Debugger Meetings and Specifications

4 SEP 72

MPS 5.1

James G. Mitchell
Edwin H. Satterthwaite*

Xerox Palo Alto Research Center*
3180 Porter Drive
Palo Alto, CA 94304
(415) 493-1600

Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, CA 94025
(415) 326-6200

Table of Contents:

(Minutes)

Minutes of debugger meeting 6-21-72

There was some initial discussion of the types of breakpoints to be provided in the initial debugger. It was agreed that the following types would be the only ones initially available:

breaking before a given statement,

breaking on calls to and from a given routine over a given port

this will allow all calls to and from a given routine to be caught by guarding all ports attached to a routine by the mechanism specified in the minutes of 6-8-72.

BWL: should have variables in the debugger so that one can assign pointers (or other values) to them for use in displaying and setting the values of items.

Some of these might have a priori definitions such as FRAME (the current frame being looked at), PROCESS, etc.

The control actions one can perform over MPS routines must include at least the following:

the ability to call a function with simple argument specifiers (allowable variable specifiers are described below), or equivalently to send control to a routine via some port (without disturbing the port connections as an option)

Variable specifiers must include at least the equivalent of the MPS '. and '$ operations and simple arithmetic expressions.

JGM suggested that the runtime variables could be put in the outermost symbol table used for segment naming. BWL pointed that this prevented one from having separate instances of the debugger; hence, the runtime variables ought to be either global to the whole world or local to an instance of the debugger.

BWL suggested using assignment operator to set runtime variables (what a radical

ideal). It was also pointed out that printing items in a format provided by a record definition could not be handled by the '. and '$ operators alone, so a command to "print x as record-name" is needed.

Afterthought (JGM): why couldn't x$recordname be used for printing purposes? Then printing a simple variable x can be viewed as a short form for FRAME./x; this should not cause any more trouble than the meaning of x in "x←value;".

This would also allow one to display a given item in a specific format simply by saying DISPLAY x$INTEGER, for instance.

CHI suggested the need for a facility to map an address into the name of a variable, if possible, including which routine it belongs to. JGM later suggested the same facility for routine bodies: i.e., map an address into an NLS statement number.

Facilities for looking backwards or forwards n stack frames or n incarnations of a specific procedure were also suggested. This agrees with the general philosophy stated in the minutes of 6-8-72 that one can move around by following pointers such as return links, port connections, etc.

It was also pointed out that facilities are needed to map an NLS statement number to an address (for setting breakpoints) and for getting the statement number of the next statement following a given statement number. This is sufficient to allow one to sequence through the statements of a routine and to find the first and last words of the code for a given statement.

Minutes of meeting 6-27-72. Attendees JGM, BWL, LPD, WHP

JGM's afterthought concerning the use of $typefield to specify how to format an object for output was generally accepted.

LPD stated that we need to determine what the handle on an object should be, e.g., a base address in memory and a pointer to a symbol table entry which describes the object. He also mentioned that a reasonable search strategy for names might be to first look for a match in the procedure to which FRAME points, then in its lexically containing routine (later in lexically containing procedures also) and then back up one frame and do the same thing for that frame.

Afterthought (JGM): a reasonable thing to do at the routine level would be to look at the brothers of a routine before backing up to the parent routine's level in the

routine tree. In this way, the normal binding rules for objects such as procedures can be used here as well.

In any case, there will be a sequencer which will take a simple name and produce an (address, symbol table address) pair following some well-defined search path, starting at the zeroth table of the local run-time variables and context for the running instance of the debugger.

Minutes of meeting 8-23-72. Attendees JGM, EHS, DDC.

The debugger will mainly be a collection of MPS procedures, notably including

(Interpret) PROCEDURE (⟨String⟩),

where the string is a sequence of commands in the debugging language (see Specifications for syntax).

It was agreed that expressions in the debugging language should follow normal MPL rules of precedence and nesting; this imposes no real burden on the implementation.

