Ten Dollars 84

Cromemco

68000 Symbolic Debugger

Reference Manual

[illegible]

Cromemco

68000 Symbolic Debugger

Reference Manual

June 1984

CROMEMCO, Inc.
280 Bernardo Avenue
Mountain View, Ca. 94043

023-4080
Rev. A

This manual was produced using a Cromemco System Three computer running under the Cromemco Cromix Operating System. The text was edited with the Cromemco Cromix Screen Editor. The edited text was proofread by the Cromemco SpellMaster Program and formatted by the Cromemco Word Processing System Formatter II. Camera-ready copy was printed on a Cromemco 3355B printer.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

OVERVIEW

The Cromemco symbolic debugger allows the interactive execution and debugging of programs written in Cromemco **FORTRAN**, Cromemco **Pascal**, and Cromemco **C**.

Execution of a target program can be "break pointed" or "traced" at the entry points or exits of subroutines, or at any statement boundary within a subroutine. After execution is suspended at a break point, the values of variables and data structures can be examined and altered using their symbolic names in the environment in which the break point occurred. When debugging **Pascal** and C programs, the active subroutines can be displayed as a calling-sequence back trace, and the debugger can be directed to change its current symbol-naming environment to any active subroutine. The target program can be continued or terminated from a break point, possibly after break and/or trace points are set and/or cleared.

Low-level operations for displaying and setting memory locations by address are also available. Break and trace points may also be set at arbitrary addresses, although the debugger offers a more limited set of functions at break points outside the normal break-pointing areas.

Tables of information describing the program being debugged drive the debugger. Thus, the executable image of a target program is the same, whether or not the program is run under the symbolic debugger. This allows debugging sessions after programs are placed into production, eliminates problems related to finding program behavior that comes and goes with minor changes in the program or in the code generated for it, and allows full-speed execution under the debugger.

Debugger tables are created under control of compiler option flags, which can be turned on or off as desired. The size of unlinked object code will be larger, due to the amount of symbolic information included. The linker can combine the debugging information from the object files being linked, and combine that information in a single file, which can then be used by the debugger. It is possible to link object files created from more than one compilation and from more than one source language, some with, and some without symbolic debugging information included. The symbolic debugger operates with partial information and with routines originating from multiple source languages.

Chapter 2

RUNNING THE DEBUGGER

RUNNING THE DEBUGGER

To use the symbolic debugger, a source program should be compiled with the debugging flag set, the linker should create a file of information for the debugger to use, and the target program should be run under the debugger.

SETTING THE COMPILER DEBUG FLAG

By default, the compilers do not place symbolic information into the generated unlinked object code. The Cromemco **Pascal**, Cromemco **FORTRAN**, and Cromemco C compilers insert this information when invoked with the +d command line option. This option may appear anywhere on the command line. Alternatively, symbolic information can be turned ON and OFF for each procedure as follows:

Pascal	Comment toggle **$D+** for debugger information ON, **$D-** for debugger information OFF.
FORTRAN	Compiler control line beginning in column 1: **$DEBUG** for debugger information ON, **$NODEBUG** for debugger information OFF.
C	Compiler control line beginning in column 1: **#option debug** for debugger information ON, **#option nodebug** for debugger information OFF.

Setting the debugger flag has no effect on the generated object code except that certain additional tables of information are placed into the unlinked object code. This information is either consolidated or ignored by the linker. The executable image produced by a program is the same regardless of whether or not the debug flag is set when the program is compiled.

OBTAINING A DEBUG INFORMATION FILE FROM THE LINKER

The linker creates an executable program by linking object-code files (**.obj** files) with each other and with appropriate run-time libraries. The linker can consolidate all the debugging information in the object files being linked, creating (in addition to its object code output file) an output file containing this debugging information for later use by the debugger. This file has the filename extension **.dbg**. Alternatively, the linker may ignore the debugging information in its input files.

