CALIFORNIA INSTITUTE OF TECHNOLOGY

TTM: A MACRO LANGUAGE FOR BATCH PROCESSING

by

Stephen H. Caine

E. Kent Gordon

Willis H. Booth Computing Center

Programming Report No. 8

May 1969

TABLE OF CONTENTS

INTRODUCTION

TTM is a recursive, interpretive language designed primarily for string manipulation, text editing, macro definition and expansion, and other applications generally classified as systems programming. It is derived, primarily, from GAP [12] and GPM [11].

Initially, TTM was planned as the macro processing portion of an assembler for the IBM System/360 and, as such, was designed to overcome the restrictions and inconsistencies which existed in the standard assemblers for that system [9,10]. In addition, it was designed to have all of the power possessed by earlier general macro assemblers but with the unfortunate syntactic and semantic difficulties removed [4,5,6,7,8].

During the development of TTM, it became apparent that applications other than assembler macro processing were possible. These include data editing, text manipulation, expression compiling, and macro processing for language processors other than assemblers.

The initial version of TTM was implemented to run in a conversational manner under the Caltech Basic Time Sharing System for the IBM System/360-50 [13]. That version is described in [14]. This report describes a version which can be run as a self-contained processor under Operating System/360 or can be interfaced directly with other language processors.

LANGUAGE STRUCTURE

Characters and Strings

The basic unit of information in TTM is the character. The characters acceptable to TTM are the EBCDIC characters whose hexadecimal representation are X'40' through X'F9' inclusive. In addition, an internal character is used to represent the end of an input line.

In certain contexts the special characters

< > # ; @ end-of-line

act as control characters. The contexts under which this special treatment occurs are discussed below.

For ease of processing, characters are normally collected into units known as strings. A string is a sequence (possibly empty) of characters which is usually manipulated as a single entity. Strings may be placed into the TTM dictionary (memory), in which case they must be named. The name of a string consists of an arbitrary number (including zero) of characters.

Functions

The contents of a string may be arbitrary text of some kind or it may represent a program which is to be executed by TTM. In general, program and text (data) are indistinguishable except by the context in which they are used.

When a string is fetched out of memory and executed, text may be substituted for previously declared substrings or bound variables. Thus, a string to be executed may be considered, in some sense, as a function. In addition to such programmer-defined functions, TTM contains a number of built-in functions.

To invoke a function, the following notation is used:

$$\#\langle s_1;s_2;s_3;\ldots;s_n\rangle$$

where the s_i are strings and $n \geq 1$. The first string, s_1, is taken to be the name of the function and the remaining strings to be its parameters.

This form is known as an active call. A passive form also exists:

$$\#\#\langle s_1;s_2;s_3;\ldots;s_n\rangle$$

The differences between these two forms will be discussed in the section on the processing algorithm.

The pound sign (#) is recognized as a control character only when it appears in the context #< or ##<. In all other cases, it is simply considered to be part of a string.

The number of parameters supplied to a function is usually dependent upon the use of the function and the number of bound variables it is defined or declared to have. If fewer than the indicated number of parameters are supplied, programmer-defined functions will consider the remaining ones as null. However, some of the primitives cannot operate with fewer than a certain number of parameters. If more than the indicated number of parameters are provided, the excess ones will be ignored by both programmer-defined and primitive functions.

The Processing Algorithm

A description of the actual processing algorithm will prove useful in learning to write progrmas for TTM. With the processing algorithm in mind it is possible to completely predict the results of passing any sequence of characters to TTM for processing.

In general, TTM can be thought of as a computer with a somewhat unusual organization. To cause this "computer" to operate, a "program" in the language of TTM is placed in an area known as the program string. The TTM processor starts scanning at the beginning of the program string, executing function calls when encountered. Some of these may cause additional characters to be inserted into the program string. When the end of the program string is reached, TTM terminates processing. Depending upon the particular version of TTM used, the initial program may be entered from a conversational terminal, the input stream of a batch processor, or as an argument to a call on TTM from some other routine.

The actual processing consists of the three steps shown below. In this discussion, the symbols {, }, and ^ are used strictly as metalinguistic symbols. They are introduced solely as a notational convenience in explaining the algorithm and never occur in actual practice.

The processing algorithm can be expressed as:

1. Scan the program string, replacing #< with {, ##< with }, and ; with ^. If an end-of-line character is encountered, it is deleted. If a @ is detected, it is deleted and the following character is skipped without any examination for special significance. If a < not preceded by # or ## is detected, it is deleted and a scan is made for the matching > which is also deleted. During this "balancing bracket" scan, only the two brackets and the @ are of

significance. When encountered, the @ and the following character are passed over. The constructions #< and ##< as well as the ; and the carriage return are simply passed over. In any case, if the end of the program string is reached, processing is terminated.

2. When a > is encountered outside of a bracketed string scan, it signifies the end of the function call which begins with the immediately preceding { or [and whose parameters are separated by _. The first parameter is taken as the function name, the function is invoked, and the characters from the { or [to the matching >, inclusive, are deleted.

3. If the call was initiated by a {, the function value is placed to the right of the current scan position so that it will be encountered when scanning is continued. If the call was initiated by a [, the function value is placed to the left of the current scan position. In either case, scanning is continued at step one above.

Several examples may clarify the operation of the algorithm. In these, an upward-pointing arrow will be used to indicate the position of the scanner.

Example 1

The Print String function is written as

```
#<PS;string>
```

which has the side effect of causing string to be printed on an output device and the result of returning a null value. Consider the processing of

```
#<PS;123>
↑
```

which becomes

```
{PS;123>
  ↑
```

then

```
{PS^123>
    ↑
```

and finally prints 123 and results in a null program string.

Example 2

```
Given:    #<PS;<1;<2;3>;4>>
          ↑
Yields:   {PS;<1;<2;3>;4>>
             ↑
          {PS,1;<2;3>;4>
                      ↑
Result:   Print 1;<2;3>;4
```

Example 3

```
Given:    #<PS;@>>
          ↑
Yields:   {PS,@>>
             ↑
          {PS,>>
                ↑
Result:   Print >
```

Example 4

```
Given:    #<PS;<@<>>
          ↑
Yields:   {PS,<@<>>
             ↑
          {PS,@<>>
             ↑
          {PS,@<>>
                ↑
          {PS,@<>
                ↑
Result:   Print @<
```

Example 5

The add function is written as

#<AD;n_1;n_2>

and results in a value which is the sum of the numbers n_1 and n_2.

Given: #<PS;#<AD;1;2>>

Yields: {PS^#<AD;1;2>>

{PS^{AD^1^2>>

{PS^3>

{PS 3>

Result: Print 3

Example 6

Assume that the string named X contains the characters

1;2

Given: #<PS;#<X>>

Yields: {PS^#<X>>

{PS^{X>>

{PS^1;2>

{PS^1 2>

Result: Prints 1 because PS expects only one parameter, thus causing the second parameter to be ignored.

