* Psdog-10 with swappable segments *

1. System organization

The modules which make up this system are divided into three sections, common, interpreter and compiler.

- common section:

  (defined in common.ccl)

  Put in this section any module that satisfies any of the following conditions:

  * Contains writeable (ie. low segment) locations
  * Contains the entry points for evaluable predicates or for ~~in-core~~ run-time support routines called from in-core compiled co
  * Contains entry points of run-time support routines for in-core compiled code
  * Contains predicates used by modules in both of the other sections

- interpreter section:

  (defined in interp.ccl)

  Put in this section any module which is used only to support the running of user (interpreted or in-core compiled) programs, and which is not in the common section. Also, put here the system starting point (currently in file PLIUI2.MAC), and system initialisation procedures (like those in JIHTAB.PL).

- xchange section
  ( defined in lxchan.ccl )

  Contains the basic segment xchange routines. It is definitely unhealthy to try to change these modules

- compiler section
  ( defined in compil.ecl )

Put here any modules that are used only during in-core compilation, ie., execution of the evaluable predicate 'compile/1'.

2. Loading

When appropriately loaded, the three sections merge into two executable files, the interpreter segment foo.EXE and the compiler segment PLCOMP.EXE. Note that the later file must be called exactly that, and be placed exactly in the same place as the other, otherwise the compile/1 evaluable predicate will not work, and the Prolog system will complain in no uncertain terms.

To start a run of Prolog, just type R(UN) foo, where foo is the name of the interpreter segment file.

To load the system, several MIC files are available:

- INT.MIC loads the interpreter segment, and saves it as SCRA:PROLOG.EXE[,]
- DINT.MIC loads the interpreter segment with symbols (in the low segment) and saves it as above
- COMP.MIC loads the compiler segment and saves it as SCRA:PLCOMP.EXE[,]
- DBOTH.MIC makes a combined one segment system, with symbols to simplify debugging, and saves it as SCRA:PLBOTH.EXE[,].

(3) Notes on loading the system:

* Do not panic if some undefined globals appear when executing INT, DINT or COMP. This is expected. However, if $ARMM.'s symbols (atom entries) appear as undefined, that is serious. You have somehow miscompiled some Psdos module. (Multiply defined symbols are not expected, and they reveal something quite wrong (maybe you forgot the /S switch when compiling a new main module).

* If you used DDT, it is simpler to use it on the combined system, loaded by DBOTH. If, however, you really want to have fun, try the two segment debug system (DINT+COMP). Note that a bug in the production system can go away in any of the debugging systems, as the memory layout cannot be exactly the same. Furthermore, with the DINT+COMP system:
  * Doesn't know about symbols in the compiler section are not known
  - symbols are in the low segment, to survive the segment swap; they may thus be clobbered by a freak execution
  - if you set a breakpoint when executing the interpreter segment, even in the common section of modules, it may not (can't?) still be there after segments are swapped, to be on the safe side, always

If some of the undefined globals seem definitely wrong, try to do the DBOTH load. If they disappear, either they are ok after all, or some module that should be in the common section has been put somewhere else. Check it.

(4) execute the exchange routine (entry $nwseg) in single step mode and reset break points somewhere after the crucial

CALLI ...

instruction, which tells the monitor to get the other segment

3. Changing the system

If you (lucky boy!) invent a new module, first classify it according to the classification in 1. Then edit the relevant CCL file (also defined in 1.) and add a new line with the name of the new module. Try to keep the common section as small as possible, which makes the EXE files (and the system at runtime!) smaller and speeds up segment exchange, by minimising the swapping of common code. If your module is in the common section because it contains writeable locations, put these locations in the common section as globals and delete them from your module. If the new module contains an evaluable predicate, make two modules of it in the case it contains much more code than just the clauses for the evaluable predicate. The new part which doesn't contain the eval pred might not need to go into the common section.

## To get a Prolog system with (global) symbols

```
.R LINK
*/SYMSEG:HIGH                ; symbols to hiseg
*<our modules, etc>
*program/SAVE
*/GO
```

To run it

```
.RUN program
```

# DDT ing Prolog

- To have DDT from the start

```
. GET program ↵
. DDT ↵

   < set breakpoints, etc. >

   ⓈG
```

- To get DDT in the middle of a session

```
1C to get Prolog interrupt routine
Function ( h for help ) : m
. DDT
    < set breakpoints, etc. >
JRST @.JBOPC Ⓢ X          to continue
```

- To get DDT after a bomb-out

```
. DDT ↵

< examine locations, etc.
```

# Shift/GC Strategy

initial allocations for stacks

global : 1K + (something < 1 page)
local : 1K
trail : 1/2K

$\alpha$: cost ratio for moving trail : 1/10
$\beta$: " " " local stack : 1/10
$\gamma$: " " updating " : 1/10
} Changed to 1/3 on 26Oct78

trail full : increment = localsize * $\gamma$
local stack full : increment = trailsize * $\alpha$
global stack full : increment = localsize * $\beta$ + trailsize * $\alpha$

garbage collection :

successful garbage collection : reclaims $\geq$ 1K

$S = \max$(nº of successful GCs, 1)
$N$ = nº of GCs

To a GC each $LN/S$th global overflow

All current local frames accessible via X, VV chains.

GC: requires:

X, VV pointing at ~~locate~~ the chain of local frames

~~X pointing at the last local frame~~

V " " " top of local stack

VV ~~backtrack frame~~

V1 " " " " " global "

TR " " " " " trail

all accessible from local & global frames complete

all trail pointers pointing at accessible stack locations

updates:

global stack

V1 pointers in local stack, global references

V1

VV1

Global stack oflo:

requires: X, V, VV, V1 ant TR as above

acessible local ~~regis~~ stack frames as above

~~from~~

updates: local stack (shifting ~~V~~ and sometimes TR fields)

~~$LOSE~~ $VMAX $V1MAX

$TRB

$TRTOP

$TRSIZ (sometimes)

TR, $TR0

V, VV, X

---

| | Assumes | Updates |
|---|---|---|
| GSO | NoGC | |
| | GC | |
| LSO | | |
| TRO | | |