The linker accepts an optional command line argument:

```
+sxxx.dbg
```

in any position on the command line. This creates the information file and names it **xxx.dbg**. This is the preferred name for an information file associated with an executable program named **xxx.bin**. If the linker is operated in prompting mode, it will prompt:

```
Symbol file -
```

Pressing the RETURN key in response to this prompt indicates that a debugger information file should not be created. Entering a filename directs the linker to create a debugger information file with the specified filename. If a filename extension is not specified, the linker will supply the filename extension **.dbg** to the specified filename.

After creating the debugger information file, the remaining steps of the compilation should be carried out, producing the executable image of this symbolic information. When debugging a given program, make sure the **.dbg** file provided to the debugger matches the program being debugged.

DEBUGGING A TARGET PROGRAM

The debugger is a program named **dbg.bin.** To debug a program named **x**, supplying command line arguments (p1, p2, ... pn) to the program being debugged, use the command:

```
dbg [-sxxx[.dbg]] x p1 p2 ...  pn
```

The optional **-sxxx** (or **-sxxx.dbg**) instructs the debugger to use the file **xxx.dbg** as the debugger information file. If this argument is omitted, the debugger will look for a file name **x.dbg** as the debugger information file.

When the debugger has read the **.dbg** file and prepared the target program for execution, the debugger prompt (a minus sign) appears and debugger commands may be entered. At this point, the target program is not broken inside any environment, so variables cannot be examined or set (refer to the discussion of environments in the next chapter). The program can be run with break and/or trace points set, and the debugger tables can be examined. Running the target program after setting a break point at the entry of the main program (named in the program statement in **Pascal** or **FORTRAN**, and named **main** or **_main** in **C**) allows the debugger to access static (global) data areas.

Chapter 3

DEBUGGER CONCEPTS AND COMMANDS

CONCEPTS AND DEFINITIONS

A knowledge of the following concepts is useful in understanding the debugger commands.

Environments

When interacting with the debugger, the program is, in general, suspended at a break point. If this break point is within the bounds of a subroutine (used throughout to mean any procedure, function, or main program body), the symbolic names available to the target program within that subroutine are those in the current naming environment. Data is interpreted based on the name's attributes in that environment. This concept also applies to static (global) data areas. The interpretation of static data areas may depend on the common and/or equivalence statements in the broken subroutine, whether or not the subroutine is within a **Pascal** unit, and whether or not the current environment belongs to a subroutine written in the **C** language.

The debugger can be suspended outside of an environment. This is the case immediately after a debugging session is started. This also occurs: 1) when the debugger encounters a break point at an address that precedes the completion of the entry code or follows the start of the exit code of a subroutine, or: 2) at a break point in a routine which was not compiled with the debugging option enabled. When the debugger is outside an environment, variables cannot be accessed symbolically.

The debugger allows display and manipulation of the current environment from certain break points. Where possible, the debugger allows interactive changing of the current environment to the environment from which the current subroutine was called (potentially, all the way back to the main program). As the current environment changes, the debugger reflects the local declarations and source language of the new environment.

Naming conventions altered by the scope of **Pascal WITH** statements do not affect the naming conventions used by the debugger.

Break Points

Under interactive control, certain addresses in the target program's executable object code can be designated break points. If such an address is executed, except as noted below, control is returned to the debugger command level for further interactive debugging. The address must correspond to the first word of a 68000 instruction. The debugger automatically ensures this for entry, exit, and statement breaks (i.e., all break points except those set using the arbitrary address break point specification).

The debugger allows only one break point per address, even if that address can be described in several different ways. For example, an address that is the entry address of a subroutine is also usually the address of the first statement of that subroutine. Several "null" statements may begin at the same address. The debugger designates a break point relative to entries, exits, line numbers, and actual addresses, regardless of which method was used to set the break point. If the user attempts to set more than one break point at the same address, the debugger produces an error message describing the situation and showing the result of the conflicting commands.

An address on which a break point is set can have "skip counts" associated with it, specifying the number of executions of the break point necessary to return control to the debugger. Refer to the explanation of the break point command.

