional statements such as AX TO TTO. Memory locations may be referenced as symbolic entities in high level statements such as COUNT P1. Other statements are provided for control purposes in the assembler itself. Statements such as *END denote the end of a program. Statements like this are directives to the assembler itself and do not generate binary information for the object program.

The assembler interprets each such statement and either generates the appropriate binary object information or performs the implied assembler functions.

The programs are generally prepared on punched paper tape, although other media such as punched cards or magnetic tape may be used.

CHAPTER TWO

FAST LANGUAGE ELEMENTS

This chapter describes the various elements that are used to form FAST language source program statements. When learning the FAST language, a cursory reading of this chapter will suffice -- the details contained herein may be referred to from time to time as the user gains experience with FAST programming. Throughout this manual, the following notational conventions will be employed when presenting general forms of language elements:

[] Brackets - used to contain an optional item. The language element may be written with or without the item -- generally, the meaning of the statement containing that language element is changed when such an item is omitted.

{} Braces - used to contain alternate items. These items will be arranged vertically within the braces -- the language element must include one, and only one, of the alternate items.

... Ellipses - used to denote permissible repetition of the immediately preceding language element.

When braces are enclosed within brackets, then either the entire form in brackets is omitted or the form is included with the appropriate alternate item selected. Some examples of this notational usage are:

Form	May be Written as
A [TO] B	A B
	A TO B
A $\left\{\begin{matrix}B\\C\end{matrix}\right\}$ D	A B D
	A C D
A $\left[\left\{\begin{matrix}B\\C\end{matrix}\right\}\right]$ D	A D
	A B D
	A C D
A [,A] ...	A
	A,A
	A,A,A
	etc.
A [B [C]]	A
	A B
	A B C

2.1 Character Set:

The FAST assembler processes source program statements composed of 8-bit ASCII characters and recognizes two distinct categories of characters: general usage characters and reserved characters.

General usage characters are used to form symbols (2.2) and simple numeric constants (2.5):

Character	External	Internal
Alphabetics	A through Z	301 through 332
Numerics	0 through 9	260 through 271
Dollar Sign	$	244
Percent Sign	%	245
At Sign	@	300

Reserved characters are used to impart special meanings to the assembler, to separate or delimit certain language elements, or to enable error-recovery within source lines:

Character	External	Internal	
Colon	:	272	Delimits a label (2.4)
Semi-colon	;	273	Delimits a comment (2.7)
Equals	=	275	Used to define parameters (4.1)
Plus	+	253	Denotes addition in expressions (2.6)
Minus	-	255	Denotes subtraction in expressions (2.6)
Comma	,	254	Separates data definitions (4.5)
Ampersand	&	246	Denotes logical "AND" in expressions (2.6)
Exclamation	!	241	Denotes logical "OR" in expressions (2.6)
Asterisk	*	252	Precedes certain assembler instructions (4.2-4.4)
Period	.	256	Represents the assembler's LOCATION COUNTER
Quote	'	247	Delimits text definition (4.5.1)
Pound	#	243	Used to define new system symbols (5)
Carriage-return	(CR)	215	Delimits source statements (2.8)
Line-feed	(LF)	212	Optionally follows CR at end of source statement (2.8)
Back-arrow	←	337	(1)
Rubout	(RO)	377	(1)
Block-mark	(BL)	375	(1)
Space		240	Used to separate language elements

(1) Back-arrow and rubout are used for error-recovery within a source line. Block-mark serves to delimit a source text block. See section 1.2, "SYSTEM CONVENTIONS" in the GRI-909 Manual M44-3, "SOURCE TEXT EDITOR".

NOTE: Although the FAST assembler recognizes only no-parity 8-bit ASCII characters internally, the characters punched onto a source tape may be 8-bit, even-parity, odd-parity, or no-parity codes since the text input routine logically OR's the high-order bit into each character read.

2.2 Symbols:

A FAST language symbol consists of one or more general usage characters (2.1), the first of which must not be numeric. Since only the first five characters are stored in the assembler's symbol table, symbols of greater length must be unique in the first five characters -- the assembler ignores all characters in a symbol after the fifth.

The following character strings could validly be used as symbols:

START

LOOP

N23@

PARA11

PARA21

These character strings are invalid as symbols for the reason given:

8ABC	First character numeric
GO*	Reserved character, *
AB LE	Embedded blank
PARAM1 PARAM2	Not unique in the first five characters

In FAST, as in any symbolic programming language, a symbol must ultimately represent some numeric value. Many commonly-used symbols are built into the assembler's standard symbol table (APPENDIX C). These, of course, already have numeric values associated with them. Other symbols are defined by the user in his source program -- these have values assigned to them during the assembly process. User-defined symbols and their associated values are added to the assembler's symbol table -- they reside there only during the assembly of a specific source program. As the assembler translates source statements into their binary or machine language equivalents, it merely replaces each symbol it encounters with its specific numeric value.

It is possible to use a symbol in a source program in such a fashion that the assembler cannot associate a numeric value with it. Symbols being defined by the user must appear in a symbol-defining context at least once in the source program. These symbol-defining contexts are described in detail in sections 2.4, 4.1, and 5 of this manual. If, by the end of the second pass of the assembly, there are still undefined symbols (i.e. symbols with no numeric values assigned), the assembler will flag each statement containing such a symbol with the error code "U" as it generates the assembly listing.

Pre-defined symbols in the assembler's symbol table have mnemonic value -- for instance, AO represents the arithmetic operator, HSP represents the high speed punch, LTZ represents the data condition of less than zero, and CLIF represents a pulse code to clear an input flag. In order to enhance the overall utility of assembly listings, the user should attempt to compose his symbols with mnemonic value as well.

Symbols in the FAST language are of the following categories:

Key Words	(2.3)
Labels	(2.4)
Parameters	(4.1)
System Symbols	(5)

Key words are pre-defined in the assembler's symbol table. These symbols comprise the framework of specific FAST language statements and enable the assembler to distinguish the various types of machine and assembler instructions.

The standard symbol table does not contain either labels or parameters -- these are always defined by the user in his program. A label is prefixed to a FAST statement so that the associated instruction or data item may be referenced symbolically from other points in the same program. A parameter is used to represent constants that may be referred to many times in a program -- a constant is often given a symbolic equivalent because it is more meaningful to the user than are octal or decimal numbers.

System symbols represent devices (registers), output pulse codes, status test codes, path codes for the GRI-909 Bus Modifier, data test codes, and pseudo-codes (5). The standard symbol table contains the more commonly used system symbols. Other system symbols may be defined by the user -- when doing so, the user is cautioned not to employ any symbols already in the standard table unless it is his specific intent to alter their standard meaning.

2.3 Key Words:

Key words are reserved character strings that have particular meanings to the FAST assembler. No key word may be redefined by the user nor may it be used in a context other than those shown in the general forms of FAST language instructions. Included in the FAST language key words are all the reserved characters -- see section 2.1 for a tabulation of these characters and their functions. The other FAST key words are as follows:

Symbol	Function
END	Program terminator (4.4)
EOT	Program terminator (4.4)
OCT	Radix Control (4.2)
DEC	Radix Control (4.2)
SKIP	Indicates sense function instruction (3.5)
IF	Indicates data test instruction (3.3)
TO	Used in data transmission (3.1, 3.2), data test (3.3) and function output instructions (3.4)
GO	Combined with TO to indicate transfer of program control in data test instructions (3.3)
C	Indicates taking of one's complement in register-to-register data transmission instructions (3.1)
I	Indicates immediate addressing mode in memory reference data transmission instructions (3.2)

Symbol	Function
D	Indicates deferred addressing mode in memory reference data transmission instructions (3.2)
ID	Indicates immediate-deferred addressing mode in memory reference data transmission instructions (3.2)

2.4 Labels:

A FAST statement may be labeled or tagged by the use of the form

symbol :

(a symbol followed by the reserved character :) as the first element of the statement. The assembler assigns the current value of its LOCATION COUNTER (4.3) to this symbolic label -- this will be the memory address of the first word assembled after the label is encountered. If, for instance, a label is prefixed to a statement which assembles as a two-word instruction, the label's value will be the memory address of the first word of the instruction. A label may also be associated with a statement which assembles as data.

A label, then, is a symbolic equivalent of the address of a specific machine instruction or data word in the user's program. The label may be used to symbolically reference the instruction or data word from other points in the same program. Therefore, each label must be unique. An attempt to use the same symbol more than once as a label will be flagged with the error code M (multiply-defined symbol) on the assembly listing.

When the assembler encounters a symbolic label in a context other than the one which defines it (above), the assigned numeric value is substituted for the symbol. Given the labeled statements:

```
                LOOP:  AO TO HSP     (1-word instruction)
and             TABLE: 1,10,100,1000 (4-word table)
then
                GO TO LOOP     transfers program control to the
                               instruction at LOOP,
and             TABLE TO AX    loads the first word of TABLE
                               (the value 1) into the register AX.
```

Addresses of machine words near these labeled may be accessed by forming a symbolic relative address in a FAST statement. Such an address expression (2.6) consists of a label plus or minus a constant (2.5). For instance,

```
                        GO TO LOOP+1    transfers control to the next
                                        instruction after the one at LOOP,
and                     TABLE+2 TO AX   loads the third word of TABLE
                                        (the value 100) into the register AX.
```

These expressions result in addresses of words relative to labeled instructions or data. Note that if the instruction at LOOP were two words long, the expression pointing to the next instruction would be LOOP+2.

2.5 Constants:

A simple constant is represented by one or more successive numeric characters. The assembler converts the character string into its equivalent binary value -- the digits are interpreted according to the setting of the assembler's RADIX (4.2). The user may write constants as either octal or decimal numbers provided he has specified the appropriate radix. The range of a constant, so as not to arithmetically overflow out of the fifteen magnitude bits of a signed machine word, is:

0 to ±32767 decimal

or 0 to ±77777 octal

A constant may be preceded with a minus sign (e.g. -20) and the assembler will form the two's complement value of the number (e.g. 177760). If the assembler's RADIX is set to octal, any single occurrence of the character 8 or 9 in a constant will be flagged with the error code D on the assembly listing and converted to octal. Multiple occurrences of 8 or 9 in a constant will result in an incorrect conversion to octal.

2.6 Expressions:

Compound addresses or data values may be formed by combining simple values in FAST language expressions. An expression consists of a numeric operand, or a series or operands separated by arithmetic and/or logical operators. The first operand in an expression may be preceded by a leading sign (arithmetic operator). Any given operand may be one of the following:

Label	(2.4)
Parameter	(4.1)
Constant	(2.5)
	represents the current value of the assembler's LOCATION COUNTER.

