***SST* Software, Inc.**
P.O. Box 26
Cedarburg, WI 53012
(414) 771-8415

SST EXPANDED BASIC COMPILER SYSTEM

by

***SST* SOFTWARE, INC.**

TABLE OF CONTENTS

INTRODUCTION

The ***SST*** Expanded Basic compiler system is divided into five programs *EDITOR/EX, EDITOR, COMPILER, LOADER* and *FASTLOAD*, for efficiency and to maximize use of available memory. Through this technique, it is possible to provide most of the commands available in Extended Basic plus many more.

The ***SST*** Compiler includes as many commands as possible while still affording you the opportunity to write fairly large program segments. Up to 470 line Basic programs can be compiled at one time. A number of programs of this length may be in memory and linked together as one large system. To allow for programs of this length all floating point arrays are stored in the T.I. console memory.

Machine language allows you to write very efficient computer programs. However, it is more difficult and time consuming to code and debug in machine language than in Basic.

The ***SST*** *EXPANDED BASIC COMPILER* system converts a *BASIC* program directly into a machine language program, completely bypassing the assembly language stage. With the ***SST*** *COMPILER*, the power and speed of machine language are available without actually having to write the program in machine language.

With *BASIC* it is easy to write a program, but *BASIC* is slow in execution. This is because each time a *BASIC* command is encountered, the *BASIC* interpreter must translate the command into machine language before executing the command. As a result the same statement may be translated several thousand times in the same program. With the ***SST*** *EXPANDED BASIC COMPILER*, a statement will be translated only once. Once the ***SST*** *COMPILER* has finished translating a *BASIC* program, the machine language code will be stored on disk, and will never need to be compiled again. The machine language code can be loaded into memory whenever it is required.

The ***SST*** *EXPANDED COMPILER* system provides many of the advantages of both *BASIC* and machine language. The ***SST*** *EDITOR* is designed to allow you to write and debug *BASIC* programs using the editing and debugging features of *T.I. BASIC*. Once the *BASIC* program is found to be correct, the Editor program is run and the *BASIC* program is stored on disk for compiling. The ***SST*** *BASIC COMPILER* then translates the *BASIC* code into machine language code and stores it permanently on disk. Anytime the program is to be run, the ***SST*** *LOADER* will load it into memory. The ***SST*** *EDITOR/EX* performs the same function as the Editor using the Extended Basic module.

To increase speed and efficiency, we have included the option to code in integer arithmetic. The *T.I. BASIC* language works only in 8 byte floating point arithmetic. While this provides great accuracy, it also causes slow execution. Therefore, to increase the speed of your program, and to conserve memory, we encourage you to do as much computation as possible using integer arithmetic. The *T.I.* processor has machine language integer addition, subtraction, multiplication and division commands. These instructions are extremely fast. In floating point, these operations must be done by subroutines, and are, therefore, slow. In addition, an integer variable requires only 2 bytes, while a floating point variable requires 8 bytes of memory. It should be noted that mathematical functions like *SIN(X)* are extremely slow. These functions are provided in the *T.I.* console and consist of large subroutines written in the Graphics Programming Language. To increase speed and efficiency we have also included the option to work in 6 byte or 4 byte floating point arithmetic. Depending upon the program this can increase speed anywhere from 6 to 50 percent.

We have also included Bit Mapping mode for high resolution graphics. This allows you to access each of the approximately 49000 pixels of the screen.

DETAILED DISCUSSION OF PROGRAMS

EDITOR:

The *EDITOR* allows the user to write a *BASIC* program using the editing and debugging features of the console. This permits you to test run the *BASIC* program before it is compiled. The Mini-Memory or Editor/Assembler module must be in place for the Editor, Compiler and Loader programs.

How to use the *EDITOR* program:

1. Load the Editor, using old *DSK1.EDITOR*.
2. Enter the *BASIC* program to be compiled, using line numbers 11 to 32000. You may use the editing features of *T.I. BASIC*. Be sure to end the *BASIC* program with a *STOP* statement.
3. The program may be tested by using the command *RUN*, with the starting line number. (Example: *RUN 11*). Errors may be corrected in the usual fashion. (This may not be possible if you are using commands not available in *T.I. BASIC*).
4. When the program is correct, save it on disk for future reference. (*SAVE DSK1.name*) This saves the editor and the program to be compiled.
5. Before running the *EDITOR* program, the *EDITOR* must be resequenced, using the command *RES 1,1*. Once this has been completed, simply type: *RUN*.
6. The ***SST*** title screen will appear, and the program will ask for an output file name. The file name will be *DSK1.name*. As the *EDITOR* is processing each line of *BASIC* code, the following statement will appear on the screen:

   *NOW PROCESSING LINE*__(starting with 11)

   When the *EDITOR* encounters a call statement, the name will be printed under *"NOW PROCESSING LINE"*. This can help you keep track of the locations in your program. When the *EDITOR* encounters the *STOP* statement it will terminate and tell you the number of bytes used by your BASIC program. This can give you a rough idea of the memory needed for your compiled program. The resulting file will be used by the ***SST*** *EXPANDED BASIC COMPILER*.

EDITOR/EX:

This program performs the same function as *EDITOR* except that it is for the Extended Basic module. You may use this editor for debugging your program. *EDITOR/EX* contains most of the call statements used by *SST* with the exception of *SUBIN* and *SUBOUT*. Although not all will function in Extended Basic, they may be called and no error statement will be given. In the summary of commands list, the commands which will cause no change as your program is being tested are indicated by an *"NA"* after the command. All of the simulated subroutines are located at the end of *EDITOR/EX*. In some cases you may want to change the simulated routine. Examples for doing this are the *USERA...USERE*, the *SCHARA*, and the *CALL OPEN* routine. The *CALL OPEN* routine only opens a *DISPLAY/VARIABLE 80 FILE*. You may want to change this to a different format. Other commands to check are *COINCA* and *DISTANCEA*. There are several forms of these commands. Check the detailed description of these commands for simulating them properly.

COMPILER:

This program takes the code generated by the *EDITOR* program and translates it into machine code.

To operate the *COMPILER* program:

1. Load the *COMPILER* program using *OLD DSK1.COMPILER.CALL FILES(1).NEW* must have been performed since the last time the computer was turned on.
2. When the cursor appears after the long pre-scan, immediately type *RUN*. If you *LIST* or try to *EDIT* the *COMPILER*, the program may not be executable, and should be reloaded again. After *RUN* has been entered, there will be approximately a 70 second pre-scan before the title screen appears.
3. The program will ask for *"INPUT, OUTPUT"*. At this time enter for the *INPUT* file name the name which was used in the *EDITOR*. Then choose an *OUTPUT* name for the compiled code that will be generated. The input file will be loaded from the input file into *MEMORY EXPANSION*.

   There will be a 30 second delay after the input file has been loaded.

4. At this point, the compiler will ask you to *ENTER* the starting location for the machine code and the name of the program. Enter a decimal number of an *EVEN* location in *MINI-MEMORY* or *MEMORY EXPANSION*. The following are the areas of memory that are available.

MINI-MEMORY	*28952 to 32700*	*(HEX. 7118 to HEX. 7FBC)*
MEMORY EXPANSION	*-24064 to -35[illegible]*	*(HEX. A200 to HEX. ~~FF66~~ FFC0*

   The name that is entered must be 1 to 6 characters in length, and must follow the characteristics of T.I. variables.

   Example of input: -8000,TEST (ENTER)

   The loader will start the program at location -8000 in *MEMORY EXPANSION*, and give it a name *(TEST)* to be used in *[CALL LINK("TEST")]* command.

5. The *COMPILER* will display the following as each line is b[illegible] compiled.

   COMPILING (line #) (instruction location)

   The *COMPILER* checks for several types of errors:

   VARIABLE NOT FOUND (Variable was not defined with a *LET* statement)

   PROGRAM TERMINATED AT LINE 2290 (instruction not allowed)

   If any of these error messages appear, the *COMPILER* will stop and you must check the source file for errors and rerun the *EDITOR* and *COMPILER*.

   IMPORTANT: The compiler does not check for all errors!

   Example:

   ```
   11 LET C@=2
   12 PRINT C
   13 STOP
   ```

   In this case, even though (C) is not defined, *"VARIABLE NOT FOUND"* will not appear, and the program will finish compiling. However, the program will not run properly. No error message is given because this error should be detected by test runs before the program is compiled. In T.I. Basic, "0" will be printed for variable C. In a similar manner, a compiled program does not give error messages for the mathematical functions like *SQR(X)*, where "X" is a negative number. These errors should be detected by test runs in *T.I. BASIC*.

6. If there are no errors the *COMPILER* will end when a *STOP* is encountered. The last statement the *COMPILER* will display is:

   LAST ADDRESS USED (address) (starting address of first executable statement minus 6)
   DONE Y/N (N: if you are linking programs together with same variable set.

   If you wish to compile another program using the same list of variables, you should *ENTER "N"*. This option is useful if you want to write programs in blocks of up to 470 lines and have them all linked together using the same variable list. You may do this as many times as you wish. However, when you use the *LOADER* only seven Compiled programs can be called from TI Basic. Within any compiled program you may branch to another compiled section by using the *CALL LINKER* routine. If you wish to call any section from *T.I. BASIC*, you may do this with the *CALL LINK* command. The first program compiled must have all the variables and constants defined in it. Subsequent programs must not have *LET, DIM OR DISPLAY* commands in them. Each program must be terminated with a STOP. The first executable instruction, located after all of the *LET* statements, is found by adding 6 to the starting address given by the *COMPILER*.

If you wish to compile another program using a different set of variables, you should *ENTER "Y"* and then *RELOAD* the Compiler from disk.

Compiled object code is now in the *OUTPUT* file(s) you had specified. If an error should occur in compiling, reload the *COMPILER* before recompiling.

LOADER:

The *LOADER* uses the code saved in the file generated by the *COMPILER*. In order to use this program, follow the steps below.

1. *ENTER* the *LOADER* program.
2. The *LOADER* may be executed by typing *RUN (press ENTER)*. The title screen will appear. You must have the disk with ***SST*** utilities *MOVE, DUMP* and *USERS* in *DSK1* since they will be loaded at this time.

   You will be asked: *"OBJECT FILE NAME?"* [file name] Enter the file name used as the *OUTPUT* file in the Compiler program.
3. You will be asked *"TI 99/4A Y/N"*. If you are using a *TI 99/4A* console enter *"Y"*, if a *TI 99/4* console enter *"N"*.
4. While the *LOADER* is running it is putting the object code into *MINI-MEMORY* or *MEMORY EXPANSION*, depending on the address given in the *COMPILER* program. The name chosen in the *COMPILER* is also being loaded into *MINI-MEMORY* or *MEMORY EXPANSION*.
5. After the code and the name are loaded, the program will ask:

   WOULD YOU LIKE A STARTED RUN: (Y/N)

   If the response is "Y", the compiled program will be run immediately.

   If the program contains more than 3200 bytes of floating point arrays, the compiled program may not return to the *LOADER* program. In that case, choose "N" and run the compiled program by the following:

   CALL LINK("name") [ENTER]

   Calculating the number of bytes used in the console is the user's responsibility. To determine the number of bytes used for floating point arrays, multiply the number of elements in the array by eight.
6. If you *ENTER "N"* for the question *"DONE Y/N"* you will be allowed to enter another compiled program into memory. Only 7 names are allowed. If you load more than 7, only the first seven names are used.

It is recommended that the user make copies of all *EDITOR*, *LOADER* and *UTILITY* programs. The *COMPILER* program, because of its structure, may not be copied or edited. Do not take the write protection off of the disk.

FASTLOAD

FASTLOAD is a program that will take the contents of different areas of memory and put them into the correct form for the Assembler. This program allows you to bypass the ***SST*** *LOADER* and only use the *CALL LOAD* command. This increases the loading speed of a compiled program. It is also useful when you want to define *USERA...USERE COMMANDS*.

HOW TO USE THE PROGRAM:

1. Use the ***SST*** *LOADER* to load your compiled programs into memory. Load the program you wish to call from TI Basic last. You are allowed only one name for every use of this program. At the time of compiling you should note the starting and ending locations of your programs.
2. Load the ***SST*** *FASTLOAD* program into memory and enter "RUN".
3. Make certain you have a nearly empty disk in the disk drive. This program may require almost a complete disk. You will be asked for a file name. The name you choose will be the source file created for the Assembler. (e.g. *DSK1.TEST*)
4. You will be asked the name of your program to be used in future *CALL LINK* commands. This can be any one to six letter name. The name given will be assigned the location of the program last loaded by the ***SST*** *LOADER*.
5. You will be asked the number of sections to be transferred. Enter the number of different areas of memory to be connected. (up to 7)
6. You will then be asked the beginning and ending address of each section. These addresses must be even numbers. You should enter the addresses starting high in memory and going down. That is, if you have these sections the starting address might be:

STARTING	ENDING
9846	15999
-24000	-20001
-8000	-7001
7. You will then be asked if all the entries are correct. If not, Enter "N". You will then return to the top of the address area. A "Y" will begin the conversion.

On comple of the program you will have a source file that contains the name of the program, a number of *AORG* statements, a number of data statements, and a branch instruction to the starting location. You may now take this source file and assemble it using the *EDITOR/ASSEMBLER* module.

To load the program you enter:

```
        CALL INIT
        CALL LOAD("DSK1.MOVE","DSK1.DUMP","DSK1.USERS",
                  "DSK1.your program")
```

The first three programs must always be loaded into memory with your program. These programs contain utilities necessary for your program. These three programs may be copied onto another disk by use of the *DISK MANAGER MODULE*.

By using the *FASTLOAD* program on only your compiled program and using the loading sequence above, your program may be loaded using Mini-Memory or Editor/Assembler. However, if you wish to restrict yourself to only Mini-Memory or E/A, you may use the *FASTLOAD* program to convert the three Utility packages. To do this you must have loaded your program with the desired module in place and have *FASTLOAD* in memory with this module in place. The advantage of this is that you need only one object file in the *CALL LOAD*. However, you must always use the module that was in place when *FASTLOAD* was run.