Trace Points

Setting a trace point is an alternative to setting a break point on a given address. When executed, a trace point causes an informative message to be displayed, while the program continues its execution. Trace points may also use "skip counts" to control the frequency with which the trace message is displayed.

If a trace and break are set at the same address, the break point takes precedence. The trace point becomes active if the break point is cleared. As is true for break points, the address has the trace point properties, regardless of which method was used to create the trace point. The address also has the skip count properties, making it impossible to set the counts independently for a trace point and a break point with the same address.

Pname

A pname refers to any object code entry point that is resolved by the Cromemco linker. In general pnames are procedure, function or subroutine names. Pnames can also refer to external or global entry points in assembly-language modules.

A pname can be either a user name or a link name. A user name is the name of a procedure in the user's source program. The debugger treats user names as does each programming language. Case is ignored when compared with **Pascal** or **FORTRAN** entry points; case is preserved for **C** entry points. As a result of nested scopes, static functions, etc., the same user name may appear several times in one executable program.

A pname can also be a link name, the name used by the linker to resolve code entry points. Link names are local or global. A local link name is a dollar sign ($), followed by digits, such as $12000. A global link name is a string in double quotation marks, such as "GLOBAL". Trailing blanks may be omitted. Local link names are also enclosed in double quotes, "$12", and trailing zeroes may be left off. Only the first eight characters in a link name are significant.

Users may specify pnames in one of two ways. The first is as an ordinary identifier, referring to any or all matching user names. Lowercase letters are automatically converted to uppercase in **Pascal** and **FORTRAN** modules, while case is preserved for **C** modules.

The second way to specify a pname is as a string enclosed in double quotes, "xxx", referring to the link name of an entry point.

Var

Var is a variable. All variables must contain an identifier. Structured variables may be indexed by constant indices, dereferenced, or fields or members selected, according to its type. The syntax used for describing subvariables includes most legal forms for variables in each language, and may allow non-conflicting syntax from other languages.

Value

Value refers to a constant value. A value may be of type integer, floating-point, or string. Integers may be specified in either decimal form (as an ordinary string of digits) or in hexadecimal form by preceding the value with a dollar sign ($). Floating-point values consist of a decimal integer containing either a decimal point, exponent, or both. Strings consist of singly quoted lists of characters.

Negative values are normally preceded by a minus sign.

DEBUGGER COMMANDS - GENERAL

The debugger prompt character is the minus sign. Any debugger commands can be entered in response to the prompt. (The system may not accept certain commands in inappropriate environments.)

In general, blanks (spaces) are not necessary in debugger commands unless omitting the blank changes the command's meaning. The debugger commands:

```
B 32 E F
```

and

```
B32EF
```

(refer to the break point command) are both acceptable and equivalent to the debugger. The command:

```
B 3 2 E F
```

is not. (The break command accepts one numeric value between the B and the E, and the blank creates two such values.) Except for **C** names, link names, and string values, debugger commands can be uppercase or lowercase.

In response to a partial or incorrect command, the debugger provides a summary of commands available when the error was detected.

DEBUGGER COMMANDS - SPECIFIC

R - Run

The run command transfers control to the target program.

```
     R
```

The program executes normally until a break point is encountered, with the exception that messages are printed on each instance of execution at a trace point. When a break point is encountered, control is returned to the debugger for further interactive command dialog. If the program terminates normally, an appropriate message is printed. In most instances in which fatal run-time errors are detected in the program, control returns to the debugger; the debugger will generally not allow the program to be restarted in this situation.

Q - Quit

The quit command ends a debugging session.

```
     Q
```

This command prompts for confirmation:

```
     Exit program (Y/N) ?
```

Entering **Y**, followed by a RETURN, ends the debugging session. Any other input cancels the quit command, and the debugging session continues.

