

\$SPAD/src/interp Makefile

Timothy Daly

July 28, 2014

Abstract

Contents

1	Notes	3
2	The Environment	3
3	Proclaim optimization	8
4	The warm.data file	9
5	Building DEPSYS	10
5.1	save depsys image	10
6	Building SAVESYS and AXIOMSYS	14
7	Building debugsys	16
8	The Interpreter files	16
8.1	debugsys.lisp [?]	16
9	The databases	17
9.1	autoload dependencies	17
10	The Makefile	17

1 Notes

Notes for understanding this makefile:

IMPORTANT: all source file names in this Makefile must be lowercase This is for cross-platform compatibility and also makes getting them into Lisp much easier at the Makefile level.

2 The Environment

We define 4 directories for this build. The first three are the traditional **IN**, which is where the source pamphlets are, **MID** which is where we will put the compiled intermediates and **OUT** which is where we will put the binaries.

In this case the **IN** files are usually written in **boot** [?]. These will be compiled in a **BOOTSYS** image to translate from **boot** to Common Lisp. Since the Common Lisp code is system independent and machine generated code the **MID** directory is built in the **\$SPAD/int** subtree. See the **\$SPAD/Makefile.pamphlet** [?] file for more details.

The Common Lisp code will be compiled from the **MID** directory into the **OUT** directory. This is system-dependent, machine-generated code and belongs in the **\$SPAD/obj** subtree.

The dvi files will be generated from the pamphlet files in the final ship doc/src/ directory. Since they are system independent but machine generated and part of the final ship they will exist in **\$SPAD/mnt/sys/doc/src/interp**.

— environment —

```
IN=${SRC}/interp
MID=${INT}/interp
OUT=${OBJ}/${SYS}/interp
DOC=${MNT}/${SYS}/doc/src/interp
BOOK=${MNT}/${SYS}/doc
```

In order to minimize the size of the Axiom image at load time we put some of the compiled files into a separate directory that will be autoloaded on demand. This directory of code will be shipped with the final system and so it belongs in the **\$SPAD/mnt** subtree.

— environment —

```
AUTO=${MNT}/${SYS}/autoload
```

We need a raw lisp image that we can use as a base to construct the other images. This is called **LISPSYS** and is system-dependent and machine-generated. It belongs in the **\$SPAD/obj** subtree.

— environment —

LISPSYS= \${OBJ}/\${SYS}/bin/lisp

We need to extract the Numerics chunk from bookvol10.5. We do this using the tangle function in the lisp image.

— environment —

LISPTANGLE= \${OBJ}/\${SYS}/bin/lisp

Most of the interpreter is written in **boot** [?]. Thus we need a program to translate **boot** to Common Lisp. This is called the **BOOTSYS** image (because the translator is written in **boot** and needs to translate itself to bootstrap the system). This image is assumed to have been built by a previous step in the make process.

— environment —

BOOTSYS= \${OBJ}/\${SYS}/bin/bootsys

Some of the Common Lisp code we compile uses macros which are assumed to be available at compile time. The **DEPSYS** image is created to contain the compile time environment and saved. We pipe compile commands into this environment to compile from Common Lisp to machine dependent code.

— environment —

DEPSYS= \${OBJ}/\${SYS}/bin/depsys

The **DEP** variable contains the list of files that will be loaded into **DEPSYS**. Notice that these files are loaded in interpreted form. We are not concerned about the compile time performance so we can use interpreted code. We do, however, care about the macros as these will be expanded in later compiles. All macros are assumed to be in this list of files.

— environment —

DEP= \${MID}/vmlisp.lisp

Once we've compile all of the Common Lisp files we fire up a clean lisp image called **LOADSYS**, load all of the final executable code and save it out as **SAVESYS**. The **SAVESYS** image is copied to the \${MNT}/\${SYS}/bin subdirectory and becomes the axiom executable image.

— environment —

```
LOADSYS= ${OBJ}/${SYS}/bin/lisp
SAVESYS= ${OBJ}/${SYS}/bin/interpsys
AXIOMSYS= ${MNT}/${SYS}/bin/AXIOMsys
```

Occasionally we need to really get into the system internals. The best way to do this is to run almost all of the lisp code interpreted rather than compiled (note that cfuns.lisp and sockio.lisp still need to be loaded in compiled form as they depend on the loader to link with lisp internals). This image is nothing more than a load of the file src/interp/debugsys.lisp.pamphlet. If you need to make test modifications you can add code to that file and it will show up here.

