Proposal for directory tree arrangement for SAX software

Edited by L.Chiappetti (IFCTR)
with substantial input by D.Dal Fiume (ITESRE)

Version 0.0	07 Jan 92	L.Chiappetti - mail to D.Dal Fiume
Version 0.0b	17 Feb 92	L.Chiappetti -other mail to D.Dal Fiume
Version 0.1	20 Feb 92	revised after extensive comments by D.Dal Fiume
Version 0.2	24 Feb 92	final editing after more comments by D.Dal Fiume

Table of content

0) Introduction

It is proposed to have separate directory trees for different components as follows:

1 one tree for the frozen version of the public software
2 possibly separate trees for development versions (TBD)
3 separate (private) trees for observational data
4 a separate tree for DAWG stuff (e.g. SPRs)

Tree 1	is what shall exist at any site (within or outside the SAX Consortium) wanting to install the SAX public software
Tree 3	is typically a private arrangement of each local user wanting to use such software (many of such trees shall exist in a generic site)
Trees 2 and 4	concern only SAX Local Centres, and allow to separate what is public and what is DAWG business only. If a beta-test version of the public software will exist, this could be rooted somewhere in a sub-tree of tree 4 (as beta-testing will be an activity for DAWG).

All the above trees shall be rooted in a relocatable way. The root name shall be kept in a logical name (VMS) or in an environment variable (Unix). Such variables shall be defined in dedicated .COM file (VMS) or csh script (Unix) invoked either at system login (`SYLOGIN`: VMS, `.cshrc` or `.login`: Unix) or private login (`LOGIN.COM`: VMS, `.mycshrc`: Unix/IFCTR) or manually.
Namely it is proposed that root names for cases 1 and 4 be defined at system login level (also for case 2 if a "public" development version exists), and for cases 2-3 be defined at user private level.

For compatibility it is proposed to use names constructed in the following way (example):

VMS: logical name `$SAXTOP` (generated with `ASSIGN/SYSTEM`)
Unix: environment variable `SAXTOP` (generated with `setenv`)
and referred as `$SAXTOP`.

Usage of $ as first character will allow to use **IDENTICAL** names in VMS and Unix. For the same reason it is suggested the name be *upper case*. In general such global variables shall be accessed within the program using a VOS call (It is TBV whether in VMS there are also any reasons to duplicate the logical name into a global symbol with same name).

The choice of the *VALUES* of such global variables will be free and *site dependent*.

1) Main software directory tree

The logical root shall be called `$XASTOP`. It, and all subdirectories and files in it shall be owned by a *dedicated SAX account*.

Under this relocatable root, one shall have a tree of subdirectories. In principle *no subdirectories but the terminal ones shall contain files other than subdirectories.* The only exception to this are `makefile`s and (possibly) other few procedures needed for maintenance (e.g. Real Programmer Tool `@configure` files) or `README` files.

```
root --+-- local
       +-- bin
       +-- lib
       +-- vos
       +-- doc
       +-- include
       +-- libsource +-- library1
       |             |
       |             +-- libraryn
       +--source
       |
       +--calib -sax-+-- exp1
       |             |
       |             +-- expn
       +--external
       +--temp
```

In the above tree, the first directory listed (`local`) shall contain **site-dependent** stuff, the next ones (`bin`, `lib` and `vos`) shall contain **system-dependent** stuff, all the rest should contain **system independent** stuff. See Appendix B for a proposal about *multi-architecture support* and *software distribution*.

1.1) local

This shall be the login directory of the SAX account and contains any file necessary for login et sim. It shall also contain any site dependent files (e.g. peripheral settings). All customization shall be reflected **ONLY** in editing files in this subdirectory.

1.2) bin

This shall contain the program executables of SAX software (`EXE` files on VMS, executables on Unix). It might also additional contain executable procedures (`.COM` files in VMS, shell scripts in Unix).

This directory shall be in the *path* in Unix, while in VMS all `EXE` and `COM` files shall be equated to a symbol (e.g. using XANADU `KNOW` command or Real Programmer `MAKEKNOWN` or alike). In addition it can be noted that *a path mechanism can be implemented in VMS using logical names*. If one does the following

```
DEFINE $PATH DUA0:[DIR1],DUA0:[DIR2]
PROG == "$$PATH:PROG.EXE"
```

one can issue the command `PROG` to execute a program residing in either directory. If there is a `PROG.EXE` in both directories, it will be executed the one residing in the directory coming first in the `DEFINE` statement. This will allow different versions of the same program (e.g. a frozen and a test version, or a public and an user customized version) to co-exist easily.