However, the same example but using a passive call on X gives:

```
Given:    #<PS;##<X>>
          ↑

Yields:   {PS_##<X>>
            ↑

          {PS_[X>>
             ↑

          {PS_1;2>
             ↑

Result:   Prints 1;2
```

Example 7

Assume that the string named Z has the value

```
PS
```

then,

```
Given:    #<##<Z>;123>
          ↑

Yields:   {[Z>;123>
            ↑

          {PS;123>
            ↑

          {PS_123>
             ↑

Result:   Prints 123
```

Example 8

Assume that the string named X has the value

```
##<AD;6;4>
```

then,

```
Given:    #<PS;##<X>>
          ↑
```

Yields: {PS^##<X>>

{PS^[X>>

{PS^##<AD;6;4>>

Result: Print ##<AD;6;4>

However,

Given: #<PS;#<X>>

Yields: {PS^#<X>>

{PS^{X>>

{PS^##<AD;6;4>>

{PS^[AD^6^4>>

{PS^10>

Result: Prints 10

TTM FUNCTIONS

The built-in TTM functions are described in this section. They are divided into groups of functions with similar format or operation and each group is prefaced by an introductory description.

A standard form is used to describe each operation:

Data: name n_1,n_2,r

where name is the function name as used in the call;

n_1 is the minimum number of parameters required;

n_2 is the maximum number of parameters used (an asterisk is used to denote an arbitrary number);

r is either V or S or VS to indicate that the function can have a non-null value, produces side effects, or both has a non-null value and produces side effects.

Call: Format of a call on the function

These are followed by a description of the parameters, values, and side effects.

DICTIONARY OPERATIONS

These functions are designed to define and erase named strings and to scan such strings for occurrences of specified substrings. A string so scanned is said to be segmented and the specified substrings will be replaced by segment marks.

When a named string is called out of the dictionary, the parameters specified in the call are positionally substituted for the corresponding segment marks. Missing parameters are taken as null and excess parameters are ignored.

In addition to segmenting, a string may also be scanned for substrings which are to be replaced by unique numbers each time the string is invoked. These numbers are useful in generating created symbols.

Within a single string, the number of segment marks may not exceed 62.

Segmentation is cumulative. That is, if a string is segmented on AA and BB in one operation and then later segmented on CC, it is equivalent to having segmented on AA, BB, and CC in a single operation.

The search for substring matches is performed on a character by character basis while scanning the string from left to right. Each substring is searched for separately with as many passes being made over the string as there are substrings.

Special treatment is not given to any character in the string and it is, therefore, possible to search for characters such as the brackets, the semicolon, or the end-of-line character.

If the characters of one substring are contained in another substring, the larger of the two should be searched for first. Thus, is is possible to scan for both THIS and IS if THIS is searched for first.

Associated with each string are two pointers: the initial pointer and the residual pointer. The initial pointer locates the true start of the string. When a string is assigned a value, the residual pointer is set equal to the initial pointer. Whenever TTM examines a string, it begins at the point specified by the residual pointer. As the various functions in the string selection group work on a string, the residual pointer is moved down the string, thus truncating it from the left. At any time, the programmer may cause the residual pointer to be reset to the value of the initial pointer.

Define a String

DS 2,2,S

#<DS;name;text>

Side Effects: the string text is placed in the dictionary and assigned the name of name.

Note: If name has been previously defined, it will be redefined with the new text. Built-in functions can be redefined in this manner.

Append to a String

AP 2,2,S

#<AP;name;text>

Side Effects: the string text is added to end of the named string and the residual pointer is positioned to the new end of the string.

Note: Append allows successive additions to be made to a string without copying the whole string each time an addition is made. The residual pointer will always be left at the end so it is necessary to reset it before using the string. However, for purposes of efficiency, the residual pointer should be left at the end as long as additional calls on AP are expected. If name is not defined, AP will be treated as DS.

Erase Strings

ES 0,*,S

#<ES;$name_1$,$name_2$,...>

Parameter: the $name_i$ are the names of strings.

Side Effects: The given strings and their names will be deleted from the dictionary.

Note: Strings which are no longer needed should be erased as the space occupied by them is thereby recovered. ES can be applied to built-in functions in which case the name of the function will be deleted.

Copy a Function

CF 2,2,S

#<CF;new-name;old-name>

Parameters: Two strings to be used as names.

Side Effects: New-name is assigned a value which is a copy of the value of old-name starting at the residual pointer.

Note: Segment marks and created symbol marks are copied. If old-name is the name of a built-in function, then new-name will be defined as if it were also that built-in function.

Segment a String

SS 1,63,S

#<SS;name;S_1;S_2;...;S_n>

Parameters: name is the name of a string.

The S_i, $1 \leq i \leq 62$, are strings.

Side Effects: All occurrences of the S_i in string name are replaced by segment marks. Scanning begins at the position marked by the residual pointer.

Segment and Count

SC 1,63,VS

#<SC;name;S_1;S_2;...;S_n>

Parameters: name is the name of a string.

The S_i, $1 \leq i \leq 62$, are strings.

Side Effects: All occurrences of the S_i in string name are replaced by segment marks. Scanning begins at the position marked by the residual printer.

Value: An unsigned decimal integer representing the number of segment marks placed into the string by this call.

Mark for Creation

CR 2,2,S

#<CR;name;S>

Parameters: name is the name of a string and S is a string.

Side Effects: All occurrences of S in string name are replaced by a special creation mark. Scanning begins at the position marked by the residual pointer.

Notes: Substring matching operates in the same way as SS operates. However, no matter how many different substrings are marked in a given string they will each be replaced by the same quantity when the string is invoked. This quantity will be a four-digit decimal number and will be different for each invocation.

STRING SELECTION

These functions provide various ways of selecting portions of named strings for processing.

Call One Character

CC 1,1,VS

#<CC;name>

Value: the next character from the position in string name marked by the residual pointer

Side Effect: The residual pointer is advanced by one character.

Call N Characters

CN 2,2,VS

#<CN;num;name>

Parameters: num is a positive decimal integer. name is the name of a string.

Value: the next num characters from the position in string name marked by the residual pointer.

Side Effect: The residual pointer is advanced by num characters.

Skip N Characters

SN 2,2,S

#<SN;num;name>

Parameters: num is a positive decimal integer. name is the name of a string.

Side Effect: the residual pointer of the string is advanced by num characters.

Initial Character Scan

ISC 4,4,VS

#<ISC;string$_1$;name;string$_2$;string$_3$>

Value: string$_2$ if the characters of string$_1$ match the characters of the string name which immediately follow the residual pointer

string$_3$ if there is not a match

Side Effect: if the characters do match, the residual pointer is advanced past these characters.

Character Scan

SCN 3,3,VS

#<SCN;string$_1$;name;string$_2$>

Value: If the string string$_1$ is a substring of the string name starting at the residual pointer, then the characters from the residual pointer to the beginning of the first occurrence of this substring are returned. Otherwise string$_2$ is returned.

Side Effect: If string$_1$ was a substring of string name starting at the residual pointer, than the residual pointer is moved past the first occurrence of this substring.