If you wish to convert the utility packages the addresses of the first two sections are:

START	END
9846	16001
-24576	-24065

The address -24065 may be changed if you add *USER* defined functions. The next sections may be your program locations.

After the source file is assembled the loading procedure is:

```
        CALL INIT
        CALL LOAD("DSK1.your program")
```

CALL USERA...USERE

These are call statements reserved for your definition and your use. At present they contain no code. You may define them permanently be writing an assembly language routine or by compiling a program and making it part of the compiler by using the *FASTLOAD* program.

How to define *USERA...USERE*:

You may first change the name to any 3 to 10 character name you desire. This name must be changed in the *EDITOR* or *EDITOR/EX* program at line 30150. The name must take up 10 spaces. If the name is less than 10 characters, you must fill the rest with blanks. The names must be unique.

EXAMPLE: *MYPROGRAM(SPACE)*

You are provided with an assembly language source file called *USER*. This program has been assembled and is loaded in the *SST LOADER* program at line 30009, with an object file name *USERS*. When using the *SST LOADER*, the utility program must be in *DSK1*.

The source file contains a series of labeled branch statements. If you plan to write your own assembly language routine for a *USER* call statement, you may do so just before the *END* directive. You must change a corresponding *USER* branch instruction to branch to the beginning of your routine. Make certain that when you compile a program, you do not use the area occupied by your *CALL USER* routine. You may now assemble *USER* into the object file named *USERS*. Anytime you do a call to *USER* (etc.), your program will be executed.

Instead of writing an assembly program you may use a compiled program. You must compile and load the program into memory and run the *FASTLOAD* program on it. You may then transfer it to a source file *USER*. That is, use the *INSERT* command of the *EDIT* program in *EDITOR/ASSEMBLER* to put it into *USER*. You must then change one of the *USER* branches to point to an executable instruction, remove the *DEF* name from your source file, take out an extra *END* and assemble the *USER* program into the object file *USERS*. The first executable instruction is 6 bytes below the starting instruction given by the Compiler.

PASSING PARAMETERS WITH THE USER DEFINED ROUTINES:

Passing parameters with the *USER* defined call statement, when the *USERA...USERE* are written in assembly language, is performed with register 12. R12 is used as a stack pointer and reserved along with R13,R14,R15 by the *SST EXPANDED BASIC COMPILER*.

When you call one of your *USER* functions, the compiler in descending order will store the location of the next line(or first parameter) on the stack, and the contents of R12 will be pointing to this location on the stack. The program then branches to the appropriate branch instruction in source file *USER*. It will then branch to the location stated in the branch instruction. This must be the starting location of your program.

You may now use R12 to find the location where the pointers to your parameters are stored.

This is done by an instruction such as: *MOV *R12,R4*. *R4* now contains the location in your main program where the pointer to the parameters is stored. The following instruction can then be used.

*MOV *R4+,R5.*

R5 now contains the location in memory where the first parameter is stored. *R4* is incremented by 2 and is now pointing to the next instruction (or the next parameter)

On completion of your routine you must load *R1* with the number of parameters that were passed multiplied by 2, and then branch to 2E9A.

EXAMPLE:

```
                LI   R1,4    2 PARAMETERS PASSED
                B    @>2E9A
```

Your subroutine will then branch to the next executable instruction of your main program.

For example, the way you would write the *CALL PEEK* instruction, if it were not included, is:

1. Change one of the *USERA...USERE* names in the *EDITOR* or *EDITOR/EX* to *"PEEK "*
2. In *USER*, the source file, it would appear as follows:

```
        .
        .
        .
USERA   B       @PEEK
        .
        .
PEEK    MOV     *R12,R4     LOCATION OF POINTERS
        MOV     *R4+,R2     LOCATION OF ADDRESS OF VARIABLE
        MOV     *R4,R1      LOCATION OF SECOND PARAMETER
        CLR     *R1         CLEAR SECOND VARIABLE
        MOV     *R2,R2      PUT ADDRESS IN R2
        MOVB    *R2,*R1     MOVE CONTENTS OF THAT ADDRESS
        SWPB    *R1         INTO SECOND VARIABLE
        LI      R1,4        LEAVE SUBROUTINE
        B       @>2E9A
```

In the *T.I. BASIC* the command would be *CALL PEEK(A@,B@)*.
See *SUBIN* and *SUBOUT* for passing parameters with a compiled Basic Program.

CALLING COMPILED PROGRAMS FROM BASIC

You may call up to 7 compiled programs from *T.I. BASIC*. However, there are several restrictions.

1. Your compiled programs cannot use *CALL SPRITEMODE* or *CALL PLOTMODE*. Unpredictable results may occur if you use these commands.
2. If your compiled programs contain floating point arrays, you must make sure your *T.I. BASIC* program and the floating point arrays do not overlap. That is, the Compiler stores all floating point arrays in the main console memory. To avoid any conflicts you can relocate where your *T.I. BASIC* program will reside. This is done by first calculating how many bytes your floating point arrays will take (8 times the number of elements). Then you must follow the procedure outlined below.

   A. You must first do a *CALL FILES(1)* followed by a *NEW* command.

   B. Enter the command.

   CALL LOAD(-31952,X,Y,X,Y)

   Where
   X=59 - INT(bytes in arrays/256)
   Y=227 - [Bytes in array -INT(Bytes in array/256)*256]
   This creates a buffer in memory.

   C. Put in any one line of code.

   D. Use the *SAVE* command (SAVE DSK1.name) This saves the buffer you have just created (step B).

   E. Turn the computer off and on. Do a *CALL FILES(1)* followed by a *NEW* and then reload the program you have just saved (step B).

   F. You may now write your *T.I. BASIC* calling program without fear of overlapping with your floating point arrays in the compiled programs. You may not resequence your programs after this process.

EXAMPLE: If you have an array with 800 elements you would proceed as follows. Recall that the *SST EXPANDED BASIC COMPILER* uses option base zero. Number of bytes in array=8×800=6400.

A. CALL FILES(1)
NEW

B. X=59-INT(6400/256)
=59-25
=34
Y=227-(6400-INT(6400/256)*256)
=227
CALL LOAD(-31952,34,227,34,227)

C. Enter 100 *STOP*

D. SAVE DSK1.name

E. Turn console off and then on.
CALL FILES(1)
NEW
OLD DSK1.name

F. You may now write your *T.I. BASIC* calling program.

SUMMARY OF AVAILABLE OPERATORS, FUNCTIONS AND COMMANDS

The available *BASIC* operators, functions and commands are summarized in *TABLES 1-3*.

TABLE 1: Available floating point operators and functions in the ***SST*** *EXPANDED BASIC COMPILER*

OPERATOR	***FUNCTION OF OPERATOR***
+	Add
-	Subtract
*	Multiply
/	Divide
ABS	Absolute Value
ATN	Arctangent
COS	Cosine
EXP	e^x
INT	Returns the greatest integer contained in the value
LOG	Natural log of a number
SIN	Sine
SQR	Square Root
TAN	Tangent

TABLE 2: Available integer operators and functions in the ***SST*** *EXPANDED BASIC COMPILER*

OPERATOR	***FUNCTION OF OPERATOR***
+	Add
-	Subtract
*	Multiply
/	Divide
ABS	Absolute Value

TABLE 3: Summary of available commands

BASIC COMMAND	***SUMMARY OF COMMAND***	***EXAMPLES***
LET	All variable names must be declared and defined at the beginning of the program. An integer variable must be followed by an @ symbol and a string variable by a **$**	100 LET **A$="TEST"** 110 LET **A=5.6** 120 LET **A@=1**

BASIC COMMAND	*SUMMARY OF COMMAND*	*EXAMPLES*
INPUT	A number or string may be entered from the keyboard.	100 INPUT A 110 INPUT B@ 120 INPUT C$
IF	A jump will be performed if the variable is <=0, > =0 or =00	100 IF A<=0 THEN 100 110 IF B@>=0 THEN 200 120 IF C@=00 THEN 300
GOTO	Unconditional jump	100 GOTO 1000
GOSUB	Jump to a subroutine	100 GOSUB 1500
RETURN	Needed to transfer control from a subroutine to the location following the GOSUB.	2000 RETURN
REM	Allows REMARKS to be entered into the program.	100 REM By ***SST***
PRINT	PRINT the value of the variable on the screen	100 PRINT A 110 PRINT B$
FOR-NEXT	The FOR-NEXT statements are used for looping. The parameters must be integer variables. Integer arrays are not allowed	100 FOR I@=K@ TO N@ 110 PRINT I@ 120 NEXT I@
STOP	A STOP statement is used only to signify the physical end of the *BASIC* program.	1000 STOP
DIM	Allows the dimensioning of either integer or floating point variables. One or two subscripts may be used. Option base 0 is assumed.	100 DIM A(10) 110 DIM B@(10,10)
DISPLAY	Must be used in place of DIM statements for large arrays.	100 DISPLAY A(1400) 110 DISPLAY B@(8000)
FLOAT(floating variable, integer variable)	Converts an integer into a floating point value and stores that value in a floating point variable.	100 CALL FLOAT(Y,X@)
INTER(integer variable, floating variable)	Converts a floating point value into an integer value and stores that value in an integer variable.	100 CALL INTER(X@,Y)

BASIC COMMAND	*SUMMARY OF COMMAND*	*EXAMPLES*	
COLOR (Character-set, foreground color, background color)	Allows screen character colors to be specified. (See T.I. Basic)	100 CALL COLOR(I@,J@,K@)	
CHAR (Character code, pattern-identifier)	Allows character definition (See T.I. Basic)	100 CALL CHAR(I@,A$)	
VCHAR (Row No., Col. No., Character code)	Allows placement of a character anywhere on the screen (See T.I. Basic)	100 CALL VCHAR(I@,J@,K@)	
GCHAR (Row No., Col. No., Character code)	Returns the character code located at the specified row and column of screen	100 CALL GCHAR(I@,J@,K@)	
KEY (Key unit, ASCII Code, status)	Allows input from keyboard without interrupting program	100 CALL KEY(I@,J@,K@)	
CLEAR	Allows the screen to be cleared.	100 CALL CLEAR	
PEEKV(VDP address, variable)	Returns the contents of a VDP memory location.	CALL PEEKV(A@,B@)	NA
PEEK(CPU address, variable)	Returns the contents of a CPU memory location.	CALL PEEK(A@,B@)	
LOAD(CPU address, value)	Stores a value into CPU memory.	CALL LOAD(A@,B@)	
POKEV(VDP address, value)	Stores a value into VDP memory.	CALL POKEV(A@,B@)	NA
OPEN(File #, file type, record length, record #, file name).	Opens a file to a device.	CALL OPEN(A@,B@,C@, D@,A$)	

BASIC COMMAND	SUMMARY OF COMMAND	EXAMPLES	
PRINT(File #, record #, string variable)	Prints a string to a file.	CALL PRINT(A@,B@,B$)	
INPUT(File #, record #, string variable)	Input a string from a device.	CALL INPUT(A@,B@,A$)	
CLOSE(File #)	Closes a file.	CALL CLOSE(A@)	
LINKER (Address, additional parameters (optional))	Links a main program to other compiled subroutines. You may pass parameters.	CALL LINKER(A@,B@,A$,C)	
SCRON(On-off)	Turns the screen scroll on or off.(1=on,O=off)	CALL SCRON(A@)	NA
PRINTAT(row, column)	Prints to a specified location.	CALL PRINTAT(A@,B@)	NA
INPUTAT(row, column)	Accepts data from any location of the screen.	CALL INPUTAT(A@,B@)	NA
RESETAT	Restores input and prints position on screen.	CALL RESETAT	NA
INSTRINGA (element, integer or string variable before string array, string)	Used to store string values in an integer array (i.e., creates a string array) Also used to subscript strings	CALL INSTRINGA(A@,B@,A$) CALL INSTRINGA(A@,B$,A$)	NA
OUTSTRINGA (element, integer or string variable before string array, string)	Retrieves a string from a integer array or subscripted strings.	CALL OUTSTRINGA(A@,B@,C$) CALL OUTSTRINGA(A@,B$,C$)	NA
POS(Integer position,string, string to find, where to begin looking)	Returns the position of a sub-string.	CALL POS(X@,A$,B$,C@)	

BASIC COMMAND	SUMMARY OF COMMAND	EXAMPLES	
SEG(New string, string to segment, start in string, # of characters)	Segments a string into a substring	CALL SEG(B$,A$,A@,B@)	
VAL(Variable, string)	Converts a string into a floating point number.	CALL VAL(F,F$)	
LEN(Length, string)	Returns the length of a given string.	CALL LEN(A@,A$)	
SOUND(Duration, frequency, volume)	Produces a tone of the given frequency.	CALL SOUND(A@,B@,C@)	
ADDSTRING (String 1, string 2, string 3)	Adds two strings together and stores the result in string 1.	CALL ADDSTRING(A$,B$,C$)	
STR(String, variable)	Converts a floating point number into a string.	CALL STR(D$,C)	
CHR(ASCII code, string)	Puts a character defined by its ASCII code at the end of a string.	CALL CHR(A@,A$)	
ASC(Result, string, where in string)	Returns the ASCII code from a location in a string.	CALL ASC(A@,A$,B@)	
FLOATIN (Floating point variable)	Brings a floating point number from a TI BASIC program into a compiled program.	CALL FLOATIN(A)	NA
FLOATOUT (Floating point variable)	Passes a floating point number to a TI BASIC program	CALL FLOATOUT(B)	NA
SUBIN (Location, type, variable)	Obtains a parameter from a CALL LINKER OR USERA .E COMMAND in a main program.	CALL SUBIN(D@,A@,A)	NA
SUBOUT (Type, variable)	Passes a variable back to the main program.	CALL SUBOUT(A@,A$)	NA