— environment —

```
DEBUGSYS=${OBJ}/${SYS}/bin/debugsys
```

These are the files that need to be compiled (in **BOOTSYS**), loaded into a clean lisp image (**LOADSYS**) and saved as a runnable Axiom interpreter (**SAVESYS**) usually named **interpsys**. Most of these files are translated from **boot** to Common Lisp and then compiled. There are two exceptions, **bootfuns.lisp** and **setq.lisp**. The **bootfuns.lisp** [?] file contains forward references for **boot** code. The **setq.lisp** file contains constant initialization code which gains nothing by being compiled.

The file http.lisp contains code to enable browser-based hyperdoc and graphics.

— environment —

```
OBJS= ${OUT}/vmlisp.${0}      \
      ${OUT}/buildom.${0} \
      ${OUT}/cattable.${0}    \
      ${OUT}/cfuns.${0} \
      ${OUT}/clam.${0}        ${OUT}/clammed.${0} \
      ${OUT}/compress.${0} \
      ${OUT}/format.${0} \
      ${OUT}/g-boot.${0}      ${OUT}/g-cndata.${0} \
      ${OUT}/g-error.${0}    \
      ${OUT}/g-timer.${0}    ${OUT}/g-util.${0} \
      ${OUT}/http.${0} \
      ${OUT}/hypertex.${0}    ${OUT}/i-analy.${0} \
      ${OUT}/i-code.${0}      ${OUT}/i-coerce.${0} \
      ${OUT}/i-coerfn.${0}    ${OUT}/i-eval.${0} \
      ${OUT}/i-funsel.${0}    ${OUT}/bookvol15.${0} \
      ${OUT}/bookvol19.${0}   ${OUT}/bookvol10.5.${0} \
      ${OUT}/i-intern.${0}    ${OUT}/i-map.${0} \
      ${OUT}/i-output.${0}    ${OUT}/i-resolv.${0} \
      ${OUT}/i-spec1.${0}    \
      ${OUT}/i-spec2.${0}    \
```

```

${OUT}/i-util.${0} \
${OUT}/lisplib.${0} \
${OUT}/match.${0} \
${OUT}/msgdb.${0} \
${OUT}/newfort.${0} \
${OUT}/nrunfast.${0} \
${OUT}/nrungo.${0}      ${OUT}/nrunopt.${0} \
${OUT}/posit.${0}      \
${OUT}/record.${0}      ${OUT}/regress.${0} \
${OUT}/rulesets.${0} \
${OUT}/server.${0} \
${OUT}/sfsfun-l.${0}    ${OUT}/sfsfun.${0} \
${OUT}/simpbool.${0}    ${OUT}/slam.${0} \
${OUT}/sockio.${0}      \
${OUT}/template.${0}    ${OUT}/termrw.${0} \
${OUT}/fortcall.${0} \
${OUT}/parsing.${0} \
${OUT}/c-util.${0}      ${OUT}/profile.${0} \
${OUT}/category.${0}    \
${OUT}/functor.${0} \
${OUT}/info.${0}        ${OUT}/iterator.${0} \
${OUT}/nruncomp.${0} \
${OUT}/htcheck.${0}

```

Before we save the **SAVESYS** image we need to run some initialization code. These files perform initialization for various parts of the system. The **patches.lisp** [?] file contains last-minute changes to various functions and constants.

— environment —

```
INOBJS=  ${OUT}/interop.${0}      ${OUT}/patches.${0}
```

Certain functions do not need to be in the running system. If the running image never calls the compiler or does not use the hypertext browser we will never call the functions in these files. The code that calls these functions in the running system will autoloading the appropriate files the first time they are called. Loading the files overwrites the autoloading function call and re-calls the function. Any subsequent calls will run the compiled code.

The **OPOBJS** list contains files from the old parser. The use of “old” is something of a subtle concept as there were several generations of “old” and all meaning of the term is lost.

— environment —

```
# These are autoloading old parser files
OPOBJS=
```

The **OCOBJS** list contains files from the old compiler. Again, “old” is meaningless. These files should probably be autoloaded.

— **environment** —

```
tpdhere
OCOBJS=
```

The **BROBJS** list contains files only used by the hypertext browser. These files should probably be autoloaded.

— **environment** —

```
BROBJS= ${AUTO}/br-con.${0} \
${AUTO}/topics.${0}
```

The **NAGBROBJS** list contains files used to access the Numerical Algorithms Group (NAG) fortran libraries. These files should probably be autoloaded. Note that `${AUTO}/nag-e02a.${0}` is not included in this list as it is a subset of `${AUTO}/nag-e02.${0}`.