The permissible operators are:

+	denoting addition
-	denoting subtraction
&	denoting logical AND
!	denoting logical OR

A general expression, e, is assembled into a 16-bit value. The resultant value may be positive or negative -- addresses, of course, must be positive and result in a valid machine address. The general form of an expression e is:

$$\left[\begin{Bmatrix}+\\-\end{Bmatrix}\right]\quad\begin{Bmatrix}\text{Label}\\\text{Parameter}\\\text{Constant}\\.\end{Bmatrix}\left[\begin{Bmatrix}+\\-\\\&\\!\end{Bmatrix}\begin{Bmatrix}\text{Label}\\\text{Parameter}\\\text{Constant}\\.\end{Bmatrix}\right]\ldots$$

An expression is evaluated by the assembler in a single left-to-right scan: no priorities are assigned to the operators. Some examples are:

```
15
-237
LOOP-25
A + B - C
. + 4
X1 & X2 ! X3
```

2.7 <u>Comments</u>:

Comments are used to augment the source program with documentation meaningful to the user. Such documentation often explains the use of a program or subroutine, describes the functions performed by a sequence of instructions, gives the reasons for various specific steps or statements, etc. A comment may be appended to a FAST statement by prefacing the comment with the reserved character semi-colon (;). Also, an entire statement may consist of only such a comment. When the assembler encounters the semi-colon, it ignores the rest of the statement until it reaches the carriage-return terminator (2.8). Comments are reproduced on the assembly listing -- only as much of a comment as will fit on an assembly listing line will be printed, however. Any printable ASCII character may be included in the body of a comment. Examples are:

```
;THIS ROUTINE PACKS CHARACTERS
IF AO LTZ GO TO ERR ;LIMIT EXCEEDED?
```

2.8 Statements:

A source program statement (line) is a meaningful arrangement of FAST language elements and is terminated by the reserved character carriage-return, which itself may optionally be followed by a line-feed character. This latter character is ignored by the assembler -- it is usually included in a source tape so that it might be listed by an off-line device which requires the line-feed to advance the paper. A FAST statement may contain no more than 80 characters including spaces (blanks). The assembler will ignore all characters beyond the 80th.

The general forms for machine instructions and assembler instructions are presented in chapters 3 through 5 of this manual. Other than the rules given for specific instructions, there are no formatting requirements imposed upon a source statement. The major elements of a free-form source statement are label, instruction and comment -- the assembler isolates these elements and arranges them in columns on the assembly listing.

The most basic elements, symbols and constants, must be separated or delimited from each other. Since these basic elements consist solely of general usage characters (2.1), an expression such as

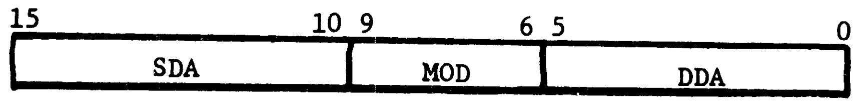
```
V1+V2&V3!25
```

is easily understood by the assembler since the basic elements are separated by reserved characters. Therefore, the rule to be observed when preparing source statements is:

> When any two successive basic elements are not separated by a non-blank reserved character, then they must be separated by at least one blank (space).

CHAPTER THREE
MACHINE INSTRUCTIONS

The bulk of the FAST language statements written by a user assemble into GRI-909 machine instructions. A sequence of machine instructions, and their associated data, constitutes a program to perform a specific task on some GRI-909 machine configuration. All basic machine instructions are described by a single internal format:



where SDA is the source device address,

MOD contains modifier, addressing mode, and function information, and

DDA is the destination device address.

The execution of any machine instruction causes, in effect, the transmission of information in the form of data or control signals <u>from</u> the source device specified by SDA <u>to</u> the destination device specified by DDA. The qualities of the transmission and/or the end result of the instruction is influenced by the specification of MOD. GRI-909 complete machine instructions are either one or two words in length. A machine instruction consists of either 1) a basic instruction in the above format or 2) a basic instruction followed by a word containing a memory address or data for the instruction.

Once an assembled program has been loaded and started, the GRI-909 control logic normally fetches and executes machine instructions from sequential locations. The computer's sequence counter (SC) contains the memory address of the next instruction to be executed. As one or two-word machine instructions are executed sequentially, the SC is automatically updated by the GRI-909 control logic. During such normal flow of program control, the SC is analagous to the assembler's LOCATION COUNTER (4.3). Some instructions, however, cause this normal program flow to be altered. The sense function or SKIP instruction causes, under certain conditions, a skip over the _next two memory locations_. The data testing or GO TO instruction can cause an absolute jump to some new location. Also, since the SC can be modified by the user, a transmission of data to the SC causes an absolute jump to some new location. In any case, after any one of these instructions alters the program flow, the sequential execution of instructions begins anew.

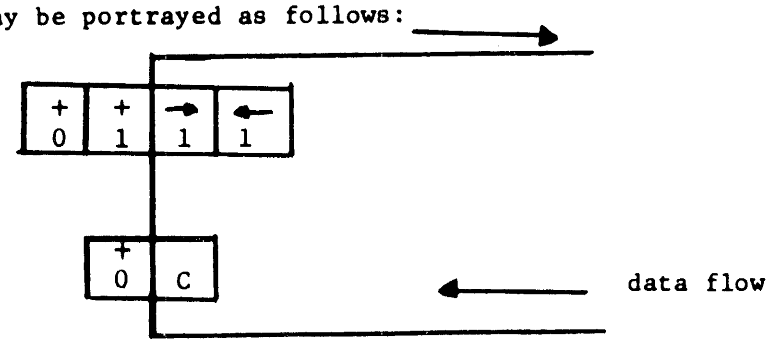
FAST source program machine instructions are oriented to the functional organization of the GRI-909. The major element of a machine instruction statement, for the most part, corresponds to and is written in the same order as the SDA, MOD, and DDA components of the basic instruction itself. Specific key words (2.3) are used by the assembler to distinguish five major classes of machine instructions. In some cases, the class of instruction implies a standard SDA or DDA (or both), and these are not supplied by the user but are automatically filled in by the assembler.

In most of the FAST language examples in this chapter, the corresponding machine language instruction will be printed to the left of the statement as it would appear on an assembly listing. Comments to the right are intended to amplify the meaning of the example and are not part of the instruction itself.

3.1 Register-to-Register Data Transmission Instructions:

A register is a storage place for data that has been produced by an input device, for data to be accepted by an output device, for data to be operated upon by a firmware device, or for data that is merely to be saved for temporary use. The data register associated with most input devices may only be used as a source of data. Generally, all other registers in the system may be used as both sources of and destinations for data. In other words, given a GRI firmware device or user-developed device with a data register, the nature of the device determines that the register will be interfaced to the source bus, to the destination bus, or to both.

The GRI-909 architecture enables the transmission of data between any two registers in the system. This is accomplished via a direct connection between the source bus and the destination bus. For data transmission there is no distinction between internal registers, firmware device registers, and input/output device registers. In addition, the connection between the source bus and destination bus contains a special device called the Bus Modifier, which enables simple modifications to data during the transmission process. This Bus Modifier may be portrayed as follows:



The modification(s) to data as it passes from one register to another is specified by the MOD portion of the data transmission instruction. If no modification is selected (MOD = 0000), then data passes through the two +0 paths, which results in a direct 16-bit parallel transfer. The FAST key words (2.3) used to specify modifications to data are:

C	one's complement
P1	plus one (increment)
R1	shift right one bit
L1	shift left one bit

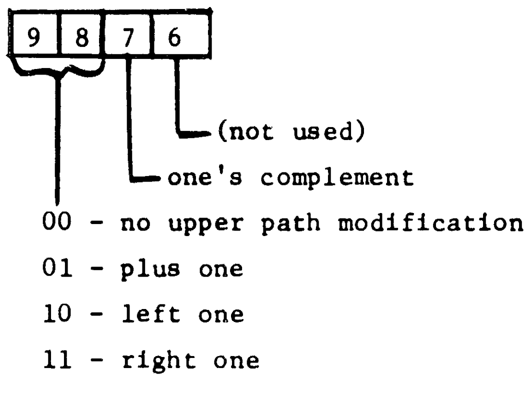
If the one's complement is selected, it occurs before other modification, if any. Only one of the upper paths may be specified during a data transmission instruction. There are 2 X 4 or 8 possible paths through the Bus Modifier.

The increment (P1) path has a bus overflow indicator associated with it. When this path is selected, the overflow indicator will be set (true) if, and only if, the source data was equal to -1 (all one's). If such overflow did not occur, then the indicator will be cleared. This indicator, referenced as BOV, may be tested after a data transmission instruction with the appropriate Sense Function instruction (3.5).

The shifting paths (R1 or L1) have a one-bit LINK register associated with them. Any shift, right or left, is performed circularly through this link (LNK). For instance, during a right shift, the entire word is displaced right one bit, and the low-order bit of the source word goes into the LNK. The previous contents of the LNK goes into the high-order position of the data word as it is sent to its destination. After any shift the new state of the LNK may be tested with the appropriate Sense Function instruction (3.5). If it is desired to shift a zero (or a one) into the word being transmitted, the LNK is first cleared (or set) with the appropriate Function Output instruction (3.4).

A further feature of the GRI-909 is the bus address 00, referenced as ZERO, which is a source of a zero data word. This special source may be used to clear any data register. ZERO through the P1 path can be used to transmit plus one and ZERO through the complement path yields minus one. ZERO may be used as a destination for data. Then tests of the BOV (or LNK) can be done <u>without</u> affecting the source data itself.

The MOD format for register-to-register data transmission instructions is:



These statements have the general form

$$\text{[symbol :] [C] register} \left[\begin{Bmatrix}\text{P1}\\ \text{L1}\\ \text{R1}\end{Bmatrix}\right] \text{[TO register] [; comment]}$$

If the transmission is between different registers, the source register is left unchanged and the contents of the destination register is replaced by the source data with the selected modifications. If the destination is omitted in this statement, then the operation is upon the contents of the source register -- the assembler copies SDA into DDA when constructing the instruction word.

The statement

```
77 0000 11    TTI TO AX
```

transmits the teletype input register to the arithmetic operator X-register. To transmit the incremented value of AX to AY, one writes.

```
11 0100 12    AX P1 TO AY
```

To send the one's complement of the high speed reader input to AX, we write

```
76 0010 11    C HSR TO AX
```

The one's complement is formed by merely inverting all the bits in the data word. The two's complement is formed by following the one's complement with an increment. To transmit the two's complement in the above instruction, we would change it to

```
76 0110 11    C HSR P1 TO AX
```

Keep in mind that the arithmetic operator assumes that negative numbers are in two's complement notation. Given two general purpose registers, GR1 and GR2, the following code computes their sum and difference and leaves the sum in GR1 and the difference in GR2 (of course, the original contents of GR1 and GR2 will be destroyed)

```
        GR1 TO AX           ;1st argument
        GR2 TO AY           ;2nd argument
        ADD                 ;select add function (3.4)
        AO TO GR1           ; store sum
        C AY P1 TO AY       ;negate 2nd argument
        AO TO GR2           ;store difference
```

The initial values of the arguments need not have been positive -- either one or both of them could have been negative (in two's complement notation). Note that to subtract argument 2 from argument 1, we add the two's complement of argument 2. The instruction that negated the data in AY could, of course, have been written

```
C AY P1
```

Some examples of the use of the special address ZERO are

```
00 0000 11      ZERO TO AX        ;CLEAR AX
00 0100 76      ZERO P1 to HSP    ;PUNCH +1
00 0110 12      C ZERO TO AY      ;AY = -1
11 1000 00      AX L1 TO ZERO     ;COPY AX SIGN TO LNK
```

This last example does not alter the contents of AX, but since the source word is sent through the Bus Modifier shifted left, the sign bit of AX is now also in the LNK.