B - Break Points

The break point command sets new break points in the program being debugged. The general form of a break point command to set a "controlled" break point is:

```
                      {E   }
     B   [nnn[*]]     {X   }    pname
                      {Lnnn}
```

The general form of a break point command to set an "uncontrolled" break point (set by address) is:

```
B   [nnn[*]]   A    {pname [+ hexconstant]}
                    {hexconstant [+ pname]}
```

The optional integer count "nnn" which follows the B command means "break after encountering nnn instances of the specified break point." If this argument is omitted, a count of 1 is assumed. If the asterisk follows the count, the break point is activated on every nnn'th encounter. If no asterisk is specified, the first "nnn" instances of the break point are skipped, but each subsequent instance causes a break.

Pname is the entry point name of the subroutine in which the break point is set. The break point may be set at the entry (E) of the subroutine, the exit (X) of the subroutine, or before the nnn statement (Lnnn) of the subroutine. Compiler-generated program listings can be used to determine the statement number of each source code line in the target program.

A given subroutine gets its run-time conditions (stack frame creation, code to copy value parameters, initialization of register pointers, etc.) from the subroutine entry code (code at the start of the routine). Breaking on entry to a subroutine occurs after the run-time conditions for the routine have been set (before this time, the environment of the subroutine is not meaningful). Similarly, breaking on exit occurs just before the subroutine's run-time conditions are unwound. As a result, it is only possible to break on entry to and/or exit from a subroutine that was compiled with the debugging option on, because the debugger requires the length of the subroutine entry and exit code, in addition to the simple address at which the entry point occurs.

In uncontrolled break points, pname, if specified, refers to the address that is the start of the code for the subroutine. The value **hexconstant** is specified without a leading dollar sign character. If the hexadecimal address immediately follows the "A", the constant must begin with a recognizable digit (a leading zero is permissible), to prevent confusion with the pname syntax. Using uncontrolled break points, any address may be specified. It is the user's responsibility to ensure that the break point is set at an address corresponding to the first word of a 68000 instruction.

T - Trace Points

The trace point command sets new trace points in the program being debugged. The general form of a trace point command to set a "controlled" trace point is:

```
                    {E   }
T   [nnn[*]]        {X   }    pname
                    {Lnnn}
```

The general form of a trace point command to set an "uncontrolled" trace point (set by address) is:

```
T   [nnn[*]]   A    {pname [+ hexconstant]}
                    {hexconstant [+ pname]}
```

The trace point command is exactly the same as the break point command except that the effect of encountering a trace point while executing the target program is to print a message, and then to continue execution without breaking. If break points and trace points are set at the same address, the trace point is inactive until the break point is cleared.

Like break points, trace points must be set at addresses corresponding to the first word of multiword 68000 instructions.

Counts are associated with the address on which the break or trace point occurs. Thus, it is not possible to set a break point and a trace point on the same address, but with different "skip counts."

C - Clearing Break and Trace Points

The clear command clears a break point. The address may be specified by entry, exit, and line number:

```
                         {E   }
C   [B]   [nnn[*]]       {X   }    pname
                         {Lnnn}
```

A break point may also be cleared by referring to the address directly:

```
        C    [B]    [nnn[*]]    A    {pname [+ hexconstant]}
                                     {hexconstant [+ pname]}
```

To clear trace points, the following commands are accepted:

```
                                {E   }
        C    T    [nnn[*]]      {X   }    pname
                                {Lnnn}
```

or

```
        C    T    [nnn[*]]    A    {pname [+ hexconstant]}
                                   {hexconstant [+ pname]}
```

The optional count and asterisk provide consistency with the break point and trace point command but--they are ignored by the clear command. The optional "B" in the clear break point command provides consistency with the clear trace point command and is ignored.

A break or trace point is associated with an address. The break or trace point can be cleared using any of the available methods of specifying the address, regardless of which description of the address was used when the break or trace point was set.

If both a break point and a trace point are set at the same address, clearing one leaves the other set.

