UNIX/IRAF Site Manager's Guide

Doug Tody

Central Computer Services
National Optical Astronomy Observatories†
June 1989

ABSTRACT

An IRAF *site manager* is anyone who is responsible for installing and maintaining IRAF at a site. This document describes a variety of site management activities, including configuring the device and environment tables to provide reasonable defaults for the local site, adding interfaces for new devices, configuring and using IRAF networking, the installation and maintenance of layered software products (external packages), and configuring a custom site LOCAL package so that local software may be added to the system. Background information on multiple architecture support and the software management tools provided with the system is presented. The procedures for rebooting IRAF and performing a sysgen are described. The host system resources required to run IRAF are discussed.

April 12, 1990

†Operated by the Association of Universities for Research in Astronomy, Inc. under cooperative agreement with the National Science Foundation.

Contents

1. **Introduction** 1

2. **System Setup** 2
   - 2.1. Installing the System 2
   - 2.2. Configuring the Device and Environment Tables 2
     - 2.2.1. Environment definitions 2
     - 2.2.2. The template LOGIN.CL 3
     - 2.2.3. The DEVICES file 3
     - 2.2.4. The DEVICES.HLP file 3
     - 2.2.5. The TERMCAP file 4
     - 2.2.6. The GRAPHCAP file 4
     - 2.2.7. Configuring IRAF networking 4
     - 2.2.8. Configuring the IRAF account 6
     - 2.2.9. Configuring user accounts for IRAF 6
   - 2.3. Tuning Considerations 6
     - 2.3.1. Caching TERMCAP and GRAPHCAP device entries 7
     - 2.3.2. Stripping the system to reduce disk usage 7

3. **Software Management** 7
   - 3.1. Multiple architecture support 7
   - 3.2. Layered software support 9
   - 3.3. Software management tools 9
   - 3.4. Modifying and updating a package 10
   - 3.5. Installing and maintaining layered software 11
   - 3.6. Configuring a custom LOCAL package 12
   - 3.7. Updating the full IRAF system 13
     - 3.7.1. The BOOTSTRAP 13
     - 3.7.2. The SYSGEN 13

4. **Interfacing New Graphics Devices** 14
   - 4.1. Graphics terminals 14
   - 4.2. Graphics plotters 14
   - 4.3. Image display devices 15
   - 4.4. Network access to graphics devices 15

5. **Host System Requirements** 16
   - 5.1. Memory requirements 16
   - 5.2. Disk requirements 16
   - 5.3. Diskless nodes 16

Appendix A. The IRAF Directory Structure 17

UNIX/IRAF Site Manager's Guide

Doug Tody

Central Computer Services
National Optical Astronomy Observatories†
June 1989

1. Introduction

Once the IRAF system has been installed it will run, but there remain many things one might want to do to tailor the system to the local site. Examples of the kinds of customizations one might want to make are the following.

- Edit the default IRAF environment definitions to provide reasonable defaults for your site.
- Make entries in the device descriptor tables for the devices in use at your site.
- Code and install new device interfaces.
- Enable and configure IRAF networking, e.g., to permit remote image display, tape drive, or file access.
- Perform various optimizations, e.g., stripping the system to reduce disk usage, or caching terminal device entries to minimize runtime searches of the device tables.
- Extend the system by installing layered software products.
- Configure a custom LOCAL package so that locally developed software may be installed in the system.

This document provides sufficient background information and instructions to guide the IRAF site manager in performing such customizations. Additional help is available via the IRAF HOTLINE (`602 323-4160`), or by sending mail to `iraf@noao.edu` (internet) or `5355::iraf` (SPAN). Contributions of interfaces developed for new devices, or any other software of general interest, are always welcome.

The IRAF software is organized in a way which attempts to isolate, so far as possible, the files or directories which must be modified to tailor the system for the local site. Most or all changes should affect only files in the `local`, `dev`, and `hlib` (`unix/hlib`) directories. Layered software products, including locally added software, reside outside of the IRAF core system directory tree and are maintained independently of the core system.

To facilitate periodic core system updates it is suggested that the site manager keep a log of all changes to files in the core system. A complete record of all modifications made to the core system by NOAO to prepare the current release will be found in the **system notes file** in the IRAF directory `local`; this is the primary source of technical documentation for each release and should be consulted if questions arise regarding any of the system level features added in a new release of the core system. The system notes files for older releases will be found in the `doc` directory.

---

†Operated by the Association of Universities for Research in Astronomy, Inc. under cooperative agreement with the National Science Foundation.

2. System Setup

2.1. Installing the System

The procedure for installing or updating a UNIX/IRAF system is documented in the appropriate UNIX/IRAF Installation Guide (e.g., the *BSD-UNIX/IRAF Installation Guide* for BSD/IRAF). In short, an IRAF tape or network distribution is obtained from NOAO and installed according to the instructions given in the installation guide. The result is a complete IRAF system, including both sources and executable binaries for the architectures to be supported. The system will have been modified to reflect the new IRAF root directory and should run, but will otherwise be a generic IRAF distribution. To get the most out of an IRAF installation it will be necessary to perform some of the additional steps outlined in the remainder of this document.

2.2. Configuring the Device and Environment Tables

Teaching IRAF about the devices, network nodes, external programs, and other special resources available at a site is largely a matter of editing a standard set of device descriptor and environment setup files, all of which are simple text files. The versions of these files provided with the distribution are simply those in use on the NOAO system from which the tapes were made, at the time the tapes were generated. Hence while these files may be useful as examples of properly configured descriptor files, the defaults, and many specific device entries, may be meaningless for a different site. This is harmless but it may be confusing to the user if, for example, the default printer doesn't exist at your site.