Call Parameter

CP 1,1,VS

#<CP;name>

Value: all of the characters from the position in string name marked by the residual pointer up to, but not including the next zero-level semicolon.

Side Effect: The residual pointer is spaced past the zero level semicolon.

Note: A zero-level semicolon is one not enclosed in brackets. In searching for a zero-level semicolon, TTM will process, but not delete, escape characters. Therefore, an escape character may be used to protect a bracket or a semicolon.

Call Segment

```
CS  1,1,VS
#<CS;name>
```

Value: all of the characters from the position in string name marked by the residual pointer up to, but not including, the next segment mark

Side Effect: The residual pointer is spaced past the segment mark.

Reset Residual Pointer

```
RRP  1,1,S
#<RRP;name>
```

Side Effect: The residual pointer of string name is returned to the beginning of the string.

STRING SCANNING OPERATIONS

These functions allow various portions of a string to be selected for processing or allow a string to be reformated without requiring the string to be defined as a named string.

Give N Characters

GN 2,2,V

#<GN;num;string>

Parameters: num is a signed or unsigned decimal integer.

string is an arbitrary string.

Value: If num is positive, the value is the first num characters of string. If num is negative, the value is all but the first num characters in string. If num is zero, the value is null.

Zero-Level Commas

ZLC 1,1,V

#<ZLC;string>

Value: The value is string with each zero-level comma replaced by a semicolon.

Notes: Zero-level commas are those not enclosed in parentheses. Brackets are ignored. The escape character can be used to protect parentheses and commas.

Zero Level Commas and Parentheses

ZLCP 1,1,V

#<ZLCP;string>

Value: The value is string with all zero-level commas and parentheses replaced by semicolons.

Note: Zero-level parentheses and commas are those not enclosed in other parentheses. In replacing commas and

parentheses by semicolons, care is taken not to change the number of delimited units in the string. Thus, A(B) will give A;B and (A),(B),C will give A;B;C.

Flip a String

FLIP 1,1,V

#<FLIP;string>

Value: string with its characters in reverse order.

Translate to Lower Case

TRL 1,1,V

#<TRL;string>

Value: string with all upper case alphabetic characters changed to lower case.

Translate Hexadecimal to Decimal

THD 1,1,V

#<THD;string>

Parameter: string is up to eight hexadecimal characters.

Value: signed decimal integer with leading zeros suppressed.

CHARACTER CLASS OPERATIONS

These operations allow string selection to be performed on the basis of the class to which the characters in the string belong.

The character classes have names but are quite different from strings and cannot be used as strings. It is possible to have a character class and a string with the same name.

Define a Class

DCL 2,2,S

#<DCL;clname;chars>

Side Effects: The class clname is defined to consist of the characters chars

Note: If the class clname has been previously defined, it will be redefined.

Define a Negative Class

DNCL 2,2,S

#<DNCL;clname;chars>

Side Effect: The class clname is defined to contain all characters except those in chars.

Note: If chars is null, the class of all characters will be defined.

Erase a Class

ECL 0,*,S

#<ECL;$clname_1$;$clname_2$;...>

Side Effect: The classes named by the $clname_i$ will be deleted.

Call Class

CCL 2,2,VS

#<CCL;clname;name>

Value: All of the characters from the position in string name marked by the residual pointer up to but not including the first character which is not a member of the class clname.

Side Effect: The residual pointer is advanced past the characters returned as the value.

Skip Class

SCL 2,2,S

#<SCL;clname;name>

Side Effect: The residual pointer of name is advanced until it reaches a character which does not belong to the class clname.

Test Class

TCL 4,4,V

#<TCL;clname;name;$string_1$;$string_2$>

Value: If the character at the position in string name marked by the residual pointer belongs to the class clname, the value is $string_1$, otherwise it is $string_2$.

Note: If name is null the value will be $string_2$.

ARITHMETIC OPERATIONS

The arithmetic functions perform fixed-point decimal arithmetic. The addition and subtraction functions accept fifteen digit operands and produce fifteen digit results. The multiplication function accepts fifteen digit operands and produces a thirty digit result. Division accepts a thirty digit dividend and a fifteen digit divisor and produces a fifteen digit result.

The operands for these functions may be signed or unsigned and leading zeros need not be supplied. Null operands are treated as zero. Results will be signed if negative and will have leading zeros suppressed. A zero result will consist of a single zero digit.

Add

AD 2,2,V

#<AD;num_1;num_2>

Value: the sum of integers num_1 and num_2

Notes: Overflow will produce a number modulo 10^{15}.

Subtract

SU 2,2,V

#<SU;num_1;num_2>

Value: the difference between integers num_1 and num_2

Notes: Overflow will produce a number modulo 10^{15}.

Multiply

MU 2,2,V

#<MU;num_1;num_2>

Value: The product of integers num_1 and num_2.

Divide and Give Quotient

DV 2,2,V

#<DV;num_1;num_2>

Value: the quotient resulting from dividing num_1 by num_2

Notes: A quotient of more than 15 digits will terminate processing and produce an error message.

Divide and Give Remainder

DVR 2,2,V

#<DVR;num_1;num_2>

Value: the remainder resulting from dividing num_1 by num_2

Notes: If the operands are such that the quotient would exceed 15 digits, processing is terminated and an error message is produced.

Obtain Absolute Value

ABS 1,1,V

#<ABS;num>

Parameter: A signed or unsigned decimal integer of up to 15 digits.

Value: the absolute value of num

NUMERIC COMPARISONS

The functions in this group provide for the comparison of two signed or unsigned decimal integers, each a maximum of 15 digits in length. The result of such a comparison is one of two strings also provided as parameters to the function. In practice, these strings will often contain function calls to be executed depending on the result of the comparison.

Compare Numeric "Equal"

EQ 4,4,V

#<EQ;num_1;num_2;$string_1$;$string_2$>

Value: If num_1 equals num_2, the result is $string_1$; otherwise, it is $string_2$.

Compare Numeric "Greater Than"

GT 4,4,V

#<GT;num_1;num_2;$string_1$;$string_2$>

Value: If num_1 is greater than num_2, the result is $string_1$; otherwise, it is $string_2$.

Compare Numeric "Less Than"

LT 4,4,V

#<LT;num_1;num_2;$string_1$;$string_2$>

Value: If num_1 is less than num_2, the result is $string_1$; otherwise, it is $string_2$.

LOGICAL COMPARISONS

The functions in this group provide for the comparison of two arbitrary strings. Comparison begins from the left and proceeds character by character. If the strings are of unequal length, the shorter string is treated as if it were filled-out with characters which collate low to all other characters. The EBCDIC collating sequence is used (special characters low to lower case alphabet which is low to upper case alphabet which is low to the decimal digits).

The result of a logical comparison is one of two additional strings which are also provided as parameters. In practice, these strings will often contain function calls which will be executed depending on the result of the comparison.

Compare Logical "Equal"

EQ? 4,4,V

#<EQ?;$string_1$;$string_2$;$string_3$;$string_4$>

Value: If $string_1$ is equal to $string_2$, the result is $string_3$; otherwise, it is $string_4$.

