Introduction

The nucleus of the MPS debugger is an interpreter which, when invoked, decodes and executes a command supplied as a string of text. The syntax of the debugging command language is approximately a subset of the syntax of MPL. In addition, the rules governing the interpretation of commands generally conform to the rules of MPL. The mechanism by which identifiers are bound to values is significantly extended in the debugger, however; bindings are determined dynamically and depend upon the context in which the debugger is invoked.

The standard debugger also includes several collections of procedures which are useful in debugging. These are designed to be called either directly by compiled code or indirectly through the command interpreter. Procedures are provided for examining and modifying data structures, for tracing, and for conducting a conversation with the interpreter from a terminal. Breakpoints are available and are implemented as calls of the debugger's command interpreter which are temporarily inserted into the compiled code. Procedures for setting and clearing breakpoints are part of the debugger.

The following specialized packages, which are documented separately, are designed to be used in conjunction with the debugger:

meter, a set of measurement routines *[see (docmps,meter,:wy)]*

ddt, a machine-oriented debugger *[see (docmps,ddt,:wy)]*

The Debugging Command Language

The syntax and semantics of the debugging command language are described below. The notation and conventions used in the MPS Language Reference Manual have been adopted. Except as noted, the semantic interpretation of each syntactic entity is the same as the interpretation of that entity in MPL.

command = $<;>exp / ;

A command is a possibly empty sequence of expressions. Interpretation of the command is performed by sequential evaluation of the expressions. In the MPL syntax, function calls and assignments are valid expressions. Thus expression evaluation can involve side effects which change the state of the computation or produce output. In addition, the value of each expression in the sequence is printed at the terminal if the debugger's variable evalTrace is nonzero when evaluation of that expression is completed *[see (,variables)]*.

The value is printed with appropriate identification and with a format corresponding to the type of that value

(as computed by the debugger). The values of certain MPS quantities are descriptors or pointers to descriptors. The tests for validating such descriptors are not absolutely reliable, but if one does not appear to be valid, a "?" is typed as the value.

The radix for numerical output is determined by the value of RadixOut , another variable within the debugger, and initially is 8.

```
exp = $<'\>relation;
```

The operator "\" provides for local changes of context, i.e., of the symbolic environment in which identifiers are bound. It is not available in MPL. The form

e1 \ e2

has the value of e2 determined in the context defined by e1. The value of e1 must be the address of a valid frame *[see (,contexts)]*.

```
relation = sum [binrel];
binrel = ('=/'#/'</'>) sum;
```

Relations are evaluated as in MPL. The relational operators are defined only for operands of type INTEGER and yield a result of type INTEGER. The value of that result is determined by normal MPS conventions.

```
sum = $<'+/'->prod;
prod = $<'*/"MOD"/'/>factor;
factor = prim / '( exp ') $qualifier / '-factor / factor path;
```

Arithmetic expressions are evaluated as in MPL. Operands of arithmetic operators must be of type INTEGER, and the results are of type INTEGER.

Field selection can be performed upon a value of any type to extract a component of its representation; the result is of type INTEGER.

A factor qualified by a path (q.v.) is used to describe the location of a statement in an MPS source file. This construct is not available in MPL. The value of such a factor has a composite type which is defined within the debugger (i.e., it is a multiword scalar given an interpretation by the declaration of the record StatementLabel).

```
prim = lhs [ args $qualifier / '← exp ] /
       '$ fwlhs / '↑ .ID / .SR / .dNUM /
       "TRUE" / "FALSE" / .SR1;
```

The following rules govern the treatment of any result words returned by a function. (These ad hoc rules arise

from the fact that the debugger can discover only whether the number of such words is zero or nonzero; they are believed to be compatible with similarly ad hoc rules used by the compiler.)

A call of a function which returns no result can validly occur as one of the elements in an expression sequence which is a command, and no value is printed for that element. Use of such a function in any other context, or with an argument list including a result variable, is considered an error.

Any other function is assumed to return an arbitrarily long list of words. The type of the value returned by a function is determined by the context in which its call occurs.