The device and environment files also contain much material which any site will need, however, so care must be taken when editing the files. Important changes may be made to the global portions of these files as part of any IRAF release. To facilitate future updates, it is wise where possible to isolate any local changes or additions so that they may be copied into the new (distributed) version of the file in a future update.

2.2.1. Environment definitions

Since IRAF is a machine and operating system independent, distributed system it has its own environment facility apart from that of the host system. Host system environment variables may be accessed as if they are part of the IRAF environment (which is sometimes useful but which can also be dangerous), but if the same variable is defined in the IRAF environment it is the IRAF variable which will be used. The IRAF environment definitions, as defined at CL startup time, are defined in a number of files in the `hlib` directory. Chief among these is the `zzsetenv.def` file. Additional user modifiable definitions may be given in the template `login.cl` file (see §2.2.2).

The `zzsetenv.def` file contains a number of environment definitions. Many of these define IRAF logical directories and should be left alone. Only those definitions in the header area of the file should need to be edited to customize the file for a site. It is the default editor, default device, etc. definitions in this file which are most likely to require modification for a site.

If the name of a default device is modified, the named device must also have an entry in the `termcap` file (terminals and printers) or the `graphcap` file (graphics terminals and image displays) in `dev`. There must also be an *editor*`.ed` file in `dev` for the default editor; EDT, EMACS, and VI are examples of currently supported editors.

Sample values of those variables most likely to require modification for a site are shown below.

```
set editor      = "vi"
set printer     = "versatec"
set stdimage    = "imt512"
set stdplot     = "lw"
```

For example, you may wish to change the default editor to "emacs", the default printer to "lw", or the default image display to "iism75". Note that the values of terminal and stdgraph, which also appear in the zzsetenv.def file, have little meaning except for debugging processes run standalone, as the values of the environment variables are reset automatically by stty at login time. The issues of interfacing new graphics and image display devices are discussed further in §4.

2.2.2. The template LOGIN.CL

The template login.cl, hlib$login.cl in IRAF filename nomenclature, is the file used by mkiraf to produce the user login.cl file. The user login.cl file, after having possibly been edited by the user, is read by the CL every time a new CL is started, with the CL processing all environment and task definitions, package loads, etc., in the login file. Hence this file plays an important role in establishing the IRAF environment seen by the user.

Examples of things one might want to change in the template login.cl are the commented out environment definitions, the commented out CL parameter assignments, the foreign task definitions making up the default user package, and the list of packages to be loaded at startup time. For example, if there are host tasks or local packages which should be part of the default IRAF operating environment at your site, the template login.cl is the place to make the necessary changes.

2.2.3. The DEVICES file

This file dev$devices should contain an entry for all allocatable devices (primarily tape drives) for the local system. The comments in the file, plus the sample entries shown, should suffice to document what should be entered. The logical names of magtape devices must begin with "mt"; IRAF magtape devices are typically named "mta", "mtb", and so on.

Allocation of a device in UNIX/IRAF involves modifying the ownership and access modes of the special files for the device in /dev. Since most UNIX systems do not provide any host level device allocation facilities the device files are manipulated directly by IRAF via a special program, alloc.e, which is given suid-root permissions when the IRAF install script is run.

When a magtape device is allocated by a user, the user owns the device and has exclusive access to the device. Failure to deallocate a device, e.g., because the user kills the CL rather than logging out (as may happen when a network login connection is broken) can leave the device still allocated to the user, rendering it inaccessible by other users unless they log into IRAF to reallocate the device. A host level program allocate, currently not part of the standard UNIX/IRAF distribution, is available if it is desired to be able to control device allocation at the host level without having to log into IRAF. The IRAF device allocation facilities will work properly even when a tape drive is remotely accessed over the network.

2.2.4. The DEVICES.HLP file

All physical devices that the user might need to access by name should be documented in the file dev$devices.hlp. Typing

```
cl> help devices
```

or just

```
cl> devices
```

in the CL will format and output the contents of this file. Note that it is the IRAF name of the device, as given in files such as `dev$devices`, `termcap`, etc., which should appear in this help file.

2.2.5. The TERMCAP file

There must be entries in this file for all local terminal and printer devices you wish to access from IRAF (there is currently no `printcap` file in IRAF). The entry for a printer contains one special (nonstandard termcap) entry, called DD. This consists of three fields: node!device, the template for the temporary spoolfile, and the UNIX command to be used to dispose of the file to the printer. On Berkeley UNIX derived systems it is rarely necessary to make use of the node name capability, since `lpr` already provides this capability.

If you have a new terminal which has no entry in the IRAF termcap file, you may already have an entry in the host system termcap file. Simply copy it into the IRAF file; both systems use the same basic termcap file format. If the terminal in question is a graphics terminal a corresponding entry is required in the `dev$graphcap` file, and you must add a '`:gd`' capability to the termcap entry to signify that the terminal supports vector graphics. If the graphcap entry has a different name than the termcap entry, the format '`:gd=`*gname*' should be used instead.

2.2.6. The GRAPHCAP file

There must be entries in this file for all graphics terminals, batch plotters, and image displays accessed by IRAF programs. New graphics terminals will need a new entry. The IRAF file `sys$gio/doc/gio.hlp` contains documentation describing how to prepare graphcap device entries. A printed copy of this document is available upon request, however once IRAF is up you may find it easier to generate your own copy using `help`, as follows:

```
cl> cd sys$gio/doc
cl> help gio.hlp fi+ | lprint
```

which will print the document on the default IRAF `printer` device.

The HELP pages for the IRAF **showcap** and **stty** tasks should also be printed as these utilities are useful for generating new graphcap entries. Help preparing new graphcap device entries is available if needed. We ask that new graphcap entries be sent back to us so that we may include them in the master graphcap file for all to benefit.