1.3) lib

This shall contain relocatable libraries (VMS: `.OLB` or shared `.EXE`; Unix: `.a`). All relocatable libraries, including external ones shall be present here (directly or, in case of external ones, perhaps as soft links).

1.4) vos

This shall contain the source code for the routines in the VOS library. The reason to put them here instead than in `libsource/vos` is to keep system dependent stuff clearly separated.
It is yet TBV whether `INCLUDE` files used by VOS routines can reside in this same directory, or in the `include` subdirectory, or in a separate place.

1.5) doc

To contain documentation files (manuals) and help files. There will be two primary subdirectories : `manual` and `help`.

The `manual` subdirectory is intended for manuals to be printed (e.g. Postscript compressed, ASCII, TeX or TBD)
.
The `help` subdirectory is intended for on-line help files. It could be investigated whether it is easy to have system independent help files and compile them into a system dependent format, but one can note that the help utilities are too much different, and almost every astronomical system (MIDAS, IRAF) developed its own help facility.

The policy for manual and help files will come later, is outside of the scope of the present note and has to be discussed later.

1.6) include

This shall contain `INCLUDE` files. These shall have type `.inc` (for Fortran include files) or `.h` (for C header files).

All these files (those not used by VOS routines) could be links to equivalent files on another machine. Keeping VOS `INCLUDE` files outside of here, allows to link the entire subdirectory to the equivalent one on another machine.

1.7) libsource

This shall contain *a subdirectory for each library*. Each subdirectory will contain the source code, *one file per routine*. The file shall have the same name of the routine, and type `.for` (VMS) or `.f` (Unix). See 1.7 below. For C routines the file has the name of the routine as called and type `.c` (even if the routine name within the source has an underscore appended to the name for linker reasons).
An exception will be for those routines which are called exclusively by another routine (and therefore always relocated with it), which may appear in the same source file as the calling routine. *Block data routines* shall be called `blkname`, where name is the name of the (main) common block they initialize.

All these files (being not VOS routines) could be links to equivalent files on another machine. Keeping VOS routine sources outside of here, allows to link the entire subdirectory to the equivalent one on another machine.

For each `libsource/lib`*n* subdirectory (and for the `vos` subdirectory) there will be a `lib`*n*`.OLB` or `liblib`*n*`.a` file under `lib`.

Current test version at IFCTR has here libraries `general`, `vos` (of these under VMS there are a few test variants) and `fitsio`. One could foresee at least one dedicated library per experiment (`lelib`, `melib`, `hplib`, `pdslib`, `wfclib`).

1.8) source

This shall contain the source code of all main programs. Each main program file may include also the code for subroutines called exclusively by it (not in library).*No other subroutine shall be allowed to be either than in a main source or library source file.*

The files shall have the name of the program, and type `for` (VMS) or `.f` (Unix) or `.c` (C, both). In line of principle the entire directory could be a soft links to equivalent directory on another machine.

NOTE: so far we assume it is best to have `.f` files on Unix, not `.for`. We are afraid `.for` files are compiled differently (this occurs on Sun, TBV on DEC). This is the only thing which precludes linking the entire subdirectory from Unix to VMS (link from VMS to Unix will be possible only with UCX 2.0, but in this case one should arrange a way for VMS to compile a `.f` file). Handling of `.f` and `.for` files in Unix in the same way requires some further investigation.

1.9) calib

This shall have subdirectories for `LECS`, `MECS`, `HPGSPC`, `PDS`, `WFC1`, `WFC2` (to allow mission independence, i.e. incorporation of calibration data for other missions, these should be subdirectories of `calib/sax`) Each one of them shall contain all instrument dependent files, e.g. the ones needed as input for the response matrix generation.

If some common tables are needed they could be kept in `calib/general` (or as appropriate `calib/sax/general`).

1.10) external

This is just a root for *SMALL* external packages for which there is no best place (one subdirectory per package). E.g. a thing like `PGPLOT` could go in here (this is a suggestion not a recommendation) and preserve its own directory structure (while the compiled libraries goes into lib).
However it could be desirable to place external *OUTSIDE* of this directory tree (i.e. at the same level as `$XASTOP`).