When the call is itself the right hand side of an assignment, the type is assumed to be identical to the type of the left hand side of that assignment?

In all other contexts, the type is assumed to be INTEGER.

A number of words appropriate for the implied type are first removed from the result list and taken as the value of the function; for each result variable appearing in the argument list, a number of words appropriate for the type of that variable are then removed from the list and assigned to the variable.

External procedure identifiers are bound, if possible, before the procedures they designate are called.

In the debugger, an assignment is allowed only if the left and right hand sides have identical types (as computed by the debugger).

Note that this computation involves only the left hand side when the right hand side is a function call; (assumed) type agreement is automatic.

Field selection can be used to perform componentwise assignment when the types are not identical.

Procedure values can be assigned to external procedure identifiers. External procedures are bound, if possible, before they are assigned.

The operator "↑" is used in the designation of an MPS process. It is not available in MPL. In a primary of the form

↑ .ID

the identifier is interpreted as the name of a process.

Search through the tree of processes begins with the process containing the current context and follows the search rules used in the (default) binding of external procedures /see (,contexts) for further explanation/. The value of the primary is the address of the youngest frame on the stack of the named process.

Auxiliary storage for the descriptor and text of a string constant is released at the end of each debugger command. If retention is desired, the string must be copied into a suitable storage zone.

A number (.dNUM) is a digit sequence without further punctuation. The radix is determined at the time of interpretation by the value of RadixIn , a variable within the debugger with an initial value of 8.

The words "TRUE" and "FALSE" are equivalent to values of type INTEGER which represent the corresponding Boolean values according to the conventions of MPS (currently, to 1 and 0 respectively).

A character preceded by an apostrophe (.SR1) represents the value of type INTEGER which is the ASCII encoding of the character.

lhs = fwlhs $qualifier;

Field selection can be performed upon a left hand side of any type to extract or change a component of the representation of the contained value. The type of a qualified left hand side is INTEGER.

fwlhs = scalar / arrayelt / contents;

scalar = .ID;

The rules used for binding identifiers in the debugger are a superset of the rules used in MPS. They are explained in (,contexts).

arrayelt = .ID '[exp '];

contents = '/ exp '];

qualifier = ('$/'.) field;

field = .ID / '/ .ID;

An identifier used as a field name must be of (implicitly or explicitly declared) type FIELD. The unary operator "/" forms a field selector from any identifier which is bound to a variable (except a REGISTER variable). In conjunction with a pointer to a suitable frame, the selector extracts the first word allocated to the corresponding variable.

```
args = '( [$<',>exp] [': $<',>lhs] ');
```

See the preceding discussion of function calls (under prim) for the rules governing the assignment of values to any variables appearing in the result list.

```
path = '@ $( [.dNUM] move );
move = 'd / 's;
```

A path is used to specify the relationship between two statements in an MPS source file. Paths are not available in MPL. That portion of the path following the "@" is a restricted form of the NLS strucrel, and its interpretation is based upon the tree structure imposed upon each source file by NLS, as follows:

the move "d" stands for the NLS relation "down" (or "first son")

the move "s" stands for the NLS relation "successor" (or "next brother")

a string of digits followed by a move abbreviates replication of that move as many times as specified by the value of the digit string, interpreted as a (decimal) number

the relation described by a path is the product of the relations described by the component moves, taken in order.

Paths describe relations among NLS statements. The use of paths in the MPS debugger is explained in (,breakpoints).

Contexts and Binding

Binding of identifiers in debugger commands is performed in a context which is represented by the address of a frame (possibly the pseudoframe established for the body of a program itself and overlaying the dseg of a process). That frame is called the "current context". It may be any valid frame except one in the active process which is a dynamic successor of the frame for the most recent activation of the interpreter.

Binding of identifiers is done by searching the "visible" frames on the stacks of all "visible" processes in the process tree.

In each frame, matches are attempted in order against

the local identifiers of the corresponding procedure instance (if applicable),

the local identifiers of the process containing the declaration of that procedure (or program),

the identifiers occurring in any modules INCLUDEd in the

DIRECTORY of the program for that process and mentioned in that program (last module to first module).