BASIC COMMAND	*SUMMARY OF COMMAND*	*EXAMPLES*	
PLOTMODE	Command to enter into bit-map mode.	CALL PLOTMODE	NA
PLOTCHR (ASCII code, x coor.,y coor.,fore-ground color, background color,on/off.)	Puts a character anywhere on the screen while in BIT-MAP MODE.	CALL PLOTCHR(A@, B@,C@,D@,E@,F@)	NA
PLOT(X position, Y position, fore-ground color, background color, on-off)	Turns the point on or off at a given position, with color specifications.	CALL PLOT(X@,Y@,CF@, CB@,S@)	NA
GPLOT(X position, Y position, flag)	Checks to see if a pixel at location X, Y is on or off (1=on; 0=off).	CALL GPLOT(X@,Y@,Z@)	NA
USING (Number of digits to right of the decimal.)	Specifies format of variables to be printed.	CALL USING(A@)	NA
UNUSE	Returns to normal ***SST*** print format.	CALL UNUSE	NA
SIG(Number of bytes).	Specifies the number of bytes to work with in floating point.	CALL SIG(A@) A@=0 → 8 bytes A@=1 → 6 bytes A@=2 → 4 bytes	NA
JOYST(key unit, X position, Y position)	Returns the position of Joystick into X position and Y position.	CALL JOYST(Z@,X@,Y@)	
SPRITEMODE	Command to get into Sprite mode.	CALL SPRITEMODE	NA
SPRITEA (Sprite number, character value, color, row, column)	Creates a sprite at a given position.	CALL SPRITEA(S@,CH@, C@,R@,CL@)	

BASIC COMMAND	*SUMMARY OF COMMAND*	*EXAMPLES*
MOTIONA (Sprite number, row velocity, column velocity)	Specifies the row and column velocity of a sprite.	CALL MOTIONA(S@,Y@,X@)
SCHARA (Character number, pattern string 1, string 2...string n, string of length not 16).	Changes only the character pattern of a sprite.	CALL SCHARA(CH@,A$,B$.., F$,N$)
PATTERNA (Sprite #, character #)	Changes the character code of a sprite.	CALL PATTERNA(A@,B@)
COLORA (Sprite number, foreground color)	Specifies foreground color for a sprite.	CALL COLORA(S@,C@)
LOCATEA (Sprite #, row, column)	Changes the location of a sprite.	CALL LOCATEA(S@,R@, C@)
POSITIONA (Sprite #, row, column)	Returns the position of the given sprite.	CALL POSITIONA(S@,R@, C@)
MAGNIFYA (magnification)	Specifies the size of the sprite.	CALL MAGNIFYA(A@)
DELSPRITEA (Sprite number)	Deletes sprites.	CALL DELSPRITEA(A@)
DISTANCEA (Type, sprite #1, sprite #2, distance)	Returns the square of the distance between 2 sprites. Type = 1	CALL DISTANCEA(A@, AS@,BS@,X@)
DISTANCEA (Type, sprite #, row, column, distance)	Returns the square of the distance between a sprite and a location. Type = 2	CALL DISTANCEA(A@, AS@,R@,C@,X@)

BASIC COMMAND	***SUMMARY OF COMMAND***	***EXAMPLES***	
COINCA(Type, sprite #1, sprite #2, tolerance, hit or miss)	Detects a coincidence between two sprites. Type = 1	CALL COINCA(A@,AS@, BS@,T@,X@)	
COINCA(Type, sprite #, row, column, tolerence, hit or miss)	Detects a coincidence between a sprite and a location. Type = 2	CALL COINCA(A@,AS@,R@, C@,T@,X@)	
COINCA(Type, hit or miss)	Detects a coincidence between any sprites. Type = 3	CALL COINCA(A@,X@)	
SCREEN (Color)	Changes the color of the screen	CALL SCREEN(A@)	
SCROLL(Type, left or right, amount)	Scrolls screen to the left or right. Type = 0	CALL SCROLL(A@,D@,L@)	NA
SCROLL(Type, row, column, amount, top or bottom)	Allows scrolling of part of the screen. Type = 1	CALL SCROLL(A@,R@,C@, L@,E@)	NA
RANDOMIZE	Set a new seed for RND command.	CALL RANDOMIZE	
RND(interval, random variable)	A random number is returned in the interval from 0 to the value of the first variable minus 1.	CALL RND(A@,B@)	
SCREENON	Keeps screen active	CALL SCREENON	NA
USERA-USERE (parameters optional)	User definable subprograms.	CALL USERA(A@,B,C$)	NA

NOTE: Most of the parameters in the call statements above are integer variables unless otherwise specified.
NA: Not Available for simulating with *EDITOR/EX*.

SST BASIC REQUIREMENTS

The basic requirements needed to use the *SST COMPILER* are.

1. All variables are to be one or two letters. All one letter variables can use the letters A through Z.

   EXAMPLE: A,B,C,D.. Z

   In all two letter variables, the first letter must be one of the letters A, B, or C, while the second letter can be any letter A through Z.

   EXAMPLE: AA,AB,AC..AZ
   CA,CB,CC...CZ

   Reserved words are: AJ,AJ@,AJ$,BL,BL@,BL$
   These words may not be used for variable or array names.

   Integer variables are to be followed by an @ symbol. String variables are to be followed by a $.

2. All variables and constants used in the program must be defined by a *LET* statement at the beginning of the program(s). Failure to do so will result in an error.

3. *DIM* statements must be used in a specific order, which is coordinated with the *LET* statements. Please be sure to read the sections discussing *LET* statements and *DIM* statements, to ensure correct order of operation.

4. There can be no more than one function operation per line.

   EXAMPLE: Y=INT(X)
   Z=SIN(W)

   More than one arithmetic operation per line can be performed if the operations are of the same order (all additions or all multiplications etc.). If the operations are not all of the same order they will be performed from left to right, regardless of the operations. Array elements may not be used if you have more than one arithmetic operation per line. The variable for the result may only appear once and must be the first or second variable to the right of the equals (=) sign.

   EXAMPLE: A=A+B+C
   OR A=B+A+C
   NOT A=B+C+A

5. Line numbers must be between 11 and 32000.

DETAILED DESCRIPTION OF COMMANDS

The following pages describe each command in greater detail. Because the ***SST EXPANDED BASIC COMPILER*** requires Memory Expansion, the ***SST EXPANDED BASIC*** is a large subset of *EXTENDED BASIC*. Therefore, most of the floating point operations and functions are included as well as integer arithmetic for speed and efficiency. Also included are graphics capabilities, Bit Map mode, sound, sprites and the ability to work in only 6 or 4 byte floating point arithmetic.

LET STATEMENTS

All variables and constants must be defined in a *LET* statement at the beginning of the program. For arithmetic statements the word *LET* is not allowed. When the ***SST COMPILER*** sees *LET*, it reserves space, computes and stores the value of the variable. *LET* statements are not executable statements in the compiled version, and it is not permissible to jump to these statements using GOTO, GOSUB or IF statements. If a string variable is longer than 26 characters, it will be truncated.

```
EXAMPLE  LET A$="SST EXPANDED BASIC"
         LET A=32 12
         LET C=COS(3 14/4)
         LET B@=2
      The following is not valid
         LET A=32 1
         LET B=A
```

Only constants or arithmetic expressions are allowed in LET statements

Note: An integer variable must be followed by an @ symbol, and a string variable by a $. String variables must be declared first, followed by floating point variables, and finally integer variables. It is possible to create strings longer than 26 characters with some of the available string commands. However, you must allocate the needed size with several *LET* statements.

```
EXAMPLE: LET B$=""
         LET B$=""
         LET B$=""
```

The string variable B$ will have space large enough to hold an 83 character string. Each string variable is 26 bytes long plus 2 bytes. One byte is used for the length of the string.

DIM STATEMENTS

The *DIM* statements must be located at the beginning of your program. You may use the *DIMENSION* statement with either integer or floating point variables. Floating point *DIM* statements must be located with floating point *LET* statements. Integer *DIM* statements must be located with integer *LET* statements. One or two subscripts may be used. String variables may not be used in *DIM* statements. (See *INSTRINGA* and *OUTSTRINGA*).

```
EXAMPLE: 100 LET A$=".. "
         110 LET I=1 2
         120 DIM V(3,3)
         130 DIM K@(2,5)
         140 LET M@=2
```

Floating point arrays are stored inside the T.I. console. Thus, maximum number of elements is limited only by the size of console memory. Integer arrays are stored in *MINI-MEMORY* or *MEMORY EXPANSION*. Your program, plus integer arrays, must be less than approximately 3700 bytes for Mini-Memory, and less than 24000 bytes for Memory Expansion.
There can be only one variable dimensioned per line. Therefore, to dimension an integer variable and a floating point variable, you would write:

```
100 DIM A(12,4)
110 DIM V@(10)
```

Floating point arrays take up a considerable amount of storage space in T.I. Basic. For example, the array B(10,10) has 121 elements in it. Each element of B takes up 8 bytes, so the entire array B takes up 968 bytes. Because of these storage requirements floating point arrays are stored in the T.I. console. Anytime an operation is performed on floating point variables or floating point array elements, 8 bytes must be manipulated. Therefore, use as few floating point arrays as possible. The overuse of these arrays will decrease the efficiency of your program. The integer array B@(10,10) also has 121 elements, however each element takes up only 2 bytes, so the entire aray takes up only 242 bytes. For string arrays see the *INSTRINGA* and *OUTSTRINGA* commands.

DISPLAY STATEMENT

This command is used only to dimension larger arrays.

```
DISPLAY F@(8000)
```

If you attempt the command *DIM F@(8000) T.I. BASIC* will give an error message when you attempt to *RUN* the *EDITOR* program. This Command is not simulated in either *EDITOR* program.

INPUT STATEMENT

The *INPUT* statement may be used to input data into your program. The variable following the *INPUT* statement may be an integer, floating or string variable. It must, however, first be declared in a *LET* statement. A string may be up to 26 characters long. The value of an integer variable must be between -32768 and +32767. A floating point variable has the same range as *T.I. BASIC*. Only one variable may be read in per *INPUT* statement. The *INPUT* statement must have the form:

```
100 INPUT A
200 INPUT A@
300 INPUT A$
400 INPUT B(A@)
```

NOTE: Due to programming considerations, the input statement scrolls the screen after input rather than before, as in TI BASIC. To remove the "?" on *INPUT* put a 128 at location 10334 with the *CALL LOAD* statement.

PRINT ST' 'EMENT

The *PRIN*ı statement allows you to print an integer, floating point, or string variable. There can only be one variable per *PRINT* statement. Due to programming considerations the *SST COMPILER'S* print format is different from *T.I. BASIC*. When a print floating point or string is performed, the resulting characters have a starting location on the screen of row 24, column 3, and are printed from left to right. An integer *PRINT* places the characters on the screen from right to left starting at location row 24, column 9. Unless you are in *SPRITE MODE* you may not print a string longer than 26 characters. This statement will only scroll the screen one line. Therefore you should use *CALL PRINTAT* to position your printing of longer strings high enough. If you print beyond the screen, you will cause a disruption. The *PRINT* statement must be of the following form:

```
100 PRINT A
200 PRINT A@
300 PRINT A$
400 PRINT B(A@)
```

IF STATEMENT

The form of the *IF* statement must be:

```
100 IF A<=0 THEN 200          100 IF A@<=0 THEN 200
        or                            or
100 IF A>=0 THEN 200          100 IF A@>=0 THEN 200
        or                            or
100 IF A=00 THEN 200          100 IF A@=00 THEN 200
```

A jump will be performed if the variable is less than or equal to zero in the first case, greater than or equal to zero in the second case, or strictly equal to zero in the third case. The variable may be either integer or floating point. Most other tests may be performed by a proper combination of these three *IF* statements.

NOTE: Both zeros are required for the test *"IF A =00"*.

GOTO STATEMENT

This statement provides an unconditional jump to another statement. The form is:

```
100 GOTO 300
```

GOSUB STATEMENT

This statement allows you to jump to a subroutine. It will return to the statement following *GOSUB* with the use of the *RETURN* statement.

NOTE: You cannot have more than 62 nested subroutines.

RETURN STATEMENT

This statement must be used at the end of a subroutine. It tells the Compiler to return to the location following the *GOSUB*.

STOP STATEMENT

A *STOP* statement must be at the end of your *BASIC* program. A *STOP* statement cannot be used in any other place in a *BASIC* program.

FOR-NEXT STATEMENTS

The *FOR-NEXT* statements work similar to the *FOR-NEXT* statements in *T.I. BASIC*. The form is:

```
FOR I@=N@ TO M@
. . . . .
. . .
NEXT I@
```

In the above example if M@ is changed inside the loop, the loop will be changed accordingly. This is unique to *SST EXPANDED BASIC*. The variables in the *FOR* statement must be integer variables (array elements are not allowed). If N@ is larger than M@, the loop will not be performed. Although the step statement is not included in the *FOR* statement, it can be easily accomplished by using an arithmetic statement.

```
T.I. BASIC                       SST COMPILER

100 FOR I=3 TO 1000 STEP 2       100 LET I@=1
. . . . . . . . . . .            110 LET M@=1000
. . . . . . . . . . .            120 LET J@=3
260 NEXT I                       130 LET K@=1
270 STOP                         140 FOR I@=J@ TO M@
                                 . . . . . . . . .
                                 . . . . . . . . .
                                 250 I@=I@+K@
                                 260 NEXT I@
                                 270 STOP
```

CALL FLOAT(FLOATING VARIABLE, INTEGER VARIABLE) STATEMENT

This statement allows you to convert an integer variable X@ to a floating point variable Y. Both X@ and Y must be declared and given a value in a *LET* statement. Arrays are allowed in this statement. The form of the statement is:

```
100 LET Y=0
110 LET X@=5
120 CALL FLOAT(Y,X@)
130 PRINT Y
```

CALL INTER(INTEGER VARIABLE, FLOATING VARIABLE) STATEMENT

This statement converts a floating point variable Y to an integer variable X@. Both Y and X@ must appear in a *LET* statement at the beginning of the program. Arrays are allowed in this statement. The form of the statement is:

```
100 LET Y=5
110 LET X@=0
120 CALL INTER(X@,Y)
130 PRINT X@
```

This function will perform rounding. For example, 8.5 will be converted to 9. The *FLOAT* and *INTER* functions will allow one of the variables to be an array element, not both. The floating point number must be in the range of -32768 to 32767.

NOTE: The *INT* command returns an 8 byte floating point value while the *CALL INTER* command returns a 2 byte integer.