Anyhow this shall not be the place for *BIG* packages (that is, things like IRAF, MIDAS, XANADU shall have their own independent tree, and even be owned by an independent maintenance account each).

1.11) temp

This is a scratch area (publicly writable ?), where temporary stuff is kept (e.g. relocatable objects before being linked or put into libraries). Could be cleaned daily. Could also be a pointer to somewhere else in the system.

1.12) dependencies and makefiles

The issue of finding proper dependencies is crucial for the construction of good `makefile`s or for good manual maintenance (that is, to know what to recompile when something is changed).

Please note that it is not envisaged to keep object (`.OBJ`, `.o`) files anywhere in the system but *to delete them* as soon as a program is linked or a member is inserted into a library.

A *main* program depends on :

all include files referred in the main source (these could be found with a "`grep .inc sourcefile`")

all called library routines (these could be found looking at some form of compiler output or load map), hence on the library where they are contained (but it is silly to recompile a main program calling `suba` and subb, because they are in the same library as subc, and only subc has been modified : however this is what most makefiles do)

included the block data routines which are not obvious to spot

it also depends on the dependencies of all such routines

A *library* depends on the sources of its routines :

A *routine* depends on

all include files referred in its source
all called library routines including block data

Now while the compiler and linker can resolve references without special documentation files, building correct makefiles might need some special tools (unless one is prepared to make redundant recompilations and relinks).

What it is meant by resolution using self documentation is that a compiler automatically finds all include files, and a linker finds all subroutines provided it is given a list of libraries in the right order (called a loader file). Note, just a list of libraries, not a list of individual routines.

Therefore general usage at IFCTR foresees just a few (3-4) standard *loader files* (e.g. for Exosat on IBM there were a DEFAULT one, an EXOSAT one, an ME and an LE one, on HP there were a general one, and an IA one), with the possible exception of a *VERY FEW* special programs which require an individual one. This *greatly minimizes the number of files* floating around.
It is wasteful to have a loader file for each main program (or a special line or statement in a makefile for each main program). Of course this scheme requires **one knows where his towel is** (that is, what to recompile when something is changed). We believe that *building a makefile which knows where his towel is, requires some more investigations and experimenting*.

We would mention here also additional useful tools, like IBM VS Fortran compiler ICA (Inter-Compilation Analysis), VMS SCA (Source Code Analyzer), lintfor (HP-UX) etc.. They should allow to check consistency of calling sequences between routines (e.g. to flag an error if a routine is called with a real argument when an integer is expected, etc.), and to produce statistics and maps. It is noted that one such tool could be used at a single site (with no difficulty if the programs are standard Fortran 77, or even with some widely diffused extensions) even if the programs are not to run on the same machine where the tool resides.

1.13) Soft links

Soft links are a great way of including external stuff in one's package, as well as useful for many other things. We are faced here with two little problems :

One is that at the moment we can easily create an Unix link to a VMS file (meaningful for text files only if the VMS file is STREAM_LF), but not the other way round. Unfortunately we have (and will go on having) more disk space under Unix. This possibly will be true in both directions with UCX 2.0.

The second is that links are not provided as an user facility in VMS. However they exist (in the form of files with different directory entries; these are widely used in SYSTEM directories) even if with some differences (see the UCX documentation for explanation). It could be useful to find a *PUBLIC* way to create such links in VMS. Usage of QIO calls will possibly allow to emulate hard links (within same physical disk).

It is also TBV the impact of using soft links with makefiles (in particular for date checking)

2) Development software directory tree

We suggest a solution in which each private sw developer has a *private* copy of the necessary branches (and leaves, i.e. files) of the main tree. He shall copy only the main program sources, library routine sources and relocatable libraries which he is actually working on.

In LC's opinion he would not see, at least at IFCTR site, and possibly at any site but SDC, a public development tree. This is also due to the fact that one envisages loosely coupled modules (no major releases of the entire sw but independently updated modules, the coupling been essentially at linker level)

To be investigated : need of access to "unchanged" components remaining in the main tree, use of links (Unix; is there a way to emulate them under VMS ?), use of paths (in VMS a logical name could be defined to be a list of directories in a given search order)

Concerning a *beta-test release* of the software, to be distributed and tested (for instance at the SAX Local Centres), this will be in a tree similar to the main software tree (see 1 above), but of course it should be rooted elsewhere. This root could be arranged somewhere under the DAWG stuff root (see 4 below)

3) Observational data directory tree

Since *each user* shall be handling his/her own data, it is necessary a mechanism to notify the analysis software where it could find the user's data. This mechanism shall use global variables to identify the location of common classes of data (e.g. FOT files, reduced data files, background subtracted cleaned files, sorted by experiment and/or object and/or date of observation).