In each (possibly degenerate) stack, frames are examined from youngest to oldest.

In the process tree, stacks are examined in the following order:

the stack containing the frame specified by the current context (and beginning with that frame),

the immediate sons of that process, youngest to oldest,

all proper brothers of that process, youngest to oldest,

all sets of brothers including a proper ancestor of that process (youngest set and youngest brother first).

This binding scheme uses

MPL binding rules for each frame,

LISP-like binding rules for each stack,

The debugger's procedures TraceBack and StackDump can be used to display the identity of the frames which will be searched *(see (,dumps))*.

MPS' external procedure binding rules for the process tree.

The debugger's procedure PrintProcessPath can be used to display the names of the processes in the search path. A statement of the form

```
PrintProcessPath()
```

causes the names of such processes to be typed in the order in which they are examined. A statement of the form

```
PrintProcessTree()
```

causes the names of all processes to be printed with indentation showing the associated tree structure *(see (,procedures))*.

Warning:

Identifiers declared in INCLUDEd DATA modules will be matched properly, but the displayed values of those which are BASEd variables will be wrong (unless the data module is mpsDATA). The terminal's bell will be sounded as a warning. Variables in such data modules can be accessed using an appropriate base value and the unary "/" operator (or possibly by changing the context).

The frame for the activation of the debugger's interpreter itself establishes the initial current context /see, however, (,signals) and (,tracing)/.

In that context, there will be immediate matches with identifiers known in the debugger /see (,variables)/.

Debugging commands can be used to modify the context. The operator "\" specifies a temporary change in the context. The current context is restored after evaluation of its second operand (even if that evaluation is aborted). The debugger's procedure SetContext is available to change the current context. A call of the form

SetContext(exp)

where exp evaluates to a frame address, resets the current context of the most recent activation of the interpreter to the context represented by the frame.

The conventions for naming processes (using the operator "↑") are useful for specifying contexts, as are the debugger's procedures FramesBack and ProcessFrame /see (,procedures)/. The debugger's variable dC$callFrame contains a pointer to the frame for the most recent activation of the interpreter itself.

If the current context is set to a frame which is not within the active process, it can be invalidated by return from a procedure which changes the state of the inactive process. When this occurs, the interpreter resets the current context to the frame overlaying the dseg of the inactive process or, when that fails, to its own frame. The debugger's procedure ShowContext can be used to verify the setting of the current context.

Establishing Breakpoints

Explicitly programmed invocation of the command interpreter can be achieved within an MPS program by a statement of the form

Interpret(string)

or of the form

CondInterpret(predicate, string) .

In the latter case, interpretation occurs only if the value of predicate is TRUE (nonzero). After the string is interpreted as a debugging command according to the conventions described in the previous sections, control returns and execution continues in the normal way.

A similar effect can be obtained during conversational debugging without altering and recompiling the source text. Procedures are available in the debugger which dynamically insert and remove calls of the interpreter. The location of such a call, called a

breakpoint, is specified to the debugger by a statement label, a value of composite type constructed by the interpreter. A statement label designates an NLS statement in that source file from which the object code was compiled. In the debugger's command language, a statement label is denoted by an MPS procedure or label identifier and a path /see the discussions of factor and path in (,language)/. A call of the interpreter can be inserted immediately before any MPS statement if the corresponding NLS file is structured appropriately.

The forms listed below are available in the command language for constructing statement labels:

an MPL statement label identifier

This designates the NLS statement in which the label occurs. Note that the scope rules of the MPS debugger, not those of NLS, are applied in binding such identifiers.

an expression in the command language which evaluates to a procedure, qualified by a path

This designates that NLS statement standing in the relation, defined by the path, to the NLS statement containing the heading of the corresponding procedure declaration.

an expression in the command language which evaluates to a statement label, qualified by a path

This designates that NLS statement standing in the relation, defined by the path, to the NLS statement designated by the expression.