An arithmetic right shift is often required in computer arithmetic. The arithmetic right shift displaces bits 0-14 each time it is performed and leaves the sign bit the same. Thus, if a word started with a 1 in the sign bit and were continually, arithmetically, shifted right, the word would eventually fill up with 1's from the left hand end.

During an arithmetic right shift, the sign of the argument being shifted is copied into the word as it is shifted right. Since the previous contents of the LNK is always shifted into the bit position vacated, the following two instructions perform an arithmetic right shift on the contents of AY

```
12 1000 00      AY L1 TO ZERO
12 1100 12      AY R1             ;AY/2
```

An arithmetic right shift is equivalent to dividing an argument by two. Similarly, a left shift is equivalent to multiplying an argument by two. In order to ensure that a zero is shifted into the low-order bit of the destination word, the LNK should be cleared before the shift. For example, the sequence

```
CLL         ;LNK = 0
AX L1       ;AX*2
```

shifts AX left one bit. It is not always necessary to set or clear the LNK before a shift. Sometimes it is only desired to examine a data word bit by bit. Such a data word might represent up to 16 switch settings. A device might be implemented such that each bit in its data register represented a contact setting or switch setting in some portion of the overall system. The GRI-909 console switch register continuously represents the settings of the 16 data switches. Since it is a source of data only, it must be transmitted to some other register before shifting its value. For example

```
10 0000 11      SWR R1 TO AX
               ;bit 0 of SWR in LNK here
11 1100 11      AX R1
               ;bit 1 of SWR in LNK here
11 1100 11      AX R1
               ;bit 2 of SWR in LNK here
11 1100 11      AX R1
               ;bit 3 of SWR in LNK here
                .
                .
                .
                etc.
```

As mentioned before, the LNK may be tested after each shift to see if the bit shifted out of the data word was a zero or a one. The data is sent back to AX so that after each shift, the next bit to be examined moves to the low-order position.

Sometimes data is packed into a 16-bit machine word. The word may contain two or more pieces of information with one or more bits used to contain each item. For example, sometimes two ASCII characters are packed into a word, 8 bits each. Suppose we wish to isolate the high order three bits of AX by shifting AX left three places and storing the item in the low order bits of AY. We write

```
AX L1
ZERO L1 TO AY
AX L1
AY L1
AX L1
AY L1
```

Note that the second instruction in this sequence ensures that the 13 high order positions of AY will be zeros when the unpacking operation is complete.

As a final register-to-register data transmission example, suppose we wish to decrement (subtract one from) the contents of a register, say GR1. We can use the arithmetic operator for this purpose.

```
GR1 TO AX
C ZERO TO AY      ;AY = -1
ADD               ;SELECT ADD FUNCTION
AO TO GR1         ;STORE RESULT
```

This code requires four machine words and ties up the arithmetic operator. The third instruction (ADD) could be left out if the AO were already in the required state. A better way to do this in the GRI-909 is:

```
C GR1 P1
C GR1
```

To demonstrate this, assume we have an 8-bit register containing the binary equivalent of 5.

00000101 $(2^0 + 2^2 = 1 + 4 = 5)$

taking the two's complement, we have

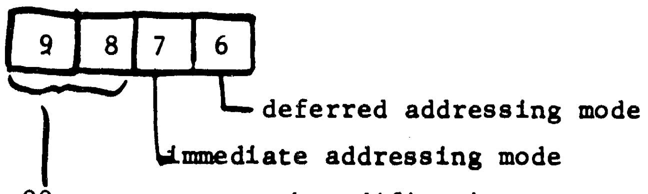
$$\begin{array}{r} 11111010 \\ +\ \underline{00000001} \\ 11111011 \end{array}$$

following this with the one's complement, we have

00000100 $(2^2 = 4)$

3.2 Memory Reference Data Transmission Instructions:

Main memory of the GRI-909 may contain from 1024 to 32,768 (decimal) locations. Any of these 16-bit words may be used as a source or destination in a data transmission instruction. These instructions are two words long -- the first word is in the usual SDA MOD DDA format, and the second word is either an address of some other memory location or is data to be operated upon. The overall memory system is accessed whenever the bus address 06 (memory buffer) appears as either the SDA or the DDA (or both) of a machine instruction. The assembler infers memory referencing from the nature of the FAST statement and fills in SDA and/or DDA appropriately. Further information, which specifies the memory address of the particular location involved in the data transmission, is supplied by the combination of MOD and the second word of the instruction. The MOD format for these instructions is:



00 - no upper path modification
01 - plus one
10 - left one
11 - right one

Note that only the modifications available in the upper portion of the Bus Modifier may be selected during memory reference data transmission -- bit 7 of the instruction word is here used to specify an addressing mode.

Data transmission statements involving main memory are of three types:

1) Register-to-memory - The contents of the hardware (non-memory) register at SDA is transmitted via the upper half of the Bus Modifier to the specified memory location. The assembler sets DDA = 06. This type has the general form:

$$\text{[symbol:] register } \left[\begin{Bmatrix}\text{P1}\\ \text{L1}\\ \text{R1}\end{Bmatrix}\right] \text{ TO [I] [D] e [;comment]}$$

2) Memory-to-register - The contents of the specified memory location is transmitted via the Bus Modifier to the hardware register at DDA. The assembler sets SDA = 06. This type has the general form:

$$\text{[symbol:] [I] [D] e } \left[\begin{Bmatrix}\text{P1}\\ \text{L1}\\ \text{R1}\end{Bmatrix}\right] \text{ TO register [;comment]}$$

3) Memory-to-self - The contents of the specified memory location is transmitted via the Bus Modifier back to the same memory location. Here, SDA = DDA = 06. This type has the general form:

$$\text{[symbol:] [I] [D] e } \begin{Bmatrix}\text{P1}\\ \text{L1}\\ \text{R1}\end{Bmatrix} \text{ [;comment]}$$

In the foregoing general forms, e is a general FAST language expression (2.6) whose value assembles into the second word of the instruction. The key words (2.3) I and D are used to select the memory addressing mode.

The address of the particular memory location into which data is transmitted, or from which data is transmitted, is called the effective address. The effective address is determined differently for each of the memory addressing modes. Given the form [I] [D], any one of the four GRI-909 addressing modes may be selected in a FAST memory reference statement.

Form	Mod	Mode
	0000	Direct (3.2.1)
D	0001	Deferred (3.2.2)
I	0010	Immediate (3.2.3)
ID	0011	Immediate - deferred (3.2.4)

3.2.1 Direct Addressing Mode

The direct mode is implied by the absence of both of the codes I and D in the FAST statement. In the direct addressing mode, the second word of the instruction contains the effective address -- the GRI-909 control logic fetches this second word and uses its value to access the location involved in the data transmission. The statement

```
06 0000 11   100 TO AX
000100
```

loads the contents of location 100 into the AX register. The statement

```
12 0100 06     AY P1 TO 1234
001234
```

stores the incremented value of AY into location 1234. To increment or shift the data word labeled X1 (assume it is at location 501), we could write

```
06 0100 06   X1 P1
000501
```

```
or              06 1000 06    X1 L1
                000501

or              06 1100 06    X1 R1
                000501
```

Note that, unless the user alters the second word during program execution, the same location is accessed every time a direct mode memory reference instruction is encountered.

3.2.2 Deferred Addressing Mode

The key word D selects the deferred mode in a FAST memory reference statement. In this mode, the second word of the instruction contains the address of another location whose incremented contents is the effective address. The second word of the instruction is fetched and its value is used to access another location whose contents is incremented and written back into the same location. The control logic then uses this incremented value as the effective address.

If location 5 contains the value 200, then

```
11 0001 06    AX TO D 5
000005
```

stores AX into location 201. Location 5 now contains the value 201. If we do not change the value in location 5 before executing the above (or an identical) instruction again, then AX will be stored into location 202 and location 5 will be left at 202.

The deferred mode is sometimes called "indirect with auto-indexing". "Indirect" means that the address in the memory reference instruction is not itself the effective address, but is the address of the effective address. "Auto-indexing" means that the effective address is incremented before it is used in the data transmission.

The deferred mode (and the immediate-deferred mode, below) is ordinarily used to access the sequential words in a data table, work area, or buffer area. Assume location 5 is initialized to 200 before each of the following examples. Then,

```
06 0001 12      D 5 TO AY
000005
```

can be successively executed to load the contents of location 201, 202, 203, etc. into AY. Obviously, the data in AY will be processed by other instructions in the program before the next item is retrieved. The instruction

```
00 0001 06      ZERO TO D 5
000005
```

can be used to clear successive locations beginning at 201. Also, the instruction

```
06 0101 06      D 5 P1
000005
```

can be used to increment these sequential locations.

Often the data entries in a table will consist of two or more words each. The first word of an entry is fetched and, depending on its value, a decision is then made either to fetch the rest of the entry or to skip over the current entry and retrieve the next entry. If our auto-indexing location (assume location 5 again) is labeled INDX, the instruction

```
06 0100 06    INDX P1
000005
```

executed n times will, in effect, skip over n words in the table in order to point us to the next entry.

3.2.3 Immediate Mode Addressing

The key word I selects the immediate mode in memory reference statements. "Immediate" implies that the instruction itself provides or receives the data being transmitted. Therefore, the effective address in this mode is merely the address of the second word of the instruction itself.

To load the constant 14 into AX, one writes

```
06 0010 11      I 14 TO AX
000014
```

Note that the data to be transmitted has been assembled into the second word of the instruction. To initialize an auto-indexing location (INDX) so that successive entries of a table (TABLE) may be retrieved using a deferred mode, we could write

```
I TABLE-1 TO TRP
TRP TO INDX
```

Sometimes counting of events, like the number of times through a given program loop, is done with an immediate mode instruction. The statement

```
06 0110 06    I 0 P1
000000
```

causes the second word of the instruction to be incremented. The statement could be written with other than 0 as the value for the second word -- in any case, before entering a loop containing this counter, the second word of the instruction must be set to some appropriate initial value.