Compare Logical "Greater Than"

GT? 4,4,V

#<GT?;$string_1$;$string_2$;$string_3$;$string_4$>

Value: If $string_1$ is greater than $string_2$, the result is $string_3$; otherwise it is $string_4$.

Compare Logical "Less Than"

LT? 4,4,V

#<LT?;$string_1$;$string_2$;$string_3$;$string_4$>

Value: If $string_1$ is less than $string_2$ the result is $string_3$: otherwise it is $string_4$.

LIBRARY OPERATIONS

TTM supports a direct access library into which collections of named strings may be stored. Such collections are known as programs and each must be named. A complete program name consists of an ampersand, a qualifier, a period, and member name. Qualifiers consist of one through eight characters and member names consist of one through twenty characters. For example, &TEST.LOAD would be a complete name consisting of qualifier, TEST, and the member name, LOAD.

When a new program is to be added to the library (STORE function) or an old program is to be deleted (DELETE function), the complete program name must be given. Complete names may also be given when searching the library for an existing program (COPY, NAMES, PDF functions). However, a set of standard qualifiers may be declared (LIBS function) which will automatically be placed, one at a time, in front of the given unqualified member name whenever search is requested. The default qualifier, SYSLIB, will be used in the absense of any declared standard qualifiers. The SYSLIB qualifier will also be used in a search which exhausts the declared qualifiers without finding the desired member.

For example, assume that the following named programs have been stored: &SYSLIB.A, &XX.B, &YY.B, &YY.C, &YY.D. Assume also that YY and XX have been declared standard qualifiers, in that order. Then, a search for B would

retrieve YY.B, a search for &XX.B would retrieve XX.B and a search for A would retrieve SYSLIB.A.

In addition to explicit retrieval of programs, TTM supports implicit retrieval. If a function is called which is not defined, the name is assumed to be a library member name and a search is made. If the member is found, its strings are brought into core and defined. If one of these has the same name as the function, it is executed with the original parameters; otherwise, an error message is issued.

Store a Program

STORE 2,*,S

#<STORE;programname;$name_1$;$name_2$;...>

Parameters: the $name_i$ are the names of functions to be stored under the name program name.

Side Effects: The named functions are placed in the library under the specified name. The residual pointers of the named strings are not stored.

Delete a Program

DELETE 1,1,S

#<DELETE;programname>

Side Effects: The named program is deleted from the library.

Note: Library space is dynamically allocated and emptied space is automatically recovered.

Copy a Program

COPY 1,1,S

#<COPY;programname>

Side Effect: The strings stored under the given program name will be copied from the library and put into the TTM memory.

Obtain String Names

NAMES 0,1,V

#<NAMES;programname>

Value: If program name is absent the value will be the names of all strings in the TTM memory, arranged in sorted order and separated by commas. If program name is present, the value will be the names of all strings stored in the library under the given name, separated by commas, and in the order in which they appeared in the original store function.

Determine if a Program is in the Library

PDF 3,3,V

#<PDF;programname;$string_1$;$string_2$>

Value: If a program in the library has the given name, the value is $string_1$; otherwise, it is $string_2$.

Declare Standard Qualifiers

LIBS 0,10,S

#<LIBS;$string_1$;$string_2$;...;$string_{10}$>

Parameters: Each string should be one through eight characters in length.

Side Effects: The strings become the standard qualifiers which will be applied, in the order given, whenever an unqualified member name is being searched for. If no parameters are given, only the default standard qualifier will be used.

Notes: Any subsequent calls to LIBS will redefine the whole set of qualifiers.

UTILITY OPERATIONS

The operations in this group perform certain utility functions which are not easily classifiable into one of the other groups.

Determine if a Name is Defined

NDF 3,3,V

#<NDF;name;$string_1$;$string_2$>

Value: If a string in the dictionary has been given a name of name, the value is $string_1$; otherwise, it is $string_2$.

Obtain the Norm of a String

NORM 1,1,V

#<NORM;string>

Value: The number of characters in the string will be returned as an unsigned decimal integer with leading zeros suppressed.

Program Break

BREAK 0,1,S

#<BREAK;string>

Side Effect: Processing of the present program string is terminated. All information contained in the program string is deleted, but any value produced so far is retained. If BREAK has a parameter it becomes the new program string and processing resumes. Otherwise, TTM returns to its caller.

Turn Trace On

TN 0,0,S

#<TN>

Side Effect: The trace mode of operation will be activated.

Note: In the trace mode, TTM will print the name of each function, as well as its parameters, just before a function is executed.

Turn Trace Off

TF 0,0,S

#<TF>

Side Effects: The trace mode of operation will be deactivated.

Test for End of String

EOS 3,3,V

#<EOS;name;$string_1$;$string_2$>

Value: If string name, starting at the residual pointer, is null, the value is $string_1$; otherwise, it is $string_2$.

Function Definition

Since TTM is oriented toward systems programming it is natural to write some TTM functions which will make TTM programming easier. The most basic of these functions is DEF. It is written as:

```
#<DS;DEF;<##<DS;NAME;<TEXT>>
##<SS;NAME;SUBS>>>
```

and

```
#<SS;DEF;NAME;SUBS;TEXT>
```

To define the string XX as 12345 and then segment XX on 34, write:

```
#<DEF;XX;34;12345>
```

The call

```
#<XX;0000>
```

will get

```
1200005
```

This function is so convenient that it will be used in presenting the following examples.

In addition to defining and segmenting a string it is frequently desirable to mark it for name creation also. This can be done with DEFCR which is defined using DEF as:

```
#<DEF;DEFCR;<NAME;SUBS;CRS;TEXT>;
<##<DS;NAME;<TEXT>>##<SS;NAME;SUBS>
##<CR;NAME;CRS>>>
```

This description in the "def" format is equivalent to:

```
#<DS;DEFCR;<##<DS;NAME;<TEXT>>
##<SS;NAME;SUBS>##<CR;NAME;CRS>>>
```

followed by

```
#<SS;DEFCR;NAME;SUBS;CRS;TEXT>
```

Basic Examples

1. A function which, given the name of a string, will return that string with a plus sign following each character but the last can be defined as:

```
#<DEF;PLUS;SN;<#<CC;SN>
#<PLUSX;SN;#<CC;SN>>>>
```

and

```
#<DEF;PLUSX;<SN;CH>;<#<EQ?;
CH;;;<+CH#<PLUSX;SN;#<CC;SN>>>>>>
```

Writing functions in pairs with one function to start the process and another to extend it for a number of iterations is quite useful.

```
Given X ≡ ABCD
then #<PLUS;X>
will yield A+B+C+D
```

2. A function which given a number N will return the power of 2 which is equal to or next greater than N can be defined as:

```
#<DEF;POWER;N;<#<PW;1;#<SU;N;1>>>>
```

and

```
#<DEF;PW;<PP;XX>;<#<EQ;XX;0;PP;
<#<PW;#<MU;pp;2>;#<DV;XX;2>>>>>>
```

when called as

```
#<POWER;12>
```

it will return 16

3. A function to convert a decimal number into a binary number is defined as:

```
#<DEF;BINARY;<NN;BNUM>;
<#<EQ;NN;0;BNUM;<#<BINARY;
##<DV;NN;2>;##<DVR;NN;2>BNUM>>>>>
```

The call

```
#<BINARY;37>
```

will yield 100101

4. A function to construct a table of length 2^n which contains in each position the count of the number of one-bits in the binary number representing that position can be defined as:

```
#<DEF;BITCT;<LL;NN>;<#<eq;LL;1;
  (NN);<#<BITCT;##<DV;LL;2>;NN>
#<BITCT;##<DV;LL;2>;##<AD;NN;1>>>>>>
```

To see how this function works, note that a table of length 2^n can be constructed by appending to a table of length 2^{n-1} the same table with all entries increased by one.