Short commands are accepted by the debugger to clear all break points, all trace points, or all break points and trace points. These commands are:

```
CB* - Clear all break points.
CT* - Clear all trace points.
C*  - Clear all break points and trace points.
```

P - Print the Value of a Variable

The print command prints either the value of a single variable or the value of all variables in the current local scope. The general form is:

```
P [var]
```

Without an argument, P prints the current value of all printable variables in the present local scope. With a variable as an argument, P prints the value of that variable. Normally the only values that can be printed are simple variables. Structured values such as arrays, records, or unions cannot be printed. However, a single element of an array, a field of a record, or a member of a union can be printed using the normal syntax in the appropriate programming language.

If the target program is suspended outside a known environment, variables cannot be printed symbolically.

S - Set the Value of a Variable

The S command sets the values of variables in the current environment.

```
S var [=, :=] value
```

Either assignment operator (or none at all) may be used. The variable may be simple, or include indexing (by constant indices), field reference, indirection, etc. The value assigned to the variable must be an integer or real constant (optionally preceded by a minus sign), or a string constant. The value must be appropriate for assignment into the variable. This means the variable with its qualifications cannot be a structured variable. The acceptable variable type and value type combinations are summarized in the following table:

Language	Var Type	Value Type	Notes
Pascal	integer	integer	
	scalar	integer	first=0, next=1, ...
	real	floating-point or integer	includes double-precision
	Boolean	integer	0=FALSE, 1=TRUE or TRUE or FALSE
	char	string	length must be 1
	string	string	
	packed array of char	string	trailing blank-filled
	pointer	integer or NIL	
FORTRAN	integer	integer	
	real	floating-point or integer	includes double-precision
	logical	integer or TRUE or FALSE	0=.FALSE., 1=.TRUE.
	character	string	trailing blank-filled
C	int	integer	
	char	string	
	float	floating-point	includes double-precision
	pointer	integer or NIL	

If the target program is suspended outside of a known environment, variables cannot be set symbolically.

M - Memory Set/Print

In addition to symbolic access to data areas, the debugger also allows the low-level operations of printing and setting memory locations by address. All addresses and values input to or printed by the memory commands are in hexadecimal. The commands are:

```
M P xxx [xxx]
M S xxx xxx [xxx] ...
```

The memory print command expects the initial address to be specified. This is optionally followed by another hexadecimal number. This optional number is interpreted as the number of bytes to print if it is less than or equal to the initial address. The optional number is interpreted as the final address to print if it is greater than the initial address. If the optional number is omitted, 16 bytes of memory are displayed.

The memory set command expects an initial address, followed by one or more values to be placed into successive locations beginning at that address. Each value is interpreted as a single byte, a pair of bytes, or four bytes, depending on the number of hex digits specified in the value. That is, one or two hex digits sets a single byte, three or four sets two bytes, and five or more sets four bytes. If more than eight contiguous hex digits are specified only the last eight are used.

W - Walkback: Print Calling Sequence

When a program is suspended at a break point, it is often possible to determine where the current subroutine was called from, where that calling context was called from, etc. Where the information is available, this calling sequence walkback can be printed using the following command:

```
W [nnn]
```

The optional integer argument limits the number of levels back that the debugger will show. The default number of levels is three. If the debugger cannot show the walkback, an appropriate message is printed. Sometimes the debugger attempts to walkback beyond the main program, resulting in an indication of an environment for which no information is known.

U - Move Environment Up

If a target program is suspended at a break point for which it is possible to walk back through the calling environments, it is also possible to change the current environment of the debugger to be one of those calling environments. It is therefore possible to operate on the state of the target program in these other environments, including accessing data structures that are not visible in the environment of the break point itself. Moving "up" an environment corresponds to changing to the calling context of the current environment. The form of the U command is:

```
U [nnn]
```

The optional count allows moving up through more than one environment at a time.

Note: If the target program is restarted using the run command, execution continues from the break point's environment, regardless of the debugger's current environment.

