MIT Dyn

IDENTIFICATION

A MEDDLE MANUAL

Greg Pfister

30 January 1972

DRAFT

MOTIVATION

I wanted to be able to define events and put breakpoints in MUDDLE programs.

Doing that implied being able to do two things: 1) say where the breakpoints/events were; 2) modify the program (in the absence of monitor bits) to insert breakpoint and event "functions".

Since the above adds up to about 7/8 of an editor, I figured that I might as well finish the job. It became somewhat larger than I expected.

The acronym, by the way, stands for Muddle Editor, Dynamic Debugger, and Little Else.

REFERENCES

1. Pfister, Greg, A MUDDLE PRIMER -- SYS.11.01.
2. Daniels, Bruce, Micro Muddle Manual -- SYS.11.03.

BODY

The body of the manual follows.

1 LOADING

MEDDLE is an editor/debugger written in, written for, and running under MUDDLE.

To load it, type to MUDDLE

```
    <FLOAD "MEDDLE" ">" "DSK" "MUDDLE">$
```

and wait a while.

Loading MEDDLE inserts 4 atoms into the root oblist. Conflicts with their atom names must be mediated by the user. They are:

```
MEDDLE
MEDDLE_BLOCK
MEDDLE_Q
MERDE
```

More may be added in the future; they will be of the same general form. You have been warned.

Loading MEDDLE also causes a modified copy of BKD's pretty-printer to be loaded for MEDDLE's use. It and a frame typeout function using it have atom names put into the root oblist so you have them available. The atoms and calls are:

```
<PPRINT <object>>
<FRAMES <n>>
```

The first pretty-prints <object>, the second pretty-prints <n> levels of FUNCT and ARGS.

MEDDLE will automatically start up when the FLOAD finishes. To start it manually, for instance after an ERRET, apply MEDDLE to no arguments. It will produce some form of "hello" message, followed by a note on the use of <>'s.

2 THE READER

The routine in MEDDLE which controls typein is the reader. It has two purposes, the first of which cannot, under the current MUDDLE, be achieved without the second:

1) Lexically blocking the ATOMs which designate MEDDLE's commands away from the user's ATOMs, while allowing access to both sets of ATOMs.
2) Making it possible to avoid typing < and > to apply MEDDLE's functions (commands). In the best case -- which occurs often -- this halves the number of keystrokes needed.

Since the outermost < and > are provided by the MEDDLE reader, where you would normally say, for instance,

```
<SET THING 1>$
```

you now need only say

```
SET THING 1$
```

(although <SET THING 1>$ works too). The difference between the two, as explained below, is the structure of the oblist used. In the above cases, since none of the ATOMs held MEDDLE functions, there was no difference.
This means, by the way, that you can't type in more than one FORM at a time.
Basically, you can use MEDDLE's functions naively, with little chance of conflict. The occurence of conflicting ATOM PNAMEs requires a little more work to get around, but only when you wish to evaluate a function of your own under MEDDLE.
The way to do this when there is conflict is to precede the use of the ATOM name at the highest level with ^N (ctrl-N). E.g., to apply your function P (not MEDDLE's), to the LVALs of your ATOMs S and B (which also conflict), type either of the following two:

```
^NP .S .B$
^N<P .S .B>$
```

The way the reader works is as follows:
The reader looks up the first MUDDLE object typed on a special oblist, namely

```
(,MEDDLE_BLOCK !.OBLIST)
```

where ,MEDDLE_BLOCK is an oblist containing only MEDDLE's commands. All other objects typed are looked up on .OBLIST.
After the lookup, a FORM is made out of the entire input and EVALed. (Unless there is only one object in the input and it is a FORM -- e.g., .FOO -- in which case the one object is

EVALed. Or unless the first element of the FORM is an undefined ATOM, in which case it tells you that the ATOM is undefined.)