If this function is called as:

```
#<BITCT;8;0>
```

it will yield

```
(0)(1)(1)(2)(1)(2)(2)(3)
```

5. Ackerman's function is a recursive function which grows more rapidly than any primitive recursive function.

The function is

```
A(m,n) = A(m-1,A(m,n-1))

A(m,0) = A(m-1,1)

A(0,n) = n+1
```

The TTM function to evaluate Ackerman's function for some M and N is:

```
#<DEF;A;<M;N>;<#<EQ;M;0;
<##<AD;N;1>>;<#<EQ;N;0;
<#<A;##<SU;M;1>;1>>;<#<A;
##<SU;M;1>;#<A;M;##<SU;N;1>>>>>>>>>
```

When called as

```
#<A;3;1>
```

it will have the value

```
13
```

6. The following is an example of a function which creates another function and then executes the created function. The purpose of this function is to duplicate a section of text and increment numbers within the text.

The first function is defined as:

```
#<DEFCR;DO;<FROM;BY;TO;SUBS;TEXT>;
NN;<#<DEF;DO-NN;SUBS;#@<#<GT;
BY;0;GT;LT><;SUBS;TO;;<TEXT#<DO-NN
##<AD;SUBS;BY>>>>@>>#<DO-NN;FROM>
##<ES;DO-NN>>>
```

The call:

```
#<DO;1;2;7;K;(K)>
```

will first define a function with a created name. This function would be

```
#<DEF;DO-0005;K;<#<GT;K;7;;
```

```
<(K)#<DO-0005;##<AD;K;2>>>>>>
```

The function DO uses the escape character instead of brackets around all of the text of the DEF because it needs to evaluate the GT to see if the increment is going up or down.

Now the newly created function will be called as:

```
#<DO-0005;1>
```

and it will produce

```
(1)(3)(5)(7)
```

If DO is called as

```
#<DO;-2;3;8;AA;XXAA >
```

it will produce

```
XX-2 XX1 XX4 XX7
```

If DO is called as

```
#<DO;1;1;3;I;#<DO;2;-1;1;J;(I,J)>>
```

it will produce

```
(1,2)(1,1)(2,2)(2,1)(3,2)(3,1)
```

If DO is called as

```
#<DO;0;1;4;AA;<(#<MU;AA;AA>)>>
```

it will produce

```
(0)(1)(4)(9)(16)
```

Since DO always erases the functions it creates it does not fill memory with useless strings.

Keyword Parameters

The following example demonstrates the writing of a TTM function which

can be called with keyword parameters.

Construct a function, FF, which will produce:

(AA)(BB)(CC)(DD)(EE)

when it is called with AA, BB and CC specified by position; and with DD and EE specified by the keywords X= and Y= respectively. In addition, DD is to have the default value 17 if it is not specified.

For example if FF is called as

#<FF;1;2;3;Y=5;X=4>

it should produce

(1)(2)(3)(4)(5)

or if it is called as

#<FF;10;Y=25>

it should produce

(10)()()(17)(25)

Such a function is constructed as two functions. The first, FF, is the one which is called. The second, FF.K, is the one which actually produces the result.

The function FF will be defined as:

```
#<DEF;FF;<P1;P2;P3;P4;P5>;
<##<DS;KW;<;P1;P2;P3;P4;P5>>
#<FF.K;#<KW1;X;17>;#<KW1;Y;>
#<KW>>>>
```

This function expects from 0 to 5 parameters with the keyword parameters following the positional parameters. It will rearrange these into positional parameters using the keywords and default text it contains and will call FF.K with these parameters.

The function FF.K will be defined as:

```
#<DEF;FF.K;<DD;EE;AA;BB;CC>;
  <(AA)(BB)(CC)(DD)(EE)>>
```

It is important to note that in the call to FF.K the two parameters which were specified by keywords will come before the three parameters defined by position.

In addition to FF and FF.K functions KW1 and KW2 will be needed. These functions can be shared by all functions having keyword parameters. These will be described later.

The operation of the keyword functions is as follows. First the parameters of FF are defined into a string, KW, with semicolons before each parameter including the first. Next, KW1 is called with a keyword and default text. KW1 will return the keyword parameter. This is done for each keyword. When all of the keywords have been searched for, the string KW will contain only the positional parameters separated by semicolons. An active call on KW is used to finish the parameter list for FF.K and a call on FF.K is made. An active call on KW is used to break it into parameters. A semicolon is not put between the last call on KW1 and the call on KW because the first character of KW will be a semicolon. Since the keyword parameters are now before the positional parameters, FF.K should be defined accordingly.

The function KW1 is defined as

```
#<DEF;KW1;<KEY;DF&T>;
<#<EQ;##<SC;KW;<;KEY=>>;0;<DF&T>;
<#<KW2;##<CS;KW>;##<CP;KW>;##<KW>>>>>
```

This function first segments the string KW on ;KEY= . If the key is not

in the string, that is if SC returns 0, then KW1 returns the default text. If the key was in the string, it calls KW2 with that part of the string KW up to the key, the keyword parameter, and the remaining part of the string KW. It is very important to use passive calls on ##<CS;KW> and ##<KW> because the strings that these functions return may have semicolons in them. However, they are to be passed to KW2 without being broken apart.

The function KW2 is defined as:

```
#<DEF;KW2;<FIRST;PARM;LAST>;
  <PARM##<DS;KW;<FIRST;LAST>>>>
```

it returns the keyword parameter and redefines KW to be what remains of the parameter string after ;KEY=PARM has been removed. The primitive SC removed ;KEY= and CP removed PARM; so a semicolon must be put between FIRST and LAST.