When an interrupt is detected, some portion of the interrupt-handling routine usually saves the contents of crucial registers in the system. This is often done with a sequence of immediate mode instructions.

```
03 0010 06    SAVE:  TRP TO I
000000

17 0010 06    MSR TO I
000000

11 0010 06    AX TO I
000000

12 0010 06    AY TO I
000000
              etc.
```

When the interrupt has been processed, the following direct mode statements could be used to restore the saved registers

```
SAVE+1 TO TRP
SAVE+3 TO MSR
SAVE+5 TO AX
SAVE+7 TO AY
etc.
```

3.2.4 Immediate-Deferred Addressing Mode

The key word ID selects the immediate-deferred mode in memory reference statements. This mode combines the immediate and deferred features. "Immediate" here implies that the address of the auto-indexing location is merely the address of the second word of the instruction itself. The GRI-909 control logic fetches the second word of the memory reference instruction, increments it, and writes the incremented value back into the second word. This incremented value is then used as the effective address in the data transmission. The statement

```
11 0011 06      AX TO ID 200
000200
```

can be used to store AX into locations 201, 202, 203, etc. It is the second word of the instruction itself which is auto-indexed. Of course, this second word must be re-initialized before a new set of AX values can be stored into the same locations. The statement

```
GET:  ID TBL-1 TO AY
```

loads AY with the word labeled TBL the first time it is executed. The next n times, without being re-initialized, it loads the words at TBL+1, TBL+2, TBL+3, ..., TBL+n. This instruction at GET would probably be the first instruction in a program loop which processes the entries in the data table, TBL. Before entering the loop, the instruction at GET could be initialized by writing

```
I TBL-1 TO TRP
TRP TO GET+1
```

where the expression GET+1 represents the address of the second word of the immediate-deferred instruction at GET. If each entry in TBL were two words long, then some other instruction in the loop, after the instruction at GET, might be used to load the second word of the entry into a register, say AX. The _same_ auto-indexing location is used for this purpose. For example,

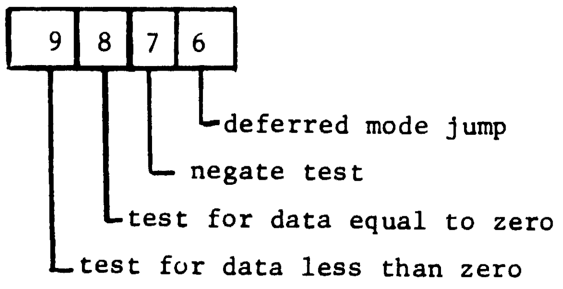
```
D GET+1 TO AX
```

loads the second word of the entry provided that the instruction at GET has already loaded the first entry word. If it is desired to bypass this second word and point to the next entry, then the auto-indexing location can be updated by the instruction represented by

```
GET+1 P1
```

3.3 _Data Testing Instructions_:

These instructions are used to test the data in a register relative to zero. The DDA is always the data tester and is filled in by the assembler. SDA refers to any non-memory data register, and MOD specifies the nature of the test and the addressing mode. The MOD format is:



If the test specified by MOD is not true, then the GRI-909 control logic procedes to the next sequential instruction in the program. If the test is true, the second word of the instruction is taken as a jump address or, in the deferred mode, as the address of the jump address. If a jump occurs (test is true), the address of the second word of the data test instruction is stored in the trap register (TRP), and the effective jump address replaces the contents of the computer's sequence counter (SC) (the TRP is used for subroutine linkage -- see section 6.2). The FAST statement for data testing is an exact description of the action performed by the GRI-909, and has the general form

```
[symbol:] [IF register test] GO TO [D] e [;comment]
```

where e is a general expression (2.6) whose value assembles into the second word of the instruction, and the word "test" in this context refers to a standard or user-defined symbolic data test code (6). The standard data test codes, their MOD values and associated meanings are:

ETZ	0100	equal to zero
LTZ	1000	less than zero
LEZ	1100	less than or equal to zero
NEZ	0110	not equal to zero
GEZ	1010	greater than or equal to zero (not less than zero)
GTZ	1110	greater than zero (not less than <u>and</u> not equal to zero)

In order to jump to location 100 if the AX register contains zero, one writes

```
11 0100 03      IF AX ETZ GO TO 100
000100
```

Note that the first word, as in all GRI-909 instructions, is a data transfer format telling the processor to connect device 11 (AX) to device 03 (data tester). The non-memory source <u>and</u> the 03 destination indicates the data test instruction to the processor. The MOD in this case has the meanings shown above. For the purpose of the example, we have used a constant as the jump address -- the statement may be written with any meaningful expression to represent the jump address.

Because of the jump address, the data testing instruction is a form of memory reference instruction. The deferred mode jump operates similarly to the deferred mode memory reference data transmission instruction (3.2.3). If location 200 contains the value 543, then the statement

```
13 1001 03      IF AO LTZ GO TO D 200
000200
```

causes a jump to location 544 if the arithmetic operator output (AO or device 13) is less than zero. The second word of the data test instruction is used by the processor to fetch the jump address at 200 -- this address is first incremented and the incremented value replaces the contents of location 200 before the jump address is transmitted to the SC. Normal program flow begins at location 544. Since location 200 now contains the value 544, the next jump affected by the foregoing data test instruction would cause a transfer of control to location 545 unless the user reinitialized location 200. The major use for the deferred mode jump is to return from subroutine (6.2).

In order to apply a data test to the contents of a memory location, it is first necessary to load it into some non-memory register. Suppose the data word X1 is at location 1234 and the instruction OVER is at location 157. In order to jump to OVER if X1 is greater than zero, we could write

```
06 0000 11      X1 TO AX
001234
11 1110 03      IF AX GTZ GO TO OVER
000157
```

If the AX register were being used for some other purpose and we did not wish to destroy its contents, the TRP register associated with the data tester could be used to contain the data for testing. We could accomplish the foregoing test by writing

```
06 0000 03      X1 TO TRP
001234
03 1110 03      IF TRP GTZ GO TO OVER
000157
```

Regarding this last example, we note the following: 1) since the TRP register has the same address as the data tester, it cannot be loaded with data from another non-memory register since this combination defines the data testing instruction, 2) the TRP can be loaded from memory (SDA=06) -- this is the only instance of DDA=03 that is not a data testing instruction, and 3) if the jump occurs, the data in TRP will be lost, since it is automatically loaded with the address of the second word of the data test instruction.

Other than the aforementioned restriction on the TRP, it can be used as a general purpose register. Its contents can be transmitted to another register or to some memory location. For example,

```
03 0000 12      TRP TO AY
03 0000 06      TRP TO 501
000501
03 0010 06      TRP TO I ; STORE IMMEDIATE
000000
```

This last example instruction is often used to save the contents of the trap upon entry to a subroutine (6.2).

Often we require an instruction which causes a jump every time it is encountered. This enables us to jump back to the beginning of a loop, to call or enter subroutines, etc. Since device ZERO is a source of a zero data word, we could always employ the following artifice to jump to some location, say 533

```
00 0100 03      IF ZERO ETZ GO TO 533
000533
```

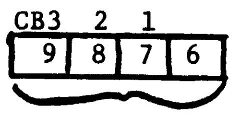
The FAST language provides a short form for this particular instruction, a simple GO TO

```
00 0100 03      GO TO 533
000533
```

The deferred mode jump may also be used with this short form jump instruction.

3.4 Function Output Instructions:

Function output instructions are used to deliver control or function pulses to those system devices which require them for mechanical or electrical control. A function output instruction always has the function generator as its SDA. Its code, 02, is supplied by the assembler. The DDA is some controllable system device, and the MOD specifies up to four pulses to be transmitted in parallel to the device at DDA. The MOD format is:



correspond to the four pulse output lines

The four pulses are transmitted in parallel. Therefore, up to 16 unique pulse patterns may be transmitted to a single device. Of course, the device must be interfaced so as to discriminate between the patterns. Simple devices usually associate a single line with a specific function. The general form of a function output statement is:

```
[symbol:] pulse [pulse] ... [[TO] device] ['comment]
```

The word "pulse" in this context refers to a standard or user-defined symbolic pulse code (5). The symbolic destination device code must be included unless the pulse code used has a destination built into it (5.1).

Standard pulse codes for operating devices such as teletype input/output, high speed reader, and high speed punch are

Mnemonic	Definition	Mod Code
STRT	-start	0001
CLIF	-clear input flag	1000
CLOF	-clear output flag	0010

The STRT pulse causes a paper tape reader to advance and read the next frame of data.

```
02 0001 77     STRT TO TTI      ;ADVANCE TTY READER
02 0001 76     STRT HSR         ;ADVANCE HS READER
```

Note that the "TO" may optionally be omitted from the statement. It is possible to combine start and clear flag commands in one instruction.

```
02 1001 77      STRT CLIF TO TTI
02 1001 76      CLIF STRT TO TTO
```

The ordering of multiple pulse codes is immaterial -- as each one is encountered, the assembler OR's its value into MOD. Depending upon how a device is being operated, the clear flag commands may be issued separately.

```
02 1000 77      CLIF TO TTI
02 0010 77      CLOF TO TTO
02 1000 76      CLIF TO HSR
02 0010 76      CLOF TO HSP
```

Standard pulse codes corresponding to internal functions of the GRI-909 have been defined so as to contain their destination. This means that the entire function output statement consists of merely the pulse code (5.1).

```
02 0100 00      HLT      ;HALT MACHINE
02 0010 00      STL      ;SET LINK
02 0001 00      CLL      ;CLEAR LINK
02 0011 00      CML      ;COMPLEMENT LINK
02 0000 13      ADD      ;SET AO TO ADD
02 0100 13      AND
02 1000 13      XOR
02 1100 13      OR
02 0010 04      ICO      ;INTERRUPT CONTROL ON
02 0001 04      ICF      ;INTERRUPT CONTROL OFF
```

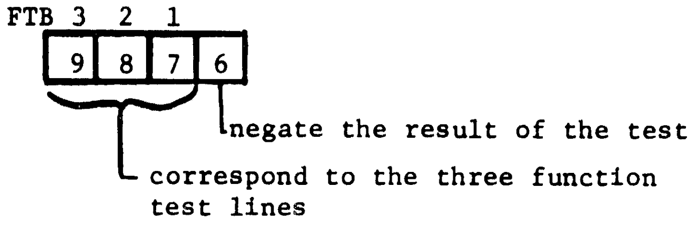
Note that the CML command is equivalent to

```
02 0011 00      STL CLL
```

The combined command is included in FAST as a convenience to the user.

3.5 Sense Function Instructions:

Sense function instructions are used to test the status or function signals of various types of devices. The device's status indicators are often used to represent such conditions as "ready", "busy", "overflow", "data error", etc. In a sense function instruction, the DDA is always the function tester, and its code (02) is filled in by the assembler. The SDA is any system device that has one or more status indicators associated with it. The MOD format is:


FTB 3 2 1
9
8
7
6
negate the result of the test
correspond to the three function test lines

If the test specified by MOD is true. then the GRI-909 control logic causes a skip over the next two machine words. Two words are skipped because a SKIP is usually followed by a jump (GO TO) instruction. The general form of a sense function is

[symbol] SKIP [IF] [device] status [status] ... [;comment]

The word "status" in this context refers to a standard or user-defined status test code (5). The symbolic source device code must be used unless the status test code has a source built into it (5.1).

Standard status codes for testing devices such as teletype input/output, high speed reader, and high speed punch are:

Mnemonic	Definition	Mod Code
IRDY	-input ready	1000
ORDY	-output ready	0010

For example:

```
77 1000 02    SKIP IF TTI IRDY ;CHARACTER READ?
76 0010 02    SKIP HSP ORDY    ;READY TO PUNCH?
```

Note that the word "IF" may be omitted as desired. The standard status code NOT causes the low-order bit of MOD to be set.

```
76 1001 02    SKIP IF HSR NOT IRDY
77 0011 02    SKIP TTO NOT ORDY
```

If the status code NOT is included, then a skip occurs only if the selected condition is not true. Standard status codes corresponding to internal conditions of the GRI-909 have been defined so as to contain their sources. This means that the sense function instruction may be written without the source device.

```
13 0010 02    SKIP IF AOV    ;ARITHMETIC OPERATOR OVERFLOW
00 0010 02    SKIP BOV       ;BUS OVERFLOW
00 0100 02    SKIP LNK       ;LINK SET
00 1000 02    SKIP POK       ;POWER OK
```

The code NOT may be included if it is desired to skip on the falsity of a given condition. Codes for status indicators residing on the same device may be combined in the same sense function instruction. For example,

```
00 0110 02      SKIP BOV LNK
```

skips if the OR of the selected conditions is true, that is, if either BOV or LNK is true. Setting the NOT bit complements this inclusive OR. Thus, the instruction

```
00 0111 02      SKIP NOT LNK BOV
```

skips only if BOV is not true and LNK is not true. Note that in neither of the foregoing instructions the setting of the POK indicator affected the tests in any way.