Each of these forms designates an NLS statement. The MPL statement which is referenced is the first one which begins within the NLS statement. If no such MPL statement exists (because the NLS statement contains only comments, declarations, punctuation, etc.), the debugging system will try to complete a path to an acceptable statement by appending to the path first a minimum number of "s" moves and then the necessary number of "d" moves. When this happens, the appended moves are also printed at the terminal.

The debugger's procedures SetBreak and ClearBreak insert and remove breakpoints. A call of the form

SetBreak(label,string)

effectively inserts a call of the interpreter immediately before the MPL statement referenced by the statement label. A private copy of the string becomes the command to be interpreted when the breakpoint is reached by the executing program, and the interpretation is performed in the context of the broken routine. A call of the form

ClearBreak(label)

removes any breakpoint preceding the referenced statement, and the call

ClearAllBreaks()

removes all breakpoints. The placement of breakpoints is subject to the following restrictions:

a breakpoint cannot be placed where one already exists

the broken statement cannot be an MPL syscall, i.e., a JSYS

breakpoints should be used with caution in INLINE coding

The first compiled instruction of the broken statement is replaced by a JSR and will be executed out-of-line. Skip instructions are correctly processed, but other location-dependent instructions generally are not. In addition, the interpreter does not restore the scratch registers.

Breakpoints are disabled when control resides within the interpreter but are reenabled during the execution of any function calls explicitly appearing in the debugging command.

This scheme is intended to prevent infinite recursion when a breakpoint is set within code used by the debugger. It can be defeated by performing the assignment

bpMask ← 1 .

After the next function call is interpreted, breakpoints will be permanently enabled.

Tracing of Procedure Calls

The standard MPS debugger includes limited facilities for obtaining a trace of the calls of selected procedures. If addr evaluates to an address falling within the stack or data segment of some MPS process, a call of the form

BeginTrace(addr)

begins the tracing of all procedures declared within that process (not the equivalence class of processes sharing the same program). The matching call to terminate tracing is

EndTrace(addr) .

The standard system action upon entry to a procedure being traced is to print the name of that procedure and to await input from the terminal. The name is indented a number of spaces equal to the depth of the procedure call (less 3) relative to the process in which it occurs. One of the following responses, each

terminated by a space, can follow the prompt ">":

[:] (.dNUM / .ID)

A string of digits is interpreted as a (decimal) number; if the number n is specified, control of the trace will return to the terminal only after n subsequent entries to traced procedures have occurred. If an identifier is specified, control will not return to the terminal until a traced procedure with an identifier matching the given identifier is entered. The identifier of each traced procedure which is encountered before control returns to the terminal is normally printed; printing is suppressed by a prefixed ":" in the response. Calls of procedures not being traced are ignored; they do not affect either the counting or the identifier matching.

The tests controlling the tracing options can also be changed at any time by a procedure call of the form

ptEnable(string)

where the string contains a message of the form given above.

<ESC>

The escape character requests a break and establishes a new conversation with the debugger in the context of the traced procedure. When the conversation is terminated, the tracing routine prompts for another response.

<null>

A null message (i.e., a blank) resumes execution. Control will return to the terminal upon the next entry to a traced procedure.

The trace routines are protected from self-tracing. During an escape into the debugger from the tracing package, tracing is suppressed but can be reenabled by a call of the form

. EnableTrace() .

Conversational Command Interpretation

The debugger's procedures BkPtConverse and CondBkPtConverse provide an interface for interactive command interpretation and are designed to be called as part of a debugging command. Interpretation of the expression

BkPtConverse()

causes further commands to be accepted from the terminal and to be interpreted in the context already established for the command string within which it is embedded. Interpretation of the

expression

CondBkPtConverse(expr)

has the same effect if the value of expr is TRUE (nonzero) and null effect otherwise.

Since these two procedures rely upon previous action of the command interpreter to establish part of the environment, they should not be called directly by compiled code (unless that code is always entered through the interpreter).

The debugger's string variable chat is initialized to "BkPtConverse()". The debugger's procedure variable ChatIf is initialized to CondBkPtConverse. These variables are convenient for setting breakpoints *[see (,variables)]*.