— **environment** —

```
NAGBROBJS= ${AUTO}/nag-c02.${0}  ${AUTO}/nag-c05.${0} \
            ${AUTO}/nag-c06.${0}  ${AUTO}/nag-d01.${0} \
            ${AUTO}/nag-d02.${0}  ${AUTO}/nag-d03.${0} \
            ${AUTO}/nag-e01.${0}  ${AUTO}/nag-e02.${0} \
            ${AUTO}/nag-e04.${0}  ${AUTO}/nag-f01.${0} \
            ${AUTO}/nag-f02.${0}  ${AUTO}/nag-f04.${0} \
            ${AUTO}/nag-f07.${0}  ${AUTO}/nag-s.${0}
```

The **ASCOMP** list contains files used by the **Aldor** [?] compiler. These files should probably be autoloaded.

— **environment** —

```
ASCOMP= ${OUT}/hashcode.${0}
```

Axiom versions are given as a string of the form: “Sunday September 21, 2003 at 20:38:05 ” which describe the day, date, and time of the build. This is used for reporting bugs. It is only partially useful in identifying which source code was used. Ideally we could create a tar file of all of the date/time stamps of all of the source files and use the MD5 hash of that file as the version stamp. Ultimately though, this would be chasing the elusive “perfect information” idea.

A new variable `boot:*build-version*` is set and used by the `yearweek` function to display the version number of the Axiom build. This information is set by hand in the top level Makefile.

— **environment** —

```

TIMESTAMP=${MNT}/${SYS}/timestamp
YEARWEEK=(progn (setq timestamp "${TIMESTAMP}") \
               (setq boot::*build-version* "${VERSION}") \
               (yearweek))

```

The **.PRECIOUS** setting will keep make from deleting these images if the build is stopped. Since once they are built they are likely to be useable we don't need to redo the work if they exist.

— environment —

```

.PRECIOUS: ${BOOTSYS}
.PRECIOUS: ${DEPSYS}
.PRECIOUS: ${SAVESYS}
.PRECIOUS: ${AXIOMSYS}

```

3 Proclaim optimization

GCL, and possibly other common lisps, can generate much better code if the function argument types and return values are proclaimed.

In theory what we should do is scan all of the functions in the system and create a file of proclaim definitions. These proclaim definitions should be loaded into the image before we do any compiles so they can allow the compiler to optimize function calling.

GCL has an approximation to this scanning which we use here.

The first step is to build a version of GCL that includes `gcl_collectfn`. This file contains code that enhances the lisp compiler and creates a hash table of structs. Each struct in the hash table describes information that about the types of the function being compiled and the types of its arguments. At the end of the compile-file this hash table is written out to a ".fn" file.

The second step is to build axiom images (depsys, interpsys, AXIOMsys) which contain the `gcl_collectfn` code.

The third step is to build the system. This generates a .fn file for each lisp file that gets compiled.

The fourth step is to build the proclaims.lisp files. There is one proclaims.lisp file for boot (boot-proclaims.lisp), interp (interp-proclaims.lisp), and algebra (algebra-proclaims.lisp).

To build the proclaims file (e.g. for interp) we:

- (a) cd to obj/linux/interp
- (b) (yourpath)/axiom/obj/linux/bin/lisp
- (c) (load "sys-pkg.lsp")
- (d) (mapcar #'load (directory "*.fn"))
- (e) (with-open-file


```
(out "interp-procliams.lisp" :direction :output)
(compiler::make-procliams out))
```

Note that step (c) is only used for interp, not for boot.

The fifth step is to copy the newly constructed procliams file back into the src/interp directory (or boot, algebra).

In order for this information to be used during compiles we define

— **environment** —

```
PROCLAIMS=(progn (load "${OUT}/sys-pkg.lisp") \
                  (load "${IN}/interp-procliams.lisp"))
```

—————

4 The warm.data file

This is a file of commands that will be loaded into interpsys at the last minute. It execute functions that will likely be used in a running system so all of the required routines will be in the lisp image thus minimizing their startup time.

— **warm.data.stanza** —

```
${INT}/algebra/warm.data:
@ echo si001 building warm.data
@ ${BOOKS}/tanglec ${IN}/Makefile.pamphlet warm.data >${INT}/algebra/warm.data
```

—————

— **warm.data** —

```
(in-package 'boot)
(setq |$topicHash| (make-hash-table))
(setf (gethash '|basic| |$topicHash|) 2)
(setf (gethash '|algebraic| |$topicHash|) 4)
(setf (gethash '|miscellaneous| |$topicHash|) 13)
(setf (gethash '|extraction| |$topicHash|) 6)
(setf (gethash '|conversion| |$topicHash|) 7)
(setf (gethash '|hidden| |$topicHash|) 3)
(setf (gethash '|extended| |$topicHash|) 1)
(setf (gethash '|destructive| |$topicHash|) 5)
(setf (gethash '|transformation| |$topicHash|) 10)
(setf (gethash '|hyperbolic| |$topicHash|) 12)
(setf (gethash '|construct| |$topicHash|) 9)
(setf (gethash '|predicate| |$topicHash|) 8)
(setf (gethash '|trigonometric| |$topicHash|) 11)
```

—————

5 Building DEPSYS

The depsys image proceeds all else. it is the compile-time environment for all interpreter code.