Multiple incarnations of the debugger are possible. Runtime variables can be associated with each incarnation as follows: entry to the debugger creates an instance of a dummy program, say VARS. (At least by default) runtime variables defined by that incarnation become local variables of the corresponding VARS process, which is also made the owner of any processes created by the debugger-provided Create procedure local to that incarnation.

The user can, if he wishes, explicitly create instances of VARS anywhere in the tree structure which defines the name space.

This implies an extension of the syntax for declarations.

Each instance of VARS will be associated only with a data (i.e., writable) segment; however, that segment will look (and be linked) as a program segment (including symbol tables) so that no special symbol table packages are required.

The structure previously proposed for object handles does not cater for, e.g., array elements (whose "names" such as a[2] are not in the symbol table), pointers, etc. It was suggested that object handles should be internally complete without reference to symbol tables; they might have the following components:

ObjectAddress: the base address of the object.

SemanticEntry: the symbol table semantic entry (possibly condensed or synthesized) for the object.

PrintName: a string which is an arbitrarily complex external name for the object.

The problem of deciding whether to pass objects or object handles to procedure calls encountered by the interpreter within debugging command strings was raised by JGM.

One level of "evaluation" is required for ordinary MPS procedures but not for (some) debugging procedures. The latter should not, however, be subject to special syntactic or semantic rules.

In a fully typed language, coercion rules would help somewhat, but note (JGM) that

object handle -> object handle

is a possible coercion, and there is no way for the interpreter to know when this should be applied.

Another potential solution is to have the interpreter create object handles for each argument of the (interpreted) call, save them on the side, and provide a procedure for accessing them (e.g., by index) from other routines.

For example, a procedure p(x) would ignore the actual parameter, which had been evaluated and placed on the stack, and obtain the corresponding object handle by calling, say, GetObjectHandle(1).

Minutes of meeting 8-24-72. Attendees JGM, EHS, DDC.

The best solution to the problem of passing objects vs. object handles (see Minutes 8-23-72) seems to be the provision of pairs of functions, e.g., DisplayObjHandle and Display.

One member of the pair (of no concern to most users of the interpreter) will accept object handles as arguments.

Users wishing to write specialized debugging packages will normally use these routines.

The other member, which expects to be called from the interpreter, will ignore its (evaluated) arguments but will use a system function (say, GetObjectHandle) to retrieve the necessary object handles (constructed by the interpreter) and to fabricate a call upon the first member of the pair.

This implies that certain MPL procedures will (appear to) accept arguments of arbitrary type.

An arbitrary type includes multi-word scalars in place of one word entities. Hence, the number of arguments checking which is currently done should be done away with. (JGM)

GetObjectHandle must be designed to work properly in recursive situations.

Data Structures.

JGM suggested that the parser within the interpreter should produce tree structures following the conventions of Tree META as closely as possible.

Object handles should include the following information:

Object address

Dseg number -- to tie the object to a particular routine (which otherwise would be impossible)

Semantic information (to be further elaborated)

Frame pointer (as a segmented address)

Name string.

A statement handle should resemble an object handle as closely as possible and should include:

Relative code address (starting location)

Dseg number (for distinguishing processes)

Code length (for delimiting statements)

Name string (as an MPL/NLS statement number).

Breakpoints will initially be set by replacing the first compiled instruction for the broken statement with a jump to a data structure established for that breakpoint.

That data structure will have the following format:

JSP (to code simulating a call of Break)

saved link (overwritten instruction)

return jump (to continue execution)

statement handle (describing the location of the breakpoint)

string (to be interpreted at the breakpoint).

The procedure Break will receive the address of this data structure as a parameter.

If the locations of breakpoints are suitably restricted, Break should be able to terminate with a transfer to the saved isnstruction. This will avoid the need for writing a PDP-10 simulator.

This scheme requires private copies of code segments. JGM agreed to implement this change.

Minutes of meeting 8-25-72. Attendees JGM, EHS, DDC.

There was considerable discussion of naming rules suitable for use by the debugging interpreter (and other components of the runtime system). The following scheme emerged from these discussions.