A series of functions can be defined which will aid in producing pairs of functions like FF and FF.K. These functions are given below. They are called by:

```
#<DEFKW;NAME;SUBS;KEYS;TEXT>
```

where NAME, SUBS, and TEXT are the same as they are in DEF and KEYS has the form

```
<KEYWORD=DEFAULT;----;---->
```

So to produce FF and FF.K we would call DEFKW as:

```
#<DEFKW;FF;<AA;BB;CC>;
  <X=17;Y=>;<(AA)(BB)(CC)(X)(Y)>>
```

It is important to note that there are two independent sets of functions here. Those beginning with DEFKW are used when we are defining a function which can be called with keywords. The two functions KW1 and KW2 are needed when we actually call such a function.

```
#<DEF;DEFKW;<NAME;SUBS;KEYS;TEXT>;<##<DS;KEY;<KEYS>>
#<DKW1;#<DKW2>>#<RRP;KEY1>#<RRP;KEY2>#<RRP;KEY3>
#<DKW3;<NAME>;<SUBS>;##<KEY1>;##<KEY2>>
#<DKW4;<NAME>;<SUBS>;##<KEY3>;<TEXT>>
#<ES;KEY;KEY1;KEY2;KEY3;KEY4>>>

#<DEF;DKW1;<KEYW;DFLT>;<#<EQ?;KEYW;;;<##<AP;KEY1;<KEYW*;>>
##<AP;KEY2;<;#<KW1;KEYW;DFLT>>>##<AP;KEY3;<KEYW;>>
#<DKW1;#<DKW2>>>>>>

#<DEF;DKW2;<>;<##<DS;KEY4;##<CP;KEY>>##<SS;KEY4;=>
##<CS;KEY4>;##<KEY4;=>>>

#<DEF;DKW3;<NAME;SUBS;KEYS;CALLS>;<#<DEF;<NAME>;<KEYSSUBS>;
<##<DS;KW;<;KEYSSUBS>>#<NAME.KCALLS#<KW>>>>>>

#<DEF;DKW4;<NAME;SUBS;KEYS;TEXT>;<#<DEF;<NAME.K>;
<KEYSSUBS>;<TEXT>>>>
```

Expression Evaluation

The following TTM functions form a recursive expression evaluator. This expression evaluation will handle expressions containing unary '+' and '-' and any level of parentheses.

It is entered by calling Eval with an expression as the parameter. The Eval function is defined by:

```
#<DEF;EVAL;EXP;<##<DS;EXP;EXP>

#<SUM>>>
```

The sum function will return the value of the expression. The other functions are:

```
#<DS;SUM;<#<SUM1;#<TERM>;#<OP>>>>

#<DEF;SUM1;<NUM;PR;OPER>;
<#<EQ?;PR;SS;<#<SUM1;##<OPER;NUM;
#<TERM>>;#<OP>>>;<NUM>>>>

#<DS;TERM;<#<TERM1;#<PRIME>;#<OP>>>>

#<DEF;TERM1;<NUM;PR;OPER>;
<#<EQ?;PR;TT;<#<TERM1;##<OPER;NUM;
#<PRIME>>;#<OP>>>;<NUM>>>>
```

```
#<DS;PRIME;<#<TCL;DIGIT;EXP;
<##<CCL;DIGIT;EXP>
#<DS;OP;##<PR|##<CC;EXP>>>>;
<#<DO##<CC;EXP>>>>>>

#<DCL;DIGIT;0123456789>

#<DS;DO;0>

#<DS;DO+;<#<PRIME>>>

#<DS;DO-;<##<SU;0;#<PRIME>>>>

#<DS;DO(;<#<SUM>##<DS;OP;
##<PR|##<CC;EXP>>>>>
```

In addition to these functions there is also what might be thought of as a translate table. That is, the string PR|+ is defined to be SS;AD. The complete table is

```
PR|+      SS;AD

PR|-      SS;SU

PR|*      TT;MU

PR|/      TT;DV

PR|)

PR|
```

the last two entries having null values.

It can be seen that this expression evaluator follows closely the BNF description of an expression:

```
<SUM>::=<TERM>|<SUM>+<TERM>|<SUM>-<TERM>

<TERM>::=<PRIME>|<TERM>*<PRIME>|<TERM>/<PRIME>

<PRIME>::=<INTEGER>|+<PRIME>|-<PRIME>|(<SUM>)

<INTEGER>::=<DIGIT>|<INTEGER><DIGIT>

<DIGIT>::=-|1|....|9
```

In SUM1 and TERM1 the class of operator is tested instead of testing for both operators separately. The class of the operator and its TTM name are in the string OP so an active call on OP is needed to separate them. However when translating the operator into its class and name and putting it into the string OP, it is very important to do a passive call on the string PR|OPERATOR jhich does this translation. Otherwise, only the class will be in the string OP.

Also note that in PRIME when it is determined that a a character is not a digit, a four way branch is made to the strings which process the unary operators, left parentheses, and null expressions.

FUNCTIONS FOR THE BATCH VERSION

The functions described in the previous sections are available in the base version of TTM. The batch version augments these by a number of additional functions which primarily control input and output operations of the system.

The batch version consists of two logical parts--an input monitor and TTM. The input monitor scans the input stream and passes control to TTM whenever a TTM program is detected. This is detected by recognizing #< or ##< starting in column one of an input card. While TTM is in control it may call upon the input monitor for more input. If there is no more input, processing is terminated. When a TTM program ends, control is returned to the input monitor. Output may be done by both the input monitor and TTM.

MONITOR CONTROL FUNCTIONS

Control Output of Input Monitor

INSW 2,2,S

#<INSW;PR;PU>

Parameters: PR and PU are integers.

Side Effects: If PR = 0 no input lines are printed; if PR =1 only lines not passed to TTM are printed; if PR = 2 all lines are printed. PU takes the same values as PR but controls punching of input lines.

Note: The default values are PR = 2 and PU = 0.

Control Handling of TTM Programs

TTMSW 2,2,S

#<TTMSW;ACT;EOL>

Parameters: ACT and EOL are integers.

Side Effects: If ACT = 0 TTM complete programs are returned by the functions CD, PK, and FOR. If ACT = 1 the current TTM program is terminated and control returns to the input monitor which then calls TTM with the new program. If ACT = 2 TTM programs are given no special significance. If EOL = 0 end of line characters will not be inserted between lines of a TTM program. If EOL = 1 end of line characters will be inserted.

Notes: The default values are ACT = 1 and EOL = 0. A TTM program is continued by placing a non-blank character in column 72 of the card to be continued. All lending and trailing blanks will be stripped. Columns 73 through 80 are not examined and may be used for any purpose.

INPUT FUNCTIONS

Input One Card

```
CD  0,0,V
#<CD>
```

Value: The next card image in the input stream as controlled by the CDSW function.

Control CD Input

```
CDSW  2,2,S
#<CDSW;SUP,NN>
```

Parameters: SUP and NN are integers.

If SUP = 0, trailing blanks will not be suppressed.

If SUP = 1, trailing blanks will be suppressed.

NN ($1 \leq NN \leq 80$) specifies the number of columns of a card to be returned by DC.

Notes: The default values are SUP = 1 and NN = 80. Suppression of trailing blanks always begins in column NN

Input Next Complete Fortran Statement

FOR 0,0,V

#<FOR>

Value: The next card in the input stream is considered to have the form of a Fortran statement and that statement, including any continuation, is returned.

Note: Columns 73 through 80 of all cards and columns 1 through 6 of continuation cards are deleted.