D - Move Environment Down

After a move environment up command, it is possible to move back down through the active calling environments. This is accomplished with one of the following commands:

```
D [nnn]
D *
```

The move environment down command, without a count (nnn), moves down one environment. If a count is specified, that number of environments is passed. If the * is specified in place of the count, the current environment is set back to the environment in which the break point occurred.

L - List

The debugger can display certain information about the program being debugged other than the values of variables. This information is available using the various forms of the list command described below.

L B - List Break Points

```
L B
```

This command displays the currently set break points and trace points. The address at which the break and/or trace point is set is identified in all appropriate ways, regardless of the method used to set it. In the event that "skip counts" are associated with the break point or trace point, the counts are displayed in the following form:

```
Count=nnn/mmm
```

in which nnn is the number of remaining occurrences which will be skipped, and mmm is the reinitialization value for nnn.

L E - List Entry Point Attributes

```
L E [pname]
```

If a pname is specified, detailed information is displayed about the particular entry point. If pname is omitted, all entry point names and their attributes are shown (except those for names beginning with a percent sign, primarily used by the run-time libraries). Attributes always include the address at which the entry point is located and, where known by the debugger, the language which generated the entry point. For assembly and unknown languages, ??? is displayed as the language. If the pname specified is a percent sign, all entry points are displayed, including those which begin with a percent sign.

L V - List Variable Attributes

```
L V var
```

This command can be applied only to variables accessible in the current environment. Attributes of variables include their type or type number and their storage location.

L F - List Record Fields / Struct or Union Members

```
L F var
```

For record variables accessible in the current environment, this command lists information relating to each field and its addressing attributes.

L T - List Type Description

```
L T [nnn]
```

If the optional integer is omitted, information is displayed about all "numbered" types. If the optional integer is specified, the debugger displays information about that type only.

L S - List Segments and Attributes

```
L S
```

A list of the program's segments and their attributes is displayed. This information is primarily meaningful under operating systems for which segments are usefully managed by programmers.

L D - List Data Areas and Attributes

```
L D
```

A list of the program's static data areas and their attributes is displayed.

L R - List Registers Values

```
L R
```

This command displays the contents of the registers whose values can be determined in the current environment.

< - Take Debugger Commands from a File

A command of the form:

```
< filename
```

is interpreted by the debugger as a directive to accept commands from the named file instead of from the standard input. When the file has been processed, the debugger returns to the interactive debugging mode. Using this feature, it is possible to set break points and trace points for repeated debugging sessions. The file may include other commands as well.

> - Save Break and Trace Points in a File

A command of the form:

```
> filename
```

creates a file with the current break and trace points. The file is suitable for reloading using the "<" command described above. The break and trace points are saved by address (not by entry, exit, or line number) so that care should be taken if the saved break and trace points are reloaded to debug **.bin** files not identical to the one that created the command file.

The debugger places count directives into the created file; these correspond approximately to the counts associated with the break and trace points.

I - Execute a Single Instruction

Programs can be executed on an instruction-by-instruction basis. A command of the form:

```
I [nnn] [q]
```

instructs the debugger to execute "nnn" machine instructions (one instruction if "nnn" omitted) and then return to the debugger as if a break point had occurred. The optional quiet (q) argument silences the default output the debugger otherwise produces on each instruction.

Notes: Using this instruction results in extremely slow target program execution. It is also possible to get the debugger break-pointed in program sections that were not compiled with the debug option set, that are part of the language's run-time system, or even inside the operating system.

In single instruction mode, code located in ROM and/or trap instructions with local parameter and return conventions may be encountered, causing the instruction mode to fail. The instruction mode should therefore not be considered a fully supported feature.

Chapter 4

.DBG FILE FORMAT

All symbolic information required by the Cromemco debugger is contained in an auxiliary file. This file is created, upon request, by the linker and is given the suffix **.dbg**. The executable object code generated by Cromemco **Pascal**, **FORTRAN** and **C** compilers is not altered by setting the debug option. Thus, a production program can be debugged without recompilation if the **.dbg** file is saved.