3 MEDDLE COMMANDS

3.1 GENERAL

MEDDLE commands are all functions; using them evaluates them. You are somewhat buffered from typing errors by the reader, which will not permit top-level application of anything with no value, and will tell you so. Any other mistakes will lead you to a MUDDLE error. To return to the MEDDLE reader from a higher level (say, an ERROR), type

<MERDE>$

(MEddle ReaDer Exit). Usually there are no bad results. (If you get any, tell me about them)

Note that ALL arguments to MEDDLE functions must be legal MUDDLE objects. In particular, you can't search for <SET , since the <>'s aren't balanced. Nor can you insert it. (But you can, for instance, search for and insert <SET THING 1> and SET THING.)

Unless otherwise stated, every command (function) prints the object immediately to the right of the pointer when it is done -- unless it would print more than a specified (changeable) number of characters. In the latter case, it returns the ATOM Moby. See the V command.

The notation _#_ after a command means that an optional arg of type FIX causes the command to be repeated arg times.

The notation _#_#_ after a command means that it takes two optional args of type FIX. These are used to refer to a segment around the cursor, with positive to the right and negative to the left. Specifying one arg causes the other to be taken as 0; specifying neither causes the arguments 1 and 0 to be used.

The format used in each of the following command descriptions is the following:

COMMAND

ENGLISH EQUIVALENT

DESCRIPTION (text)

3.2 General Commands

?

huh?

Prints a list of the currently available commands and their arguments.

O

Open

Open an object for editing. Takes 0, 1 or 2 args. Positions the pointer just to the left of the entire object specified. Returns strings to indicate whether it opened the global or local value. If it can't find anything to open, it tells you so.

No args -- Uses the last specified ones.

1 arg -- Must be an ATOM, namely the one whose "value" you wish to edit. Uses global value if there is one, otherwise local.

2 args -- First the same as above; existence of second forces use of local value.

P_#_#_

Print

Pretty-prints objects around the pointer, ignoring anything done via the V command.

Arg1 is FIX -- Prints a segment as described above.

Otherwise -- Prints the entire object Arg2 levels above the current one. Default on Arg2 is 0, i.e., the current level. Prints ^!^ when in this mode to indicate where the cursor is.

No args -- Assumes an args of 1 and 0.

PC

Print Comment

Prints the comment associated with the object immediately to the right of the pointer.

V

Verbosity

No args -- Toggles the verbosity mode between normal (all commands print something) and silent (only P prints).

1 arg -- (of type FIX) Sets the maximum number of characters a non-P command will print in normal mode. Over that limit, they print the ATOM Moby.

3.3 Movement Commands

R _#_

Right

Move the pointer 1 object to the right.

L _#_

Left

Move the pointer 1 object to the left.

DL _#_

Down Left

Position the pointer just to the LEFT of the LEFTmost object within the object to the RIGHT of the pointer.

DR _#_

Down Right

Position the pointer just to the RIGHT of the RIGHTmost object within the object to the RIGHT of the pointer.

D _#_

Down

Equivalent to DI.

U _#_

Up

Position the pointer just to the left of the object within which the pointer currently is.

WR _#_

Walk Right

Position the pointer just to the left of the next object it would hit on a depth-first, left-first tree walk, starting at the object to the right of he pointer.

WL _#_

Walk Left

Like **WR**, but walk is depth-first, right-first.

MR _#_

Monad Right

WR until you hit a monad.

ML _#_

Monad Left

WL until you hit a monad.

S

Search

WR until the pointer is just to the left of a segment with elements successively =? to its args. On success, prints the segment. On failure, returns you to the place you were when you issued the command. With no args, moves as far right as it can without changing levels.

3.4 Editing Commands

I

Insert

Inserts all its args into the structure being edited at the position of the pointer. All args are quoted, so you can insert unevaluated forms. The pointer ends up to the right of the last object inserted. (If you attempt to insert other than a CHARACTER or a STRING into a STRING, you will lose.)

I:<type>_#_#_

Insert a type around a segment.

Grabs the segment specified, deletes it, and re-inserts it as an object of TYPE<type>.

DEL _#_

DELete

Delete the object to the right of the pointer.

K_#_#_

Kill

Deletes the segment specified, as described above.

C

Change

Equivalent to I followed by DEL.

Z

Zap

Equivalent to:
1) Delete N objects, where N is the number of objects last searched for
2) Insert Z's arguments.