The prompt used by the conversational interpreter when it expects input is "/". When the command interpreter is used recursively, a number of spaces equal to the depth of recursion (less 1) precedes the prompt. The proper response to the prompt is a command, as described in (,syntax), or the single character "!", which terminates the conversation.

A limited amount of intraline editing, similar to that provided by NLS, is available. The following control characters are recognized:

↑a backspace character

↑w backspace visible

(delete to a visible, then delete to an invisible)

↑q backspace alphanumeric

(delete to alphanumeric, then delete to nonalphnumeric)

↑x delete line

↑r retype current line

↑v literal next character

Each line is terminated by an EOL.

When the standard MPS save file is loaded, BkPtConverse is in control. Commands are accepted and interpreted as described above. Note that additional processes can be created and run by using such system procedures as Create (or mplCREATE), Run (or mplRUN), and Control. Termination of the top-level conversation with the debugger returns control to the process MPLRUN.

Signal Monitoring

In conversational mode (i.e., when BkPtConverse or CondBkPtConverse has been invoked), all signals generated during the interpretation of debugging commands are intercepted by the debugger and analyzed by the procedure assigned to the debugger's variable SignalAnalyzer . That procedure should conform to the following specifications:

The input parameters are, in order, the signal code and the message associated with the signal.

The single returned value specifies the action to be taken:

1: resume

2: exit to the command interpreter

other: allow the signal to propogate.

The default signal analyzer is the debugger's procedure SignalTrap. It clears the terminal's input buffer and reports the signal, with symbolic identification, to the user, who chooses the disposition of the signal with one of the following responses:

r: resume

x: exit

p: propogate

b, <ESC>: break (recursively)

When a recursive break is requested, a new conversation with the debugger is initiated in the context defined by the frame which originated the signal. Note, however, that the immediately accessible variables are those of mplSIGNAL *(see (,contexts))*. When the conversation is terminated, the debugger prompts for a further disposition.

SignalTrap treats the UnboundFunction signal as a special case. If the external reference responsible for the signal can successfully be bound (by the procedure BindProcedure), such binding takes place and computation resumes without comment. If the debugger's attempt to bind the external reference fails, however, the unbindable identifier is printed and the UnboundFunction signal is processed as described above.

Dumps

A procedure declared with the following heading is available for examining the state of any process:

```
(StackDump) PROCEDURE(address);
```

If the address falls within the stack or data segment of some MPS

process, both the stack and the data segment of that process are dumped as described below. In particular, the address can be a dseg or frame pointer associated with the process. A similar procedure, declared with the following heading,

(TraceBack) PROCEDURE;

will dump the stack containing the debugger's current context; that dump begins with the current context frame.

When called, StackDump responds by typing the name and dseg address of the selected process. Both StackDump and TraceBack then prompt for parameters specifying the format and contents of the dump.

The following responses to the prompt ">" set the output mode:

"f" for formatted output (default)

"u" for unformatted output.

Setting the output mode is optional. The following responses to the (perhaps repeated) prompt ">" select the values to be dumped:

"n"	none (call trace only)
"p"	parameters of each procedure
"l" or "v"	locals (including parameters) of each procedure
"c"	user's choice (prompt repeated for each frame)
"q"	discontinue the dump .

Frames are dumped from youngest to oldest, i.e., in the order imposed by the dynamic links (OldM values) in the stack of the selected process. If the active process is selected, the dump begins with the frame for StackDump itself.

The name and state word of each procedure on the call chain are always printed. The state word has the form (PC, FRAME). PC is the value of the (saved) program counter for the procedure, e.g., the return address pointing into the procedure. It is expressed as an octal offset in the code generated for the procedure. FRAME is the absolute octal address of the origin of the procedure's frame.

Catch phrases are identified as such. The frame address appearing in the state word of a catch phrase designates the frame created for that phrase; the origin of the frame of the containing procedure instance, the signal code, and the signal message are printed as supplementary information.