In the XAS sw spec chapter 6 (/xas/scratch) and 9.2 there was a possible example. This should now be modified so that the system should be flexible enough to allow the user to construct alternate hierarchies (from a flat one, "all under current directory", to any arrangement of choice)

A precise definition of this could be left for later, and ad interim (e.g. for simulation sw) being contented by the sw reading "standard" input files from `calib` and reading/writing "user" files from/to the current directory.

3.1) Concept

The idea is that generally the user provides to the program just a *fileNAME* (e.g. `pinco`). The program will provide to add a **default type according to context** (e.g. an image display program will expect `pinco.image`, a fitting program `pinco.spectrum`, a Fourier transform `pinco.rate` etc.). The program will also add a **default VOS path** according to current settings in the XAS environment (see below). E.g. an analysis program could look for a spectrum as `/somewhere/lucio/amher/19jan/mecs/pinco.spectrum`, or an accumulation program could look for a telemetry file as `/elsewhere/daniele/pds/7feb/gammacas/obs0003.hk`.

Of course it will be possible to *override* the default type specifying one (e.g. a spectrum could be `pinco.panco`) or the default path specifying a full or relative path (anything containing a slash, eg. `/tizio/caio/sempronio/pinco` or `somewhereelse/panco/pinco`), although this will **not be encouraged**.

3.2) The environment

We could foresee the use of a `XASSET` program to set such global variables (LC has not yet tested this for Unix, while for VMS he made tests both using logical names and global symbols, each with pros and cons; more info available on request), since they will be used only internally. Of course a frequently used configuration could be stored in some sort of profile.

3.3) Data directories

The idea is therefore that programs will find data files well classified in specific places (according to user's taste) and not just in the current directory (actually, *independently* of the current working directory).

There are some **physical end directories** where different kind of data are to be put, and some **logical specifiers** which may make easy to classify the data (an example was given in chapter 6 of XAS spec, but it has to be changed now)

3.3.1) End directories

It would be appropriate to provide separate physical locations for:

- FOTDIR:	"telemetry" files as filed from a FOT (or RRD)
- DATADIR:	reduced data files produced by accumulation/reduction
- PRINTDIR:	"other" files if any, e.g. log and output files to be printed (or preserved)
- ??	custom calibration files (IF and WHEN allowed, these could be used in preference to `$XASTOP/calib`)

- ??	custom versions of programs (but a pseudo-"path" mechanism or redefinition of global symbols could be used for this in absence of a XAS monitor)

3.3.2) User controlled specifiers

The user may want to specify the following things :

- ROOT:	the root directory (this could either be a public area, or the user home directory - this would be a choice in VMS - or an user directory in a public data filesystem - this would be a choice in Unix - or anything else, including areas sharable between more than one user)
- CELES:	one may want to separate data by the celestial object observed
- DATE:	one may want to separate data of the same object by date of observation, for repeated observations
- INSTR:	one may want to separate MECS data from PDS data from WFC data etc. even on same object

or one may not want to make some or all of the above separations

3.3.3) Cases

One may therefore be able to chose one of the following arrangements, setting appropriate XAS environment global variables, to be tested by programs

- **all under current** working directory (all variables *NULL*)
- **all under a same** fixed directory (ROOT non null, all other *NULL)*
- separate "end directories" as given above as subdirectories of a fixed directory (ROOT, FOTDIR, DATADIR etc. not null). Example :
  ```
  ROOT=/somewhere/else/again
  FOTDIR=tlm
  DATADIR=data
  ```
  results in `/somewhere/else/again/tlm` and `/somewhere/else/again/data`
- **separate** "end directories" as **terminal directories** of a tree allowing a hierarchy of levels by specifiers (some of CELES, DATE, INSTR not null, plus also ROOR, FOTDIR, DATDIR etc.). One shall be allowed to specify the order of the levels e.g.
  ```
  ROOT=/dua0
  FOTDIR=fot
  DATADIR=red
  CELES=pks2155   DATE=7dec95   INSTR=lecs
  ```
  `ORDER=cdi` results in
  `/dua0/pks2155/7dec95/lecs/red` or `/dua0/pks2155/7dec95/lecs/fot`
  but
  `ORDER = icd` results in `/dua0/lecs/pks2155/7dec95/red`
  and
  `ORDER = ic` results in `/dua0/lecs/pks2155/red` etc.