2.2.7. Configuring IRAF networking

The `dev` directory contains several files (`hosts`, `hostlogin`, and `uhosts`) used by the IRAF network interface. The IRAF networking capabilities are used to access image displays, printers, magtape devices, files, etc. resident on any node in a local area network. Nodes do not necessarily have to have the same architecture, or even run the same operating system, so long as they can run IRAF.

To enable IRAF networking for a UNIX/IRAF system, all that is necessary is to edit the file `dev$hosts` (the `uhosts` file is not used with UNIX/IRAF). Make an entry for each logical node in the local network in the format

nodename [*aliases*] ":" *irafks.e-pathname*

Note that there may be multiple logical entries for a single physical node. There must be an entry in the host system file `/etc/hosts` for each physical node referenced in the IRAF table.

To enable IRAF networking for a node, the BSD/UNIX **hostname** must appear as a primary name or alias somewhere in the IRAF host table. During process startup, the IRAF VOS looks for the system name for the current host and automatically disables networking if this name is not found. Hence IRAF networking is automatically disabled when the distributed

system is first installed - unless you are unlucky enough to have installed the system on a host with the same name as one of the nodes in the NOAO host table.

Once IRAF networking is configured, the following command may be typed in the CL to verify that all is well:

```
cl> netstatus
```

This will print the host table and state the name of the local host. If there are any problems a message will be printed describing the problem.

For IRAF networking to be of any use, it is necessary that IRAF be installed on at least two systems. In that case either system can serve as the server for an IRAF client (IRAF program) running on the other node. Note that it is not necessary to install IRAF separately on each node, i.e., a single copy of IRAF may be NFS mounted on all nodes. If it is not possible to install IRAF on a node for some reason (either directly or using NFS) it is possible to manage by installing only enough of IRAF to run the IRAF kernel server. Contact IRAF site support if you need to configure things in this manner.

UNIX/IRAF currently supports only TCP/IP networking using the BSD interface. Networking between any heterogeneous collection of systems is possible provided they support TCP/IP based networking. If the local network includes any VMS systems, a VMS client can access a Sun server, but the reverse is not possible as at present we do not support a TCP/IP based IRAF kernel server for VMS.

Once IRAF networking is enabled, objects resident on the server node may be accessed from within IRAF merely by specifying the node name in the object name, with a "*node!*" prefix, e.g., if `foo` is a network node,

```
cl> page foo!hlib$motd
cl> allocate foo!mta
cl> devstatus foo!mta
```

The user will be prompted for their password on the remote node *each time an IRAF client process connects to the node*. This can be defeated if desired by creating a `.irafhosts` file in the user's UNIX home directory containing their login name and password for each node they may wish to access with IRAF (see `dev$hostlogin` for information on the file syntax). Obviously, there may be security problems associated with placing a password in a file in this manner, and the file should be read protected if used. A possible alternative is to set up a public account on the server node, possibly with restricted permissions. It is also possible to use a combination of both public and private accounts by making multiple logical entries for the server in the IRAF host table.

To keep track of where files are in a distributed file system, IRAF uses **network pathnames**. A network pathname is a name such as `foo!/tmp3/images/m51.pix`, i.e., a host or IRAF filename with the node name prepended. This feature can confuse the user when combined with the use of NFS to cross mount disks at the host level, due to the use of a network pathname for the pixel file pathname in an IRAF image. There are various ways the problem can be avoided, e.g., [1] use only IRAF networking for image access, [2] set `imdir` so that pixel files are located relative to the header file directory, rather than by absolute network pathname, or [3] use only NFS for image access, e.g., configure the host table so that all client hostnames appear to be aliases for the server (effectively disabling IRAF networking in the default case), using secondary logical entries to enable networking for things like remote image display and magtape accesses. At present there is no ideal solution to this problem but any of the above approaches will work.

Even on a workstation where NFS is routinely available, IRAF networking can be an interesting alternative to NFS for file accesses, especially where the files being accessed are IRAF images. The problem is that accessing bulk image data over the network via NFS can place a very heavy load on the server. While IRAF networking may not be able to match the

i/o bandwidth of NFS, it does limit the load on the server and hence may improve the system throughput and response time. This is especially important when the server in question is not a dedicated file server but rather a general purpose compute and file server, which users may log into directly, or which is also used for large batch jobs due to its great disk, memory, and number crunching capabilities. In such a case doing heavy image processing via NFS will drive the load on the server way up, aggravating any users trying to work directly on the server. A combined strategy of running large jobs directly on the server, plus using IRAF networking for IRAF based interactive image processing on the clients, can avoid this problem.

2.2.8. Configuring the IRAF account

The IRAF account, i.e., what one gets when one logs into UNIX as "iraf", is the account used by the IRAF site manager to work on the IRAF system. Anyone who uses this account is in effect a site manager, since they have permission to modify, delete, or rebuild any part of IRAF. For these and other reasons (e.g., concurrency problems) it is recommended that all routine use of IRAF be performed from other accounts.

If the system has been installed according to the instructions given in the installation guide the login directory for the IRAF account will be `iraf$local`. This directory contains a `.login` file which defines the host environment seen by the IRAF account. The `.login` may be customized to reflect one's personal preferences if desired, but care must be taken to avoid deleting environment definitions or other statements which are essential to the correct operation of IRAF.

