Inter-Office Memorandum

To	IFS Project	Date	December 1, 1981
From	Ed Taft	Location	Palo Alto
Subject	IFS Software Maintenance (version 1.35)	Organization	PARC/CSL

XEROX

Filed on: [Maxc2]<IFS>IFSSoftwareMaint.bravo

This is a brief description of how the IFS software is organized and maintained.

Organization

The pieces from which IFS is constructed fall into three major categories:

1. Components specific to IFS; all of these have file names that begin with Ifs', and are kept on the master IFS directory (presently [Maxc2]<IFS>).

2. Standard Alto software packages (including pieces of the Alto Operating System), obtained from the <Alto> and <AltoSource> directories. In general, IFS always uses the latest version of these.

3. Modified Alto software packages. In general, these are intended to be released as the standard packages at some later time. Packages that have been so extensively modified for IFS that they are no longer suitable for general release are renamed and moved into category 1.

An official' release of IFS consists of the following files, all kept on [Maxc2]<IFS>:

1. IFS.run, IFS.syms, and IFS.errors, the files needed to operate an IFS server.

2. A command file IfsDevDisk.cm that initializes a blank Alto disk with the minimum set of programs and other files required for IFS development.

3. A dump-format file IfsCm.dm that contains all the command files useful for IFS development.

4. Dump-format files containing all the IFS-specific sources, divided into functional groups. At present, these consist of the following:

IfsDecl.dm	All the Bcpl declaration (.decl) files. These are also contained in the appropriate dump files listed below; they are collected together here because may of them are required throughout the system, not just in one group.
IfsKernel.dm	Basic underlying mechanisms (virtual memory, overlays, storage allocation), plus system initialization.

IfsFileSys.dm	Directory and file access machinery.
IfsRsMgr.dm	Rendezvous socket and server management.
IfsFtp.dm	FTP server.
IfsMail.dm	Mail server and forwarder.
IfsCopyDisk.dm	CopyDisk server.
IfsLeaf.dm	Leaf server.
IfsMisc.dm	Miscellaneous servers (name, time, boot, and Press printing).
IfsTelnet.dm	Telnet server and command interpreters.
IfsBackup.dm	Backup system.
IfsMc.dm	Microcode.
IfsLeftovers.dm	Everything that doesn't seem to fit into one of the above categories.

5. A dump-format file IfsBrs.dm that contains all the Br files (both IFS code and standard packages) for this release.

The IFS maintenance procedures have been rather carefully organized so that you can do IFS development on a single Diablo disk, with considerable support from one or more file servers. The disk contains all necessary subsystems and all the Br files for loading an IFS, and has enough room (about 400 pages) for a fair number of source files being actively worked on. The idea is that you fetch source files from a file server, modify them, get the modified IFS working again, store the source files back on the file server, and delete them from your Alto disk.

Recreating IFS from sources

This procedure assumes you have access to a file server and directory containing a released version of IFS on a directory called <IFS>.

First, boot an OS from the network and use it to erase' a disk. In response to the question Do you want a big SysDir' you should answer y'.

Next, fetch <IFS>IfsDevDisk.cm from the file server and execute it. (The command file on Maxc2, naturally, assumes that all standard Alto software should be obtained from Maxc2. If you want to change this you should edit the file first.)

Toward the end of this command file, you are asked for the contents of an unknown command file FileServerForIfsSoftware', at which point you should type the name of the file server that has all the IFS dump files. IfsDisk.cm now loads IfsCm.dm and IfsDecl.cm. Finally, it calls the command files IfsPackages.cm, CompileIfs.cm, and LoadIfs.cm, described below.

The command file IfsPackages.cm fetches all the standard and modified software packages (categories 2 and 3). Again, the one on Maxc2 assumes you will obtain the standard software from Maxc2. This command file fetches a whole lot of stuff, deletes some of it, and recompiles several packages in non-standard ways. This takes about 30 minutes. It concatenates all the error files (*.bt) into one file called IfsPackages.errorFiles, for your later examination.

Next, CompileIfs.cm compiles all the IFS-specific software. This first asks you what file server the IFS dump files live on. The command file then loads each dump file in turn, compiles all the source files, and deletes them. This procedure takes about two hours. It concatenates all the error

files (*.bt) into one file called Ifs.errorFiles.