There is a "current context" (perhaps represented by the runtime variable FRAME?) which designates a process and, when appropriate, a procedure instance within the process.

The current context is set to designate the newly created process by the Create procedure.

When a breakpoint is encountered, a new current context is established (recursively) and is the context of the broken procedure (if any) and of the process whose Dseg address is in the D register (see MPS 10.0, page 32).

The user may explicitly reset the current context using a processname (see below).

A name has the form

name ::= [processname '\ ] objectname

where

processname ::= (.ID / empty) $(': .ID)

objectname ::= .ID $(('. / '$) .ID)

The processname, if present, sets the context for interpretation of the objectname.

The initial identifier names a process and is matched within the current context as described below. The empty alternative designates the root of the process tree (which thus never becomes inaccessible).

Any subsequent identifiers select successive offspring processes (in the process tree established by process ownership). At each step, the newly selected process is that offspring of the established process with a name matching the corresponding identifier.

Since this match is not performed in the interpreter's current context, analysis of the processname will probably occur in the lexical scan.

Object names are to be interpreted as if they appeared in the text of a program existing within the context established by the processname (or the current context, if the processname is omitted). Matches are performed:

(1) against identifiers local to the procedure (if any) defined by the established context,

(2) against identifiers local to the process defined by the established context,

(3) against identifiers local to all those processes which are siblings of the process defined by the established context in the process tree,

(4) recursively, against identifiers local to that process which is the parent of the most recently considered process or to any of its siblings.

If either step (3) or step (4) produces more than one match for the corresponding identifier, the objectname is ambiguous (and will hopefully be reported as such).

As a convenience, search contexts should probably be pushed and popped over expressions or commands decoded by the interpreter, i.e., the process designated by a processname should become the initial search context for further names until a new processname is encountered or until the (textual) end of the command string is reached. It should not, however, become the more permanent current context.

(Specifications) Initial Debugger Design Specification (JGM)

Introduction:

Eventually, many of the facilities provided by the debugger will be implemented by

having an MPS interpreter available at run time. Therefore, the specification for a debugger language which follows stays as close as possible to the spirit of MPS in its syntax. This may mean that it is not the most suitable interface to present to the programmer; that, however, can be provided by a program.

It is also assumed that the core debugger is a program which simply accepts strings in this language and executes them. Any program may pass it such a string by a port call to its INPUT port. Any replies which the debugger makes in response to an input string are given as a sequence of strings which are sent out a port called BUGOUT. If the debugger detects an error in a request, it will send some output over the BUGOUT port and will also send an error number (which is > 0) back over its INPUT port. Of course, any functions provided by the debugger can be called directly from MPS programs without being interpreted as strings first.

The output produced by the debugger when a statement is incorrect may contain some valid output as well as an error message. It is for this reason that an error indication in the form of a non-zero number is sent over INPUT when control returns to the sender of the string. If a user interface would like to suppress all printing if an error occurs, it can buffer all output from the debugger until control returns to it with verification that no error occurred and then do whatever it wishes with the output. If an error does occur, the last string sent will be the error message put out by the debugger.

The debugger will allow breakpoints to be placed on statements; when a breakpoint has been set on a statement, it occurs every time the statement is to be executed. The user can specify a string which will be interpreted each time the breakpoint occurs. A set of functions and a simple convention in the debugger get conditional breakpoints out of this simple mechanism. It also has the nice feature that the language for specifying what is to happen on a breakpoint is the same as the rest of the debugger language.

The debugger also provides for the creation and manipulation of runtime variables. These variables may be any of the basic MPS types, including INTEGER, STRING, or ARRAY. One particular function provided by the debugger is worthy of note: it allows one to call the interpreter to interpret a string. This facility and runtime variables make possible debugger "procedures" to be executed by a simple call such as Interpreter(string). Thus an action which is common to a number of breakpoints need only be specified once as a runtime string variable.