Before starting up IRAF from the iraf account, following the installation of a new system, it will be necessary to execute the `mkiraf` task. This will initialize (optionally) the `uparm` directory for the iraf account, and create a new `login.cl` file. It may also be necessary to edit the `.login` file to modify the way the environment variable `IRAFARCH` is defined. This variable, required for software development if the system is configured for multiple architecture support, but optional for merely running IRAF, must be set to the name of the desired machine architecture, e.g., `vax`, `mips`, `sparc`, etc. If it is set to the name of an architecture for which there are no binaries, e.g., `generic`, the CL may not run. The alias `setarch`, defined in the iraf account `.login`, is convenient for setting the desired architecture for IRAF execution and software development.

2.2.9. Configuring user accounts for IRAF

User accounts should be loosely modeled after the IRAF account. All that is *required* for a user to run IRAF is that they run `mkiraf` in their desired IRAF login directory before starting up the CL. Defining `IRAFARCH` in the user environment is not required unless the system has been configured for more than one architecture and the user will be doing any IRAF based software development (including IMFORT).

2.3. Tuning Considerations

There are two things that are commonly done to tune UNIX/IRAF for a particular host system:

- Precompile selected `termcap` and `graphcap` entries.
- Strip the system to reduce disk consumption.

The most important optimization is precompilation of the termcap and graphcap entries of the devices most commonly used at the local site, particularly when running IRAF on a slow machine. Stripping the system is undesirable if the system is to be used for software development, but is normally desirable when installing a production version of IRAF on a small system with limited disk space.

2.3.1. Caching TERMCAP and GRAPHCAP device entries

Precompilation of a termcap or graphcap entry is a technique used to speed runtime access of the entry for that device. If the entry is not precompiled the termcap or graphcap file must be physically opened and scanned at runtime to read the desired entry. This causes a noticeable delay of as much as a second when clearing the terminal screen or plotting a graph, hence it is usually worthwhile to cache the entries for commonly used video and graphics terminals. It is not worthwhile for printers, plotters, and image displays.

The system comes with selected termcap and graphcap entries already precompiled. To see which devices are precompiled, page the cache data files, `dev$cachet.dat` (for termcap) and `dev$cacheg.dat` (for graphcap). To cache a different set of entries one must regenerate these files with the **mkttydata** task in the `softools` package, and then do a full sysgen-relink with the `mkpkg` utility. Detailed instructions are given in the manual page for `mkttydata`.

Note that if you wish to relink the system to cache selected termcap or graphcap device entries and plan to strip the system as well to save disk space, *you must cache the termcap and graphcap entries before stripping the system*, because once the system is stripped you cannot update the system libraries.

2.3.2. Stripping the system to reduce disk usage

Depending upon the local system configuration, it may be necessary or desirable to strip the system of all non-runtime files to save disk space. This means deleting all the sources and all the reference manuals and other documentation, excluding the online manual pages. A special utility called **rmfiles** (in the `softools` package) is provided for this purpose. It is not necessary to run `rmfiles` directly to strip the system. The preferred technique is to enter the commands given below. The example is for the cshell for consistency with the rest of this document, but this could be done from within the CL as well.

```
% cd $iraf
% mkpkg strip
```

This will preserve all runtime files, permitting use of the standard system as well as user software development. Note that only the IRAF core system is stripped, i.e., if you want to strip any external layered software products, such as the NOAO package, a `mkpkg strip` must be executed separately for each - `cd` to the root directory of the external package first. A tape backup of a system should always be made before the system is stripped; keep the backup indefinitely as it may be necessary to restore the sources in order to, e.g., install a bug fix or add-on software product.

3. Software Management

3.1. Multiple architecture support

It is becoming increasingly common for the computing facilities at a site to consist of a heterogeneous network of workstations and servers. These machines will often have different architectures, or will be configured with different hardware, e.g., different floating point accelerators, or single and multiple cpus.

Since IRAF is a large system it is undesirable to have to maintain a separate copy of IRAF for each machine architecture or hardware configuration on a network. For this reason IRAF provides support for multiple architectures within a single copy of IRAF. To be accessible by multiple network clients, this central IRAF system will typically be NFS mounted on each client.

Multiple architecture support is implemented by segregating the IRAF sources and binaries into different directory trees. The IRAF sources are architecture independent and hence sharable

by machines of any architecture. All of the architecture dependence is concentrated into the binaries, which are collected together into the so-called BIN directories, one for each architecture. The BIN directory contains all the object files, object libraries, executables, and shared library images for an architecture, supporting both IRAF execution and software development for that architecture. A given system can support any number of BIN directories, and therefore any number of architectures.

In the case of the core IRAF system, there must be a separate copy of IRAF for each *operating system*, since each operating system will in general require a different HSI (IRAF host system interface). The HSI for a given operating system can support any number of different architectures if properly configured, but a separate copy of the HSI will be required for each operating system. On the other hand, layered packages such as NOAO or LOCAL are operating system independent, hence a single copy can include BINs for any combination of operating systems or architectures.

In IRAF terminology, when we refer to an **architecture** what we really mean is a type of BIN. The correspondence between BINs and hardware architectures is not necessarily one-to-one, i.e., multiple BINs can exist for a single compiler architecture by compiling the system with different compilation flags, as different versions of the software, and so on.

Examples of some currently supported UNIX operating systems and architectures are shown below.

Architecture	*Operating System*	*Description*
vax	VAX BSD/UNIX	Standard Berkeley UNIX
vax	VAX Ultrix	Similar to VAX/BSD UNIX
mips	MIPS Ultrix	MIPS dependencies (RISC architecture)
ieee	Convex	Convex with IEEE floating point
native	Convex	Convex with native floating point
generic	any	no binaries
pg	any	compiled with -pg option for profiling

Most of these correspond to different operating systems or hardware architectures. The exceptions are the `generic` architecture, which is what the distributed system is configured to by default (to avoid having any architecture dependent binary files mingled with the sources), and the `pg` architecture, which is not normally distributed to user sites, but which is a good example of a custom software architecture used for software development.