Nominally one can have the following ordering :

```
 instr/celes/date
 instr/date/celes     (does not make much sense)
 celes/instr/date
celes/date/instr
 date/instr/celes     (do not make much sense as on one date one has
 date/celes/instr      only one object at one time)
 celes/instr          (see note *)
 celes/date
date/instr
 date/celes           (does not make much sense as above)
 instr/celes
 instr/date
 instr
 date
 celes
```

(*note* * there might be problems if there are observations taken at different dates when filing FOT data; we assume data are filed with names like `0001.hk` where the name is the obs number and the type is the data type; one may also have problems filing more instruments in same place; may be names shall be *`eeennnn.typ`*, eg. `pds0001.hk`, or even *`ppeeennn.typ`* with *`pp`* a generic user specified prefix, eg. `mypds0001.hk`)

- finally could one allow to put **end directories at different levels** in the tree ? e.g. to have a tree like

```
 /mydata/printdir                 level 1 under root
 /mydata/ngc4151/reduced                 2
/mydata/ngc4151/mecs/fot                 3
 /mydata/ngc4151/pds/fot                 3
```

etc.

or even structures like

```
/mydata/ngc4151/reduced/17jan95
/mydata/ngc4151/reduced/20feb95
```

where "end" specifier is not at the end ?

Is all the above meaningful, sensible, easy to implement ?

4) DAWG stuff directory tree

The logical root shall be called `$DAWGTOP` [?]. It, and all subdirectories and files in it shall be *owned by the DAWG account*. This is *separate* from the account owning the software. [The name DAWG is proposed instead of SAXDAWG, since this at IFCTR is already a logical name pointing to the SAXDAWG Listserv list, and since the latter resides on a non-SCS machines where one could have other DAWGs, it was best to use a self-documenting name.

Structure: TBD (e.g. this should host the SSREP stuff) The idea is to keep here any stuff related to the DAWG but not part of the "standard sw distribution" to be given to ANY site (including external sites). That is, this directory tree shall exist only at the Local Centres.

Appendix A) VOS name mapping

In contrast with the current provisional VOS pathname mapping. (used by LC for i/o tests) which roots absolute paths under a fixed root, i.e. VOS=`/a/b/c` is translated into SYS=`DUA0:[lucio.a.b.c]` or SYS=`/poseidon/lucio/a/b/c`, one shall have.a **definitive VOS mapping**.

A.1) Unix

In Unix this could just be a **no-op**. VOS names equal to system names. In particular if the first character is a dollar, the first component of the path is unchanged also in case (to allow *use of shell variables*). It is TBV whether it is perhaps desirable to *force the rest in lower case*, or one should *preserve case* as typed by the user. We could perhaps add to the Unix notations `~ .` and `..` some vms-like notations as `- -- ---` (to go up one, two, three steps etc.).

A.2) VMS

VMS is more tricky because of different physical disks (unlike Unix where all filesystems are under `/`).

Anyhow we should add the following translations (and *NOT translate* anything containing a [to allow usage of VMS notation, though **discouraged**):

```
 ~/dir1/dir2      into SYS$LOGIN:[dir1.dir2]
 ./dir1/dir       into [.dir1.dir2]
../dir1/dir2      into [-.dir1.dir2]
```

and perhaps allow VMS-like VOS notations like :

```
-/dir1/dir2 for [-.dir1.dir2] same as ../dir1/dir2
--/dir1/dir2      for [--.dir1.dir2] etc.
```

For the rest there is no problem in translating pathless filenames (just check name syntax and goes under current directory) or relative paths (ie. `dir1/dir2/dir3` is `[.dir1/dir2/dir3]`), but for absolute paths one can :

A) `/disk:/dir/dir` maps to `disk:[dir.dir]`
`/disk/dir/dir` maps to `[disk.dir.dir]` or `DUA0:[disk.dir.dir]`

B) `/disk/dir/dir` maps to `disk:[dir.dir]`

C) `/first/dir/dir` maps either as `first:[dir.dir]` or `[first.dir.dir]` ie `DUA0:[first.dir.dir]` according to a *site dependent* arrangement

Case A is the "*compulsory colon*" rule. Anything which is a disk, physical or logical, shall have a colon.