A **.dbg** file consists of three major sections: link-map information; variable and type definitions; and statement-beginning offsets in the object code. Each major section is created by a different component in the compilation sequence. The linker produces the link-map information, the language front end generates the variable and type definitions, and the code generator produces the object code statement offset tables.

.DBG HEADER

The first 16 bytes of a **.dbg** file are the header. The contents of the header depends upon the target operating system.

.DBG LINK MAP

The first section of a **.dbg** file describes the link map of the associated program. This section contains three subsections: segment definitions, link name entry points, and data area names.

The form of the segment definition table is:

```
+----+----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----
| NumSegs | Segment1 Name | Size1 | Addr1 | ...
+----+----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----
 1    2    3                           18

   -----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    ... | SegmentN Name | SizeN | AddrN |
   -----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                   16N+2
```

where:

NumSegs	- is the number of segments.
SegmentN Name	- is the 8-character name of segment N.
SizeN	- is the size in bytes of segment N.
AddrN	- is the load address of segment N.

The form of the entry point table is:

```
+----+----+-+-+-+-+-+-+-+-+-+-+-+-------+-+-+-+-+-+-+-+-+-+-+
| NEntrys |  Link Name 1  | Addr1 | ... |  Link Name N  | AddrN |
+----+----+-+-+-+-+-+-+-+-+-+-+-+-------+-+-+-+-+-+-+-+-+-+-
 1    2    3                  14                         12N+2
```

where:

NEntrys	- is the number of entry points.
Link Name N	- is the link name of entry point N.
AddrN	- is the address of entry point N. The first byte is the segment number, and the last three bytes are the segment relative offset.

The form of the data area table is:

```
+----+----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----
| NoDatas |  Date Name 1  | Size1 | Addr1 | ...
+----+----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-----
 1    2    3                           18

   -----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    ... |  Data Name N  | SizeN | AddrN |
   -----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                   16N+2
```

where:

NoDatas	- is the number of data areas.
Data Name N	- is the name of data area N.
SizeN	- is the size in bytes of data area N.
AddrN	- is the load address of data area N.

VARIABLE AND TYPE DESCRIPTIONS

The general form of this section is:

```
+---------------------+-----+---------------------+-----+
| Procedure 1's Info  | ... | Procedure N's Info  | FF  |
+---------------------+-----+---------------------+-----+
```

It is a list of subsections, each describing the types and variables of a single procedure, terminated by a single byte of $FF. Each procedure's information is in the form:

```
+-----+-----+-----+-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+------
| Lan | Ver | Sub | Lev |   Link name    | Outer Lnkname | ...
+-----+-----+-----+-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+------
   1     2     3     4     5           12 13            20

    -----+-----------+-------+---+---+-----------+-----+
     ... | User Name | Types | 00 00 | Variables | 00  |
    -----+-----------+-------+---+---+-----------+-----+
          21                                       ??
```

where:

Lan	- is the language in which this procedure is written. 0=**Pascal**, 1=**FORTRAN**, 2=**BASIC**, and 3=**C**.
Ver	- is the language version number.
Sub	- is the language sub-version number.
Lev	- is the procedure's static level.
Link Name	- is the link name of this procedure.
Outer Lnkname	- is the link name of an enclosing procedure. If none exists, this field is blank-filled.

User Name	- is the user name of this procedure. The first byte gives the length of the name and the remaining bytes give the value of the name.
Types	- is a description of any types defined by this procedure. Its format is given below.
Variables	- is a list of any locally defined variables. The format is given below.

Type Descriptors

The form of a type descriptor is:

```
+----+----+-----+-----+
| TypeNo. | Kind | ... |
+----+----+-----+-----+
 1        2     3    ??
```

where:

TypeNo.	- is a two-byte positive integer referring to this specific type.
Kind	- is a byte containing a packed flag (bit 4) and a variant tag (bits 0 through 3). The format of the following information depends upon the value of the tag.