As a further clarification of sense function instructions, assume a device Q that has three status indicators X, Y, Z corresponding to bits 9, 8, 7 of MOD respectively. Further assume that we wish to test for unique combinations of the status indicators -- given the three indicators, there are 8 unique combinations to test for. In the examples which follow, program control passes to YES if the desired combination test is true, and to NO if it is not true.

To test for the condition 000 (X, Y, Z all false), we write

```
         SKIP Q NOT X Y Z
         GO TO NO
YES:     test true here ;NONE ON
```

To test for a single indicator on, such as 010, we could write

```
         SKIP Q Y
         GO TO NO
         SKIP Q NOT X Z
         GO TO NO
YES:     test true here ;ONLY Y ON
```

To test for two indicators on, such as 110, we could write

```
         SKIP Q NOT Z
         GO TO NO
         SKIP Q X
         GO TO NO
         SKIP Q X
         GO TO NO
YES:     test true here ;ONLY X and Y
```

To test for all three indicators on, we write

```
         SKIP Q X
         GO TO NO
         SKIP Q Y
```

```
        GO TO NO
        SKIP Q Z
        GO TO NO
YES:    test true here   ;ALL ON
```

Of course, the assumption we must make here is that the state of the device Q remains constant when testing for any unique combination.

CHAPTER FOUR

ASSEMBLER INSTRUCTIONS

Some FAST statements merely act as directives to the assembler during the assembly process and do not result in object code output. Other assembler instructions enable the inclusion of numeric or textual data into the user's program. All of these so-called assembler instructions (pseudo-ups), with the exception of the one for defining new system symbols (Chapter 5), are presented in this chapter.

4.1 Parameter Definition:

It is often desirable to establish a numeric value or constant and to be able to refer to it symbolically. For example, instead of using the constant 215 (the teletype carriage-return code), it might be more meaningful to use the symbol CR. Such a symbolic parameter can be defined by the statement:

```
CR = 215   ;CARRIAGE RETURN
```

We could then write:

```
I  CR  TO  TTO  06 0010 77
                000215
```

The general form of a FAST parameter definition statement is:

```
symbol = e [;comment]
```

where e is a general expression (2.6). The value of the symbolic parameter will be the assembled value of the expression with which it is associated. A parameter is used to represent a numeric value -- this parameter may be used as an operand in other expressions for address or data values. Note that no object code is generated by a parameter definition statement -- the statement merely generates an entry in the assembler's symbol table.

Examples:

```
TEN = 10
CRLF = 106612 ;CARRIAGE RETURN AND LINE FEED
DIFF = A - B
```

A symbolic parameter may be redefined within the same program. If a parameter does take on more than one value, then its initial value must be defined in the source program before it is first used to reference a numeric value. Also, the expression e, used to specify the value of a parameter, must be fully resolvable by at least the end of pass 1 (so that it will have the correct value during passes 2 and 3). In other words, the value of any symbol in the expression must be established within at most one forward reference. For example, in the sequence

```
A = B + 5
B = 22
```

the correct value of A is not established when the statement is first encountered during pass 1 since B is not yet defined (A will have the value 5 since undefined symbols are assigned the value 0). Nevertheless, when the definition of A is encountered during pass 2, it will be assigned the correct value, 27, since B is now defined.

Parameter definition statements are often used for the purpose of system linkage -- see section 6.1.

4.2 Radix Control:

FAST language constants (2.5) are converted to binary and are interpreted according to the setting of an assembler variable called the RADIX. Constants may be written as either octal numbers or decimal numbers. The assembler's RADIX is set to octal at the beginning of each pass. In other words, the assembler assumes all constants to be in octal.

The user may switch the RADIX from one mode to another at will. The form of the RADIX control statement is:

$$* \left\{ \begin{matrix} \text{DEC} \\ \text{OCT} \end{matrix} \right\} \quad [\text{;comment}]$$

If the user wishes to write constants in decimal notation, he precedes the first such constant with the statement:

*DEC

All constants between this statement and the end of the program (or between this statement and a *OCT command) will be interpreted as decimal numbers. The assembler's RADIX is like a switch -- once it is thrown to decimal it stays in that mode until the beginning of the next pass or until the assembler encounters the other RADIX command

*OCT

causing the RADIX to be set back to octal. While in the octal mode, the assembler detects the usage of the decimal digits 8 and 9. If either of these digits occur in a statement while the assembler is in the octal mode, the assembly listing of that statement will be preceded by the error code D (decimal digit in octal field).

Remember that the assembler's RADIX is automatically set to octal at the beginning of each pass.

4.3 Location Counter:

The assembler maintains a variable called the LOCATION COUNTER. During the assembly process, the LOCATION COUNTER always reflects the address of the next memory location that object code may be assigned to by the assembler. As the assembler processes each statement that generates either a machine instruction or data, it automatically updates the LOCATION COUNTER by the length (number of machine words) of that statement's object code. At the beginning of each pass, the assembler sets its LOCATION COUNTER to 0. If the user does not change the LOCATION COUNTER, then his entire program will be assembled for sequential locations starting with location 0.

A statement to set the assembler's location counter is ordinarily used to specify the first location of a program being assembled. The set-LOCATION COUNTER statement has the form:

```
*e [;comment]
```

where e is an expression (2.6). A further restriction on this statement is that the expression must not contain any undefined symbols when it is first encountered by the assembler. Also, e is influenced by the current radix (4.2). The statement

*1000 ;START AT $1000_{(8)}$

causes all subsequently encountered machine instructions or data words to be assembled for sequential locations starting at location 1000. More than one set-LOCATION COUNTER statement may be included in a given program -- if the user wishes to load various segments of his program into non-contiguous areas of memory, then a set-

LOCATION COUNTER statement at the beginning of each source program segment fixes its respective starting address.

Often it is desirable to reserve a block of memory locations to be used as a work area or input/output buffer when the object program is run. This is done by updating the LOCATION COUNTER relative to its current value. Thus, the statement

```
*.+50 ;NOTE:  50 MAY BE OCTAL OR
              DECIMAL
```

causes the assembler to reserve (skip over) 50 sequential locations, the first of which will be at the address specified by the LOCATION COUNTER when the statement is read. This current value of the LOCATION COUNTER is denoted by the special character period (.). If the current value was 101, then the next object code generated will be assigned to location 151.

A block of memory thus reserved may be labeled. The statement

```
WORK: *.+20
```

reserves 20 locations, the first of which may be symbolically referenced as WORK. Note that the label is encountered and processed before the set-LOCATION COUNTER command itself is processed.

4.4 Program Terminators:

The user must indicate the physical end of a program to the assembler. The assembler can then finalize all processing for the current pass and come to a halt ready to procede to the next pass in the assembly process, if any. The assembler command

```
*END
```

must be the last statement in the program. If a source program is made up of two or more segments of tape, then each segment but the last must have the command

```
*EOT
```

as its last statement. This end-of-tape command causes the assembler to halt for the insertion of the next tape segment in the reader -- pressing CONT on the console starts the processing of the new tape as part of the same program.

4.5 Data Definition:

Some of the data a program operates upon may be assembled into the program itself. Such items consist of numeric constants and/or textual data. Numeric constants could be upper and lower limits for checking against input values, tables of values used for code conversion or function interpolation, machine addresses of important tables or entry points in a program, etc. Of course, single-valued numeric constants are often assembled into immediate mode memory reference instructions (3.2.2). Textual data could consist of error messages to be output to an operator, or fixed heading information for printed reports.

4.5.1 Word Values

Numeric constants can be assembled for consecutive locations of memory by using the general form

```
[symbol:] e [,e] ... [;comment]
```

where each e is a general expression (2.6). The expressions representing values to be assembled for each location are separated from each other by commas (,). Each expression e results in a full 16-bit binary word in the object program. The first word assembled from this statement may optionally be labeled. Thus, the statement

```
PWR:  1750,144,12
```

causes the three consecutive octal numbers to be assembled for three consecutive memory locations, the first of which is labeled PWR. This statement is equivalent to the three statements:

```
PWR:  1750
      144
      12
```

Further examples are:

```
COUNT:  0
LNGTH:  B - A + 1
        Z!Q,ZQQ
        2,15,.+5
```

Note: If the symbol . (representing the assembler's LOCATION COUNTER) is encountered in an expression in a comma-separated list of data word definitions, its value will be the address of the memory location for which that specific expression is being assembled.

4.5.2 Text

One or more characters of ASCII text may be assembled for consecutive words, packed two characters (8 bits each) per word. If the text contains an odd number of characters, the rightmost 8 bits of the last word assembled will be set to 0's. A textual data definition statement consists of the character single-quote ('), followed by the body of the text, and is terminated by the same delimiting character that preceded the text. The general form for defining textual data is

```
'dc [c] ... d
```

where d is a delimiter chosen by the user, and the c's are the individual characters in the text. The delimiter must be chosen such that it does not occur within the body of the text. Examples are

```
'/LIMIT EXCEEDED/

MSG3: '.V1/V2  0.
```

Note that text may be labeled. The label (e.g. MSG3 above) is associated with the first packed word assembled from the text. As for word values, two or more text definitions may occur in the same source statement provided they are separated by commas.

The delimiters and text characters may be any of the printable ASCII characters, including those outside the FAST general usage and reserved character sets (2.1). Exceptions -- the following characters have special meanings to the assembler and the Source Text Editor and may not be used as a delimiter or text character:

Carriage-return
Line-feed
Back-arrow
Rubout
Block-mark

4.5.3 Combined Text and Word Values

Text and word value definitions may be freely combined in a comma-separated list. In this, and in any statement, the 80-character statement length must not be exceeded. As an example of a reasonable such combination, consider the following

```
ERR4:7,'/TEMP. 4 HIGH/,106612
```

The address of this hypothetical message, represented by ERR4, is to be sent to a general output routine which will process the data assembled from the overall statement. The first word fetched by the output routine is the value 7 which tells it to unpack and type out the next 7 words (14 characters). The second list element assembles into 6 packed words, since there are 12 characters in the body of the text. The last list element, or 7th message word, represents the packed characters carriage-return and line-feed. This mechanism is necessary because they could not be made a part of the text definition itself (4.5.2).

CHAPTER FIVE

DEFINITION OF NEW SYSTEM SYMBOLS

The most useful function an assembler provides the user is the ability to give symbolic names to memory locations and word values. In addition to this, the FAST language also provides the user the ability to name new system entities beyond those defined in the assembler's permanent symbol table. This ability is extended to encompass not only the SDA and DDA portions of the instruction but also the MOD portion of the instruction.