The value of each parameter or local variable is labeled with the corresponding identifier. If formatted output is selected

(perhaps by default), an attempt is made to display each value in a format appropriate to its declared type /see the discussion of commands in (,language)/. Otherwise, all values are printed without interpretation as strings of octal digits (grouped into halfwords).

Debugger Variables

Several variables within the debugger are explicitly available for inspection and modification by the user.

The following variables control the behavior of the debugger:

INTEGER RadixIn, RadixOut

%radices for input and output conversion /see (,language)/%

INTEGER evalTrace

%If nonzero, the value of each expression in a command sequence is printed upon its evaluation /see (,language)/. If zero, all output must be programmed.%

PROCEDURE SignalAnalyzer

%procedure which inspects and acts upon signals generated by the execution of a debugger command entered conversationally /see (,signals)/%

The following variables are provided solely to serve as scratch variables for the user.

INTEGER v0, v1, v2, ..., v8, v9

PROCEDURE p0, p1, p2, ..., p8, p9

STRING s0, s1, s2, ..., s8, s9

Note that the declaration of these string variables does not reserve any auxiliary storage for associated text. If such storage is required, it must be explicitly allocated by the user.

STRING chat ← "BkPtConverse()"

PROCEDURE ChatIf ← PROCEDURE CondBkPtConverse

Most identifiers declared in the INCLUDE modules listed below are also known in the initial context of the interpreter. These modules contain definitions of registers, fields, and constants which are useful for examining machine and system data structures.

registers:	(mps,mplregs,)
machine words:	(mps,pdp10words,)
frames, data segments:	(mps,processdefs,)

code segments:	(mps,moduledefs,)
segmentation:	(mps,segdefs,)
symbol tables:	(mps,modulesymdefs,)

Warning:

The general registers are not treated specially by the debugger. In effect, they are global variables, and inspection of one of the volatile registers will usually produce a value generated by the debugger. Assignments to the registers or to private variables of the debugger are not prohibited but must be done with considerable care.

Other Useful Procedures

The following procedures, which are listed alphabetically, are defined within the debugging package and have proved convenient for use within debugging commands. Users should normally avoid tampering with other procedures internal to the debugger.

```
PROCEDURE(addr);

  %begins tracing  of all procedures declared within the
  process in whose stack or data segment the word designated
  by  addr  occurs [see (,tracing)]%

PROCEDURE;

  %establishes a conversation with the command interpreter
  [see (,interaction)]%

PROCEDURE(label SIZE RECORDSIZE(StatementLabel));

  %removes a breakpoint [see (,breakpoints)]%

PROCEDURE;

  %removes all breakpoints [see (,breakpoints)]%

PROCEDURE(predicate);

  %conditionally establishes a conversation with the command
  interpreter [see (,interaction)]%

PROCEDURE(predicate, STRING command);

  %conditionally interprets a debugging command [see
  (,breakpoints)]%

PROCEDURE(addr,n);

  %types [addr], [addr+1], ..., [addr+n-1] in octal [see
  (docmps,ddt,) for more comprehensive machine-oriented
  debugging procedures]%

PROCEDURE(signalcode);
```

%types the symbolic name of the given signal%

PROCEDURE;

%unconditionally enables tracing of procedure calls /see (,tracing)/%

PROCEDURE(addr);

%terminates tracing of all procedures declared within the process in whose stack or data segment the word designated by addr occurs /see (,tracing)/%

PROCEDURE(n);

%returns the address of the frame which is the n'th dynamic predecessor of the current context frame /see (,contexts)/%

PROCEDURE(STRING command);

%interprets a debugging command /see (,breakpoints)/%

PROCEDURE(PROCEDURE p);

%returns TRUE unless p is the unbound function%

PROCEDURE(dsegnum);

%types the name of the process, i.e., the name of the designated segment%

PROCEDURE;

%prints, in order, the names of the processes examined in the application of the external binding rules to the debugger's current context%

PROCEDURE(segnum, level);

%types the process subtree rooted at the process with the designated data segment number and with level characters of indentation at the root%