SCALAR:

```
+----+----+----+
| 00 | Max Val |
+----+----+----+
 3    4        5
```

SUBRANGE:

```
+----+----+----+---+---+---+---+---+---+---+
| 01 | RangeOf | Minimum Value | Maximum Value |
+----+----+----+---+---+---+---+---+---+---+
 3    4       5 6            9 10           13
```

POINTER:

```
+----+----+-----+
| 02 | PointerTo |
+----+----+-----+
 3    4         5
```

SET:

```
+----+---+---+
| 03 | SetOf |
+----+---+---+
 3    4      5
```

ARRAY:

```
+----+-----+-----+----+----+   +----------+
| 04 | IndexedBy | ArrayOf |   | PckdInfo | (present if packed)
+----+-----+-----+----+----+   +----------+
 3    4         5 6       7              8
```

PckdInfo - Signed bit is 1 if signed, bits 0 through 3
           are size of element in bits.

STRING:

```
+----+-----+
| 05 | Len |
+----+-----+
 3        4
```

FILE:

```
+----+---+----+
| 06 | FileOf  |
+----+---+----+
 3    4       5
```

RECORD:

```
+----+---+---+---+---+--------+-----+-------+----+
| 07 |   Record Size   | Field1 | ... | FieldN | 00 |
+----+---+---+---+---+--------+-----+-------+----+
 3    4              7 8                        ??
```

and the form of a field descriptor is:

```
+------+-+-+-+-+------+-----+-----+----+ { +--------+---------+ ]
| NLen | N A M E ... | FldType | Offset  | { | LeftBit | NumBits | ]
+------+-+-+-+-+------+-----+-----+----+ { +--------+---------+ ]
                                           Only Present if Packed
```

CHARACTER:

```
+----+----+----+
| 09 | length  |
+----+----+----+
 1         3
```

FORTRAN ARRAY:

```
+----+------+----+----+-----
| 0A | Dims | ArrayOf | ...
+----+------+----+----+-----
 1    2      3      4
```

```
    -----+--------+--+--+--+--+--+--+--+--+--+--+--+-----
     ... | Flags1 | LoBound1  | HiBound1  | ElemSize1 | ...
    -----+--------+--+--+--+--+--+--+--+--+--+--+--+-----
         5       6         9 10       13 14       17

       FlagsK           - 00000000 = Constant Lo, Hi and ElSz
                          000000*1 = Lo computed, at LoBoundK(A6)
                          0000001* = Hi computed, at HiBoundK(A6)
                          If either Lo or Hi is computed, then
                          ElSz is also computed, at ElemSizeK(A6)
```

Predefined type numbers are:

```
 -1: integer, 1 byte
 -2: integer, 2 bytes
 -3: integer, 4 bytes
 -4: integer, 1 byte, unsigned
 -5: integer, 2 bytes, unsigned
 -6: integer, 4 bytes, unsigned
 -7: character, 1 byte
 -8: character, 2 bytes
 -9: single precision floating point (4 bytes)
-10: double precision floating point (8 bytes)
-11: logical, 1 byte
-12: logical, 2 bytes
-13: logical, 4 bytes
-14: file;
-15: complex
```

Variable Descriptors

The form of a variable descriptor is:

```
+-----+-+-+-+-+-----+---+---+----------+
| Len | N A M E ... | vtype | location |
+-----+-+-+-+-+-----+---+---+----------+
 1     2                      Len+3   ??
```

STATEMENT OFFSETS DESCRIPTIONS

```
+-----+-----+     +-+-+-+-+-+-+-+-+----+-----+----+----+
| NumEntrys | ... |    linkname     | EntryLoc | ExitLoc | * NumEntrys
+-----+-----+     +-+-+-+-+-+-+-+-+----+-----+----+----+


        +----+----+-----+-----+-----+-----+-----+-----+
    ... | NumStmt | StmtLoc 1 | StmtLoc 2 | ... | StmtLoc N |
        +----+----+-----+-----+-----+-----+-----+-----+
```

...

Cromemco®

280 Bernardo Ave.
P.O. Box 7400
Mountain View, CA 94039