The param. s in the following *CALL STATEMENTS* are integer variables unless specified otherwise.

CALL KEY (KEY UNIT, ASCII CODE,STATUS)

See *T.I. BASIC* for a complete description. For the 99/4A the key unit is changed to 5 each time an input statement is encountered by the compiled program. Caution is recommended when using key unit zero. No arrays can be used in this command.

IMPORTANT: Because of the speed of machine language, you may need a delay loop between calls to *CALL KEY*.

```
EXAMPLE: 100 LET K@=5
         110 LET X@=1
         120 LET S@=3
         130 LET L@=500
         140 LET I@=0
         150 LET J@=1
         160 CALL KEY(K@,X@,S@)
         170 PRINT X@
         180 PRINT S@
         190 FOR I@=J@ TO L@
         200 NEXT I@
         210 GOTO 160
         220 STOP
```

CALL COLOR (CHARACTER-SET,FOREGROUND-COLOR,BACKGROUND-COLOR)

(See *T.I. BASIC* Manual for a more complete description).

Screen character colors may be specified using this statement. The color of the dots making up the character are called the foreground color, while the color that makes up the rest of the character position is called background color. Arrays are allowed in this command.

```
EXAMPLE: 100 LET S@=1
         110 LET F@=2
         120 LET B@=16
         130 CALL COLOR(S@,F@,B@)
```

CALL CHAR (CHARACTER-CODE, "PATTERN-IDENTIFIER")

This statement allows definition of graphic characters. The character-code specifies the code of the character. The pattern-identifier *MUST BE* a 16 character string which specifies the pattern of the character. (See *T.I. BASIC* for a complete description and examples). This command cannot be used to define sprites. Arrays are allowed in this command. In *SPRITEMODE* you can define all characters 1-256.

```
EXAMPLE: 100 LET A$="FFFFFFFFFFFFFFFF"
         110 LET C@=63
         120 CALL CHAR(C@,A$)
```

CALL VCHAR (ROW-NUMBER, COLUMN-NUMBER, CHARACTER CODE)

A character may be placed anywhere on the screen using this statement. The row-number and column-number specify the position on the screen. Note that repetitions are not allowed in the *VCHAR* command. Arrays are allowed in this command.

```
EXAMPLE: 100 LET R@=5
         110 LET C@=10
         120 LET CH@=63
         130 CALL VCHAR(R@,C@,CH@)
```

NOTE: This command can be used in place of *HCHAR*. Repetitions can be accomplished by using a *FOR-NEXT* loop.

CALL GCHAR (ROW-NUMBER, COLUMN-NUMBER, CHAR CODE)

The *ASCII* code of the character on the screen, at the location given by row-number, column-number, is returned in *CHAR CODE*. No array may be used in this command.

```
EXAMPLE: 100 LET R@=5
         110 LET C@=10
         120 LET CH@=0
         130 CALL GCHAR(R@,C@,CH@)
         140 PRINT CH@
```

CALL CLEAR

This statement allows you to clear the screen. This command must not be used in *PLOTMODE* (Bit Map mode).

IN THE FOLLOWING COMMANDS NO ARRAYS CAN BE USED

CALL PEEKV(VDP ADDRESS, INTEGER VARIABLE)

Given an integer address in *VDP* memory as the first variable, the subroutine returns the contents of that address in the second variable.

```
EXAMPLE: 100 LET A@=1
         110 LET B@=65
         120 LET C@=0
         130 REM PUT 'A' ON SCREEN AT ROW 1,
              COLUMN 1
         140 CALL VCHAR(A@,A@,B@)
         150 REM PICKUP THE ASCII CODE FROM THE
              SCREEN i.e., B@="A"+96=161
              AND PUT IT IN THE VARIABLE B@
         160 CALL PEEKV(C@,B@)
```

CALL PEEK(CPU ADDRESS,INTEGER VARIABLE)

Given an integer address in *CPU* memory as the first variable, the subroutine returns the contents of that address in the second variable.

```
EXAMPLE: 100 LET A@=-24576
         110 LET B@=0
         120 CALL PEEK(A@,B@)
         130 REM B@ NOW CONTAINS THE CONTENTS OF
              LOCATION -24576
```

CALL LOAD(CPU ADDRESS, INTEGER VARIABLE)

Given an integer address in *CPU* memory as the first variable, this subroutine places the contents of the second variable at that location.

```
EXAMPLE: 100 LET A@=-8000
         110 LET B@=30
         120 REM PUT THE VALUE 30 AT
                LOCATION -8000
         130 CALL LOAD(A@,B@)
```

CALL POKEV(VDP ADDRESS, INTEGER VARIABLE)

Given an integer address in *VDP* memory as the first variable, this subroutine places the contents of the second variable at that location.

```
EXAMPLE: 100 LET A@=161
         120 LET B@=0
         130 REM PUT THE LETTER "A" ON THE SCREEN
                AT ROW 1, COLUMN 1
         140 CALL POKEV(B@,A@)
```

CALL OPEN(FILE #, FILE TYPE, RECORD LENGTH, RECORD #, FILE NAME)

You may open files using file #'s from 1-3, 11-30. Files may go to disk drives, RS232 or PIO, which are defined in a string variable (FILE NAMES).

You *MUST GIVE* the file *TYPE*. This is an integer number which may be evaluated from the *"FILE TYPES"* table.

You *MUST GIVE* an integer number for the length of each record in the *(RECORD LENGTH)* variable, and this number *MUST BE* between 0-255. (0-254 if the record type is variable) You must give a record number even if you are not using relative records. This has the same range as in *T.I. BASIC* and *MUST BE AN INTEGER VARIABLE*. For sequential files use a "0" for the record number.

```
EXAMPLE: 100 LET A$="RS232.BA=9600"
         120 LET B$="DSK1.TEST"
         130 LET D@=1
         140 LET A@=2
         145 LET E@=0
         150 LET B@=16
         155 LET F@=-1
         160 LET C@=80
         165 REM THE TWO FILES ARE BOTH SEQUENTIAL
               DISPLAY, UPDATE, VARIABLE 80, FILES
         170 CALL OPEN(D@,B@,C@,E@,B$)
         180 CALL OPEN(A@,B@,C@,E@,A$)
         190 CALL INPUT(D@,F@,B$)
         200 REM PRINTS B$ OUT TO THE RS232 PORT
         210 CALL PRINT(A@,F@,B$)
         220 CALL CLOSE(A@)
         230 STOP
```

How to calculate different file types.

1. Choose the parameters you wish your file to have.
   EXAMPLE: UPDATE,INTERNAL,SEQUENTIAL
2. Look in the following table for the number associated with the parameters. Take these numbers and add them to determine the type #.

FILE TYPE TABLE

1.	FIXED=0	VARIABLE=16
2.	DISPLAY=0	INTERNAL=8
3.	UPDATE=0	APPEND=6
4.	SEQUENTIAL=0	RELATIVE=1

Only one in each of the 4 categories may be choosen. Examples of different file types #'s and *CALL OPEN* statements are as follows:

```
  TI BASIC: OPEN #A@: F$,UPDATE, INTERNAL,FIXED 192,SEQUENTIAL
                             0   +   8   +   0    +    0     =8

 SST BASIC: CALL OPEN(A@,B@,C@,D@,F$)
            WHERE
            B@=8......FILE TYPE
            C@=192......RECORD LENGTH
            D@=0......STARTING RECORD

  ~~TI~~BASIC: OPEN #A@:F$,UPDATE,DISPLAY,VARIABLE 80,SEQUENTIAL
                             0   +   0   +   16   +    0     = 16

            THIS IS THE SAME AS: OPEN #A@:F$

 SST BASIC: CALL OPEN(A@,B@,C@,D@,F$)
            WHERE B@=16......FILE TYPE
                  C@=80......RECORD LENGTH
                  D@=0......STARTING RECORD

  TI BASIC: OPEN #A@:F$,RELATIVE 50,UPDATE,FIXED 80,INTERNAL
                             1     +     0  +  0    +    8     = 9

 SST BASIC: CALL OPEN(A@,B@,C@,D@,F$)
            WHERE B@=9......FILE TYPE
                  C@=80......RECORD LENGTH
                  D@=50 ......IF THE FILE IS BEING CREATED
                          D@ SPECIFIES THE INITAL NUMBER
                          OF RECORDS IN THE FILE. IF THE
                          FILE HAS ALREADY BEEN CREATED
                          IT STARTS AT RECORD # D@.
```

NOTE: There are no defaults for the *SST EXPANDED BASIC*, all parameters must be defined.

CALL PR .FILE #,REC #, STRING VARIABLE)

You may use this command to print a string to a device that has been opened with the *"CALL OPEN"* subroutine. The file number must be the same for both commands. A record variable is required *(REC #)* and must have a value of "-1", if the file is not opened as a relative file. Otherwise, it follows the same rules as specified in the TI manual for relative files.

EXAMPLE: SEE CALL OPEN EXAMPLE

TI BASIC	SST BASIC
PRINT #A@,REC B@:A$	CALL PRINT(A@,B@,A$)

NOTE: All floating point numbers must be converted to a string by the *STR* command before they are printed to a file. All integer numbers must be converted to floating point numbers by the *FLOAT* command and then to a string by the *STR* command before being printed to a file.

CALL CLOSE(FILE #)

This command will close a file that was previously opened in a *CALL OPEN* statement.

EXAMPLE: SEE CALL OPEN

TI BASIC	SST BASIC
CLOSE #A@	CALL CLOSE(A@)

CALL INPUT(FILE#,REC#,STRING VARIABLE)

Allows you to read a string from a device that has been opened with the given *FILE #*. The record # is required even if you have a sequential file, in which case the *REC#* must be equal to "-1".
For relative files you must give a starting record #. Then, if in subsequent record #'s a "-1" is used, the record # will be incremented automatically. If the file was opened as a *FIXED DISPLAY* type, the entire record will be put into the string variable, including any extra blanks which may pad it. Therefore, you must reserve enough space to hold the incoming record. (Use LET Strings).
NOTE: Only strings may be read or written to a file. Numbers must be inputed as strings and then converted.

EXAMPLE: SEE CALL OPEN

TI BASIC	SST BASIC
INPUT #A@,REC B@:A$	CALL INPUT(A@,B@,A$)

CALL LINKER(ADDRESS[,OPTIONAL PARAMETERS])

You may use this command to link your main program(s) to other compiled subroutines. If you give an address of another program (which has a *RETURN* statement), then when the *CALL LINKER* is performed, control is transferred to that subroutine. The subroutine returns to the statement following the *CALL LINKER* statement. The address must be in an integer variable in base 10.
NOTE: The address of each line that is being compiled is given next to the line number.

EXAMPLE: COMPILING LINE 20 -8564

Using this command you are able to extend your program beyond 470 lines of code.

```
EXAMPLE: 11 REM FIRST PROGRAM STARTS AT -24000
         12 LET A@=-8000
         13 REM THE FIRST EXECUTABLE INSTRUCTION
                OF THE SECOND PROGRAM IS GIVEN BY THE
                STARTING ADDRESS PLUS 6
                .
                .
                .
         467 INPUT A$
         468 INPUT B$
         469 CALL LINKER(A@)
         470 STOP

         11 REM Second Program starts at address -8006
         12 PRINT A$
         13 PRINT B$
         14 RETURN
         15 STOP
```

In this example, both programs were compiled without reloading the compiler. In this way, all variables are common to both programs. When the *RETURN* statement is encountered, control is returned to the first program and the stop instruction is executed.

The parameters after the address in *CALL LINKER* are optional. However, to pass variables from one compiled program to another, which were compiled with a different variable set, you must put the variables from the main program after the address. Each parameter passed must be seperated by a ",". See *CALL SUBIN* and *CALL SUBOUT* for information on how to accept variables in the second compiled program and how to pass them back to the main program. At most 5 variables can be passed.

CALL SCRON(ON/OFF)

You may prevent the computer from scrolling by using this command. *AN INTEGER* variable must be given. If *ON/OFF* is equal to "0" then the scroll after every *INPUT* and *PRINT* is turned off. If *ON/OFF* is "1" then the screen will scroll after every *PRINT* and *INPUT*.

This feature is useful in conjunction with *PRINTAT* and *INPUTAT* commands. It allows you to print or input more than one variable on a line.

```
EXAMPLE: 100 LET A@=10
         110 LET B@=20
         120 LET C@=1
         130 LET D@=0
         140 LET E@=24
         150 LET F@=10
         160 REM TURN OFF SCROLL
         170 CALL SCRON(D@)
```

```
180 PRINT A@
190 REM PRINT NEXT VARIABLE OVER 10 PLACE
200 CALL PRINTAT(E@,F@)
210 PRINT B@
220 REM RESTORE TO ORIGINAL PRINT AND
      INPUT FORMAT
230 CALL RESETAT
240 STOP
```

CALL PRINTAT(ROW,COLUMN)

This command will allow you to place the output on the screen at any row and column.

EXAMPLE: SEE CALL SCRON

NOTE: See print command to understand how integers and floating point numbers are displayed on the screen. As with most of the commands, parameters that exceed the allowable values will give unpredictable results.

CALL INPUTAT(ROW,COLUMN)

This command will allow you to input data from the location on the screen given by row and column. It should be noted that any existing data on the screen is ignored when you use an input statement. Only what is typed is entered into the variable.

```
EXAMPLE: 100 LET A$=" "
         110 LET X@=2
         120 LET Y@=12
         130 CALL INPUTAT(Y@,X@)
         140 REM THE CURSOR AND ? WILL APPEAR
               FOR INPUT IN THE MIDDLE OF THE
               SCREEN, TWO POSITIONS FROM THE LEFT.
         150 INPUT A$
```

CALL RESETAT

You may restore the *INPUT* and *PRINT* commands back to the default values by use of this command.
NOTE: The *PRINT* position and *INPUT* position are both affected and may not be separated.

EXAMPLE: SEE CALL SCRON

CALL INSTRINGA(ELEMENT,INTEGER OR STRING VARIABLE BEFORE STRING ARRAY,STRING)

Although the *SST* Compiler does not have dimensioning of string arrays directly, you may use this command to save a string of lengths 0-255 characters in an integer array. The integer array must be dimensioned as follows:

```
100 LET H@=1
120 DIM A@(100)
```

As you can see it takes two statements to define the string array. You must indicate the integer array you wish to use as a string array by giving the integer variable just before the integer array. The integer variable is used as an indicator and will be changed.