C:

Change Type

Change the type of the object to the right of the pointer to the type given as an argument. Attempts to do something reasonable for every type change. (If you tell it to change a STRING to a LIST, you get a LIST of CHARACTERS. If you attempt to change a structure whose elements are other than CHARACTER and STRINGs to a STRING, you will lose.)

DEL:

Delete Type

Deletes the brackets (generically meant) around an object. I.e., DELs the object and I's its members into the structure of which it was a part.

SC

Set Comment

Attach the arg to the command to the object to the right of the pointer as a COMMENT.

3.5 Breakpoints

B

Breakpoint

Inserts a breakpoint "around" the object to the right of the cursor. Takes any number of arguments. Whenever that object would have been evaluated, you instead hit a breakpoint function which:

1) Types *BREAK*

2) For each argument you gave B, types

<what you typed> = <value of what you typed>

Unless what you typed was of the form <print_thing argumnts> where print_thing can be PRINT, PRIN1, PRINC, or TERPRI. In that case, it just evaluates the form.

3) Enters the MEDDLE reader.

You continue by typing ^T0, like it says.

Breakpoints are currently inelegant. They are implemented by inserting a FORM of the form

<MEDDLE_B (<args to B>) <the object you put B on>>

in place of the object the breakpoint was put on, so don't be surprised if you come across things like that. They really screw up the printing of the object.

CAUTIONS: The breakpoint function returns EVAL of the thing it is put "around", and there are cases where this does not work. There are, however, always equivalent places which do work. Cases:

1) Breakpoint on first element of a FORM does not work. Put it on the whole form.
2) Breakpoint on a list which is an argument to a COND does not work. Put it on the first form in the list.

KBPTS

Kill BreakPoinTS

Removes all the breakpoints in the currently open object.

3.5 "Q-registers"

NOTE: "Q-registers" in an editor running under a system like MU DLE are intrinsically the wrong way to do thing . They will be replaced with something much more reasonable, including a reasonable way to do "macros", as soon as I can convince myself that it works.

QALL

who are Q-ALL?

Prints the names and contents of all current Q-registers.

X <q> _#_#_

eXtract

Gets a copy of the segment specified and inserts it into the Q-register designated by <q>. <q> may be any MUDDLE object. If such a Q-register does not yet exist, it is created and you are so informed. Otherwise, the old one is used, and you are not so informed.

Y <q>

zapY

Like X, but with segment specified as in Z.

Q <q>

Queer.

Like X, but:

1) Only the object to the right of the pointer is obtained.

2) The object itself -- i.e., shared, not a copy -- is used.

Q plus G can create circular lists - so watch out.

G <q>

Get

Inserts the contents of <q> at the position of the cursor.

4 Example

Suppose you have the function

#FUNCTION (('A) <EVAL .A>)

as the global value of the ATOM SIMP, and you wish to change it to

#FUNCTION (("BIND" B 'A) (<EVAL .A .B> .A))

using MEDDLE. Console input (ending with $) and output are on the left, and the position of the cursor (^!^) is shown at the right. The reason for the blank output lines is that there is nothing to the right of the cursor.

```
0 SIMP$
"Global value used."
                      ^!^#FUNCTION(('A) <EVAL .A>)
D 2$
'A                    #FUNCTION((^!^'A) <EVAL .A>)
I "BIND" B$
'A                    #FUNCTION(("BIND" B ^!^'A) <EVAL .A>)
S .A$
.A                    #FUNCTION(("BIND" B 'A) <EVAL ^!^.A>)
R$
                      #FUNCTION(("BIND" B 'A) <EVAL .A ^!^>)
I .B$
                      #FUNCTION(("BIND" B 'A) <EVAL .A .B^!^>)
U$
<EVAL .A .B>          #FUNCTION(("BIND" B 'A) ^!^<EVAL .A .B>)
I: LIST$
(<EVAL .A .B>)        #FUNCTION(("BIND" B 'A) ^!^(<EVAL .A .B>))
DR$
                      #FUNCTION(("BIND" B 'A) (<EVAL .A .B>^!^))
I .A$
                      #FUNCTION(("BIND" B 'A) (<EVAL .A .B> .A^!^))
```