The “oldboot” symbol is pushed on the features list because there is a function in util.lisp that emulates the new boot parser command BOOTTOCL. since we eventually plan to move to the new boot parser this function (and the push) should disappear.

The load of postpar and parse (without extensions) allows the .`{LISP}` form to be loaded in a virgin system. However, if depsys is recreated then the compiled form will get loaded.

This file contained the only mention of the AKCLDIR variable which gives the path to the version of AKCL. Now that the system is running on GCL this variable has been renamed to GCLDIR. This cannot be eliminated entirely because the system uses this variable to look up a file called collectfn.lsp which is part of the GCL distribution. This file lookup is in conditional lisp code so other lisps will not see the file load. The collectfn.lsp code is used by GCL to generate the “.fn” files which are used to optimize function calling.

5.1 save depsys image

Once the appropriate commands are in the `{OUT}/makedep.lisp` file we can load the file into a fresh image and save it. At least that’s how it used to work. In freebsd we cannot do this so we have to use a much more complicated procedure. This code used to read:

```
\begin{chunk}{save depsys image}
@ (cd ${MNT}/${SYS}/bin ; \
  echo '(progn (load "${OUT}/makedep.lisp")' \
            '(spad-save "${DEPSYS}")' | ${LISPSYS})
\end{chunk}
```

Now game is much more difficult.

```
'(progn \
```

`si::*collect-binary-modules*` instructs GCL to build a list of binary object modules loaded into the current session with (load ...) The list will be stored in `si::*binary-modules*`.

```
(setq si::*collect-binary-modules* t) \
(load "${OUT}/makedep.lisp") \
```

`compiler::link` is a lisp interface to the “ld” C-based system linker. The first argument is a list of .o binary object modules to link into a fresh gcl image. The second argument is the name of the new output image. The third argument is a string containing an initialization command to run in the new image to

reinitialize the heap. The fourth argument is a list of external C libraries, either static or dynamic, that one wishes to link into the fresh image. The last argument is a flag which indicates whether GCL should initialize all of the freshly linked in new lisp modules, or whether it should transparently redirect load calls in the new image to initialization calls for the already linked in module.

Some lisp systems, such as `acl2`, have a complex heap initialization, in which load calls must be interspersed with other form evaluation comprising the logic of the heap construction. Others, such as `maxima`, have no such complex initialization sequence.

```
(compiler::link \
```

`si::*binary-modules*` here has the list of compiled lisp binary module .o files loaded by `makedep.lisp` above.

```
(remove-duplicates si::*binary-modules* \
                  :test (quote equal)) \
```

The name of the output image.

```
"$(DEPSYS)" \
```

This will be run in the newly linked sub-image.

```
(format nil "\
```

Collect loaded binary modules again to make sure that there are none, as all should be already linked in via `ld`. For error checking purposes.

```
(setq si::*collect-binary-modules* t) \
```

We need to find `gcl_collectfn.lisp`, so set the `*load-path*`, and make sure the source, not the binary, form is loaded here, as we're only using this entire sequence on machines which cannot load binary object modules and preserve them in saved images.

```
(let ((si::*load-path* (cons ~S si::*load-path*))\
      (si::*load-types* ~S))\
```

Turn on function analyzation and autoload thereby `gcl_collectfn.lisp`.

```
(compiler::emit-fn t))\
```

Load the heap creation sequence again in the fresh new image, this time transparently redirecting all calls to load of binary modules invoked thereby into initialization calls for the already linkned in module.