Each element of the string uses 14 elements of the integer array. If your string contains more than 27 characters, for example, if it contains 50 characters, you must use two or more string elements to hold the string. String arrays must be one dimensional arrays.

```
EXAMPLE: 100 LET A$="AB...Z"
         120 LET B$="AB...Z"
         130 LET CA$="TEST2..TEST"
         140 LET CB$=""
         150 LET CB$=""
         160 LET CB$=""
         165 REM THE ABOVE 3 LET'S
         170 REM ALLOCATES AN 83 CHARACTER STRING
         175 LET H@=1
         180 DIM S@(140)
         185 REM THE ABOVE 2 LINES ALLOCATE A 10
         187 REM ELEMENT STRING ARRAY
         190 LET A@=1
         200 LET B@=2
         210 LET C@=4
         220 LET D@=8
         230 REM STORE FIRST STRING IN FIRST 14
               ELEMENT OF ARRAY S@
         240 CALL INSTRINGA(A@,H@,A$)
         250 REM CONCATENATES TWO STRINGS TOGETHER
               AND STORES IN STRING CB$. STRING CB$
               NOW CONTAINS 52 ELEMENTS
         260 CALL ADDSTRING(CB$,A$,B$)
         270 REM USED TWO ELEMENTS OF A STRING ARRAY
               WHICH IS THE SAME AS 28 ELEMENTS
               OF THE INTEGER ARRAY
         280 CALL INSTRINGA(B@,H@,CB$)
         290 REM C$ IS STORED IN THE NEXT FREE BLOCK
               OF THE STRING INTEGER ARRAY
         300 CALL INSTRINGA(C@,H@,C$)
         310 REM RETURN A STRING FROM AN INTEGER ARRAY
         320 CALL OUTSTRINGA(A@,H@,C$)
         330 PRINT C$
         340 STOP
```

REFERENCE MATERIAL

Using element numbers of 0,1,2,3... will allow strings of 0-27 characters to be stored consecutively. Using element numbers of 0,2,4,6,8... will allow strings of 0-55 characters to be stored consecutively. Using element numbers of 0,3,6,9,12... will allow strings of 0-83 characters to be stored consecutively, (etc.).

NOTE: Make certain you have allocated enough room with your *DIM* command or else a string may over-write part of your program. *OUTSTRINGA* and *INSTRINGA* can also be used to subscript existing strings. This is done by using a string variable name as the second parameter. Starting with that variable its subscripts will be "0" and subsequent 26 character strings will have subscripts 1,2...n.

```
EXAMPLE: 100 LET A$="TEST"
         110 LET B$="TEST2"
         120 LET B$="TEST3"
         130 LET C$="TEST4"
         140 LET A@=2
              .
              .
              .
         200 CALL OUTSTRINGA(A@,A$,C$)
         230 PRINT C$
         300 REM"TEST3" WILL BE PRINTED
```

This form of the command is not simulatable in *EDITOR/EX*.

CALL OUTSTRINGA(ELEMENT,INTEGER OR STRING VARIABLE BEFORE STRING ARRAY,STRING)

This command is the opposite of the *CALL INSTRINGA*. You may retrieve a string from an array by use of this command. You must make certain that the string in which the array element is going is of sufficient length before transfer or else you may write over part of your program.

EXAMPLE: SEE CALL INSTRINGA

CALL POS(INTEGER LOCATION,STRING,STRING TO FIND, WHERE TO BEGIN LOOKING)

This statement is similar to the *POS* command in *TI BASIC*, with the exception that the result is returned in the first parameter. This command returns the location of a substring in a main string. The command starts searching in the main string at the specified location.

TI BASIC	***SST*** BASIC
X@=POS(A$,B$,C@)	CALL POS(X@,A$,B$,C@)

```
EXAMPLE: 100 LET A$="ABCDEF"
         110 LET B$="C"
         120 LET A@=1
         130 LET C@=0
         150 CALL POS(C@,A$,B$,A@)
         160 REM C@ NOW =3
```

CALL SEG(NEW STRING,STRING TO SEGMENT,START IN STRIN NUMBER OF CHARACTERS)

This command is similar to the *TI BASIC* function *SEG$*, with the exception that the segmented string is store in the first variable(NEW STRING).

TI BASIC	***SST*** BASIC
B$=SEG$(A$,A@,B@)	CALL SEG(B$,A$,A@,B@)

```
EXAMPLE: 100 LET A$="ABCD"
         110 LET B$=""
         120 LET A@=2
         130 LET B@=2
         140 CALL SEG(B$,A$,A@,B@)
         150 REM B$ NOW CONTAINS "BC"
```

CALL VAL(FLOATING POINT VARIABLE,STRING VARIABLE)

You may convert a string that is a numeric representation of a number into a floating point value. This command is similar to the *VAL* function of *TI BASIC*, with the exception that the floating point variable is inside the parenthesis.

TI BASIC	***SST BASIC***
F=VAL(F$)	CALL VAL(F,F$)

```
EXAMPLE: 100 LET F$="1.56E10"
         110 LET F=0
         120 CALL VAL(F,F$)
         125 PRINT F
         130 REM YOU WILL HAVE 1.56E+10 PRINTED
                  ON YOUR SCREEN
         140 STOP
```

CALL LEN(INTEGER VARIABLE,STRING VARIABLE)

You may find the number of characters in a string with this command. It is similar to the *LEN* function in *TI BASIC*, except that the size of the string is returned in an integer variable inside the parenthesis.

TI BASIC	***SST*** BASIC
A@=LEN(A$)	CALL LEN(A@,A$)

```
EXAMPLE: 100 LET A$="ABC"
         110 LET A@=0
         120 CALL LEN(A@,A$)
         130 REM PRINT 3 ON THE SCREEN
         140 PRINT A@
         150 STOP
```

CALL SOUND(DURATION,FREQUENCY,VOLUME)

This command produces a tone of the given duration, frequency and volume. A duration between 17 and 4250 gives the length of the sound in thousandths of a second. A duration between 1 and 16 gives a sound of indeterminate length. That is, the sound will continue until another *CALL SOUND* is encountered. Frequency must be between 110 and 32767 and volume must be between 0 and 30

NOTE: "0" is the loudest volume. See the T.I. manual for a complete description. Only one sound generator is available.

```
EXAMPLE: 100 LET A@=4000
         110 LET B@=110
         120 LET C@=2
         122 REM CALL SPRITEMODE IS NEEDED FOR THE
               SOUND COMMAND
         125 CALL SPRITEMODE
         130 CALL SOUND(A@,B@,C@)
         140 GOTO 140
         150 STOP
```

CALL ADDSTRING(STRING1,STRING2,STRING3)

This instruction adds *STRING2* and *STRING3* together and stores the result in *STRING1*.

TI BASIC	***SST*** BASIC
A$=B$ & C$	CALL ADDSTRING(A$,B$,C$)

```
EXAMPLE: 100 LET A$="TEST"
         110 LET B$="SST"
         120 LET C$="SOFTWARE, INC."
         130 CALL ADDSTRING(A$,B$,C$)
         140 PRINT A$
         150 STOP
```

A$=B$ is allowed if B$ is less than 27 characters. However, you must use the *CALL ADDSTRING* command for Strings longer than 27 characters.

CALL STR(STRING,VARIABLE)

This command converts a floating point number into a string. This is needed for output to a disk or printer. This is similar to the *STR$* command in *TI BASIC*.

TI BASIC	***SST*** BASIC
B$=STR$(A)	CALL STR(B$,A)

```
EXAMPLE: 100 LET B$="TEST"
         110 LET A=12 3
         120 CALL STR(B$,A)
         130 REM A IS NOW CONVERTED
         140 REM TO STRING REPRESENTATION
         150 REM AND STORED IN B$
         160 STOP
```

CALL CHR(ASCII CODE,STRING)

This command is similar to the *CHR$* command in *TI BASIC*, with the exception that the character is added onto the end of the given string. Therefore, if you call this command and the original string was *"ABCD"* and the *ASCII* code in variable A@ is 65, the new string produced would be *"ABCDA"*.
NOTE: That the character code for A is 65.

```
EXAMPLE: 100 LET A$="ABCD"
         110 LET A@=65
         120 CALL CHR(A@,A$)
         130 REM A$ NOW CONTAINS ABCDA
```

CALL ASC(RESULTS,STRING,WHERE IN STRING)

CALL ASC is similar to the *ASC* function in *TI BASIC*, with the exception that you give the location of the character in the string. The *RESULT* has the value of the *ASCII* code for that character.

```
EXAMPLE: 100 LET A$="TESTA"
         110 LET A@=0
         120 LET B@=5
         130 CALL ASC(A@,A$,B@)
         140 REM A@ WILL NOW EQUAL 65
```

CALL FLOATIN(FLOATING POINT VARIABLE)

This command will take the contents of the first floating point variable defined in a *TI BASIC* program and puts it into the variable indicated in the *CALL FLOATIN*. The first variable defined must be a floating point variable, not an array or string variable. This command is useful for putting data into compiled programs.

```
         TI BASIC CALLING PROGRAM
EXAMPLE: 10 LET A=0
         20 DATA 1,2,3,4,5
         30 FOR I=1 TO 5
         40 READ A
         50 CALL LINK("compiled program")
         60 REM THE VALUE OF "A" WILL BE PUT
         70 REM INTO THE COMPILED PROGRAM
         80 NEXT I
         90 STOP

         SST EXPANDED BASIC
         100 LET A=0
         110 LET D@=1
         120 LET A@=5
         130 LET C@=0
         140 LET E@=4
         145 DIM B@(5)
         150 A@=A@-D@
         160 REM BRING IN FLOATING POINT NUMBER
```

```
165 CALL FLOATIN(A)
170 REM FROM TI BASIC
180 C@=E@-A@
190 REM CONVERT TO INTEGER
200 CALL INTER(B@(C@),A)
210 REM CHECK IF END OF DATA
220 IF A@>=0 THEN 1000
MAIN PROGRAM
.
.
.
990 REM RETURN TO BASIC
1000 STOP
```

After the fifth time the compiled program is called it will have loaded all the data from *TI BASIC* and will begin to execute the main program. The data stored in the compiled program will remain there until the computer is turned off or the data is changed by the program. Because of this you may use the *FASTLOAD* program to save your compiled program with the data in place. After this you do not have to use a *TI BASIC* program to reload the data.

NOTE: This command may not be used after *CALL SPRITEMODE* or *CALL PLOTMODE*, because they may erase your *TI BASIC* program.

CALL FLOATOUT(FLOATING POINT VARIABLE)

This command is similar to *CALL FLOATIN*, except the floating point number from the compiled program is passed to the first defined floating point variable in *TI BASIC*.

```
TI BASIC CALLING PROGRAM               SST EXPANDED BASIC
10 LET A=0                             200 LET B=10
20 CALL LINK("compiled program")       210 CALL FLOATOUT(B)
30 PRINT A                             220 STOP
.
.
100 REM WHEN A IS PRINTED IT WILL BE 10
```

SUBIN(LOCATION,TYPE,VARIABLE)

This command is used to accept parameters from the *CALL LINKER* or *CALL USERA...E* commands. The program or subprograms that *SUBIN* and *SUBOUT* are used in must have been compiled with their own set of variables. The subroutine must have as many *SUBIN COMMANDS* as parameters passed by the *CALL LINKER* or *USERA COMMANDS*. The first parameter in the *CALL SUBIN* must contain the location of the parameter in the *CALL LINKER* parameter list. This will be a value between 1 and 5. That is, you can pass up to 5 values.

The type parameter indicates if the parameter is a string (*TYPE*=1), a floating point number (*TYPE*=2), or an integer (*TYPE*=3). The last parameter will, after execution contain the contents of the variable being *PASSED* from the main program.

NOTE: This command is not simulated in *EDITOR/EX*. You must enter this command after you are done testing the program with *EDITOR/EX*.

SUBOUT(TYPE,VARIABLE)

This command passes a variable from a separately compiled subroutine to a main program. As in *CALL SUBIN* you must indicate the type of variable you are passing (*SEE SUBIN*). As many parameters must be passed back as are in the *CALL LINKER* or *USER* command. The parameters must also be in the same order.

```
EXAMPLE: 100 REM MAIN PROGRAM
         110 LET A$="TEST"
         120 LET B=10.3
         130 LET C@=5
         140 LET D@=-8000
         150 REM -8000 IS THE STARTING ADDRESS PLUS 6
         160 REM -8000 IS THE ADDRESS OF CALL SUBIN(D@,D@,AB$)
              .
              .
         500 CALL LINKER(D@,A$,B,C@)
              .
              .
         900 STOP
         SUBROUTINE IS STORED STARTING AT ADDRESS -8050
         100 LET AB$=""
         110 LET BB=0
         120 LET CB@=5
         130 LET D@=1
         140 LET E@=2
         150 LET F@=3
         155 REM ADDRESS -8006 IS THE STARTING
                 ADDRESS FOR THE COMPILER
         160 CALL SUBIN(D@,D@,AB$)
         170 CALL SUBIN(E@,E@,BB)
         180 CALL SUBIN(F@,F@,CB@)
         190 PRINT AB$
         200 PRINT BB
         210 PRINT CB@
         215 REM PASS VALUES BACK
         220 CALL SUBOUT(D@,AB$)
         230 CALL SUBOUT(E@,BB)
         240 CALL SUBOUT(F@,F@)
         250 RETURN
         260 STOP
```

CALL PLOTMODE

This command is needed to enter Bit Map Mode. In Bit Map Mode you can access all of the pixels on the screen individually. This command can be performed only once in any program and not from *SPRITEMODE*. While in *PLOTMODE* you are not allowed the following functions and commands:

ATN, COS, EXP, INT, LOG, SIN, SQR, TAN, PRINT, INPUT

The following *CALL* statements are not allowed in *PLOTMODE*:

SPRITEMODE, PLOTMODE, COLOR, CHAR, VCHAR, GCHAR, CLEAR, USING, UNUSE, SPRITEA, MOTIONA, SCHARA, PATTERNA, COLORA, LOCATEA, POSITIONA, MAGNIFYA, DELSPRITEA, SCROLL, DISTANCEA, COINCA, SOUND, PRINTAT, INPUTAT, SCRON and RESETAT

NOTE: You cannot enter *SPRITEMODE* from *PLOTMODE*. Also, only one disk file can be opened in *PLOTMODE*. You must do a *CALL FILES(1),NEW* before entering *PLOTMODE*.