When running IRAF on a system configured for multiple architectures, selection of the BIN (architecture) to be used is controlled by the UNIX environment variable `IRAFARCH`, e.g.,

```
% setenv IRAFARCH ieee
```

would cause Convex/IRAF to run the IRAF binaries compiled to use the IEEE floating point format, corresponding to the BIN directory `bin.ieee`. Once inside the CL one can check the current architecture by entering one of the following commands (the output in each case is shown as well).

```
cl> show IRAFARCH
ieee
```

or

```
cl> show arch
.ieee
```

If `IRAFARCH` is undefined at CL startup time a default architecture will be selected based on the current machine architecture, the available floating point hardware, and the available BINs.

The IRAFARCH variable controls not only the architecture of the executables used to run IRAF, but the libraries used to link IRAF programs, when doing software development from within the IRAF or host environment.

Additional information on multiple architecture support is provided in the system notes file for V2.8, file notes.v28 in the doc directory.

3.2. Layered software support

An IRAF installation consists of the **core system** and any number of **external packages**, or **layered software products**. As the name suggests, layered software products are layered upon the core IRAF system. Layered software depends upon the core system for all of its functionality and is portable to any computer which already runs IRAF. Any number of layered products can be combined with the core system to produce the IRAF system used at a specific site. Due to disk space limitations it is likely that a given site will not wish to have all the available layered software products installed and on line at any one time.

The support provided for layered software is essentially the same as that provided for maintaining the core system itself. Each "external package" (usually this refers to a tree of packages) is a system in itself, similar in structure to the core IRAF system. Hence, there is a LIB, one or more BINs, a HELP database, and all the sources and runtime files. A good example of an external package is the NOAO package. Except for the fact that NOAO happens to be rooted in the iraf directory, NOAO is equivalent to any other layered product, e.g., STSDAS, PROS, CTIOLOCAL, KPNOLOCAL, etc. Other layered products should be rooted somewhere outside IRAF to simplify updates.

3.3. Software management tools

IRAF software management is performed with a standard set of tools, consisting of the tasks in the SOFTOOLS package, plus the host system editors and debuggers. Some of the most important and often used tools for IRAF software development and software maintenance are the following.

mkhelpdb	Updates the HELP database of the core IRAF system or an external package. The core system, and each external package, has its own help database. The help database is the machine independent file helpdb.mip in the package library (LIB directory). The help database file is generated with mkhelpdb by compiling the root.hd file in the same directory.
mkpkg	The "make-package" utility. Used to make or update package trees. Will update the contents of the current directory tree. When run at the root iraf directory, updates the full IRAF system; when run at the root directory of an external package, updates the external package. Note that updating the core IRAF system does not update any external packages (including NOAO). When updating an external package, the package name must be specified, e.g., "mkpkg -p noao".
rmbin	Descends a directory tree or trees, finding and optionally listing or deleting all binary files therein. This is used, for example, to strip the binaries from a directory tree to leave only sources, to force mkpkg to do a full recompile of a package, or to locate all the binaries files for some reason. IRAF has its own notion of what a binary file is. By default, files with the "known" file extensions (.[aoe], .[xfh] etc.) are classified as binary or text (machine

	independent) files immediately, while a heuristic involving examination of the file data is used to classify other files. Alternatively, a list of file extensions to be searched for may optionally be given.
`rtar,wtar`	These are the portable IRAF tarfile writer (`wtar`) and reader (`rtar`) programs. About the only reasons to use these with UNIX/IRAF are if one wants to move only the machine independent or source files (`wtar`, like `rmbin`, can discriminate between machine generated and machine independent files), or if one is importing files written to a tarfile on a system such as VMS/IRAF, where the files are blank padded and the trailing blanks need to be stripped with `rtar`.
`xc`	The X (SPP) compiler. This is analogous to the UNIX `cc` except that it can compile ".x" or SPP source files, knows how to link with the IRAF system libraries, knows how to read and use the environment of external packages, and so on.

The SOFTOOLS package contains other tasks of interest, e.g., a program `mktags` for making a tags file from SPP source files for the `vi` editor, a HELP database examine tool, and other tasks. Further information on these tasks is available in the online HELP pages.

3.4. Modifying and updating a package

IRAF applications development (including revisions to existing programs) is most conveniently performed from within the IRAF environment, since testing must be done from within the environment. The usual development cycle is as follows. This takes place within the *package directory* containing all the sources and mkpkg-files for the package.

- Edit one or more source files.
- Use `mkpkg` to compile any modified files, or files which include a modified file, and relink the package executable.
- Test the new executable.

The `mkpkg` file for a package can be written to do anything, but by convention the following commands are usually provided.

`mkpkg`	Same as `mkpkg relink` below.
`mkpkg libpkg.a`	Updates the package library, compiling any files which have been modified or which reference include files which have been modified. Private package libraries are intentionally given the generic name `libpkg.a` to symbolize that they are private to the package.
`mkpkg relink`	Rebuilds the package executable, i.e., updates the package library and relinks the package executable. By convention, this is the file `xx_foo.e` in the package directory, where *foo* is the package name.
`mkpkg install`	Installs the package executable, i.e., renames the `xx_foo.e` file to `x_foo.e` in the global BIN directory for the system to which the package *foo* belongs.
`mkpkg update`	Does everything, i.e., a *relink* followed by an *install*.

If one wishes to test the new program before installing it one should do a *relink* (i.e.,

merely type "mkpkg" since that defaults to relink), then run the host system debugger on the resultant executable. The process is debugged standalone, running the task by typing its name into the standalone process interpreter. The CL task `dparam` is useful for dumping a task's parameters to a text file to avoid having to answer innumerable parameter queries during process execution. If the new program is to be tested under the CL before installation, a *task* statement can be interactively typed into the CL to cause the CL to run the "xx_" version of the package executable, rather than old installed version.