PROCEDURE;

%types the entire process tree /see (,contexts)/%

PROCEDURE(addr);

%returns the address of the frame overlaying the data segment for the process in whose stack or data segment the word designated by addr occurs%

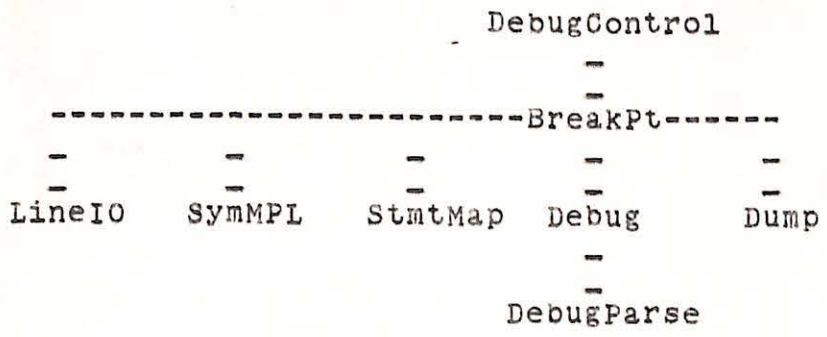
PROCEDURE(addr);

```
    %returns the number of the data segment for the process in
    whose stack or data segment the word designated by addr
    occurs%

  PROCEDURE(PROCEDURE p);

    %returns the data segment number for the process in which
    the procedure is declared%

  PROCEDURE(PROCEDURE p);

    %returns the address, relative to the origin of the
    containing code segment, of the procedure's external entry
    point%

  PROCEDURE(label SIZE RECORDSIZE(StatementLabel), STRING
command);

    %sets a breakpoint [see (,breakpoints)]%

  PROCEDURE(contextframe);

    %sets the debugger's current context to the designated
    frame [see (,contexts)]%

  PROCEDURE;

    %prints an identification of the debugger's current context
    in the form of procedure/program name, relative pc offset,
    and frame address%

  PROCEDURE(addr);

    %produces a symbolic dump of the stack of the process in
    whose stack or data segment the word designated by addr
    occurs [see (,dumps)]%

  PROCEDURE(label SIZE RECORDSIZE(StatementLabel));

    %returns the (current) virtual memory address of the
    machine instruction referenced by the statement label [see
    (,breakpoints)]%

  PROCEDURE;

    %produces a symbolic dump of the stack beginning with the
    debugger's current context frame [see (,dumps)]%

  PROCEDURE(n);

    %types n in octal, with no padding%

  PROCEDURE(n);

    %types n in RadixOut with no padding%
```

& EndPage()

Appendix 1: Process Environment

The debugging system consists of a collection of processes structured as follows:



These processes are part of the standard environment provided by the save file <MPS>MPS. The only active process is DebugControl; each of the others provides a data base and a collection of procedures for use by its parent.

If a nonstandard environment is required, a process corresponding to DebugControl must be created from the file <MPS>DEBUGCONTROL.MPS. Running that process will cause the remainder of the debugging system to be initialized properly.

& EndPage()

Appendix 2: Command Syntax

```
command   = $<';>exp / ;

exp       = $<'\>relation;

relation  = sum [binrel];

binrel    = ('=/'#/'</'>) sum;

sum       = $<'+/'->prod;

prod      = $<'*/"MOD"/'/>factor;

factor    = prim / '( exp ') $qualifier / '- factor / factor
path;

prim      = lhs [ args $qualifier / '← exp ] /

              '$ fwlhs / '↑ .ID / .SR / .dNUM / "TRUE" / "FALSE" /

              .SR1;

lhs       = fwlhs $qualifier;

fwlhs     = scalar / arrayelt / contents;
```

```
scalar    = .ID;
arrayelt  = .ID '[ exp '];
contents  = '[ exp '];
qualifier = ('$/'.) field;
field     = .ID / '/ .ID;
args      = '( [$<',>exp] [': $<',>lhs] ');
path      = '@ $( [.dNUM] move );
move      = 'd / 's;
```