You cannot return to TI Basic from *PLOTMODE* or *SPRITEMODE*.

CALL PLOTCHR(ASCII CODE,X COOR.,Y COOR.,FOREGROUND COLOR,BACKGROUND COLOR,ON/OFF)

This command will allow you to put a character on or take a character off anywhere on the screen. You must be in *PLOTMODE*. This command is the same as *CALL PLOT* with the exception of one added parameter at the beginning. Instead of just putting a dot on the screen, a character is put there. The character will be put on (or taken off) the screen starting with its upper left hand corner at the given X and Y coordinate.

NOTE: Only the pixels making up the character will be plotted, any other pixel in the area will be left unchanged. Also, you may define your own characters with the *CALL CHAR* command, but only before calling *PLOTMODE*.

```
EXAMPLE: 11 REM ASCII CODE FOR "A"
         12 LET A@=65
         20 LET B@=0
         30 LET CF@=2
         40 LET CB@=16
         50 LET D@=1
         60 CALL PLOTMODE
         65 REM GIVES A WHITE SCREEN
         70 CALL SCREEN(CB@)
         73 CALL PLOTCHR(A@,CB@,CB@,CF@,CB@,D@)
         75 REM THE LETTER "A" WILL APPEAR
         80 REM AT THE UPPER LEFT OF THE SCREEN
         90 GOTO 90
```

CALL PLOT(X POSITION, Y POSITION, FOREGROUN COLOR, BACKGROUND COLOR, ON/OFF)

This instruction allows you to do high resolution graphics. By specifing the *X* and *Y* values you can place a dot anywhere on the screen. The X-value is between 0 and 255 and the Y-value is between 0 and 191. The foreground and background colors are as specified in *TI BASIC*. The last variable must be a "1" to place a dot and a "0" to remove a dot. You must be in *PLOTMODE* to use this command.

```
EXAMPLE: 100 LET X@=20
         110 LET Y@=40
         120 LET CA@=2
         130 LET CB@=16
         140 LET Z@=1
         145 CALL PLOTMODE
         150 CALL PLOT(X@,Y@,CA@,CB@,Z@)
         160 REM PLACES A BLACK DOT AT
         170 REM LOCATION X=20, Y=40
         180 GOTO 180
```

CALL GPLOT(X POSITION, Y POSITION,ON/OFF)

This command checks the screen to see if the pixel at position X, Y is on or off. See command above for limits on X and Y. ON/OFF returns with value 1 if the pixel is on, 0 if the pixel is off.

EXAMPLE: CALL GPLOT(X@,Y@,Z@)

CALL USING(NO. OF DIGITS TO RIGHT OF DECIMAL)

This command allows you to format your output. That is, you can specify the number of digits to the right of the decimal point. Once you use this command, all floating point numbers will be printed to the screen under this format.

CALL UNUSE

This command turns off the *CALL USING* instructions

```
EXAMPLE: 100 LET A=12.1234567890
         120 LET B@=6
         130 CALL USING(B@)
         140 REM A=12.123457 IS PRINTED
         150 PRINT A
         160 CALL UNUSE
         170 REM A=12.12345679 IS PRINTED
         180 PRINT A
         190 STOP
```

CALL SIG(NUMBER OF BYTES)

This command allows you to specify the number of bytes to be used in all floating point computations. A "0" specifies 8 bytes (14 digits), a "1" specifies 6 bytes (10 digits) and a "2" specifies 4 bytes (6 digits). This command can shorten the computation time by 6 to 50 percent for arithmetic statements.

NOTE: This command saves time not space.

```
EXAMPLE: 100 LET A=1.333333333
         110 LET B=2.333333333
         120 LET A@=2
         130 CALL SIG(A@)
         140 REM THE FOLLOWING COMPUTATION
         150 REM WILL BE DONE USING ONLY 4
         160 REM BYTE ARITHMETIC
         170 A=A+B
         180 REM A=3.6666 IS PRINTED
         190 PRINT A
         200 STOP
```

NOTE: This command will reduce the accuracy of the computation.

CALL JOYST(KEY UNIT, X-VARIABLE, Y-VARIABLE)

The *JOYST* command returns data into the X and Y variables based on the positions of the joystick. See the TI manual for a more complete description.

```
EXAMPLE: 100 LET A@=1
         110 LET B@=1
         120 LET C@=1
         130 CALL JOYST(A@,B@,C@)
         140 PRINT B@
         150 PRINT C@
         155 GOTO 130
         160 STOP
```

NOTE: Due to the speed of machine language, you may need a delay loop between calls to *JOYST.*

CALL SPRITEMODE

This command is needed to get the computer ready to handle sprites. It must be used before *SOUND* or any *SPRITE* commands are specified. This command can be performed only once in any program and may not be performed while in *PLOTMODE.* The sprite commands are used to create and control continuously moving graphics in 32 different planes. This allows you to overlap graphics with sprite #1 being in front of the other sprites.

The following commands are sprite commands:

SPRITEA, MOTIONA, SCHARA, PATTERNA, COLORA, LOCATEA, POSITIONA, MAGNIFYA, DELSPRITEA, DISTANCEA and COINCA.

While in *SPRITEMODE* you cannot use the following *CALL* statements:

SPRITEMODE, PLOTMODE, PLOTCHR, GPLOT and PLOT

NOTE: You cannot enter *PLOTMODE* from *SPRITEMODE.* You should not return to TI Basic from *SPRITEMODE* or *PLOTMODE.*

CALL SPRITEA(SPRITE #, CHARACTER #, COLOR, ROW, COLUMN)

This instruction creates a sprite at a given row and column. The sprite number is a value from 1 to 32 and the character value is any integer from 1 to 112. The foreground color is any value from 1 to 16 (See the TI manual for a more complete description). The pattern of a sprite is defined by the character and its pattern description #, which can be changed by the *SCHARA* command only.

CALL MOTIONA(SPRITE NUMBER, ROW VELOCITY, COLUMN VELOCITY)

The *MOTIONA* instruction is used to specify the row and column velocity of a sprite. A positive row velocity moves the sprite down and a positive column velocity moves the sprite right. Conversely a negative row velocity moves the sprite up and a negative column velocity moves the sprite left. These values must be between -128 and +127.

CALL SCHARA(CHARACTER #, PATTERN STRING 1, PATTERN STRING 2..., STRING N, STRING OF SIZE NOT EQUAL TO 16)

This subroutine allows you to change only the character pattern of a sprite starting with the specified character number. If the character number is 4, then string 1 changes character 4, string 2 changes character 5, etc. until a string is encountered which is not 16 characters long. N must be less than or equal to 4.

NOTE: Each pattern string must be 16 characters long. The character table for sprites is completely empty and does not contain the ASCII character set. To obtain the ASCII set you can use the following example program to transfer the ASCII character set to the *SPRITE* character table.

```
100 LET I@=1
110 LET J@=2304
120 LET K@=2951
130 LET L@=1032
140 LET A@=1
150 LET X@=1
160 LET B@=1
170 LET C@=33
180 LET D@=2
190 LET E@=20
195 CALL SPRITEMODE
196 REM PUTS CHAR 32-112 INTO THEIR PROPER PLACE
        IN THE SPRITE DESCRIPER TABLE
200 FOR I@=J@ TO K@
210 X@=I@-L@
220 CALL PEEKV(I@,A@)
230 CALL POEKV(X@,A@)
240 NEXT I@
250 CALL SPRITEA(B@,C@,D@,E@,E@)
260 GOTO 260
270 STOP
```

CALL PA RNA(SPRITE #, CHARACTER #)

This command changes the character used by the given sprite to the value given by the second variable. This command allows you to quickly change the definition of a sprite. The definitions must first be set using *CALL SCHARA*.

CALL COLORA(SPRITE NUMBER, FOREGROUND COLOR)

This instruction allows you to specify the foreground color of a sprite.

```
EXAMPLE: 100 LET A$="183C7E1818181800"
         105 LET B$=""
         110 LET A@=1
         120 LET B@=1
         130 LET C@=2
         140 LET D@=20
         150 LET E@=20
         160 LET F@=4
         170 LET G@=4
         175 LET H@=2
         180 REM GO INTO SPRITEMODE
         190 CALL SPRITEMODE
         192 REM CHANGE THE PATTERN OF THE SPRITE
         194 CALL SCHARA(A@,A$,B$)
         195 REM CREATE SPRITE #1 AT ROW 20,
                 COLUMN 20
         200 CALL SPRITEA(A@,B@,C@,D@,E@)
         205 REM SPRITE MOVES WITH X VELOCITY
                 4, Y VELOCITY 4
         210 CALL MOTIONA(A@,F@,G@)
         230 INPUT H@
         240 CALL COLORA(A@,H@)
         250 GOTO 230
         260 STOP
```

CALL LOCATEA(SPRITE #,ROW, COLUMN)

The *LOCATEA* subroutine changes the location of a sprite. *ROW* is an integer variable with a value between 0 and 191. *COLUMN* is an integer variable with a value between 0 and 255.

CALL POSITIONA(SPRITE NUMBER, ROW, COLUMN)

This subroutine returns the position of the given sprite in *ROW* and *COLUMN*. See Extended Basic routine *POSITION*.

CALL MAGNIFYA(MAGNIFICATION)

This command specifies the size of sprites. The value must be between 1 and 4. A "1" is normal size and a "2" causes the sprite to be doubled (four consecutive characters are used to define the sprites). A "3" specifies the sprite to be magnified (one character is used to define the sprite but the size is magnified by "2"). And a "4" causes the sprite to be magnified and doubled (takes up 16 character positions). See Extended Basic manual.

CALL DELSPRITEA(SPRITE NUMBER)

This subroutine deletes the sprite specified. (See *DELSPRITE* subprogram in *TI EXTENDED BASIC*). If the sprite number is greater than 32 all sprites are deleted. Otherwise the value should be between 1 through 32

CALL SCREEN(COLOR)

The *SCREEN* command is used to change the color of the screen.

```
EXAMPLE: 100 LET X@=2
         110 CALL SCREEN(X@)
         120 STOP
```

CALL SCROLL(TYPE,LEFT/RIGHT,AMOUNT)

This command changes the scrolling of the screen so that it scrolls left to right or right to left a specific amount.The scroll is performed on *PRINT* and *INPUT* statements. *TYPE* must be a "0". If the second value is a 0 the scroll is to the left, and if it is a 1, the scroll is to the right. The third value indicates the number of columns to be scrolled.

```
EXAMPLE: 120 LET A@=0
         130 LET D@=1
         140 LET L@=1
         150 INPUT D@
         160 INPUT L@
         170 CALL SCROLL(A@,D@,L@)
         180 INPUT A@
         190 GOTO 180
         200 STOP
```

CALL SCROLL(TYPE, ROW, COLUMN, AMOUNT, TOP/BOTTOM)

This command changes the scrolling of the screen so that only a section of the screen scrolls up. *TYPE* must be "1". Row and column indicate the position from which the scroll will start or end. The amount indicates the number of columns that will be scrolled to the right of the position given by row and column. That is, only these columns will be scrolled. If *TOP/BOTTOM* is "0" the bottom part of the screen is scrolled up and if *TOP/BOTTOM* is "1" the top part of the screen is scrolled up.

```
EXAMPLE: 100 LET A@=1
         110 LET B@=10
         120 LET C@=11
         125 REM THIS COMMAND CHANGES THE SCROLL SO THAT
                 ONLY THE AREA OF THE SCREEN BETWEEN COLUMN
                 11, COLUMN 21
         126 REM AND ABOVE ROW 10 IS SCROLLED UP ONE ROW
                 EACH TIME A PRINT OR INPUT COMMAND IS USED
         130 CALL SCROLL(A@,B@,C@,B@,A@)
         140 INPUT A@
         150 GOTO 140
         160 STOP
```

CALL DISTANCEA(TYPE, SPRITE # 1, SPRITE # 2, DISTANCE)

This command is used to find the distance between two sprites. *TYPE* must be a "1", and *DISTANCE* is returned with the square of the distance between the sprites.

CALL DISTANCEA(TYPE,SPRITE#1,ROW,COLUMN,DISTANCE)

TYPE must be a "2" and *DISTANCE* is returned with the square of the distance between the sprite and a location. See *DISTANCE* subroutine program of TI Extended Basic. To simulate this command in *EDITOR/EX* you must refer to this as *CALL DISTANCEB*. This command must be changed back to *CALL DISTANCEA* before compiling.

```
EXAMPLE: CALL DISTANCEB(A@,B@,C@,D@,E@)
```

CALL COINCA(TYPE, SPRITE # 1, SPRITE # 2 TOLERANCE, COINCIDENCE)

This command detects a coincidence between 2 sprites. *COINCIDENCE*="-1" if there is a coincidence and equals "0" if there is no coincidence. *TOLERANCE* is the number of dots allowed between the sprites. (See *COINC* subprogram in TI Extended Basic). *TYPE* must be 1.

CALL COINCA(TYPE, SPRITE #, ROW, COLUMN, TOLERANCE, COINCIDENCE)

This detects a coincidence between a sprite and the location specified by *ROW* and *COLUMN*. *TYPE* must be a "2". To simulate this command in *EDITOR/EX* you must refer to this command as *CALL COINCB*. This command must be changed back to *CALL COINCA* before compiling.