When updating a package other than in the core system, the **-p** flag, or the equivalent `PKGENV` environment variable, *must* be used to indicate the system or layered product being updated. For example, "`mkpkg -p noao update`" would be used to update one of the packages forming the NOAO system of packages.

The CL **process cache** can complicate debugging and testing if one forgets that it is there. Recall that when a task is run under the CL, the executing process remains idle in the CL process cache following task termination. If a new executable is installed while the old one is still in the process cache, the `flprcache` command must be entered to force the CL to run the new executable. If an executable is currently running, either in the process cache or because some other user is using the program, it may not be possible to set breakpoints under the debugger.

A full description of these techniques is beyond the scope of this manual, but one need not be an expert at IRAF software development techniques to perform simple updates. Most simple revisions, e.g., bug fixes or updates, can be made by merely editing or replacing the affected files and typing

```
cl> mkpkg update
```

plus maybe a `flpr` if the old executable is still sitting idle in the process cache.

3.5. Installing and maintaining layered software

The procedures for installing layered software products are similar to those used to install the core IRAF system, or update a package. Layered software may be distributed in source only form, or with binaries; it may be configured for a single architecture, or may be preconfigured to support multiple architectures. The exact procedures to be followed to install a layered product will in general be product dependent, and should be documented in the installation guide for the product.

In brief, the procedure to be followed should resemble the following:

- Create the root directory for the new software, somewhere outside the IRAF directories.
- Restore the files to disk from a tape or network archive distribution file.
- Edit the core system file `hlib$extern.pkg` to "install" the new package in IRAF. This file is the sole link between the IRAF core system and the external package.
- Configure the package BIN directory or directories, either by restoring the BIN to disk from an archive file, or by recompiling and relinking the package with `mkpkg`.

As always, there are some little things to watch out for. When using `mkpkg` on a layered product, you must give the name of the system being operated upon, e.g.,

```
cl> mkpkg -p foo update
```

where *foo* is the system or package name, e.g., "noao", "local", etc. The **-p** flag can be omitted by defining `PKGENV` in your UNIX environment, but this only works for updates to a single package.

An external system of packages may be configured for multiple architecture support by repeating what was done for the core system (except that multiple operating systems can be supported in layered software if desired). One sets up several BIN directories, one for each architecture, named `bin.`*arch*, where *arch* is "vax", "vux3", "mips", "ieee", "sparc", etc. These directories, or symbolic links to the actual directories, go into the root directory of the external system. A symbolic link `bin` pointing to an empty directory `bin.generic`, and the directory itself, are added to the system's root directory. The system is then stripped of its binaries with `rmbin`, if it is not already a source only system. Examine the file `zzsetenv.def` in the LIB to verify that the definition for the system BIN (which may be called anything) includes the string "`(arch)`", e.g.,

```
set noaobin = "noao$bin(arch)/"
```

The binaries for each architecture may then be generated by configuring the system for the desired architecture and running `mkpkg` to update the binaries, for example,

```
cl> cd foo
cl> mkpkg sparc
cl> mkpkg -p foo >& spool &
```

where *foo* is the name of the system being updated. If any questions arise, examination of a working example of a system configured for multiple architecture support (e.g., the NOAO packages) may reveal the answers.

Once installed and configured, a layered product may be deinstalled merely by archiving the package directory tree, deleting the files, and commenting out the affected lines of `hlib$extern.pkg`. With the BINs already configured reinstallation is a simple matter of restoring the files to disk and editing the `extern.pkg` file.

3.6. Configuring a custom LOCAL package

Anyone who uses IRAF enough will eventually want to add their own software to the system, by copying and modifying the distributed versions of programs, by obtaining and installing isolated programs written elsewhere, or by writing new programs of their own. A single user can do this by developing software for their own personal use, defining the necessary *task* statements etc. to run the software in their personal `login.cl` file. To go one step further and install the new software in IRAF so that it can be used by everyone at a site, one must configure a **custom local package**.

The procedures for configuring and maintaining a custom LOCAL package are similar to those outlined in §3.5 for installing and maintaining layered software, since a custom LOCAL will in fact be a layered software product, possibly even something one might want to export to another site (although custom LOCALs often contain non-portable or site specific software).

To set up a custom LOCAL, start by making a local copy of the **template local** package that comes with the distributed system. If you make a source only tar archive of `iraf$local` and install it as outlined in §3.5, you will have a custom LOCAL. The purpose of the template LOCAL is to provide the framework necessary for a external package; a couple of simple tasks are provided in the template LOCAL to serve as examples. Once you have configured a local copy of the template LOCAL and gotten it to compile and link, it should be a simple matter to add new tasks to the existing framework.

3.7. Updating the full IRAF system

3.7.1. The BOOTSTRAP

All current IRAF distributions come with the system already bootstrapped, i.e., the host system interface (HSI) comes with the HSI binaries already built. A bootstrap should not be necessary unless one is doing something unusual, e.g., installing a bugfix or making some other revision to the HSI.

A bootstrap is like a full system sysgen, except that it is the host system interface (kernel and bootstrap utilities) which are compiled and linked, rather than IRAF itself. The system must be bootstrapped before a sysgen is possible, since the bootstrap utilities are required to do a sysgen. The two operations are distinct because only the bootstrap is machine dependent; everything else works the same on all IRAF systems.