The permanent symbol table contains definitions for standard IO devices such as TTI, TTO, HSP, and HSR. The user may desire to refer to an analog multiplexer as MUX, or an A/O converter as ADC. These symbols are added to the assembler's symbol table via a symbol definition statement.

Also included in the permanent symbol table are definitions of certain pulse patterns from the function output section as well as certain status codes utilized in sense function instructions. Examples are STRT, CLIF, BOV, ORDY, etc. The user again may wish to add his own unique codes to the symbol table, such as GRP1, STOP, LOAD, etc.

The user may also wish to develop his own pseudo code for commonly used instructions. THE DEFINITION STATEMENT THAT DEFINES A NEW SYMBOL MUST PRECEDE ANY OTHER USE OF THE SYMBOL IN THE PROGRAM. (5.1)

It is strongly recommended that the user not re-define bus modifier symbols (P1, L1, R1, C) or data testing symbols (ETZ, LTZ, etc.). These are symbol type numbers 4 and 5. Symbol types 6 and 7 are more normally defined as described in 2.4 and 4.1. They may, however, be defined by a definition statement.

The symbol definition statement may also be used to add pseudo-codes.

The general form for defining a new symbol is:

SYMBOL #t,n [;comment]

SYMBOL is the 5 character name being defined. #t is a type number that describes the type of definition being made (see table). n is the numeric value (in octal) that the assembler will use to replace the symbol when it is encountered during an assembly.

<u>Note</u>: ALL NUMERIC REFERENCES MUST BE OCTAL.

SYMBOL DEFINITION TABLE

#t	Type of Definition
1	Device Code
2	Output Pulse Code
3	Status Test Code
4	Path Code for Bus Modifier
5	Data Test Code
6	Statement Label
7	Parameter Symbol
10	Register Reference Pseudo Code
11	Memory Reference Pseudo Code

5.1 <u>Device and Device Related Codes</u>:

New devices added to the system by the user will require the use of the definition statement if they are to be referred to symbolically in the users program. <u>It is necessary that the user put these definitions at the front of the program, or perhaps on a separate tape terminated by an *EOT</u> (section 4.4). This definition tape may be used in front of all program tapes the user wishes to assemble that refer to the same system devices. This definition tape need only be read during pass 1 of the assembly.

For example, if a new device named MUX were to be defined with an octal address 56 (maximum address is 77_8), the definition statement is:

```
MUX#1,56
```

After this statement has been entered, the name MUX may be used just as any other device name in the assembler. If one wished to replace one of the existing assembler symbols with another address, the same procedure would be used. For example:

```
TTI#1,44
```

This associates the symbol TTI with the new address 44. It is useful to point out here that the values in the assembler's symbol table for type 1 symbols (devices) exist as right justified numbers. However, this is not the case when one specifies a MOD code (types 2-5). The MOD codes consist of four binary bits in the middle of a machine instruction, so defining a new pulse code will require that the octal value be located in the proper place (bits 9-6) of the octal "n" parameter.

Consider the following pulse code:

```
0110
```

Its name will be "QUIT", and it will be a type 2. It remains to define the octal "n" parameter.

Consider the string of sixteen binary digits below:

```
0000000000000000
```

Breaking these up into the more familiar SDA-MOD-DDA style and adding the binary code for the new symbol in the right spaces yields:

000000 0110 000000

If this were to be converted into an octal number, conversion sould begin by separating the number into groups of three binary digits, starting from the right.

0 000 000 110 000 000

The equivalent octal number is, therefore,

600

This, then, is the "n" parameter needed above. The pulse code digits are in the correct places in the SDA-MOD-DDA form, and the number is octal. The symbol definition would be

QUIT#2,600

Now consider the case of defining a pulse code with a specific destination attached to it in order that the destination need not be specified each time the pulse is generated. To do this, the user must define the destination as well as the MOD, and convert this entire string to an octal number. For example, if the symbol QUIT, above, had a destination device with octal address 56, then the binary string would be:

0 000 000 110 101 110

and the equivalent octal number would be

656

The symbol definition then would be

QUIT#2,656

thus incorporating the destination address as well as the pulse code. When the newly-defined symbol QUIT is subsequently encountered in a source statement, the assembler recognizes that it was defined as a pulse code (type 2) and consequently fills in SDA with 02 (the address of the function-generator) and fills the remainder of the word with the defined octal value of QUIT, i.e. (in machine language), 02 0110 56.

If the described symbol were to be a status test code and have a source device associated with it rather than a destination device, then the binary string would be

```
1  011  100  110  000  000
```

The equivalent octal number is

134600

and the symbol definition would be

QUIT#3,134600

then the statement

SKIP IF QUIT

assembles as

```
56  0110  02
```

Note that the assembler fills in the correct DDA.

5.2 Abbreviated Machine Instructions:

These FAST language pseudo-instructions are abbreviations of actual machine instructions. That is, the instruction will translate into the SDA MOD DDA form and will have the same machine language representation as the instruction to which it corresponds. The general instruction has two forms

symbol

or

symbol e

which combine into the format

$$\textit{symbol} \left[\text{[D] e} \right]$$

The D symbol signifies the optional deferred addressing mode as it did in preceding sections. Note that the immediate mode is not included in the parameters.

This first usage of pseudo codes is perhaps best illustrated by several examples. For instance, if a zero memory word were to be defined at many locations in core, the programmer would have to use the instruction below for each re-initialization.

ZERO TO *location*

To save effort, the pseudo code with operand Type 11

ZM *location*

could be used in place of the normal FAST instruction.

This pseudo code if defined using the procedure shown in section 6.1. The value is derived from the instruction that the pseudo code is to represent. The instruction ZERO TO LOCATION has the machine language equivalent

```
00  0000  06
*location*
```

for which the octal equivalent is simply the number 6. The data is put into the general form for symbol definition

```
SYMBOL#t,n
```

where t is the octal type number (from the table in 4.5) and n is the equivalent octal value.

Thus, to define the pseudo code ZM, the programmer would write

```
ZM#11,6
```

as one of the statements in his source code. Using the instruction

```
ZM 765
```

in the source program would then translate into the machine instruction

```
00  0000  06
000765
```

The deferred addressing option could also be specified for the instruction by

```
ZM D 765
```

Other instructions may be assigned pseudo code equivalents as the programmer wishes. Another example might be an instruction to copy the arithmetic overflow bit into the bus modifier link bit. In FAST, the instruction would be

MSR R1 TO ZERO

(The arithmetic overflow bit is the rightmost bit of the Machine Status Register.) If the programmer wished to abbreviate the instruction as the new instruction

COVL (for copy overflow to link)

a pseudo code without an included operand, the definition would follow the procedure outlined below.

1. Determine from the table the type of the pseudo code: no operand - Type number = 10
2. Determine the octal value: The machine language equivalent of MSR R1 TO ZERO is

   17 1100 00

   The equivalent octal number for the above digit string is

   37400
3. Define the symbol

   COVL#10,37400

5.3 Local Pseudo-Instructions:

These are codes used in conjunction with the various interpretive software routines provided with the GRI-909, such as the Floating Point Interpretive Package. This section will deal with pseudo codes only as they relate to the floating point package, but the ideas expressed here are relevant to all such routines.

The format for these codes is the same as that in the previous section.

$$\textit{symbol}\ \left[\ [\mathrm{D}]\ \mathrm{e}\ \right]$$

However, the set of instructions is not open ended but consists only of those included in the interpretive routine under discussion. The following are a few of the instructions in the Floating Point Package:

FLDA [D] *operand*	load the floating point accumulator with the quantity at the location specified by the operand.
FMPY [D] *operand*	multiply the contents of the accumulator by the value at the location specified by the operand.
FADD [D] *operand*	add to the current accumulator value the contents of the location specified by the operand.
FSTA [D] *operand*	store the value of the accumulator at the location specified by the operand.
FSQT	take the square root of the current contents of the floating accumulator and put the result back into the accumulator.

These local pseudo codes are defined on a command equate tape supplied with the Floating Point Interpretive system. The definitions which create the pseudo-instructions are either type 10 or 11 function definitions. The command equate statements for the few examples listed would be as follows:

```
FLDA #11, 01
FLDAD #11, 101
FADD #11, 10
FADDD #11, 110
FMPY #11, 12
FMPYD #11, 112
FSTA #11, 2
FSTAD #11, 102
FSQT #10, 36
```

The use of the floating point system is illustrated by the following example. Suppose one wished to compute the Pythagorean relation $A^2 + B^2 = C^2$ and solve for the C value in floating point. Further suppose that the values for A and B are stored at locations referenced by the symbols AA and BB respectively. The floating point interpretive routine is called into play by jumping to the location referenced by the symbol $SFI. The sequence of instructions would then be:

```
GO TO $SFI ;          THIS INSTRUCTION CAUSES ENTRY TO
                      INTERPRETIVE MODE

FLDA AA    ;          LOAD THE ACC WITH A

FMPY AA    ;          COMPUTE A SQUARED

FSTA CC    ;          STORE A SQUARED IN C

FLDA BB    ;          LOAD ACC WITH B

FMPY BB    ;          COMPUTE B SQUARED

FADD CC    ;          ADD A SQUARED TO B SQUARED

FSQT       ;          COMPUTE C=SQRT (A*A+B*B)

FSTA CC    ;          STORE THE C VALUE IN CC

FEXT       ;          EXIT FROM INTERPRETIVE MODE
```

The above sequence shows how these special codes are used as instructions to the floating point package. In effect, the interpretive routine executes each of the instructions following the call to the routine as if it were an instruction to a firmware device. Program control returns from the $SFI routine when an exit instruction (FEXT) is encountered. The FAST assembler assigns a specific code to each of the floating point instructions. This is not in the SDA-MOD-DDA format but rather a unique octal number which is interpreted by the floating point interpreter and causes a series of local subroutines to be called to perform the necessary calculations. The user is referred to manual 71-44-005 for a detailed description of the Floating Point Interpretive Package.

CHAPTER SIX

USAGE NOTES

6.1 System Linkage:

When implementing large software systems consisting of several sub-systems, work is often done on the sub-systems independently until they are all functioning. The entire system is then put together by combining all of the sub-systems. There are often portions of executive routines, initializing routines, linkage programs, etc. that must refer to various parts of the sub-systems. When it is time to put all of the sub-systems together, a straight-forward method calls for one large assembly to be run on all of the sub-system source tapes and other parts that tie the package together. This can involve a good deal of computer time if the system is large enough and could be extremely time consuming if high speed tape equipment is not available. A practical aspect of the problem of large system assemblies is the length of the symbol table that is built during the assembly. Profuse use of such symbols in user programs are beneficial to the understanding of the system after it is operational.

These problems can be overcome through the use of a technique called System Linkage.

Once a sub-system has been debugged and is deemed operational, a final assembly is done, assigning the starting location of the sub-system to the beginning of some known free area of memory.

After the assembly is completed, the user scans the symbol table and picks out all of the routine entry symbols and other symbols that must be referenced from outside the sub-system. A linkage tape is then generated, using a series of parameter definition statements (4.1). It is assembled with the program that ties all of the sub-systems together. A set of labels (Li) for subroutine entries, argument storage points, counters, etc. and their corresponding octal locations (Ni) is prepared. The linkage to them is established by generating a tape of the form:

```
L1 = N1
L2 = N2
L3 = N3
  .
  .
  .
Li = Ni
*EOT
```

These linkage statements will be prepared as separate tapes for each sub-system. At final assembly time, all of the linkage tapes are assembled along with the final linking routines and executive program. When assembling these tapes, it is only necessary to read the linkage statements during PASS 1 of the assembly.