CALL COINCA(TYPE, COINCIDENCE)

This subprogram detects a coincidence between any sprites. *TYPE* must be a "3" and *COINCIDENCE* is returned with a value of "-1" or "0". To simulate this command in *EDITOR/EX* you must refer to this command as *CALL COINCC*. This command must be changed back to *CALL COINCA* before compiling. The following program does nothing more than demonstrate the use of the sprite commands.

```
EXAMPLE: 100 LET A$="FFFFFFFFFFFFFFFF"
         110 LET N$=""
         120 LET A@=1
         130 LET B@=2
         140 LET C@=20
         142 LET K@=1
         145 LET H@=1
         150 LET D@=4
         155 LET I@=1
         160 LET E@=3
         162 LET X@=20
         165 REM SET UP SPRITE MODE
```

```
170 CALL SPRITEMODE
175 REM SET UP SPRITES
180 CALL SPRITEA(A@,A@,B@,C@,C@)
190 CALL SPRITEA(B@,B@,B@,C@,D@)
200 CALL MOTIONA(A@,D@,D@)
210 CALL MOTIONA(B@,D@,C@)
220 CALL SCHARA(A@,A$,A$,N$)
225 REM LIKE COINC(ALL,K@)
230 CALL COINCA(E@,K@)
240 PRINT K@
245 REM DIST. OF SPRITE TO LOCATION
250 CALL DISTANCEA(B@,A@,C@,D@,E@)
260 PRINT E@
265 REM COINC. OF SPRITE TO A LOCATION
270 CALL COINCA(B@,A@,C@,D@,E@,X@)
280 PRINT X@
290 INPUT C@
300 INPUT D@
305 REM CHANGES THE LOCATION OF THE
      SPRITE
310 CALL LOCATEA(A@,C@,D@)
313 REM GIVES THE LOCATION OF THE
      SPRITE
315 CALL POSITIONA(A@,H@,I@)
320 PRINT H@
325 PRINT I@
326 REM DELETES A SPRITE
330 CALL DELSPRITEA(A@)
340 GOTO 340
350 STOP
```

CALL RANDOMIZE

This command sets a new random seed for the *RND* command.

CALL RND(INTERVAL, RANDOM VARIABLE)

This command returns, in the second variable, a random number in the interval from 0 to the value of the first variable minus 1.

CALL SCREENON

This command keeps the screen active even if no key has been hit in 5 minutes. The following commands also keep the screen active

SPRITEA, MOTIONA, LOCATEA, POSITIONA, COINCA, DISTANCEA, SCHARA, COLORA, MAGNIFYA, DELSPRITEA, JOYST, KEY and INPUT.

ARITHMETIC STATEMENTS

Arithmetic statements may be either in integer or floating point arithmetic. They cannot be mixed. However, a call can be made to *FLOAT* or *INTER* before the computation. The ***SST*** *EXPANDED BASIC COMPILER* will allow more than one computation per line. However all arithmetic computations are performed from left to right.
NOTE: This differs from T.I. Basic.
If you access one of the mathematical functions, only one computation may be performed on the line.

EXAMPLE: A=SQR(B)

Notice also that only variables can be used in arithmetic computations; constants are not allowed. For example, the statement:

```
A=B+2
must be written
LET C=2
A=B+C
```

That is, all variables and constants must be declared in a *LET* statement at the beginning of the program. Note, the word *LET* must not be used in an arithmetic statement.

SAMPLE PROGRAMS

The following examples indicate how programs should be written using the ***SST*** *EXPANDED BASIC COMPILER*. The operating time for the TI Basic programs and the machine language programs are given for comparison.

PROGRAM 1

The first program gives an example of a loop from 1 to 30000, in steps of 1.

```
T.I. BASIC                 SST COMPILER

100 FOR I=1 TO 30000       100 LET I@ =1
200 NEXT I                 110 LET J@=30000
300 STOP                   120 FOR I@=I@ TO J@
                           130 NEXT I@
                           140 STOP

TIME: 84 Seconds           TIME: 1.40 seconds
```

PROGRAM 2

This program is similar to the previous one, except that it uses an *IF* statement in place of the *FOR* loop.

```
T.I. BASIC                 SST COMPILER

100 L=0                    100 LET A=1
110 L=L+1                  110 LET L=0
120 A=L-29999              120 LET M=29999
130 IF A<=0 THEN 110       130 LET Z=1
140 STOP                   140 L=L+Z
                           150 A=L-M
                           160 IF A<=0 THEN 140
                           170 STOP

TIME: 415 seconds          TIME: 73 seconds in floating point;
                                 3 seconds in integer arithmetic
                           (To convert to integer arithmetic,
                           put an @ symbol after each variable)
```

PROGRAM 3

If you would run Program 2 a second time, without reloading it into memory, it would not run correctly. This is because the value stored in L would not be 0, but would be the value of L at the end of the first run (i.e., L=30000). Therefore, if you are going to run a program many times while it is stored in memory, you should be certain the variables are initalized correctly. The second program should be written as shown below:

```
100 LET A=1
110 LET L=0
120 LET M=29999
130 LET K=0
140 LET Z=1
150 L=K
160 L=L+Z
170 A=L-M
180 IF A<=0 THEN 160
190 STOP
```

In this form L will be set to zero in line 150, and the program can be run repeatedly. The addition of line 150 is necessary because execution always begins at the first line following the *LET* statements.

PROGRAM 4

The next program is similar to one which appeared in the March, 1980 ***BYTE*** Magazine. It is a program designed to generate prime numbers, and is often used as a benchmark. The program was originally run in Basic on the TRS-80 computer. It took 7 hours, 12 minutes to check the first 10,000 integers for prime numbers. The program written here checks only the first 1,000 integers.

```
100 LET L@=6
110 LET E@=1
120 LET M@=1000
130 LET Z@=5
140 LET A@=1
150 LET N@=10
160 LET D@=1
170 LET B@=2
180 LET C@=2
190 FOR A@=L@ TO M@
200 A@=A@+E@
210 D@=A@/C@
220 FOR Z@=B@ TO D@
230 Z@=Z@+E@
235 REM FOR T.I. BASIC LINE 240 SHOULD BE
236 REM                 N@=INT(A@/Z@)
240 N@=A@/Z@
250 N@=N@+Z@
260 N@=A@-N@
270 IF N@<=0 THEN 300
280 NEXT Z@
290 PRINT A@
300 NEXT A@
310 STOP
```

TIME, BASIC: 1535 seconds
TIME, ***SST*** *COMPILER*: 18 seconds

If line 120 is changed from M@=1000 to M@=10000, the program will check the first 10,000 integers. The ***SST*** *EXPANDED COMPILER* completes the program in 11 minutes, 20 seconds. In *T.I. BASIC*, it took 4 hours and 15 minutes to check the first 5500 integers. The ***SST*** *EXPANDED COMPILER* took 4 minutes to check the first 5500 integers.

PROGRAM 5

This program illustrates some of the graphics capabilities of the compiler. The program creates a man using graphics commands and moves the man across the screen.

```
EXAMPLE:100 LET B$="1899FF3C3C3C2221"
        110 LET A$="995A3C3C3C3C4484"
        120 LET A@=2
        130 LET D@=42
        140 LET E@=3
        150 LET F@=32
        160 LET I@=1
        170 LET J@=12
        180 LET K@=1
        190 LET L@=1
        200 LET M@=1000
        205 LET T@=32
        207 LET S@=0
        210 LET N@=52
        215 REM DEFINES THE MAN
        220 CALL CHAR(D@,A$)
        230 CALL CHAR(N@,B$)
        240 CALL CLEAR
        250 INPUT M@
        260 FOR I@=E@ TO F@
        265 REM PRINT CHARACTER FOR MAN
        270 CALL VCHAR(J@,I@,D@)
        275 REM DELAY LOOP
        280 FOR K@=L@ TO M@
        290 NEXT K@
        295 REM PRINT CHARACTER FOR MAN.
        300 CALL VCHAR(J@,I@,N@)
        305 DELAY LOOP.
        310 FOR K@=L@ TO M@
        320 NEXT K@
        325 S@=I@-L@
        326 REM ERASES THE OLD LOCATION OF MAN.
        330 CALL VCHAR(J@,S@,T@)
        335 CALL VCHAR(J@,I@,T@)
        340 NEXT I@
        350 STOP
```

PROGRAM 6

This program is an example of *PLOT MODE*. The program prints the word *TEST* at the bottom left hand corner and then allows you to draw designs on the screen. NOTE: There are no restrictions to the inputs given by the *JOYSTICK* command. If you go off the screen you may destroy the screen image.

```
EXAMPLE: 100 LET A$="TEST"
         110 LET A@=1
         120 LET B@=4
         130 LET C@=1
         140 LET I@=1
         150 LET J@=1
         160 LET K@=1
         170 LET CF@=2
         180 LET CB@=1
         190 LET X@=128
         200 LET Y@=96
         210 LET AX@=1
         220 LET AY@=1
         230 LET BX@=48
         240 LET BY@=160
         250 LET BN@=8
         255 REM ENTERS PLOTMODE.
         260 CALL PLOTMODE
         265 REM GETS LENGTH OF STRING.
         270 CALL LEN(K@,A$)
         280 FOR I@=J@ TO K@
         285 REM GETS THE ASCII CODE OF EACH CHARACTER
         290 CALL ASC(C@,A$,I@)
         300 BX@=BX@+BN@
         305 REM PRINTS LETTER.
         310 CALL PLOTCHR(C@,BX@,BY@,CF@,CB@,A@)
         320 NEXT I@
         325 REM PLOTS THE NEW POINT.
         330 CALL PLOT(X@,Y@,CF@,CB@,A@)
         335 REM CHANGES LOCATION OF THE POINT.
         340 CALL JOYST(A@,AX@,AY@)
         350 AX@=AX@/B@
         360 AY@=AY@/B@
         370 X@=X@+AX@
         380 Y@=Y@-AY@
         390 GOTO 330
         400 STOP
```

PROGRAM 7

This program will print a file created using the *EDIT* program of the *EDITOR/ASSEMBLER* module, to a device such as a printer. It is particularly useful if you need a large number of copies. It will ask you for a file name. This should be a Disk Text File in a variable display 80 format. Then you will be asked for the Print Name. Possible answers are: PIO.EC or RS232. Next you will be asked for the number of lines to be printed. Enter a number from 1-200. Lastly you are asked for the number of copies to be printed. The program will then load the proper number of lines into a string array and display a counter for each copy printed. To stop the printing before the computer is done you may press any key. This program illustrates several commands and techniques to allocate large integer arrays without having the compiler initialize it. The program is compiled in parts. In the first part the variables and constants are defined. This section should be compiled at a starting address of -20000. The second part must be compiled without reloading the compiler. This section should be compiled with a starting address -3000. The space between the two parts will be used for a string array. Although only two elements are dimensioned, the compiled program will not check if you go beyond these.

NOTE: You execute the second part with the *CALL LINK* command in *TI BASIC*.

```
EXAMPLE: 90 REM START PROGRAM AT ADDRESS -20000
         100 LET CB$="PRINT NAME"
         110 LET CA$="FILE NAME"
         120 LET A$=""
         130 LET B$=""
         160 LET CC$="# OF LINES"
         170 LET CD$="# OF COPIES"
         180 LET CE$="PRESS ANY KEY TO ABORT"
         190 LET CF$=" COPY COUNT"
         195 REM ALOCATE SPACE FOR 80 CHARACTER STRING
         200 LET C$=""
         201 LET C$=""
         202 LET C$=""
         203 LET C$=""
         210 LET CA@=22
         215 LET CI@=2
         220 LET CB@=13
         230 LET CC@=1
         240 LET CD@=0
         250 LET A@=0
         260 LET B@=0
         270 LET CE@=12
         280 LET CF@=16
         290 LET CG@=80
         300 LET CH@=-3
         310 LET S@=0
         320 LET K@=0
         330 LET I@=0
```

```
340 LET AI@=0
345 REM E@ IS THE POINTER FOR THE STRING ARRAY
350 LET C@=0
355 LET E@=0
360 DIM AR@(1)
370 STOP

90 REM START AT ADDRESS -3000
100 CALL CLEAR
105 REM FORMAT INPUTS
110 CALL INPUTAT(CA@,CB@)
120 CALL PRINTAT(CA@,CI@)
130 CALL SCRON(CD@)
140 PRINT CA$
145 CALL SCRON(CC@)
150 INPUT A$
160 CALL SCRON(CD@)
170 PRINT CB$
180 CALL SCRON(CC@)
190 INPUT B$
200 CALL SCRON(CD@)
210 PRINT CC$
220 CALL SCRON(CC@)
230 INPUT A@
240 CALL SCRON(CD@)
250 PRINT CD$
260 CALL SCRON(CC@)
270 INPUT B@
275  CALL SCRON(CD@)
280 CALL CLEAR
290 PRINT CE$
300 CALL PRINTAT(CE@,CC@)
310 PRINT CF$
320 CALL PRINTAT(CE@,CF@)
325 REM OPEN A DISPLAY VARIABLE 80 DISK FILE
330 CALL OPEN(CC@,CF@,CG@,CD@,A$)
340 C@=CH@
350 FOR I@=CC@ TO A@
360 CALL INPUT(CC@,CD@,C$)
370 C@=C@—CH@
375 REM C@ COUNTS BY 3 FOR STRING ARAYS
380 CALL INSTRINGA(C@,E@,C$)
390 CALL KEY(CD@,K@,S@)
400 IF S@=00 THEN 420
410 GOTO 570
420 NEXT I@
430 CALL CLOSE(CC@)
435 REM OPEN PRINTER FILE
440 CALL OPEN(CC@,CF@,CG@,CD@,B$)
```

```
450 FOR AI@=CC@ TO B@
460 PRINT AI@
470 C@=CH@
480 FOR I@=CC@ TO A@
490 C@=C@-CH@
500 CALL OUTSTRINGA(C@,E@,C$)
510 CALL PRINT(CC@,CD@,C$)
520 CALL KEY(CD@,K@,S@)
530 IF S@=00 THEN 550
540 GOTO 570
550 NEXT I@
560 NEXT AI@
570 CALL CLOSE(CC@)
580 STOP
```

As you can see from the results, the ***SST*** *COMPILER* is a powerful tool, and should be used whenever speed is important. Once the program is compiled, the result is a machine language program equivalent to the Basic program. To *RUN* the program, it is only necessary to read in the machine language file. The program does not need to be compiled again.