Load has code in it to recognize when a module is already linked in, and to forgo in this case the actual load and replace with a mere initialization call instead.

```
(load ~S "$(OUT)/makedep.lisp")\
(gbc t)\
```

It is an error to load a binary module. Calling load will not reload them but only run initialization. Throw an error if we've actually loaded any binary modules.

```
(when si::*binary-modules* \
  (error si::*binary-modules*))\
```

Unset the binary module collection flags.

```
(setq si::collect-binary-modules* nil \xo
  si::*binary-modules* nil)\
(gbc t)\
```

We need to forget the build time system paths per Camm Maguire's email on July 14th, 2011

```
(si::reset-sys-paths)
```

Turn on SGC (Stratified Garbage Collection) in the final image. This is a optional gbc algorithm which is suitable for images which will not grow much further. It marks a large fraction of the heap read-only, eliminating such pages from the time-consuming gbc processing. When writes are actually made to such pages, a segfault is triggered which is handled by a function which remarks the pages read-write and continues.

```
(when (fboundp (quote si::sgc-on)) (si::sgc-on t))\
```

This is a flag which instructs the GCL compiler to make unique initialization function C names. This is necessary when using ld, as all function names must be unique.

```
(setq compiler::*default-system-p* t)\
```

si::*system-directory* goes into the *load-path*, and .lsp in the *load-types*.

```
" si::*system-directory* (quote (list ".lsp")))\
```

No C libraries to link in here.

```
" " \
```

Do not run the initialization code for the newly linked in lisp modules "by hand", but rather rely on the transparent load redirection described above to initialize at the proper moment in the heap initialization sequence.

```
nil))' | $(LISPSYS))
```

The save dephys image was supposed to read:

```
@ (cd ${OBJ}/${SYS}/bin ; \
echo '(progn' \
'(setq si::collect-binary-modules* t)' \
'(load "${OUT}/makedep.lisp)"' \
```

```

'(compiler::link' \
  '(remove-duplicates si::*binary-modules*' \
    ':test (quote equal))' \
    '$(DEPSYS)'' \
    '(format nil' \
      "(setq si::*collect-binary-modules* t)'" \
      '(let ((si::*load-path* (cons ~S si::*load-path*))' \
        '(si::*load-types* ~S))' \
        '(compiler::emit-fn t))' \
      '(load \"$(OUT)/makedep.lisp\")' \
      '(gbc t)' \
      '(when si::*binary-modules*' \
        '(error si::*binary-modules*))' \
      '(setq si::*collect-binary-modules* nil' \
        'si::*binary-modules* nil)' \
      '(gbc t)' \
        '(si::reset-sys-paths)' \
      '(when (fboundp (quote si::sgc-on)) (si::sgc-on t))' \
      '(setq compiler::*default-system-p* t)' \
        'si::*system-directory* (quote (list ".lsp")))' \
      ', ""' \
      'nil))' | $(LISPSYS))

```

This scheme does not work. It fails during loading with multiple messages of the form:

```

/home/axiom33/obj/linux/interp/parse.o(.text+0x5660):
In function 'init_code':
: multiple definition of 'init_code'
/home/axiom33/obj/linux/interp/postpar.o(.text+0x4e78):
first defined here

```

— depsys —

```

${DEPSYS}: ${DEP} ${OUT}/sys-pkg.${LISP} ${OUT}/nocompil.${LISP} \
  ${OUT}/bookvol5.${LISP} ${OUT}/bookvol9.${LISP} \
    ${OUT}/util.${LISP} \
  ${OUT}/parsing.${LISP} \
  ${OUT}/g-boot.lisp ${OUT}/c-util.lisp \
  ${OUT}/g-util.lisp \
  ${OUT}/clam.lisp \
  ${OUT}/slam.lisp ${LOADSYS}
@ echo si002 making ${DEPSYS}
@ echo '${PROCLAIMS}' > ${OUT}/makedep.lisp
@ echo '(push :oldboot *features*)' >> ${OUT}/makedep.lisp
@ echo '(load "${OUT}/nocompil")' >> ${OUT}/makedep.lisp
@ echo '(load "${OUT}/bookvol5")' >> ${OUT}/makedep.lisp
@ echo '(load "${OUT}/bookvol9")' >> ${OUT}/makedep.lisp
@ echo '(load "${OUT}/util")' >> ${OUT}/makedep.lisp
@ echo '(in-package "BOOT")' >> ${OUT}/makedep.lisp