To bootstrap UNIX/IRAF, go to the `unix` directory and enter the commands shown below. This takes 1/2 to 2 hours depending on the machine, so the output should be spooled in a file. One should only attempt a bootstrap from the IRAF account, to ensure that the environment is configured properly and that all the necessary permissions are in order.

```
% cd $iraf/unix
% reboot >& spool &
```

There are actually two types of bootstrap, the "hard" bootstrap starting from a source only system, called the NOVOS bootstrap, and the usual or VOS bootstrap, performed once the IRAF system libraries `libsys.a` and `libvops.a` exist. To configure IRAF starting from a completely source only system, one would have to do a NOVOS bootstrap, make the system libraries, then do a VOS bootstrap, after which one could perform a full sysgen to build the main IRAF system.

3.7.2. The SYSGEN

A full system sysgen is necessary when installing a source only version of IRAF, e.g., when an IRAF distribution is obtained from the IRAF network archive. We assume that the system has already been bootstrapped, since most IRAF distributions include a completely configured HSI.

To do a full sysgen of IRAF one merely runs `mkpkg` at the IRAF root. If the system is configured for multiple architecture support one must repeat the sysgen for each architecture. Since a full sysgen takes a long time and generates a lot of output which later has to be reviewed, it is best to run the job in batch mode with the output redirected. For example to update the `sparc` architecture binaries:

```
% cd $iraf
% mkpkg sparc
% mkpkg >& spool &
```

To watch what is going on after this command has been submitted and while it is running, try

```
% tail -f spool
```

Sysgens are restartable, so if the sysgen aborts for any reason, simply fix the problem and start it up again. How long the sysgen takes depends upon how much work it has to do. The worst case is if the system and applications libraries have to be recompiled. If the system libraries already exist they will merely be updated.

A full sysgen generates a lot of output, too much to be safely reviewed for errors by simply paging the spool file. Enter the following command to review the output (this assumes that the output has been saved in a file named `spool`).

```
% mkpkg summary
```

It is normal for a number of compiler messages warning about assigning character data to an integer variable to appear in the spooled output if the full system has been compiled. There should be no serious error messages if a supported and tested system is being recompiled.

The above procedure only updates the core IRAF system. To update a layered product one must repeat the sysgen process for the layered system. For example, to update the `vax` binaries for the NOAO package:

```
% cd $iraf/noao
% mkpkg vax
% mkpkg -p noao >& spool &
```

This must be repeated for each supported architecture. Layered systems are completely independent of one another and hence must be updated separately.

4. Interfacing New Graphics Devices

There are three types of graphics devices that concern us here. These are the graphics terminals, graphics plotters, and image displays.

4.1. Graphics terminals

The IRAF system as distributed is capable of talking to just about any conventional graphics terminal or terminal emulator, using the `stdgraph` graphics kernel supplied with the system. All one need do to interface to a new graphics terminal is add new graphcap and termcap entries for the device. This can take anywhere from a few hours to a few days, depending on one's level of expertise, and the characteristics of the device. Be sure to check the contents of the `dev$graphcap` file to see if the terminal is already supported, before trying to write a new entry. Useful documentation for writing graphcap entries is the GIO reference manual and the HELP pages for the `showcap` and `stty` tasks (see §2.2.6). Assistance with interfacing new graphics terminals is available via the IRAF Hotline.

4.2. Graphics plotters

The current IRAF system comes with several graphics kernels used to drive graphics plotters. The standard plotter interface is the SGI graphics kernel, which is interfaced as the tasks `sgikern` and `stdplot` in the PLOT package. Further information on the SGI plotter interface is given in the paper *The IRAF Simple Graphics Interface*, a copy of which is included with the IRAF installation kit.

SGI device interfaces for most plotter devices already exist, and adding support for new devices is straightforward. Sources for the SGI device translators supplied with the distributed system are maintained in the directory `$iraf/unix/gdev/sgidev`. NOAO serves as a clearinghouse for new SGI plotter device interfaces; contact us if you do not find support for a local plotter device in the distributed system, and if you plan to implement a new device interface let us know so that we may help other sites with the same device.

The older `NCAR` kernel is used to generate NCAR metacode and can be interfaced to an NCAR metacode translator at the host system level to get plots on devices supported by host-level NCAR metacode translators. The host level NCAR metacode translators are not included in the standard IRAF distribution, but public domain versions of the NCAR implementation for UNIX systems are widely available. A site which already has the NCAR software may wish to go this route, but the SGI interface will provide a more efficient and simpler solution in most cases.

The remaining possibility with the current system is the `calcomp` kernel. Some sites will have a Calcomp or Versaplot library (or Calcomp compatible library) already available locally. To make use of such a library to get plotter output on any devices supported by the interface, one may copy the library to the `hlib` directory and relink the Calcomp graphics

kernel.

A graphcap entry for each new device will also be required. Information on preparing graphcap entries for graphics devices is given in the GIO design document, and many actual working examples will be found in the graphcap file. The best approach is usually to copy one of these and modify it.

4.3. Image display devices

Most image display is now done with workstations, running a windowing system such as X, News, or SunView. Workstations may be used as "super terminals" for running UNIX/IRAF on any supported host system. Dedicated image display devices are still important, but are an expensive option which is justified only for particularly demanding applications.

For most workstations image display now means using the X window system, or some variant such as DECwindows (DEC) or X11/NeWS (Sun). Image display for IRAF under the MIT X window system is currently provided by the SAOIMAGE display server (originally called XIMAGE) developed by CFA and distributed by the IRAF project. The standard image display facility for a Sun workstation running the SunView window system is currently IMTOOL. More comprehensive support for graphics and image display under X, including the major vendor systems such as X11/NeWS and DECwindows, is planned for the future; contact NOAO for updated information on the degree of support available for any of these systems.