In order to allow a breakpoint routine to be attached to a port of a routine in a transparent way, every port has an indirection flag associated with it which causes control to indirect to another port of some other routine whenever control goes in or out over the port. In reality there are two ports specified when the indirection flag is on, one for control going in via the port, and one for control going out over the port. Since this mechanism is implemented at a very low level, it can be entirely transparent to all levels above it, including the JOINing of ports together, and the passing around of pointers to a port to effect connecting other ports to it. Moreover, the property of port indirection belongs to the port and not to the routine which currently owns the port (i.e., is currently using it).

How does the breakpoint routine know which of its ports it should be pending on? Control may first go into the port or out of it. The answer is that if the port has its in-use flag on, then control will first go into the port otherwise it will first go out the port.

Accessing Objects and Manipulating Them

There are a number of simple names known to the debugger which may be thought of as its static variables, and all name searches are viewed as starting at the innermost debugger context first.

SimpleItem ::= 'FRAME / 'PROCESS / LowerCaseID;

A LowerCaseID used as a SimpleItem is a runtime debugger variable as mentioned in the minutes of 6-22-72. Such a variable is created by the 'NEW declaration described below.

TypeItem ::= 'INTEGER / 'REAL / 'POINTER / 'STRING / 'ARRAY '[[exp ':] exp '] / DeclaredType;

A TypeItem is a pseudo-variable used to influence the debugger's view of the type of an expression, and may be one of the MPS builtin types or a type (read "record" currently) declared by a program.

item ::= SimpleItem / NameToBeLookedUp / TypeItem;

A NameToBeLookedUp may result in a data variable or a procedure; either is acceptable. A procedure in an object specification is called with three arguments:

(1) a pointer to the value of the object specified by the sequence to the left of the occurrence of the procedure name,

(2) 0 => return value, 1 => set value of "field" to be the value of the third parameter,

(3) the value to be stored in the field (valid only if parameter (2) is 1).

```
object ::= [SimpleItem] $( ('. / '$) item);
```

Examples:

FRAME.a

FRAME.a$INTEGER

PROCESS.port.ObjectPort$OwnerRoutine

If an object does not begin with a SimpleItem, the string 'FRAME is assumed.

```
exp ::=  prim  ('+ / '- / '* / '/ ) prim ;
```

At the moment it has been proposed that this be the only form of expression and that execution be strictly left to right (shades of DDT!).

```
prim ::= object / object '( objectlist ') / MPSvalue ;
```

Simple function calls are allowed (note that arguments may not be expressions, only objects). An MPSvalue is a string, integer, or array constant as currently allowed by MPS.

```
objectlist ::= [object $( ', object)] ;

MPSvalue ::=

 decimalnumber / octalnumber ('B,'b) [octalnumber] /
 '" $(printablecharacter) '" / 'TRUE / 'FALSE ;
```

Printable characters include the normal ASCII printable character set and backspace, line-feed, carriage-return and new-line (=line-feed followed by carriage-return).

```
decimalnumber ::= 1$(0/1/2/3/4/5/6/7/8/9) ;

octalnumber ::=  1$(0/1/2/3/4/5/6/7) ;
```

declaration ::= ('NEW / 'GLOBAL) ['INTEGER / 'STRING / 'ARRAY '[number '; number ']] LowerCaseID ['← exp] ;

A runtime variable with the name LowerCaseID is created and initialized to zero (or the null string) or the value of the expression if the assignment option is used; if a type is not specified, INTEGER is assumed. A variable which is declared GLOBAL is created in the outermost scope available at runtime.

assignment ::= object '← exp ;

This is the primary means by which debugger pointers such as FRAME are moved around. The operations may also be used to set any nameable object, including runtime variables.

simplestat ::= assignment / exp ;

Of course, a simple function call is a special case of an expression.

statement ::= simplestat $('; simplestat) ;

Mapping Object Specifiers into Internal "Handles"

The string which specifies an object to the core debugger is mapped into a data structure called an object handle (handle for short) consisting of three components:

ObjectAddress: the base address of the object,

ObjectRoutine: the base address of the dseg (more properly the context block) of the routine in whose symbol table the definition of the object has been found,

ObjectIndex: the index of the symbol table semantic entry in ObjectRoutine's symbol table which describes the object.