```

```

@ echo '(build-depsys (quote (($ (patsubst %, "%", ${DEP})))' \
    '${SPAD}' "${GCLDIR}" "${SRC}" "${INT}" "${OBJ}" "${MNT}"' \
    '${SYS}'))' >> ${OUT}/makedep.lisp
@ echo '(unless (probe-file "${OUT}/parsing.${0}"))' \
    '(compile-file "${OUT}/parsing.${LISP}")' \
    ':output-file "${OUT}/parsing.${0}"))' >> ${OUT}/makedep.lisp
@ echo '(load "${OUT}/parsing")' >> ${OUT}/makedep.lisp
@ echo '(unless (probe-file "${OUT}/clam.${0}"))' \
    '(compile-file "${OUT}/clam.lisp"' \
    ':output-file "${OUT}/clam.${0}"))' >> ${OUT}/makedep.lisp
@ echo '(load "${OUT}/clam")' >> ${OUT}/makedep.lisp
@ echo '(unless (probe-file "${OUT}/slam.${0}"))' \
    '(compile-file "${OUT}/slam.lisp"' \
    ':output-file "${OUT}/slam.${0}"))' >> ${OUT}/makedep.lisp
@ echo '(load "${OUT}/slam")' >> ${OUT}/makedep.lisp
@ echo '(unless (probe-file "${OUT}/g-boot.${0}"))' \
    '(compile-file "${OUT}/g-boot.lisp"' \
    ':output-file "${OUT}/g-boot.${0}"))' >> ${OUT}/makedep.lisp
@ echo '(load "${OUT}/g-boot")' >> ${OUT}/makedep.lisp
@ echo '(unless (probe-file "${OUT}/c-util.${0}"))' \
    '(compile-file "${OUT}/c-util.lisp"' \
    ':output-file "${OUT}/c-util.${0}"))' >> ${OUT}/makedep.lisp
@ echo '(load "${OUT}/c-util")' >> ${OUT}/makedep.lisp
@ echo '(unless (probe-file "${OUT}/g-util.${0}"))' \
    '(compile-file "${OUT}/g-util.lisp"' \
    ':output-file "${OUT}/g-util.${0}"))' >> ${OUT}/makedep.lisp

@ echo '(load "${OUT}/g-util")' >> ${OUT}/makedep.lisp
\getchunk{save depsys image}
@ echo si003 ${DEPSYS} created

```

6 Building SAVESYS and AXIOMSYS

GCL likes to tell you when it has replaced a function call by a tail-recursive call. This happens when the last form in a function is a call to the same function. In general, we don't care so we set `compiler::*suppress-compiler-notes*` to true in order to reduce the noise.

— savesys —

```

${SAVESYS}: ${DEPSYS} ${OBJS} ${OUT}/bookvol15.${0} ${OUT}/util.${0} \
    ${OUT}/nocompil.${LISP} ${OUT}/sys-pkg.${LISP} \
    ${OUTINTERP} ${BROBJS} \
${OUT}/database.date ${INOBJ} ${ASCOMP} \
${NAGBROBJS} \
    ${LOADSYS} \
${SRC}/doc/msgs/s2-us.msgs \

```

```

    ${INT}/algebra/warm.data
@ echo si004 invoking make in 'pwd' with parms:
@ echo SYS= ${SYS}
@ echo LSP= ${LSP}
@ echo PART= ${PART}
@ echo SPAD= ${SPAD}
@ echo SRC= ${SRC}
@ echo INT= ${INT}
@ echo MID= ${MID}
@ echo OUT= ${OUT}
@ echo OBJ= ${OBJ}
@ echo MNT= ${MNT}
@ echo O=${O} LISP=${LISP} BYE=${BYE}
@ cp -p ${OUT}/*.fn ${MID}
@ cp -p ${SRC}/doc/messages/s2-us.messages ${SPAD}/doc/messages
# @ cp -p ${SRC}/doc/messages/co-eng.messages ${SPAD}/doc/messages
@ echo '${PROCLAIMS}' > ${OUT}/makeint.lisp
@ echo '(load "${OUT}/nocompil")' >> ${OUT}/makeint.lisp
@ echo '(load "${OUT}/bookvol15")' >> ${OUT}/makeint.lisp
@ echo '(load "${OUT}/util")' >> ${OUT}/makeint.lisp
@ echo '(in-package "BOOT")' >> ${OUT}/makeint.lisp
@ touch ${TIMESTAMP}
@ echo '${YEARWEEK}' >> ${OUT}/makeint.lisp
@ echo '(build-interpsys (append' \
    '(quote ($(patsubst %, "%", ${OBSJS})))' \
    '(quote ($(patsubst %, "%", ${ASCOMP})))' \
    '(quote ($(patsubst %, "%", ${INOBSJS})))' \
    '(quote ($(patsubst %, "%", ${BROBSJS})))' \
    '(quote ($(patsubst %, "%", ${NAGBROBSJS})))' \
    "${SPAD}" "${LSP}" "${SRC}" "${INT}" \
    "${OBJ}" "${MNT}" "${SYS}")' >> ${OUT}/makeint.lisp
@ echo '(in-package "SCRATCHPAD-COMPILER")' >> ${OUT}/makeint.lisp
# @ echo '(|shoeInternFile| "${MNT}/${SYS}/doc/messages/co-eng.messages")' \
#     >> ${OUT}/makeint.lisp
@ echo '(boot::set-restart-hook)' >> ${OUT}/makeint.lisp
@ echo '(in-package "BOOT")' >> ${OUT}/makeint.lisp
@ echo '(load "${INT}/algebra/warm.data")' >> ${OUT}/makeint.lisp
@ echo '(|clearClams|)' >> ${OUT}/makeint.lisp
# @ echo '#+:akcl (si::multiply-bignum-stack 10)' >> ${OUT}/makeint.lisp
@ echo '#+:akcl (setq compiler::suppress-compiler-notes* t)' \
    >> ${OUT}/makeint.lisp
@ echo '#+:akcl (si::gbc-time 0)' >> ${OUT}/makeint.lisp
@ echo '#+:akcl (setq si::system-directory* "${SPAD}/bin/")' \
    >> ${OUT}/makeint.lisp
@ (cd ${OBJ}/${SYS}/bin ; \
    echo '(progn (gbc t) (load "${OUT}/makeint.lisp")' \
        '(gbc t) (user::spad-save "${SAVESYS}"))' | ${LISP} )
@ echo si005 ${SAVESYS} created
@ cp ${SAVESYS} ${AXIOMSYS}
@ echo si006 ${AXIOMSYS} created