Case B is the "*compulsory disk*" rule. First argument is always a disk, logical or physical, irrespective whether it has a colon or not.

Case C could be arranged to map either way according to a site dependent set-up. I.e. sites with a single disk force all under `DUA0:` while other sites allow specification of disk under rules A or B.

Please comment on above possibilities.

It could also be verified whether a programmatic *SAFE* way exists to verify whether a given name is a disk (physical or logical) or a directory and in such case handle the first component `/first/dir/dir` either as `first:[dir.dir]` or `[first.dir.dir]` according to such internal test.

Also, should we consider also mapping Decnet names with a node specifier ? e.g. `IFCTR::DUA0:[PINCO.]` ? Or should we leave them to local definitions of logicals ?
At the moment Decnet is the only way to see Ultrix disks from VMS, but if UCX 2.0 allows NFS client this may change.

Appendix B) Multi-architecture support and sw distribution

WARNING: t*his appendix is just a first sketch on how s/w distribution can be made. It tries to list some technical possibilities and some implications for the organization of software directories, but it does not exhaust the matter. In particular it does not imply things have to be done this way. Additionally, policy related issues (e.g. will s/w distribution be completely free - e.g. anonymous ftp - or restricted to registered user, or somehow controlled) are willingly excluded here, but shall be discussed in the future.*

The scheme of software directory trees presented in section 1 above is the one which should be implemented on each target *standalone* machine running the software.

However the portable design of the software (whose goal is to have *ALL* main programs and library routines *IDENTICAL* on all machines with the exception of the VOS routines) could allow some saving in disk space and maintenance at a site supporting more machines of different architectures (e.g. a VMS one and an Unix one, or two different flavours of Unix).

This impacts on the organizations one has at a local site, on the installation procedure, and on the distribution of the software.

B.1 Possible organizations for multi-architecture support

First of all, to clear the way of misunderstandings, in the case one has more homogeneous systems (i.e. more VMS systems, or more Unix systems with the same operative system and platform) one uses normal system facilities (VMS VAXCluster or NFS disks in Unix) to install a single copy of the software and we are not concerned with this any longer.

If instead one has heterogeneous systems, one may note that there are parts of the software directory tree which are duplicated and identical under all architectures (the `source`, `libsource` and `include` files, `documentation` and hopefully `help` files, and possibly most of the `calib` files), while other are specific of a given system (the `vos` sources and the `bin` and `lib` binaries).

The simplest possibility is of course to have *separate, fully duplicated directory trees* on each machine. This could be assumed as *default* : it works, but is just wasteful in disk space.

Another possibility is to create identical, empty directory trees, then fill them partially with stuff retrieved from the archives, and use links for the identical parts. That is, designate one machine hosting the single physical copy of the system-independent files, and set up links to them on the other machines. This is greatly simplified if one can link entire directories (the `source`, `libsource` and possibly `doc` and `calib` directories could be handled this way).

Soft links are possible (and tested) between Unix machines and from Unix linking files residing on a VMS machine running an NFS server. The converse (from VMS accessing files residing on Unix) has not been tested and situation is unknown.

B.2 Distribution of multi-architecture software

This is a possible scheme for distribution of software under a multi-architecture scheme.

a) entire separate trees for VMS and DEC Ultrix are kept at SDC/SAC for the use of Guest Observers. They also constitute the master frozen version. For these architectures distribution can occur in source or binary (executable) form.

b) for other architectures for which a version may exist (e.g. Sun, HP-UX) the distribution will occur only in source form. Executables could or could not be stored at SDC according to convenience. These architectures will **not** be officially supported.

c) It is TBD whether building the archive for distribution will occur by SDC/MSC accessing the SDC/SAC master version directly, or storing a master copy on SDC/MSC.

d) Two kind of distributions should be considered : **entire releases** (or new installations) and **updates**.

e) For **entire releases** the following kits should be created at SDC/MSC.