In the following discussion of basic debugger facilities, "object" means an object handle. MPS programs can call these debugger functions directly without sending a string to the core debugger provided they provide them with object handles. Handles can be constructed by the function MakeHandle described below.

Statement Specifiers and Statement Handles

A statement is specified as

statementobject ::= object '. statement-number;

e.g., statement lal in procedure f would be f.lal, which, if it was in routine p1 could also be be named p1.lal -- the statement number corresponds to the NLS numbering at compile time and is not local to procedures.

The debugger handle on a statement has the following components:

StmntBase: the address of the first word of code for the statement,

StmntRoutine: the address of the dseg of the routine to which the statement belongs,

StmntIndex: the index of the statement in the structure information block for the module from which the StmntRoutine was created.

Examining Objects and Setting the Value of Objects

Output(object)

construct a sequence of strings to display the object's value and send them out the BUGOUT port.

DisplayStructure(object)

Display the structure of the object in some nice form:

e.g., DisplayStructure(arecord)

```
arecord: RECORD(
  b: INTEGER,
  c: RECORD(
    c1: BYTE[1],
    c2: ARRAY[0:9]),)
  d: STRING[20];
```

MakeHandle(string)

Make an object or statement handle from the string specification of an object or statement. A handle for a null object is returned if the string specification is in error. If the string has the form of a statement specifier and the NLS number is 0, a null statement handle for the routine specified by the part of the specifier to the left of the statement number is created.

Example: in the following, P is a routine and foo is a procedure:

P.0 will return a null statement handle for the routine;

P.foo.0 will return the statement number of the statement preceding the PROCEDURE statement which begins the definition of foo.

Specifying Breakpoints

Functions Provided to be used with Breakpoint Facility

Break(statement, debugger-string)

Set a breakpoint on the statement specified. The break will occur whenever the statement is about to be executed, at which time the debugger-string will be interpreted.

At most one breakpoint may be set on a given statement at a time. Hence, setting a breakpoint on a statement which already has one will remove the previous breakpoint. In this case, the message "REPLACED BREAKPOINT AT statement" will be sent out the BUGOUT port.

(??) Perhaps we should view using Break to replace existing breakpoints as errors and require one to use a function called ReplaceBreak to do this specific action. The user interface could mask this unpleasantness from the x

RemoveBreak(statement)

Remove any breakpoint which may be on the specified statement. By restricting the number of breakpoints on each statement to one, we are able to identify each breakpoint by the statement number to which it is attached rather than by some secondary name (such as the small integers used by DDT, for instance).

DisableBreak(statement)

Temporarily remove any breakpoint attached to the given statement. the breakpoint can be turned on agin by a call on the following function.

EnableBreak(statement)

Turn on the breakpoint attached to the given statement.

DisableBreaks(object)

Temporarily turn off all breakpoints attached to statements in the given routine or procedure.

EnableBreaks(object)

Turn on all the breakpoints on statements in the given procedure or routine.

ListBreaks(object)

List all the breakpoints for a given routine or procedure along with the string to be interpreted at each breakpoint.

NextStatement(statement)

Returns a handle for the (lexically) next statement following the one given; if the given statement is the last one in a routine, a null statement handle for that routine is returned. If a null statement handle for a routine is given, a handle for the first executable statement in the routine is returned.

Creating and Destroying Routines:

Create(LowerCaseID, modulespecifier, ownerobject)

Make an instance of a routine whose name will be the specified LowerCaseID and which will be created either from an object module or an already existing routine. The owner of the new routine will be ownerobject.

Destroy(routineobject)

Destroy the named routine. Any routines which it owns will be destroyed first.

MakeOwner(routineobject,ownerobject)

ownerobject will be made the owner of the routineobject. No protection is provided to prevent a routine from altering the ownership of another.

Join(object1, object2)

Bind object1 to object2. The allowable types of object1, object2 pairs are the following:

object1	object2
port	port
pointer	anything

procedure variable	procedure variable or actual procedure
signal variable	signal code