6.2 Subroutine Linkage:

The standard transfer of control to a subroutine in the GRI-909 is via a data testing instruction (3.3). Whenever a jump is about to occur, the contents of sequence counter (SC) points to the second word of the data testing instruction just before the actual jump takes place. At this point, the SC is transmitted to the trap (TRP), a hardware register associated with the data tester. The contents of the second word (or the incremented contents of the word it points to if deferred addressing is selected) is transmitted to the SC. The SC now points to the first (or entry) instruction of the subroutine called -- this instruction is executed next by the processor.

After any data testing jump is performed, the contents of the TRP provides a linkage back to the program that called the subroutine. Note that the address value in TRP is one less than that of the next instruction of the calling routine. If the subroutine does not alter the contents of the TRP, then the subroutine may return to the next step in the program from which it was called by simply executing the instruction

```
TRP TO SC
```

Since the SC is automatically incremented after this instruction is performed, an absolute return of control to the proper location is performed.

If the subroutine is going to disturb the TRP, then the contents of the TRP must be saved immediately upon entry to the subroutine. This may be done by

```
SUB:  TRP TO I
```

which places the contents of TRP into the second word of the entry instruction (i.e. SUB + 1). The subroutine may now return to the calling program via any one of the following instructions:

```
GO TO D SUB + 1
IF DEVICE TEST GO TO D SUB + 1
SUB + 1 P1 TO SC
```

Since the last instruction is a three cycle instruction, the automatic incrementation of SC is completed before the contents of location SUB + 1 is transferred to the SC; therefore, the instruction must increment the value on its way to the SC via the P1 path of the bus modifier.

A subroutine often performs some operations on one or more data items, called the arguments of the subroutine. Arguments may be passed to subroutines by loading them into specified hardware registers before calling the subroutine. Arguments may also be passed to the subroutine with a list of word values that follow the subroutine call:

```
GO TO SUB
V1,V2,V3,....Vn
```

The word values Vi may be one of the following:

a) An address of data to be operated upon
b) A data word to be operated upon
c) An address to which return is made if errors are detected by the subroutine
d) An address into which results are to be stored by the subroutine

If the subroutine entry instruction is

```
SUB:  TRP TO I
```

then the first argument V1 can be loaded into the AX register by

```
D SUB + 1 TO AX
```

The second and successive arguments can be fetched by executing similar instructions. Note that the word stored in SUB + 1 is auto incremented each time a deferred reference to it is executed.

When all of the arguments have been picked up by the subroutine, the word at SUB + 1 contains one less than the normal return address. The

```
GO TO D SUB + 1
```

or its equivalent is used for normal return of control to the calling program.

6.3 Some Fast Examples:

As an example of typical subroutine development, we present for consideration a collection of subroutines that form a teletype input package.

6.3.1. %TBO is a simple teletype output service routine. To save time in the service routine, note that the trap is not saved on entry and therefore the instruction at %TBO + 1 is an immediate mode memory to sequence counter type of jump. This jump preserves the contents of the trap register so that the subroutine returns control at %TBO + 4 by merely sending the trap to the sequence counter.

6.3.2. %TBI uses the same technique and services teletype input, one character at a time.

6.3.3. %TLO is a processor subroutine which prints a line of ASC characters, stored one character per word right justified, to the byte output routine, %TBO. In this routine, the trap will be disturbed so it is saved in %TLO + 1 immediately on entering. The call for this subroutine carries with it two arguments that are fetched by the subroutine %TLO. These arguments are the address -1 of the line of text in memory and the count of the number of characters in the line. The subroutine call looks like this:

```
GO TO %TLO
MSG - 1          ;address -1 of message
120              ;count = 80 decimal characters
```

The subroutine will continue outputing characters via %TBO until the character count runs out or a carriage return character (215) is encountered in the string. When either of these conditions is encountered, a carriage return-line feed combination is output, and the routine returns control to the main program via a deferred jump through %TLO + 1.

6.3.4. %TLI is a subroutine that fetches a line of teletype characters into a buffer area specified by the calling routine. The characters are entered in the buffer area one per word, right justified. The calling sequence requires the user to specify two arguments, the first of which specifies the buffer address -1 and the second the character count. The subroutine call, then, is:

```
GO TO %TLI
MSG - 1          ;address -1 of input buffer
120              ;count = 80 decimal characters
```

This routine, similar to %TLO, will input a string of 80 characters and exit or input a string of characters terminated by a carriage return. Note that this routine, in addition to calling %TBI for fetching characters, also calls %TBO for echoing the typed characters. The routine also has two other features. When a back arrow character is typed (←), the buffer address and count are backed up by 1. This process will continue as long as back arrows are typed until the buffer address reaches the initial buffer address, whereupon the routine ignores further back arrows. A rubout character (377) causes the entire line to be dropped and reinitialized at the beginning of the buffer. When this occurs, the routine outputs a carriage return, followed by a double line feed.

6.3.5. %TMO is a routine to output a packed message string from memory. The call for this routine requires a single argument, the starting address -1 of the message string. The subroutine terminates when a 0 word is encountered. Notice the use of a conditional jump as a subroutine call for %TBO at %TMO1-3.

The subroutine also uses an interesting trick when it unpacks the left half character of a pair of characters. The left half character is shifted to the right 8 times, moving it to the least significant 8 bits of AX. The least significant 8 bits of AX is shifted out of the link into the left half of AY. Leading zeros fill in the left half of AX. A control bit (200) is loaded into AY and shifted along with the SH character from AX. When the control bit shifts into the link, the process is complete. When it is time to print the LSH character of the word, the character may be tested for 0 because it is present in the MSH of AY, and the LSH of AY has been filled with 0's. This routine does not supply a carriage return-line feed combination at the end of the string. As an example of usage, let us write a routine to print a string of characters from memory and terminate it with a CR-LF.

```
GO TO %TMO

MSSG-1

HLT

MSSG:  '/GRI-909 /, 106612,0 ;106612 = CR-LF
```

```
ØØ1                          ;TELETYPE BYTE OUTPUT
ØØ2 ØØØØØ 77 ØØ1Ø Ø2 %TBO:      SKIP IF TTO ORDY ;READY TO OUTPUT?
ØØ3 ØØØØ1 Ø6 ØØ1Ø Ø7            I .-2 TO SC      ;NO,WAIT
    ØØØØ2 177777
ØØ4 ØØØØ3 11 ØØØØ 77            AX TO TTO        ;YES,DELIVER CHARACTER
ØØ5 ØØØØ4 Ø3 ØØØØ Ø7            TRP TO SC        ;RETURN TO PROGRAM
ØØ1                          ;TELETYPE READER/KEYBOARD BYTE INPUT
ØØ2 ØØØØ5 Ø2 ØØØ1 77 %TBI:      STRT TO TTI      ;START READER
ØØ3 ØØØØ6 77 1ØØØ Ø2            SKIP IF TTI IRDY ;INPUT READY?
ØØ4 ØØØØ7 Ø6 ØØ1Ø Ø7            I .-2 TO SC      ;NO,WAIT
    ØØØ1Ø ØØØØØ5
ØØ5 ØØØ11 Ø2 1ØØØ 77            CLIF TO TTI      ;YES,CLEAR INPUT FLAG
ØØ6 ØØØ12 77 ØØØØ 11            TTI TO AX        ;FETCH CHARACTER TO AX
ØØ7 ØØØ13 Ø3 ØØØØ Ø7            TRP TO SC        ;RETURN TO MAIN PROGRAM
ØØ1                          ;TELETYPE LINE OUTPUT
ØØ2 ØØØ14 Ø3 ØØ1Ø Ø6 %TLO:      TRP TO I         ;SAVE TRAP FOR RETURN
    ØØØ15 ØØØØØØ
ØØ3 ØØØ16 Ø2 ØØØØ 13            ADD
ØØ4 ØØØ17 Ø6 ØØØ1 11            D %TLO+1 TO AX ;FETCH BUFFER ADDR-1
    ØØØ2Ø ØØØØ15
ØØ5                          ;                    FROM CALL
ØØ6 ØØØ21 Ø6 ØØØ1 12            D %TLO+1 TO AY ;FETCH COUNT FROM CALL
    ØØØ22 ØØØØ15
ØØ7 ØØØ23 12 Ø11Ø 12            C AY P1          ;2'S COMP THE COUNT
ØØ8 ØØØ24 11 ØØØØ Ø6            AX TO .+7        ;SAVE BUFFER ADDR-1
    ØØØ25 ØØØØ33
ØØ9 ØØØ26 12 ØØØØ Ø6            AY TO .+11       ;SAVE -COUNT
    ØØØ27 ØØØØ37
Ø1Ø ØØØ3Ø Ø6 ØØ1Ø 12            I -215 TO AY     ;CR CHAR
    ØØØ31 177563
Ø11 ØØØ32 Ø6 ØØ11 11            ID Ø TO AX       ;FETCH NEXT CHAR IN LINE
    ØØØ33 ØØØØØØ
Ø12 ØØØ34 ØØ Ø1ØØ Ø3            GO TO %TBO       ;OUTPUT BYTE FROM AX
    ØØØ35 ØØØØØØ
Ø13 ØØØ36 Ø6 Ø11Ø Ø6            I Ø P1           ;INCREMENT COUNT
    ØØØ37 ØØØØØØ
Ø14 ØØØ4Ø ØØ ØØ1Ø Ø2            SKIP IF BOV      ;COUNT=Ø?
Ø15 ØØØ41 13 Ø11Ø Ø3            IF AO NEZ GO TO .-7 ;CR?NO, CONTINUE LINE
    ØØØ42 ØØØØ32
Ø16 ØØØ43 12 Ø11Ø 11            C AY P1 TO AX    ;END OF LINE,FETCH CR CHAR
Ø17 ØØØ44 ØØ Ø1ØØ Ø3            GO TO %TBO       ;PRINT CR
    ØØØ45 ØØØØØØ
Ø18 ØØØ46 Ø6 ØØ1Ø 11            I 212 TO AX      ;FETCH LF CHAR TO AX
    ØØØ47 ØØØ212
Ø19 ØØØ5Ø ØØ Ø1ØØ Ø3            GO TO %TBO       ;PRINT LF
    ØØØ51 ØØØØØØ
Ø2Ø ØØØ52 ØØ Ø1Ø1 Ø3            GO TO D %TLO+1 ;RETURN TO MAIN PROGRAM
    ØØØ53 ØØØØ15
ØØ1                          ;TELETYPE LINE INPUT
ØØ2 ØØØ54 Ø3 ØØ1Ø Ø6 %TLI:      TRP TO I         ;SAVE TRAP FOR RETURN
    ØØØ55 ØØØØØØ
ØØ3 ØØØ56 Ø2 ØØØØ 13            ADD
ØØ4 ØØØ57 Ø6 ØØØ1 11            D %TLI+1 TO AX ;FETCH BUFFER ADDR-1
    ØØØ6Ø ØØØØ55
ØØ5                          ;                   FROM CALL
ØØ6 ØØØ61 Ø6 ØØØ1 12            D %TLI+1 TO AY ;FETCH CHAR COUNT
    ØØØ62 ØØØØ55
ØØ7                          ;                   FROM CALL
ØØ8 ØØØ63 12 Ø11Ø 12            C AY P1          ;2'S COMP COUNT
```