Finally, LoadIfs.cm creates IFS.run and IFS.syms from the .br files. This takes about 4 minutes. There are normally 5 errors (all multiply-defined symbols) which you should ignore. At the end, ListSyms is run to produce IFS.bz, which is a listing of the sizes of the overlays (all except AltoDirs and perhaps some of the initialization overlays should be 1024 or less words long).

You should inspect IfsPackages.errorFiles, Ifs.errorFiles, and Ifs.bs to make sure that no errors occured during compilation and loading, then delete *.errorFiles.

Other command files

Each of the dump files has associated with it a command file that enumerates its contents. That is, IfsKernel.cm contains a list of all the files in IfsKernel.dm. It should not have any CRs in it, even at the end (but of course ^CR' is ok). These command files are expanded in many contexts and control loading, dumping, printing, etc., of each group of files.

PrintIfs.cm is a command file that obtains all the source files, group by group, and prints them using Empress.

During active software development, it is inconvenient to have to keep loading and dumping dump-format files; it is more convenient to store all the source files separately, and retrieve, edit, and store them individually.

The command file ExpandIfs.cm loads all the dump files and puts the sources back out as individual files in <IFS>Sources>. You get to specify both source and destination file servers for this operation. (Note that the destination file server must be an IFS because Maxc2 doesn't have subdirectories. Again, you must previously have created a file whose name and contents are the server name, as described above.)

The command file DumpIfs.cm performs the inverse operation, retrieving all the source files from <IFS>Sources> and dumping them into dump-format files in <IFS>. It also stores Ifs.run, Ifs.syms, Ifs.errors, and IfsBrs.dm from your Alto disk, in preparation for an IFS release.

The command files IfsDevFromSources.cm and CompileIfsSources.cm perform the same functions as IfsDevDisk.cm and CompileIfs.cm, but they assume the IFS source files are stored individually on <IFS>Sources> rather than as dump files.

Command file maintenance

To enable all these mechanisms to work smoothly, it is important to keep the command files up-to-date. Specifically:

> When you add, delete, or rename a source file, you must change LoadIfs.cm, xxx.cm, and Compilexxx.cm, where xxx is the name of the group to which the file belongs.
>
> When you add or remove a package, you must change LoadIfs.cm and IfsPackages.cm.

IFS code organization

The IFS code is organized into three major parts: resident code, resident initialization, and overlays. The command file LoadIfs.cm controls the loading of code into these regions; you should examine this file in some detail, consulting the Bldr section of the Bcpl manual when necessary.

The resident code consists of all modules that must be resident throughout execution. Code is made resident for one of two reasons:

1. It is used to support the overlay, virtual memory, and lowest level disk I/O mechanisms, so it must itself be resident at all times.

2. It is so heavily used that to make it nonresident would introduce performance problems.

The resident initialization consists of code executed while IFS is starting up, but before the overlay and virtual memory machinery is working. This code is thrown away and the space it occupies turned into free storage after that.

The overlays contain the bulk of the code. Each overlay occupies an integral number of 1024-word Trident disk pages (normally just one), and should be packed as full as possible to minimize breakage. To first approximation, the overlays are managed by the Bcpl Overlays package, which swaps overlays from disk to memory whenever necessary. Documentation for the Overlays package may be found in the Alto Bcpl Software Packages manual.

IFS has an extended emulator' that is capable of executing Bcpl code directly from extended memory. That is, instruction fetches and literal references are made in extended memory, but data references are made to bank zero. The extended emulator has been carefully designed to permit most Bcpl code to be executed in extended memory without modification, but there are a few conventions that have to be followed in order for this to work.

If IFS is started up on an Alto that has extended memory, it loads as many overlays as will fit into the extended memory, starting in bank 1. The order of overlays in LoadIfs.cm should be most frequently executed first, so that on a 128K Alto (which does not have enough memory for all the overlays) the most frequently executed overlays will be in extended memory, and only the less frequently executed ones will swap from disk. The first 63 overlays are the ones that fit in memory in this case; currently they consist of all overlays up to and including Grapevine2'.

If there is enough memory left over after all overlays are loaded, most of the resident code is also moved into extended memory and the space reclaimed. In LoadIfs.cm, the resident code is subdivided into two sections: code which must remain in bank zero and code which may be moved into extended memory. In the latter section, the resident code for the Leaf server is at the end so that it may be thrown away (i.e., treated the same as resident initialization code) if the Leaf server is not to be enabled.