1) a Unix compressed `tar` file of the `bin`, `doc` and `calib` directories (for DEC Ultrix)
2) eventually a Unix compressed `tar` file of the same for other TBD Unix architectures (not mandatory)
3) a VMS compressed `BACKUP` saveset of the `bin`, `doc` and `calib` directories
4) a Unix compressed `tar` file of the `lib` directory (DEC/Ultrix)
5) idem for other TBD Unix architectures (not mandatory)
6) a VMS compressed `BACKUP` saveset of the `lib` directory
7) a Unix compressed `tar` file of the `source` and `libsource` directories (this will do for all Unixes)
8) some Unix compressed `tar` files of the `vos` directory (one for each officially or unofficially supported architecture)
9) a VMS compressed `BACKUP` saveset of the `source`, `libsource` and `vos` directories (for analogy with the above one could prefer to have `vos` in a separate saveset)

An user *not wanting or needing to recompile* could retrieve **only the essential kit** (1,2 or 3). With this he should be able to run the software.
An user wanting to be able to interface programs of his own should *also* retrieve the relocatable library kit (4,5, or 6).
An user wanting to recompile everything, or wanting to make modifications, or just curious to look at the source code, or needing to recompile everything on an unofficially supported architecture should retrieve *also* the other kits (7 AND the correct version of 8 for Unix, or just 9 for VMS)

The mention of `tar` files or `BACKUP` savesets above does not imply they are stored in a single file. It is possible that for convenience of the network transfer, they are split in chunks to be reconstituted at the receiving end. It is also possible that some kits are kept in compressed *and* uncompressed form for convenience of some sites.

f) for **updates** (this applies to updates of the *programs as well as* to updates of the *calibrations*) it is desirable that one could retrieve only the files changed, without doing a complete reinstallation. Of course this assumes one has done previously a *FULL* installation, and keeps the standard directory tree structure *UNCHANGED*.

For this it may look desirable that a master copy of the entire sw trees not in tar or BACKUP form is also residing at SDC/MSC.

The user will then retrieve the *necessary* files as instructed by *notifications circulated* to the community :

1) some new or changed calibration files
2) some new program executables (if he did a binary-only installation)
3) some new libraries (if he did an executable+library installation)
4) some new subroutine and main sources, plus the instruction to recompile and relink in any other case

g) It is noted that in case one wants to recompile and relink (either during full installation or updates) he needs to retrieve *only ONE* copy of the source kits or files for *ONE* of his target systems, and then he can *duplicate or link this locally*. It is to be evaluated whether this should be allowed at all, or only to experienced users (at their own risk) or whether utilities to do this will be supplied.

B.3 Installation related issues

A **new installation with a single architecture** involves the following steps if one is just installing the essential system :

retrieve from SDC/MSC the essential (binary+calibration) kit

If one instead wants to do a full recompilation the steps are :

retrieve from SDC/MSC the essential kit (for calibration etc.)
retrieve the source kit
retrieve the vos kit for the target system
issue a `make all` or run a provided installation procedure (this could look like to :

```
sourcedir source
libdir lib
incdir include
libsourcedir root
complib vos all
libsourcedir libsource
complib library1 all
....
complib library2 all
comlink program1 option1
....
comlink programn optionq
```

A **new installation for multiple architecture**, if installing the essential system involves :

retrieve the essential kit for each system
delete the stuff duplicated (`calib`, `doc` etc.) from all systems but one
create links for the duplicated stuff to the one system left with it

One might argue from this that is better to supply binaries separately from `doc` and `calib` files. This could be considered.

A new installation for multiple architecture with recompilation and relink involves :

retrieve the essential kit for one system
retrieve the source and vos kits for the same system
do a `make all` or run the installation script
repeat what follows for each other system
  create empty directory tree
  create links for duplicated stuff to first system
  retrieve the system specific vos kit
  do a `make all` or run the installation script

Installations of updates (for single or multi-architecture sites) **only with executables** or calibration files consist just in retrieving the necessary files to the right place.

Installations of **updates with compilation for a single system** involves:

retrieving all changed library routines as instructed
retrieving all changed main sources as instructed
doing a `make` or running a script provided, which may look like

```
complib lib1 sub3
complib lib1 sub7
complib lib5 sub2
comlink main1 option3
comlink main4 option2
```

Installation of **updates with recompilation** in case of **multiple architecture** involves :

```
retrieving all changed routines and mains as above for one system
running make or the script provided
for each other system
        create necessary links if not already existing
        running make or the script provided
```

The latter case does not consider updates to the VOS routines (they should be fairly stable soon), however if they are updated, a separate copy should be retrieved for each system before running the script.