PROGRAM 8

This program is Compiled and can be found on the ***SST*** *EXPANDED BASIC COMPILER* system disk. To load and run the program follow the steps below.

```
CALL INIT
CALL LOAD("DSK1.MOVE","DSK1.DUMP",
              "DSK1.GAME")
Leave Disk in Place
CALL LINK("GAME")
```

You will be given more information about this game program by the program itself.

PROGRAM 9

This is an example program to show how to use the speech synthesizer. The input on line 220 consists of the decimal equivalent of the addresses given in the *EDITOR/ASSEMBLER* manual pages 422-427. These addresses correspond to the words residing in the synthesizer.

```
100 LET A@=0
105 LET Z@=0
110 LET AA@=0
120 LET B@=0
130 LET C@=0
140 LET D@=0
150 LET E@=0
160 LET F@=256
170 LET G@=16
180 LET H@=64
190 LET I@=-27648
200 LET J@=80
210 CALL SPRITEMODE
220 INPUT A@
225 REM THIS CONVERTS THE DECIMAL ADDRESS TO THE
      CORRECT FORM. See page 351 of the EDITOR/
      ASSEMBLER manual.
230 AA@=A@/F@
240 C@=AA@*F@
242 A@=A@-C@
250 C@=A@/G@
260 B@=Z@-C@*G@+A@+H@
265 C@=C@+H@
270 E@=AA@/G@
280 D@=Z@-E@*G@+AA@+H@
285 E@=E@+H@
290 CALL LOAD(I@,B@)
300 CALL LOAD(I@,C@)
310 CALL LOAD(I@,D@)
320 CALL LOAD(I@,E@)
330 CALL LOAD(I@,H@)
335 CALL LOAD(I@,J@)
340 GOTO 220
```

IN CASE ROBLEMS . . . .

The error statement *"VARIABLE NOT FOUND"* usually indicates that a variable was not defined in a *LET* statement at the beginning of the program. The error statement *"MEMORY FULL"* is usually caused by not doing a *"CALL FILES(1)"* followed by a *"NEW"*.

CHECKING PROCEDURES

1. Do not mix integer and floating point variables. Use *CALLS* to *FLOAT* or *INTER* first.
2. If more than one arithmetic operation is performed per line, computation will be performed from left to right.
3. Integer variables must be used in *FOR* loops, and as subscripts for array elements.
4. Only three forms of the *IF* statement are allowed.

```
IF A<=0 THEN 150              IF A@<=0 THEN 150
IF A>=0 THEN 150      or      IF A@>=0 THEN 150
IF A=00 THEN 150              IF A@=00 THEN 150
```

5. Only one element is allowed in *PRINT* and *INPUT* statements.
6. A program cannot be run without reloading, unless the variables are initialized at the beginning of the program. See examples in Programs 2 and 3.
7. Do not use a *COMPILER* program that has been listed, edited, duplicated or changed in any way. The program may not run correctly. The *COMPILER* program is written in such an efficient way that any change may cause an error.
8. *CALL FILES(1)* must have been performed before the *COMPILER* is used.
9. The editor must have been resequenced using *RES* 1,1 before running.
10. Some statements are not executable in *T.I. BASIC* and cannot be included in test runs (e.g. *FLOAT, INTER*).
11. Most of the parameters in the Call statements use integer variables unless specified otherwise.
12. If the screen and characters become disrupted, check to see that you are not doing a *PRINT* or *INPUT* to a position below the screen.
13. If the computer locks up or behaves strangely check that an array or a string allocation has not been exceeded.
14. If you obtain the same string on inputs from a disk drive, you may have tried to read beyond the end of the file or you may not have changed the record #. If you use a fixed display type file the string will be filled with spaces. Make sure it is of the specified fixed length.
15. Check that all statements are exactly as in the manual.

THANK YOU!

NOTE YOU MAY NOT WRITE A NULL STRING TO A DEVICE. THAT IS, A STRING THAT HAS A LENGTH OF ZERO CANNOT BE WRITTEN TO EITHER THE RS232 CARD OR DISK DRIVE.

CHANGES TO MANUALS:

CHANGE IN MANUAL "SST EXPANDED BASIC COMPILER SYSTEM " PAGE 4 LINE:

```
     "MEMORY EXPANSION -24064 TO -1     (HEX. A200 TO FFFF)
TO   "MEMORY EXPANSION -24064 TO -256   (HEX. A200 TO FF00)
```

CHANGE IN MANUAL "SST EXPANSION PACKAGE" PAGE 3 LINES:

```
     "CALL MAGNIFY(FACTOR)"
TO   "CALL MAGNIFYP(FACTOR)"

     "200 CALL MAGNIFY(M@)"
TO   "200 CALL MAGNIFYP(M@)"

     "210 PRINTAT(A$,X@,X@,C@,G@)"
TO   "210 CALL PRINTAT(A$,X@,X@,C@,G@)"
```

INDEX OF COMMANDS

AMENDMENTS

YOU HAVE JUST RECEIVED VERSION (SST5) OF THE COMPILER SYSTEM. THIS VERSION HAS THE ABILITY TO DETECT MORE THAN ONE VARIABLE THAT HAS NOT BEEN DEFINED WITH A "LET" STATEMENT IN YOUR PROGRAM. IN MOST CASES WHEN THE VARIABLE IS NOT DETECTED A SOUND WILL BE HEARD AND THE VARIABLE WILL BE PRINTED ON THE SCREEN WITH ITS LINE NUMBER. THE COMPILER WILL CONTINUE WHEN YOU PRESS ANY KEY.

YOU WILL FIND WITH THIS NEW VERSION THAT A VARIABLE NOT BEING FOUND IS NOT ALWAYS FATAL TO THE COMPILED PROGRAM. THAT IS, IF THE VARIABLE THAT WAS NOT DEFINED IS NOT ESSENTIAL TO THE EXECUTION OF YOUR COMPILED PROGRAM THE PROGRAM MAY STILL RUN PROPERLY.

THERE ARE SEVERAL CASES IN WHICH THE COMPILER WILL NOT DETECT "VARIABLE NOT FOUND". THEY ARE:

1: IF THERE ARE NO "LET" STATEMENTS AT THE BEGINNING OF YOUR PROGRAM AND AN "A, A@ OR A$" IS USED IN YOUR PROGRAM. THESE THREE VARIABLES WILL NOT BE DETECTED, BUT OTHERS WILL BE.

2: THE VARIABLE IN A "NEXT" STATEMENT WILL NOT BE DETECTED. THIS ERROR SHOULD BE DETECTED BY THE EDITORS OR WHEN THE VARIABLE IS NOT FOUND IN THE "FOR" STATEMENT.

3: IF BY ACCIDENT YOU MIX VARIABLE TYPES IN ARITHMETIC STATEMENTS IT MAY CAUSE THE VARIABLE TO BE IMPROPERLY PRINTED TO THE SCREEN AND/OR DISRUPT THE SCREEN. BUT THE LOCATION OF THE ERROR WILL BE DETECTED. UNDEFINED DIMENSIONED VARIABLES MAY CAUSE THE SAME EFFECT.

THE SST EXPANSION PACKAGE IS INCLUDED ON YOUR SST EXPANDED BASIC SYSTEM DISK. FOR THIS REASON PLEASE IGNORE PARAGRAPH 6 ON PAGE (1).

WE HAVE FOUND THAT IN SOME RECENT VERSIONS OF THE TI 99/4A COMPUTERS OUR SST EXPANDED BASIC LOADER WILL NOT FUNCTION PROPERLY WITH THE EDITOR/ASSEMBLER MODULE IN PLACE.

IF YOUR COMPUTER IS THIS VERSION THE LOADER WILL GIVE YOU AN ERROR AT LINE NUMBER 30200 (NUMBER TOO BIG). TO MAKE YOUR LOADER WORK WITH THE EDITOR/ASSEMBLER MODULE YOU MUST MAKE THE FOLLOWING CHANGES TO THE LOADER.

1. REMOVE LINES 30016, 30017, 30018.

2. ADD LINE: 30016 CALL LOAD(11310,16)

SAVE THIS NEW LOADER ON ANOTHER DISK AND USE IT ANY TIME YOU HAVE THE ED/AS MODULE IN PLACE. USE THE ORIGINAL LOADER IF YOU ARE USING THE MINI MEMORY.

THE FASTLOAD PROGRAM MUST ALSO BE CHANGED. REMOVE LINE 210 FOR THIS RECENT VERSION.

- PRE/SST PREPROCESSOR PROGRAM -

1. - PURPOSE AND LIMITATIONS -

The purpose of this program is to aid home computer users in converting an existing TI-99 Extended BASIC or BASIC computer program into a modified version which complies with the structure and format requirements of the SST EXPANDED BASIC COMPILER. What the program does is:

- o Changes all variables into two-character variable names;
- o Changes all string and numeric constants into variables;
- o Inserts "LET" statements at the beginning of the program for all variables and constants;
- o Sequences the "LET" and "DIM"statements - strings first, followed by reals, followed by integers;
- o Flags all lines which contain string arrays;
- o Breaks all multiple statement lines (::) into separate lines;
- o Deletes in-line (!) remarks, or optionally converts them into separate-line REM statements.

The program is only an aid, and does not completely prepare the program for the SST COMPILER. After running the preprocessor program, the user still will have to:

- o Eliminate any string arrays, replacing them with integer arrays and utilization of the INSTRING and OUTSTRING subroutines (remember that the preprocessor helps you do this by flagging all lines which contain any string arrays);
- o Replace the intrinsic Extended BASIC library functions, such as OPEN, CLOSE, SEG$, LEN, etc., with their corresponding SST COMPILER subroutines;
- o Reformat IF statements per the SST COMPILER restrictions (taking special note to the IF statements in lines which were formerly all on the same line - PRE/SST helps here by flagging such lines as "Fractionated IF" occurences);
- o Identify string variables and constants which are longer than 26 characters, and add additional LET statements accordingly;
- o Convert DIM to DISPLAY for any large arrays;
- o Eliminate all mixed mode arithmetic statements, using the CALL FLOAT and CALL INTER subroutines.

The SST COMPILER will help you effect these residual modifications, because it will supply error messages for any of the above items not completed.

2. - USER INSTRUCTIONS -

To preprocess a program use Extended Basic and perform the following:

a. use the OLD command to load the program to be preprocessed;

b. make sure all OPTION BASE and DIM statements precede any executable code - move them up to the front of the program - you can use the REDO LINE function (FCTN 8) to create a copy whose statement number you can change - then go back and delete the line which was out of place; do not put your OPTION BASE and DIM statements in a multiple statement (::) line;

c. execute RESEQUENCE 400 (NOTE: if your program has within it lines containing many statements separated by double colons - :: - then you should allow a line increment greater than 10, because PRE/SST inserts new line numbers for the multiple statements, as well as adding REMARKS flagging string arrays, etc. Thus, if your program has as many as, say, 12 statements in one single line, then you should execute RESEQUENCE 400,20). Store the program back to disk, with a new name, e.g. TARGET, in the MERGE format - i.e., execute SAVE DSK1.TARGET, MERGE;

d. now load the preprocessor program - i.e., execute OLD DSK1.PRE/SST, and RUN (or in one step, RUN "DSK1.PRE/SST") or use the loader provided;

e. when prompted for the "INPUT FILE NAME", enter DSK1.TARGET - or whatever name you have given the program saved in the MERGE format;

f. next you are prompted "VARIABLES: 0-PROMPT,1-FIXED,2-FLOAT" - if you know that all of the variables in your program are integers ("fixed point", the advantage of using integers is a reduction in the run-time of the compiler version), then reply with a 1; if you know that all variables must be real ("floating point"), then reply with a 2; if you want to decide individually for each variable, enter 0 (NOTE: PRE/SST will retain as an integer any variable which ends with the "@" symbol, consistent with the SST Compiler convention - if your program follows this convention, then respond to this prompt with a 2, and all variables will be real except those which end with "@");

g. Next you are prompted "IN-LINE (!) REMARKS: 0-DELETE, 1-CONVERT TO REM, 2-LEAVE AS-IS"; based on how you respond, your same-line REMARKs will be deleted, changed to a REM statement on the following line, or left unchanged - remember, the SST COMPILER will not accept in-line REMARKs;

h. Next you are requested to SPECIFY PRINTER - respond with your printer's characteristics - PIO, PIO.LF, etc for parallel printer, or RS232.BA=1200.LF, etc., for a serial printer.

i. if you have entered a 0 in step f, then, for each variable name in the program to be processed (TARGET in the example), a prompt will appear - "INTEGER (1) OR REAL (2)", followed by the variable name - respond accordingly;

j. when processing is complete, the program will ask you "ANY MORE FILES (Y/N)?" - if you respond "Y", then the whole process repeats, starting with step e. Each related program to be processed must use the same variable set; if any new variables are defined in the new program(s) then they will appear in LET/DIM statements in that program - however, the SST COMPILER will not permit any new LET/DIM statements in the secondary programs; therefore it is necessary to move all the variable definition statements (LET/DIM) to the first-processed program, even if some of these variables are not referenced until the subsequent programs;

k. when completed, this preprocessing results in the processed programs on disk with the "^" character appended to their names - e.g. TARGET^ - now you must complete the reformating process as described in paragraph 1 above. Pay particular attention to the "FRACTIONATED IF" statements - IF statements which are within a multiple statement line. Extended BASIC provides for IF statements which, if false, result in none of the following same-line statements being executed. However, when the PRE/SST Program breaks up your multiple line statements, any IF statements included are potentially not going to perform as planned. Special attention must be given these lines. Finally, MERGE the program (e.g., TARGET^) into the SST EDITOR/EX, and proceed from there to produce the compiled version of your TARGET program.

One final suggestion: you may wish to perform some manual editing of your program into the SST Compiler format before using the PRE/SST program. For example, you can replace the intrinsic Extended BASIC functions with their corresponding SST Compiler subroutine CALL statements before applying the PRE/SST Program, while you can still use imbedded numeric constants. On the other hand, you may not want to manually revise your IF statements prior to applying PRE/SST, because PRE/SST will replace the zeros in the IF statements with variables.