Control FOR input

FORSW 2,2,S

#<FORSW;SUP;EOL>

Parameters: If SUP = 0 trailing blanks are not suppressed. If SUP = 1 trailing blanks are suppressed. If EOL = 0 end of line characters are not inserted between continuation lines. If EOL = 1 end of line characters are inserted.

Note: The default values are SUP = 1 and EOL = 0

Look Ahead One Card

PK 0,0,V

#<PK>

Value: The next input card is returned as the value of PK. However, the card is not removed from the input stream and may be read again with the CD or FOR functions.

Notes: If the card is a TTM program, the action specified by TTMSW over-rides. Successive calls on PK without intervening calls on CD or FOR cause the previously seen card to be processed by the input monitor as if the card had not been seen by TTM.

Control PK Input

PKSW 2,2,S

#<PKSW;SUP;NN>

Note: Has same effects and defaults for PK as CDSW has for CD.

Output Functions

Print a String

PS 1,1,S

#<PS;string>

Side Effects: The string, of any length, will be printed. If the string is longer than 132 characters, multiple lines will be printed.

Note: End of line characters can be used to force multiple lines.

Specify Length of Page

PAGE 1,1,S

<PAGE;NN>

NN is an integer specifying the number of lines to be printed per page.

Note: The default value of NN is 60.

Space Before Printing

SP 1,1,S

#<SP;NN>

Parameter: NN is an integer specifying the number of lines to space before printing the next line.

Notes: NN = 1 is normal single spacing and is the default for each line. A space past the end of the logical page causes an eject as does a value of NN of 100 or more. If NN = 0, no spacing will occur before printing and over-printing will result. The effect of a call on SP holds only for the next line printed.

FORMATED OUTPUT OPERATIONS

The functions in this group are used to format cards to be punched or lines to be printed. Since these lines or cards may be used as input to other language processors (e.g., assemblers, compilers) provision is made for automatic tabulation and generation of continuation cards.

Format a Line or Card

FM 0,*,S

#<FM;$string_1$;$string_2$;...>

Side Effects: One line is formated and printed or punched. The first ten strings will be placed starting at the ten tab positions. If this would place a string such that no blanks separated it from the previous string, it will be placed one blank after the end of the previous string. If there are more strings than declared tabs, the additional strings will be separated by one blank. Tabs are effective only on the first line of a line group. If continuation is required, all strings in the continuation lines will be separated by one blank.

Note: If an end of line character is detected in one of the parameters, continuation will be forced at that point.

Declare Tab Positions

TABS 1,10,S

#<TABS;tab_1;tab_2;...;tab_{10}>

Parameters: positive decimal integers specifying tab positions and lying in the range

$$1 \leq tab_i \leq tab_{i+1}$$

Side Effects: The tab positions are saved for use by the FM function.

Note: The default action of FM is as if #<TAB;1> had been issued. Tabs past the end of the line will cause continuation.

Set Continuation Convention

SCC 3,3,S

#<SCC;LCH;END;ST>

Parameters: LCH ($1 \leq LCH \leq 132$) is an integer specifying the position on a line after which a continuation line is produced. END is a string which will be placed at the end of a line to be continued. ST is a string which will be placed at the beginning of a continuation line.

Notes: The default values are LCH = 132, END = null, and ST = null. As an example, the FORTRAN continuation conventions could be established by coding

#<SCC;72;;bbbbbC>

where "b" means "blank".

Control FM Output

FMSW 2,2,S

#<FMSW;PR;PU>

Parameters: If PR = 0 output from FM is not printed; if PR = 1 output from FM is printed. If PU = 0 output from FM is not punched; if PU = 1 output from FM is punched.

Notes: The default values are PR = 1 and PU = 0. Output lines longer than 80 characters will be truncated to 80 characters before they are punched.

UTILITY FUNCTIONS

Obtain Time of Day

Time 0,0,V

#<TIME>

Value: The time of day, in hundredths of seconds since midnight, will be returned as an unsigned decimal integer.

Obtain Execution Time

XTIME 0,0,V

#<XTIME>

Value: The current execution (task) time for the job, in hundredths of seconds, will be returned as an unsigned decimal integer

Define Error String

DES 1,1,S

#<DES;string>

Side Effect: The string is considered to be a TTM program which will be called whenever a processing error occurs.

USING THE BATCH VERSION

The batch version of TTM consists of one load module in standard OS/360 format. It is designed to execute under any version of OS/360 which can provide a region of at least 56K for execution. With this size region, a work area of approximately 10K bytes is available. This can be expanded by increasing the region size as TTM performs a variable GETMAIN to obtain all available care for its own use. Sufficient care is returned to the system so that abnormal terminations will produce a core dump if a data set with a DDNAME of SYSUDUMP is provided.

TTM uses four data sets--SYSIN, SYSLIB, SYSPRINT, and SYSPUNCH. All, except SYSLIB are standard sequential sets and QSAM is used to process them. The library, SYSLIB, must reside on a direct access device and is processed at the EXCP level. However, physical unit independence is maintained by obtaining device characteristics of the library through use of the DEVTYP system service. Before first use, the library must be initialized by a special utility program.

TTM can process blocked SYSIN, SYSPRINT, and SYSPUNCH. DCB exit routines are utilized for SYSPRINT and SYSPUNCH to provide automatic default blocking of these data sets. In the absence of any over-riding DCB parameters, SYSPRINT is assigned BLKSIZE=1596 and RECFM=FBSM. The defaults for SYSPUNCH are BLKSIZE=1680 and RECFM=FBS. Either BLKSIZE or RECFM or both can be over-ridden for each data set. However, RECFM for SYSPRINT will always be forced to contain an M to indicate machine control codes . Block sizes need not be specified exactly as they are always forced down to an exact multiple of the appropriate LRECL. RECFM is automatically adjusted to indicate whether the records are blocked or not.

It is quite acceptable to specify DISP=SHR in the DD card for SYSLIB, even in an MFT or MVT environment. TTM uses the system ENQ and DEQ facilities to insure library integrity during all storing and deleting operations.

The following is sample JCL for an execution of TTM. The processor is assumed to reside on a data set named EKG.TTMLOAD and the library name is SHC.TTMLIB. The JCL may need to be modified to reflect varying installation requirements:

```
//NAME       JOB    ACCOUNTING PARAMETERS
//JOBLIB     DD     DSN=EKG.TTMLOAD,UNIT=SYSDA,DISP=SHR,
//                  VOL=SER=CITSC5
//STEP       EXEC   PGM=TTMSA
//SYSLIB     DD     DSN=SHC.TTMLIB,DISP=SHR
//SYSPRINT   DD     SYSOUT=A,UNIT=(,SEP=SYSLIB)
//SYSPUNCH   DD     SYSOUT=B,UNIT=(,SEP=(SYSLIB,SYSPRNT))
//SYSIN      DD     *
```

APPENDIX A

ERROR MESSAGES

Whenever an error occurs, TTM prints an error message and the name and parameters of the function being processed. It then returns to its caller, which in the batch version is the input monitor. The contents of the program string are lost but any side effects which occurred before the error will remain.