```

7 Building debugsys

Note that we assume you've already built interpsys so all of the files are known to exist and be up to date. We don't list any of the preconditions here.

— debugsys —

```

${DEBUGSYS}: ${MID}/debugsys.lisp
@ echo si007 building debugsys
@ (cd ${OBJ}/${SYS}/bin ; \
  echo '(progn (gbc t) (load "${MID}/debugsys.lisp"))' \
    '(user::spad-save "${DEBUGSYS}")' | ${LISPSYS} )
@ echo si008 ${DEBUGSYS} created
```

8 The Interpreter files

8.1 debugsys.lisp [?]

The **debugsys.lisp** file is used to create a **debugsys** runnable image. This image contains almost all of the lisp code that make up the axiom interpreter in lisp form. It is useful for deep system debugging but otherwise worthless. This file is certain to drift over time as changes are made elsewhere to add or remove files. It is assumed that you know what you are doing if you change this file or use debugsys.

This file is basically the same as the one created during the build of interpsys. See the echo lines in the **SAVESYS** block above. These are echoed into a temporary file which gets loaded into the lisp image. We simply captured that temporary file, replaced the .o files with .lisp files (or .lsp or .clisp) and saved it here.

— debugsys.lisp (MID from IN) —

```

${MID}/debugsys.lisp: ${IN}/debugsys.lisp.pamphlet
@ echo si011 making ${MID}/debugsys.lisp from ${IN}/debugsys.lisp.pamphlet
@ (cd ${MID} ; \
  echo '(tangle "${IN}/debugsys.lisp.pamphlet" "*" "debugsys.lisp")' \
    | ${OBJ}/${SYS}/bin/lisp 1>/dev/null 2>/dev/null)
```

9 The databases

9.1 autoload dependencies

If you are adding a file which is to be autoloaded the following step information is useful. There are 2 cases:

1. adding files to currently autoloaded parts
(as of 2/92: browser old parser and old compiler)
2. adding new files
 - case 1:
 - (a) you have to add the file to the list of files currently there (e.g. see BROBJS above)
 - (b) add an autoload rule (e.g. *AUTO/parsing.O: OUT/parsing.O*)
 - (c) edit util.lisp to add the 'external' function (those that should trigger the autoload)
3. case 2:
build-interpsys (in util.lisp) needs an extra argument for the new autoload things and several functions in util.lisp need hacking.

database.date is a marker file used to force a rebuild of interpsys if the database is rebuilt (src/algebra/Makefile).

— databases —

```

${OUT}/database.date:
@ echo 617 the database was updated...remaking interpsys
@ touch ${OUT}/database.date
```