Most of the machinery for overlay management and extended emulation is completely automatic, but there are a few rules that must be followed in order for it to function correctly. These rules may be divided into two categories: the overlay rules' and the extended memory rules'.

Overlay rules

1. Procedures that are in overlays must ordinarily be called only via their statics. A procedure variable, argument, object, etc., must not have as its value a procedure that is in an overlay. (An exception to this rule applies if the value of such a procedure variable is another procedure in the same overlay as the caller, and was assigned during the lifetime of the caller.)

2. A procedure that must be used in a manner violating rule 1 may be declared to be an Overlay Entry Point (OEP), as discussed in the Overlays package documentation. In the resident initialization there are a number of procedures with names of the form Declare*xxx*OEP' which are called from the DeclareAllOEPs procedure (in IfsOvCopy.bcpl) and passed an Overlay Entry Vector (oev) as their argument. These procedures declare Overlay Entry Points by calling

   DeclareOEP(oev, lv Proc1, lv Proc2, ...)

   for each procedure (Proc1, etc.) that must be an OEP. The effect of this is to create, for each OEP, a dummy (two-word) permanently resident procedure that in turn passes off control to the real procedure, wherever it happens to be.

3. A procedure in an overlay may reference data in the same overlay, but a pointer so generated remains valid only until the procedure returns (unless explicitly locked, using the LockCell procedure in the VMem package). Since Bcpl string and table literals are implemented as pointers to code, this means that procedures cannot return string and table literals as values.

Extended memory rules

1. To be executable in extended memory, a procedure must not execute any S-group instructions except JSRII (see Alto Hardware Manual); must not execute any PC-relative JSRs except to construct string and table literals; must not make any absolute references to literals in the code (but PC-relative references are permitted); and must not fiddle with the Bcpl stack.

2. All Bcpl procedures (i.e., procedures actually generated by the Bcpl compiler) that are in overlays must conform to extended memory rule 1 as well as all the overlay rules.

3. Assembly-language procedures that are in overlays are assumed to conform to the overlay rules but not to the extended memory rules unless declared XM Entry Points (XEPs). If a non-XEP assembly-language procedure is called, the overlay containing that procedure is automatically swapped into bank zero before execution. To declare an assembly-language procedure to be an XEP, the appropriate Declare*xxx*OEP procedure should call

   DeclareXEP(oev, lv Proc1, lv Proc2, ...)

   This permits assembly-language procedures Proc1, Proc2, etc., to be executed directly in extended memory rather than swapped into bank zero when called. Note that it is OK for a procedure to be both an OEP and an XEP, if that is appropriate.

4. A string or table literal generated within an extended memory procedure is valid only until that procedure returns, and it cannot be locked in the manner described in overlay rule 3. (The extended emulator allocates string and table literals in the procedure's stack frame, which disappears when the procedure returns.)

5. Strings and tables greater than 127 words long are illegal.

The resident code in the region that is movable to extended memory (if there is enough of it) must conform to the extended memory rules, but need not conform to the overlay rules.

Debugging note

Attempting to debug code that resides in extended memory is likely to lead to insanity. Swat knows almost nothing about extended memory; the one exception is an extended stack trace (^T) feature, enabled by installing Swat with InstallSwat/x'. Unfortunately, when this feature is enabled, each entry into Swat takes an extra 10 to 15 seconds.

Therefore, for most serious debugging, it is recommended that IFS be started up with extended memory disabled (i.e., IFS/ x'). This results in rotten performance, but at least all the code is in bank zero where you can look at it.

If you want to set breakpoints, the problem may arise that the code you want to breakpoint is swapped out, or gets swapped out between the time you set the breakpoint and the time the code is executed. For debugging purposes, a mechanism exists for pinning up to ten selected overlays permanently into bank zero.

Before starting IFS, decide which overlays are of interest, and determine their overlay numbers by counting overlays in Ifs.bz or LoadIfs.cm (the first overlay is numbered 1). Start IFS in Swat by IFS/!'. Beginning at InitIFSPart1+5, you will find a table of ten zeroes. Replace one or more of these zeroes with overlay numbers. Then proceed from Swat, to start up IFS. The specified

overlays will be pinned in bank zero until IFS is next restarted, and Swat can deal with the code in those overlays in the normal manner.