Interfaces for specific hardware image display devices are also available, although a fully general display interface is not yet provided by IRAF. Only the IIS model 70 and 75 are currently supported directly by NOAO. Interfaces for other devices are possible using the current datastream interface, which is based on the IIS model 70 datastream protocol with extensions for passing the WCS, image cursor readback, etc. (see the ZFIOGD driver in `unix/gdev`). This is how all the current displays, e.g., imtool and ximage, and the IIS devices, are interfaced, and there is no reason why other devices could not be interfaced to IRAF via the same interface. Eventually this prototype interface will be obsoleted and replaced by a more general interface.

4.4. Network access to graphics devices

Any IRAF device may be accessed either locally, with the device resident on the network node on which IRAF is being run, or remotely, using IRAF networking to access the remote device. The procedure for configuring IRAF networking is described in §2.2.7.

Unlike system resources such as directories, files, and magtape devices, for which the user specifies the node name explicitly when the device is accessed, graphics devices are normally accessed by device name only, with the system making the networking connection transparently to the user. In the case of a plotter or printer, the network connection is either defined in the IRAF graphcap or termcap entry for the device (in the `DD` or device dispose string), or is made at the host system level by `lpr`. The network node on which an image display device resides is specified in the graphcap entry for the device.

variable	*description*
`lpnode`	Line printer devices
`plnode`	Plotter devices
`node`	The user’s workstation

Network nodes in graphcap and termcap entries may be either actual network node names, or the names of environment variables that may be defined to point to network nodes, allowing the user to specify the network node dynamically at runtime. Examples of logical device nodes are shown in the table above. These appear in graphcap or termcap entries as embedded logical

node name references, e.g.,

```
:DD=plnode!apl,tmp$sgk,!{ sgidispatch sgi2uapl $F -l$(XO)…
```

The environment variable **node**, if defined, specifies the hostname of the user workstation. Currently this is only used for image display on the user workstation, but in principle it could be used to access any other workstation resident resource as well. For example, if the user is logged in on a workstation named "fred", which is running an IRAF compatible display server, one would enter

```
cl> reset node = fred
```

to cause the IRAF image display software running on the central server to display images on workstation "fred".

5. Host System Requirements

UNIX/IRAF is currently supported on VAXes running Berkeley UNIX or Ultrix, on the DECstation (MIPS cpu) running the MIPS version of Ultrix, on the Alliant, Convex, HP, Masscomp, and on all models of Sun workstations and servers. An Apollo port is in progress. Lick Observatory has ported IRAF to the ISI. The VMS and DG AOS-VS operating systems are supported in addition to UNIX based systems. New ports are always in progress, and anyone interested in running IRAF on a new computer should contact the IRAF project to determine the status of IRAF support for the system in question.

5.1. Memory requirements

The amount of memory required to run IRAF is more dependent upon how the system will be used than upon anything having to do with IRAF. Any large server should have more than enough memory to run IRAF. A workstation will need enough memory to run IRAF, UNIX, and the window system without paging heavily; 8-12 Mb is the minimum for such a system.

5.2. Disk requirements

The amount of disk required by a user depends greatly on the application, so it is hard to recommend a minimum disk size. For a system with access to a central server, a diskless system or 200-300 Mb of local SCSI disk is fine. For a standalone system with no access to large server, 500-600 Mb is about the minimum. A server should have several Gb of fast disk.

5.3. Diskless nodes

For applications such as programming or daily login support, a diskless node connected to a large file server is a cost effective approach delivering good performance. Some local disk for boot, swap, and local file storage and server independence is desirable but not essential.

For IRAF applications involving heavy image processing, one is inevitably going to want to run large batch image processing jobs directly on the server, implying that a *compute* rather than *file* server is what is needed (i.e., one will want to avoid heavy NFS loading on the server). A diskless node is still viable, but one will want to run jobs which involve heavy disk i/o directly on the server, reserving the workstation for the interactive things, e.g., graphics and image display, and compute bound image analysis tasks. Small SCSI disks are getting cheap enough that almost any color workstation equipped with say, 12-16 Mb of memory, probably warrants several Mb of local disk for server independence, swap, and local file storage.

Appendix A. The IRAF Directory Structure

A graph of the current full UNIX/IRAF directory structure is given at the end of this document. The main branches of the tree are summarized below. Beneath the directories shown are some 300 subdirectories, the largest directory trees being `sys`, `pkg`, and `noao`. The entire contents of all directories other than `unix`, `local`, and `dev` are fully portable, and are identical in all installations of IRAF sharing the same version number.

`bin`	- the IRAF BIN directories
`dev`	- device tables (`termcap`, `graphcap`, etc.)
`doc`	- assorted IRAF manuals
`lib`	- the system library; global files
`local`	- iraf login directory; locally added software
`math`	- sources for the mathematical libraries
`noao`	- packages for NOAO data reduction
`pkg`	- the IRAF applications packages
`sys`	- the virtual operating system (VOS)
`unix`	- the UNIX host system interface (HSI = kernel + bootstrap utilities)

The contents of the `unix` directory (host system interface) are as follows (this will vary somewhat depending upon the system):

`as`	- assembler sources
`bin`	- the HSI BIN directories
`boot`	- bootstrap utilities (mkpkg, rtar, wtar, etc.)
`gdev`	- graphics device interfaces (SGI device translators)
`hlib`	- host dependent library; global files
`os`	- OS interface routines (UNIX/IRAF kernel)
`reboot`	- executable script run to reboot the HSI
`shlib`	- shared library facility sources
`sun`	- gterm and imtool sources

If you will be working with the system much at the system level, it will be well worthwhile to spend some time exploring these directories and gaining familiarity with the system.