—————

10 The Makefile

— * —

```

\getchunk{environment}

all: announce ${SAVESYS} # ${DEBUGSYS}
@echo 618 finished ${IN}

announce:
@ echo =====
@ echo src/interp BUILDING INTERPRETER FILES
@ echo =====
```

```

clean:
@echo 619 cleaning ${SRC}/interp

\getchunk{savesys}
\getchunk{depsys}
\getchunk{debugsys}
\getchunk{databases}

\getchunk{debugsys.lisp (MID from IN)}

\getchunk{warm.data.stanza}

${MID}/bookvol15.${LISP}: ${IN}/bookvol15.pamphlet
@ echo si125 making ${MID}/bookvol15.${LISP} from ${IN}/bookvol15.pamphlet
@ (cd ${MID} ; \
    echo '(tangle "${IN}/bookvol15.pamphlet" "Interpreter" "bookvol15.${LISP}")' \
        | ${OBJ}/${SYS}/bin/lisp ) 1>/dev/null 2>/dev/null

${MID}/bookvol19.${LISP}: ${IN}/bookvol19.pamphlet
@ echo si128 making ${MID}/bookvol19.${LISP} from ${IN}/bookvol19.pamphlet
@ (cd ${MID} ; \
    echo '(tangle "${IN}/bookvol19.pamphlet" "Compiler" "bookvol19.${LISP}")' \
        | ${OBJ}/${SYS}/bin/lisp ) 1>/dev/null 2>/dev/null

${MID}/bookvol10.5.${LISP}: ${IN}/bookvol10.5.pamphlet
@ echo si131 making ${MID}/bookvol10.5.${LISP} from ${IN}/bookvol10.5.pamphlet
@ (cd ${MID} ; \
    echo '(tangle "${IN}/bookvol10.5.pamphlet" "Numerics" "bookvol10.5.${LISP}")' \
        | ${LISPTANGLE} ) 1>/dev/null 2>/dev/null

${MID}/http.lisp: ${IN}/http.lisp
@ echo si110 making ${MID}/http.lisp from ${IN}/http.lisp
@ ( cp ${IN}/http.lisp ${MID}/http.lisp )

${OUT}/%.${LISP}: ${MID}/%.${LISP}
@ echo siOUTfromMID2 making ${OUT}/${*}.${LISP} from ${MID}/${*}.${LISP}
@cp ${MID}/${*}.${LISP} ${OUT}/${*}.${LISP}

${OUT}/%.${LISP}: ${MID}/%.lisp
@ echo siOUTfromMID1 making ${OUT}/${*}.lisp from ${MID}/${*}.lisp
@cp ${MID}/${*}.lisp ${OUT}/${*}.${LISP}

${OUT}/%.lisp: ${IN}/%.lisp.pamphlet
@ echo siOUTfromIN making ${OUT}/${*}.lisp from ${IN}/${*}.lisp.pamphlet
@ rm -f ${OUT}/${*}.${O}
@ ( cd ${OUT} ; \
    echo '(tangle "${IN}/${*}.lisp.pamphlet" "*" "${*}.lisp")' \
        | ${OBJ}/${SYS}/bin/lisp ) 1>/dev/null

${OUT}/%.o: ${MID}/%.lisp

```

```

@ echo siOUTfromMID making ${OUT}/${*.o} from ${MID}/${*.lisp}
@ if [ -z "${NOISE}" ] ; then \
    (cd ${MID} ; \
        echo '(progn (compile-file "${*.lisp}" \
            ':output-file "${OUT}/${*.o}") (${BYE}))' | ${DEPSYS} ) ; \
    else \
        (cd ${MID} ; \
            echo '(progn (compile-file "${*.lisp}" \
                ':output-file "${OUT}/${I.o}") (${BYE}))' | ${DEPSYS} ) \
                1>/dev/null 2>/dev/null ; \
        fi

${MID}/${*.lisp}: ${IN}/${*.lisp}.pamphlet
@ echo siMIDfromIN making ${MID}/${*.lisp} from ${IN}/${*.lisp}.pamphlet
@ (cd ${MID} ; \
    ( echo '(tangle "${IN}/${*.lisp}.pamphlet" "*" "${*.lisp}"' \
        | ${OBJ}/${SYS}/bin/lisp 1>/dev/null ) )

${AUTO}/${*}.${O}: ${OUT}/${*}.${O}
@ echo siAUTOfromOUT making ${AUTO}/${*}.${O} from ${OUT}/${*}.${O}
@ cp ${OUT}/${*}.${O} ${AUTO}

${OUT}/${*}.${O}: ${MID}/${*}.${LISP}
@ echo si123 making ${OUT}/${*}.${O} from ${MID}/${*}.${LISP}
@ if [ -z "${NOISE}" ] ; then \
    (cd ${MID} ; \
        echo '(progn (compile-file "${MID}/${*}.${LISP}" \
            ':output-file "${OUT}/${*}.${O}") (${BYE}))' | ${DEPSYS} ) ; \
    else \
        (cd ${MID} ; \
            echo '(progn (compile-file "${MID}/${*}.${LISP}" \
                ':output-file "${OUT}/${*}.${O}") (${BYE}))' | ${DEPSYS} ) \
                1>/dev/null 2>/dev/null ; \
        fi
fi

```

References