The following list gives all of the error messages and a short description of those which are not completely self-explanatory. In this discussion, the terms "primitive" and "built-in function" are synonomous.

Function Not Defined

The function being called is not defined.

Name Not Defined

A primitive which uses string names has been called with an undefined name.

Class is Undefined

An undefined class name was used.

Only Strings Allowed

Some primitives which take function names as parameters will not operate on other primitives.

Too Few Parameters

Many primitives have a minimum number of parameters which they must have to operate.

Quotient is Too Large

The quotient in a division operation is too large to be represented.

Decimal Integer Required

Some primitives require certain parameters to be decimal integers.

Too Many Digits

An input parameter contained more than 15 decimal digits.

(9 for functions like TABS, CDSW, etc.)

Too Many Segment Marks

The number of segment marks has exceeded 62. The function will contain the first 62 such marks.

Dynamic Storage Overflow

No space left in core. There may be room enough to do an Erase String.

Parm Roll Overflow

Discussed in Appendix C.

Name Already on Library

An attempt has been made to store a program on top of an existing one. The existing program can be deleted.

Name Not on Library

A call to COPY or NAMES requested a non-existent program.

No Space on Library

A delete will create space.

No Library

The library data set was not provided.

Incorrect Library

The library data set does not have the format of a TTM library.

Wrong Name Format

Name is not &QUAL.NAME or NAME.

Too Many Lib Names

More qualifiers provided than can be used.

Lib Name Too Long

Qualifier is too long.

Error on Library

An error was detected in format of library.

Too Many Nested Copies

The library routine supports nested copies, this facility is not used

by TTM but this message could appear if TTM was called during a copy by an outside routine sharing the library.

Zero Level Only

Deletes cannot be nested inside copies.

Only Unsigned Decimal Integers

The numbers used in functions like TABS, CDSW, etc. must be unsigned decimal integers.

APPENDIX B

FUNCTION ATTRIBUTES

ABS	1	1	V	EQ	4	4	V	PKSW	2	2	S
AD	2	2	V	EQ?	4	4	V	PS	1	1	S
AP	2	2	S	ES	1	*	S	RRP	1	1	S
BREAK	0	1	S	FLIP	1	1	V	SC	2	63	VS
CC	1	1	VS	FM	0	*	S	SCC	3	3	S
CCL	2	2	VS	FMSW	2	2	S	SCL	2	2	S
CD	0	0	V	FOR	0	0	V	SCN	3	3	VS
CDSW	2	2	S	FORSW	2	2	S	SN	2	2	S
CF	2	2	S	GN	2	2	V	SP	1	1	S
CM	1	1	S	GT	4	4	V	SS	2	63	S
CN	2	2	VS	GT?	4	4	V	STORE	2	*	S
COPY	1	1	S	INSW	2	2	S	SU	2	2	V
CP	1	1	VS	ISC	4	4	VS	TABS	1	10	S
CR	2	2	S	LIBS	0	10	S	TCL	4	4	V
CS	1	1	VS	LT	4	4	V	TF	0	0	S
DCL	2	2	S	LT?	4	4	V	TIME	0	0	V
DELETE	1	1	S	MU	2	2	V	TN	0	0	S
DES	1	1	S	NAMES	0	1	V	THD	1	1	V
DNCL	2	2	S	NDF	3	3	V	TRL	1	1	V
DS	2	2	S	NORM	1	1	V	TTMSW	2	2	S
DV	2	2	V	PAGE	1	1	S	XTIME	0	0	V
DVR	2	2	V	PDF	3	3	V	ZLC	1	1	V
ECL	1	*	S	PK	0	0	V	ZLCP	1	1	V
EOS	3	3	V								

APPENDIX C

STORAGE OVERFLOW

The following simple programs show the kinds of overflow which are possible with the present implementation of TTM.

The first example is

```
#<DS;X;<#<X>>>#<X>
```

This program defines the string X to be an active call on the string X and then does an active call on X. This program is interesting because it does not cause an overflow. It is an infinite loop. Any program which ends in a call on itself is such a loop unless it creates strings which it does not erase. Such programs are of particular importance as idle programs for conversational programs written in TTM.

The program:

```
#<DS;X;<#<Z#<X>>>>#<X>
```

fills the program string to the left of the scan position with {Z{Z{Z...{Z, and fills the program string to the right of the scan position with >>>...>. It will eventually cause a parameter roll overflow. In this function, X is called before the right bracket of the call on Z is reached. The function X will be called about 300 times before the overflow occurs.

The program:

```
#<DS;X;<Z#<X>>>#<X>
```

will fill the program string to the left of the scan position with ZZZ...Z and will eventually cause a dynamic storage overflow. This overflow will not occur until the entire core is filled with ZZ...Z. It does not cause a

parameter roll overflow because it is not producing uncompleted function calls, but only a string of characters.

The program:

```
#<DEF;X;Z;<#<X;ZZ>>>#<X;Z>
```

will also cause a dynamic storage overflow by filling the program string but does it in a quite different way. The single parameter of X doubles in length each time X is called and eventually becomes too large for core.

It is also possible to get a dynamic storage overflow by defining too many strings.

Bibliography

1. Greenwald, I.D. and Kane, M., The Share 709 System: Programming and Modification. JACM 6 No. 2 (1959). pp. 128-133.

2. Greenwald, I.D., Handling Macro Instructions. CACM 2, No. 11 (1959), 21-22.

3. Remington Rand UNIVAC Division, UNIVAC Generalized Programming. Philadelphia, 1957.

4. Eastwood, D.E. and McIlroy, M.D., Macro Compiler Modification of SAP. Bell Telephone Laboratories Computation Center, 1959.

5. McClure, R.M., Description of CODAPT Assembler, 1960.

6. Caine, S.H., Reference Manual for CIT 7090/7040 Experimental Macro Assembly Program (XMAP). California Institute of Technology, Willis H. Booth Computing Center (1964).

7. McIlroy, M.D., Macro Instruction Extensions of Compiler Languages. CACM 3, No. 4 (1960), 214-220.

8. McIlroy, M.D., Using SAP Macro Instructions to Manipulate Symbolic Expressions. Bell Telephone Laboratories Computation Center (1960).

9. IBM, System/360 Assembler Language, C28-6514-4, (1967).

10. Caine, S.H. et al., Report of the Systems Objectives and Requirements Committee, SHARE, 1965, pp. 29-40.

11. Struachey, C., A General Purpose Macro Generator. Comput J 8, 3(1965), pp. 225-241.

12. Farber, D. J., 635 Assembly System - GAP. Bell Telephone Laboratories Computation Center (1964).

13. Caine, S.H., et al., An Operating Environment for Programming Research. California Institute of Technology, Willis H. Booth Computing Center Programming Report No. 1, 1967.

14. Caine, S.H. and Gordon, E.K., TTM:An Experimental Interpretive Language. California Institute of Technology, Willis H. Booth Computing Center, Programming Report No. 7, 1968.