FAST TELETYPE INPUT PACKAGE 71-44-002L

```
009 00064 11 0010 06            AX TO I          ;SAVE BUFF ADDR-1
    00065 000000
010 00066 12 0010 06            AY TO I          ;SAVE -COUNT
    00067 000000
011 00070 11 0000 06            AX TO %TLI3+5   ;INITIALIZE CURRENT BUFFER
    00071 000161
012 00072 12 0000 06            AY TO %TLI3+3   ;ADDR AND BUFFER COUNT
    00073 000157
013 00074 00 0100 03 %TLI1:     GO TO %TBI       ;FETCH A BYTE TO AX
    00075 000005
014 00076 06 0010 12            I 200 TO AY      ;OR A 1 IN CHANNEL 8
    00077 000200
015 00100 02 1100 13            OR
016 00101 13 0000 11            AO TO AX         ;AX NOW NO PARITY CHAR
017 00102 02 0000 13            ADD
018 00103 00 0100 03            GO TO %TBO       ;ECHO THE CHAR READ
    00104 000000
019 00105 06 0010 12            I -337 TO AY     ;CHECK FOR BACK ARROW CHAR
    00106 177441
020 00107 13 0110 03            IF AO NEZ GO TO %TLI2-4
    00110 000132
021 00111 06 0000 11            %TLI3+5 TO AX   ;BACK ARROW, DELETE LAST
    00112 000161
022                  ;                             CHAR READ
023 00113 11 0110 11            C AX P1          ;-CURRENT BUFFER ADDR
024 00114 06 0000 12            %TLI1-7 TO AY   ;INITIAL BUFFER ADDR
    00115 000065
025 00116 13 0100 03            IF AO ETZ GO TO %TLI1 ;STILL FIRST CHAR,
    00117 000074
026                  ;                               CONTINUE
027 00120 11 0010 11            C AX             ;BACK UP BUFFER ADDR 1
028 00121 11 0000 06            AX TO %TLI3+5   ;RESET BUFF ADDR TO
    00122 000161
029                  ;                             BUFF ADDR-1
030 00123 06 0000 11            %TLI3+3 TO AX   ;FETCH CURRENT COUNT
    00124 000157
031 00125 00 0010 12            C ZERO TO AY     ;FETCH -1 TO AY
032 00126 13 0000 06            AO TO %TLI3+3   ;SET CURRENT COUNT=COUNT-1
    00127 000157
033 00130 00 0100 03            GO TO %TLI1      ;CONTINUE INPUT
    00131 000074
034 00132 06 0010 12            I -377 TO AY     ;NOT BACK ARROW
    00133 177401
035 00134 13 0110 03            IF AO NEZ GO TO %TLI3 ;IS IT A RUB OUT?
    00135 000154
036 00136 06 0010 11 %TLI2:     I 215 TO AX      ;YES FETCH A CR TO AX
    00137 000215
037 00140 00 0100 03            GO TO %TBO       ;AND PRINT
    00141 000000
038 00142 06 0010 11            I 212 TO AX      ;FETCH A LF TO AX
    00143 000212
039 00144 00 0100 03            GO TO %TBO       ;AND PRINT
    00145 000000
040 00146 06 0000 11            %TLI1-7 TO AX   ;RE-INITIALIZE
    00147 000065
041                  ;                             BUFFER PARAMETERS
042 00150 06 0000 12            %TLI1-5 TO AY
    00151 000067
043 00152 00 0100 03            GO TO %TLI1-4   ;START LINE AGAIN
```

```
    00153 000070
044 00154 06 0010 12 %TLI3:   I -215 TO AY    ;NOT BACK ARROW OR RO
    00155 177563
045 00156 06 0110 06          I 0 P1          ;INCREMENT CHAR COUNT
    00157 000000
046 00160 11 0011 06          AX TO ID 0      ;STORE CHAR IN BUFFER
    00161 000000
047 00162 13 0100 03          IF AO ETZ GO TO .+7 ;CR?
    00163 000171
048 00164 00 0011 02          SKIP IF NOT BOV ;NO,IS BUFFER FULL?
049 00165 00 0100 03          GO TO %TLI2     ;YES,FULL ACTS LIKE RO
    00166 000136
050 00167 00 0100 03          GO TO %TLI1     ;NO,GET NEXT CHAR
    00170 000074
051 00171 06 0010 11          I 212 TO AX     ;CHAR=CR, ALL DONE
    00172 000212
052 00173 00 0100 03          GO TO %TBO      ;PRINT A LF
    00174 000000
053 00175 00 0100 03          GO TO %TBO      ;PRINT A LF
    00176 000000
054 00177 06 0000 11          %TLI1-5 TO AX   ;COMPUTE NUMBER OF
    00200 000067
055                  ;                         CHAR READ
056 00201 06 0000 12          %TLI3+3 TO AY
    00202 000157
057 00203 11 0110 11          C AX P1
058 00204 13 0000 11          AO TO AX        ;AX=NO. OF CHAR READ
059                  ;                        INTO BUFFER
060 00205 00 0101 03          GO TO D %TLI+1 ;RETURN TO MAIN PROGRAM
    00206 000055
001                  ;TELETYPE MESSAGE OUTPUT
002 00207 03 0010 06 %TMO:    TRP TO I        ;SAVE TRAP FOR RETURN
    00210 000000
003 00211 06 0001 11          D .-1 TO AX     ;FETCH ADDR-1 OF MESSAGE
    00212 000210
004                  ;                          FROM CALL
005 00213 11 0000 06          AX TO %TMO1     ;AND SAVE
    00214 000234
006 00215 06 0001 11          D %TMO1 TO AX   ;FETCH FIRST PAIR OF CHAR
    00216 000234
007 00217 11 0100 03          IF AX ETZ GO TO %TMO+1 ;DONE IF WRD=0
    00220 000210
008 00221 06 0010 12          I 200 TO AY
    00222 000200
009 00223 02 0001 00          CLL
010 00224 11 1100 11          AX R1           ;MOVE LH TO RH(AX)
011 00225 12 1100 12          AY R1           ;MOVE RH(AX) TO LH(AY)
012 00226 00 0100 02          SKIP IF LNK     ;DONE?
013 00227 00 0100 03          GO TO .-4       ;NO, CONTINUE
    00230 000223
014 00231 11 0110 03          IF AX NEZ GO TO %TBO ;YES,OUTPUT RIGHT
    00232 000000
015                  ;                        HALF IF NEZ
016 00233 06 0000 11          0 TO AX         ;FETCH RIGHT HALF OF WORD
    00234 000000
017       000234     %TMO1=.-1
018 00235 12 0110 03          IF AY NEZ GO TO %TBO ;OUTPUT RIGHT
    00236 000000
019                  ;                        HALF IF NEZ
```

```
020 00237 00 0100 03        GO TO %TMO+6    ;DO NEXT WORD
    00240 000215
001                         * END
```

APPENDIX A

OPERATING INSTRUCTIONS

Passes 1, 2, and 3 of the assembler perform user symbol definition, object code output, and listing output respectively. After Pass 1, the assembler will continue to Pass 2 and then to Pass 3. Any time after Pass 1 has been completed, however, the assembler may be re-started and either Pass 2 or 3 selected.

I. Load the assembler with the Absolute Loader.

II. Transmit "0" to SC.

III. Set console switches as follows:

Bit 15 selects source input device
Bit 14 selects object output device
Bit 13 selects listing output device

UP = High-speed (paper tape)

DOWN = Low-speed (teletype)

Bits 1-0 select Pass

```
01 = Pass 1
10 = Pass 2        if Pass 1 previously completed
11 = Pass 3
```

IV. Ready source tape in reader (if TTI, set reader control to START).

V. Press START.

The assembler will halt after encountering an *EOT command. Mount the next tape segment and press CONTINUE.

The assembler will halt after encountering an *END command. If another pass is either desired or necessary, remount the source tape (or the first segment thereof) and

a) Press CONTINUE to proceed to the next pass, or

b) Select the next pass by starting at II, above.

Turn the appropriate punch on before starting Pass 2 and turn it off after the pass is completed.

NOTES:

1) If bits 14 and 13 have different settings, then both the object code and the listing will be generated during Pass 2. The listing may be punched on the high speed device and later printed off-line.

2) If the user wishes to type in instructions to see how various forms are assembled, proceed as follows:

   a) Perform I and II, above.
   b) Perform III, selecting low-speed I/O and Pass 3.
   c) Press START.

Statements (each followed by a carriage-return) may now be typed on the TTY keyboard. The characters typed are not echoed on the teleprinter as they are struck. After a statement is terminated (carriage-return), the assembler responds with the corresponding listing output.

APPENDIX B

STANDARD SYMBOL TABLE

The following are the pre-defined parameters that are part of the assembler's symbol table, to which user symbols are added.

INTENDED CATEGORY	SYMBOL	VALUE	MEANING
Device Addresses	ISR	4	Interrupt Status Register
	TRP	3	Trap Register
	SC	7	Sequence Counter
	SWR	10	Console Switch Register
	AX	11	Arithmetic Operator X-register
	AY	12	Arithmetic Operator Y-register
	AO	13	Arithmetic Operator
	MSR	17	Machine Status Register
	HSR	76	High-speed Reader
	HSP	76	High-speed Punch
	TTI	77	Teletype Input
	TTO	77	Teletype Output
Status Test Codes	AOV	2	Arithmetic Overflow
	NOT	1	Negation of Test Results
	IRDY	10	Input-ready flag
	ORDY	2	Output-ready flag
	LNK	4	Bus Modifier Link
	BOV	2	Bus Overflow
	NPFL	10	No Power Failure
Transmission Path Codes	P1	1	Increment
	L1	2	Shift Left 1 bit
	R1	3	Shift Right 1 bit
Pulse Output Codes	CLL	1	Clear Link
	STL	2	Set Link
	CML	3	Complement Link
	HLT	4	Halt Machine

INTENDED CATEGORY	SYMBOL	VALUE	MEANING
Pulse Output Codes (continued)	ADD	0	Select AO "ADD"
	AND	4	Select AO "AND"
	OR	14	Select AO "OR"
	XOR	10	Select AO "XOR"
	STRT	1	General Start Pulse
	CLIF	10	Clear Input Flag
	CLOF	2	Clear Output Flag
	ICF	1	Interrupt Control OFF
	ICO	2	Interrupt Control ON
Data Test Codes	ETZ	2	Equal to Zero
	NEZ	3	Not Equal to Zero
	GTZ	7	Greater Than Zero
	GEZ	5	Greater Than or Equal to Zero
	LTZ	4	Less Than Zero
	LEZ	6	Less Than or Equal to Zero
Pseudo Codes	NOP	0	No Operation

gri-909

gri GRI Computer Corporation

320 NEEDHAM STREET, NEWTON, MASSACHUSETTS 02164