UNIX/IRAF Site Manager's Guide

Doug Tody
Mike Fitzpatrick

IRAF Group
June 1989
Revised December 1997

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, shared library 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.

December 27, 1997

Contents

UNIX/IRAF Site Manager's Guide

Doug Tody
Mike Fitzpatrick

IRAF Group
June 1989
Revised December 1997

1. Introduction

The IRAF system should be runnable as soon as it is installed, but there remain various 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.
- 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 *adass.iraf* newsgroups on USENET, by sending mail to `iraf@noao.edu`, or via the IRAF HOTLINE (520 318-8160). Contributions of interfaces developed for new devices, or any other software of general interest, are always welcomed.

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.

A summary of all modifications made to the IRAF system for a given IRAF release is given in the *Revisions Summary* distributed with the system. Additional information will be found in the system notes files (notes.v210, notes.v211, etc.) in the iraf/local and iraf/doc directories. 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.

2. System Setup

2.1. Installing the System

The procedure for installing or updating a UNIX/IRAF distribution is documented in the *IRAF Installation Guide* distributed with the software. A custom installation guide is provided for each platform on which IRAF is supported. In short, an IRAF tape, CD-ROM or network distribution is obtained and installed according to the instructions. The result is a full 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 distribution files were made, at the time the distributions were generated. Hence while these files may be useful as examples of properly configured descriptor files, the defaults, and many specific device entries, will in many cases 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, 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 simply be extracted and 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 unix/hlib directory. Chief among these is the **zzsetenv.def** file which defines the default hardcopy devices, image frame buffer, buffer sizes, etc. 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 used for text hardcopy) or the **graphcap** file (graphics terminals and image displays and graphics hardcopy printers) in iraf/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     = "lpr"
set stdplot     = "lpr"
set stdimage    = "imt512"
```

For example, you may wish to change the default editor to "emacs", the default printer to "lw5", or the default image display to "imt800". 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 §5.

2.2.2. The template LOGIN.CL

The template login.cl file hlib$login.cl, 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 TAPECAP file

Since V2.10 IRAF magtape devices are described by the "tapecap" file, `dev$tapecap`. This replaces the "devices" file used in earlier versions of IRAF. The tapecap file describes each local magtape device and controls all i/o to the device, as well as device allocation.

In V2.10 IRAF there was one tapecap file per IRAF installation and all client nodes sharing the same central version installation required device entries in the global tapecap file. In V2.11 this scheme has been generalized to allow each host to have its own private tapecap file, with a fallback to the generic tapecap file if no host-specific file is found. The system will look first for a configuration file called `tapecap.`*node* where *node* is the hostname of the server the tapecap file describes. If that is not found the default `tapecap` file will be used. In this way a separate tapecap file can be created for each node allowing a name such as ’mta’ to always refer to the first tape on that machine regardless of whether it varies in type from node to node. Alternatively, sites may wish to maintain only a single tapecap file with generic device names describing the different types of tape drives available on the local network. In either case the `devices.hlp` file described in the next section should be edited to document for the user the tape devices available at your site.

The tapecap files included in the distributed system include some generic device entries such as "mtxb1" (Exabyte unit 1, Sun ST driver), "mthp2" (HP7880 9 track drive, unit 2), and so on which you may be able to use as-is to access your local magtape devices. The exact list of available device types depend on the platform in question. Most likely you will want to add some device aliases, and you may need to prepare custom device entries for local devices. There must be an entry in the tapecap file for a magtape device in order to be able to access the device from within IRAF. All magtape device names *must* begin with the two-letter prefix "mt".

2.2.4. Configuring new TAPECAP entries

The `tapecap` file is text data base file (similar to the `termcap` and `graphcap` files) describing the capabilities and device names associated with a particular tape device on the system. For information on the format of the file see the termcap(5) man page. A listing of all recognized fields is given in the program comments for the tape driver in iraf$unix/os/zfiomt.c (more on this later). In general, creating a new tapecap entry for a device is a matter of finding a similar entry in the distributed file, and either using that directly if the device names are correct, or simply modifying it slightly to change device names so it will be appropriate for a drive on a different SCSI unit or using a different host driver. On occasion, other tapecap parameters will need to be added to correct for specific behavior that affects appending new data and tape positioning.

A tapecap entry for a device is usually divided into three different sections: a high-level entry giving the name of the drive as known to IRAF, a mid-level section defining the host device names associated with the drive, and a low-level generic section describing capabilities associated with all instances of a particular type of drive (DAT, Exabyte, 9-track, etc.). The starting point for the tapecap entry is whatever iraf name was used to access the drive. This is usually something like ’mta’, ’mtb’, etc but can be any valid name beginning with an 'mt' prefix and which defines all the needed parameters. When searching for a particular tapecap parameter the *first* occurrence of that parameter in the entry is used by the system, and a complete tapecap description is composed of all the entries which are linked by the **:tc** continuation fields.

As an example consider a typical entry for a DAT drive on unit 0 known to a Solaris/IRAF system as 'mta', the high-level entry would look like:

```
mta|Generic DAT entry, unit 0|         :tc=mtst0.solaris.dat:
```

Here we define the iraf name (which must begin with an 'mt' prefix) along with any aliases delimited by the '|'. The **:tc** field continues the tapecap at the next entry named "mtst0.solaris.dat":

```
mtsd0|mtst0.solaris.dat|DAT drive on Solaris:\\
      :al=0 0bn 0cb 0cn 0hb 0hn 0lb 0ln 0mb 0mn 0u 0ubn \\
      0b 0c 0cbn 0h 0hbn 0l 0lbn 0m 0mbn 0n 0ub 0un:\\
      :dv=0bn:lk=0:tc=solaris-dat:
```

This entry is primarily used to specify the host device names associated with the drive. The **:al** (aliases) field is a list of *all* device aliases in the UNIX /dev or /dev/rmt directories associated with this device. This is needed so the tape allocation task can properly change the permissions and ownership on *each* device name which accesses that tape drive. The **:dv** (device) field is the *no-rewind* device name and is the device file actually opened for tape I/O; this must be a no-rewind device since IRAF will maintain the tape position automatically. The actual device name typically depends on the density of the tape, whether compression is used etc. The **:lk** is used to build the name of a "lok file" that is created in the /tmp directory of the machine hosting the drive that will be used to maintain the tape status and position information; this value should be unique for each drive on the machine to avoid conflicts. When configuring a new tapecap entry, all one usually needs change is the iraf device name in the first section and the host device names in the :dv, :al and :lk fields of this entry. Finally this section continues the entry with a **:tc** field saying to branch to the "solaris-dat" generic entry:

```
solaris-dat|sdat-60m|Sun/Solaris DAT drive:\\
      :dt=Archive Python 4mm Helical Scan tape drive:tt=DG-60M:\\
      :ts#1274378:bs#0:mr#0:or#65536:fb#10:fs#127000:mf:fe#2000:
```

The low-level entry here is where parameters relating to all drives of a particular type using a particular host tape driver are maintained, e.g. the record sized used for tape I/O, positioning capabilities, filemark sizes, etc. These will rarely need to be changed from the distributed entries unless you are using a new tape driver or a different model tape drive, or a type of tape cartridge with a capacity different than that given ("tz"). See the section below for a full list of the tapecap parameters and their meanings.

For a more complicated example let's consider how to add an entry for an Exabyte 8505 drive given an existing entry for an Exabyte 8200 device. We can ignore for now the low-level entry found in the distributed tapecap and concentrate on what fields actually need changing in this case. We begin with the high-level entry defining the iraf names, we will need one name for the drive in each of three modes (8200 mode, 8500 mode, and 8500 mode w/ compression):

```
mta|Exabyte 8200, Unit 0|           :tc=mtst0.solaris.exb8200:
mtb|mtblo|Exabyte 8505, Unit 0|     :tc=mtst0.exb8505-lo:
mtbhi|Exabyte 8505, Unit 0|         :tc=mtst0.exb8505-hi:
mtbc|Exabyte 8505, Unit 0|          :tc=mtst0.exb8505-c:
```

The new iraf names are therefore *mtb* (8200 mode), *mtbhi* (8500 mode), and *mtbc* (8500 + compression). These all link to the second level entry where we make use of the existing EXB8200 entry:

```
mtsee0|mtst0.solaris.exb8200|Exabyte 8200 drive on Solaris:\\
        :al=0 0bn 0cb 0cn 0hb 0hn 0lb 0ln 0mb 0mn 0u 0ubn \\
        0b 0c 0cbn 0h 0hbn 0l 0lbn 0m 0mbn 0n 0ub 0un:\\
        :dv=0bn:lk=0:tc=solaris-exb8200:
mtsee0lo|mtst0.exb8505-lo|:dv=0lbn:tc=mtsee0:
mtsee0hi|mtst0.exb8505-hi|:dv=0mbn:fs#48000:ts#5000000:tc=mtsee0:
mtsee0hic|mtst0.exb8505-c|:dv=0cbn:fs#48000:ts#5000000:tc=mtsee0:
```

Note that the names we just created link to the one-line entries below the standard EXB 8200 entry 'mtst0.solaris.exb8200' (the *mtb* entry could just as legally have linked to this entry right away). Since all we need to change is the **:dv** field (because we're opening the same drive, but by using a different name the host system accesses it in the appropriate mode) we can simply make a new entry point, change the :dv field and then link to the existing entry where all the rest of the parameters will be the same. In this case we've also reset the **:fs** and **:ts** fields to override the values in the low-level Exabyte description since these have also changed for the new model drive. If we wished to modify this entry for a drive on e.g. unit 2 all we would need to do is modify the various :dv, :al, and :lk fields so the device names are correct, and change the name of the tapecap entry points so we avoid any confusion later on.

When configuring a new tapecap and encounter problems it is useful to turn on status output so you get a better idea of where the tape is positioned and what's going on, to do this use the **:so** field as follows:

```
cl> set tapecap = ":so=/dev/tty"
```

Alternatively, the :so can be specified on the command line, e.g.

```
cl> rewind "mta[:so=/dev/tty]"
```

Any other tapecap parameters can be specified in the same way. The quotes around the tape name are required if any special characters such as this can also be directed to an *Xtapemon* server running either locally or remotely, see the xtapemon man page for details. Help with configuring new tapecap entries is available from IRAF site support.

2.2.4.1. More on TAPECAP parameters

As we see from the previous section, in most cases the only tapecap parameters that need to be changed are **:dv**, **:al**, and maybe **:lk**. There are however a number of other tapecap parameters that sometimes must be modified to describe how the tape device operates or to optimize I/O to the device. A full listing of the available tapecap parameters can be found in the program comments for the iraf tape driver iraf$unix/os/zfiomt.c; we will only briefly discuss a few here. Any changes you make with the parameters mentioned here can usually go in the low-level tapecap entry so they will "fix" all drives of the same type, however you may also wish to modify just the high-level entry to change only one drive. For example:

```
mta|Generic DAT entry, unit 0|      :se:ow:tc=mtst0.solaris.dat:
```

would add the ":se:ow" fields (discussed below) to only the *mta* device.

Boolean tapecap parameters may be negated if you are linking to an existing entry which already defines a particular field. For example, in

```
mta|Generic DAT entry, unit 0|      :se@:tc=mtst0.solaris.dat:
```

the '@' character would negate the **:se** field regardless of whether it is defined elsewhere in the entry.

One of the most common problems encountered is that only odd-numbered images on a tape are readable by the drive. The solution to this is usually to add a **se** to the tapecap to tell the driver that the tape will position past the EOT in a read. Another common problem is with appending new data to an existing tape, this sometimes requires the addition of a **ow** field to tell the driver to backspace and overwrite the EOT when appending. A **re** is sometimes needed if there is a problem sensing the EOT when reading all images from a tape, this tell the driver that a read at EOT returns an ERR.

The parameter **fb** may be specified for a device to define the "optimum" FITS blocking factor for the device. Unless the user explicitly specifies the blocking factor, this is the value that the V2.11 *wfits* task will use when writing FITS files to a tape. Note that for cartridge devices a FITS blocking factor of 22 is used for some devices; at first this may seem non-standard FITS, but it is perfectly legal, since for a fixed block size device the FITS blocking factor serves only to determine how the program buffers the data (for a fixed block device you get exactly the same tape regardless of the logical blocking factor). For non-FITS device access the magtape system defines an optimum record size which is used to do things like buffer data for cartridge tape devices to allow streaming.

Some devices, e.g. most Exabyte drives, are slow to switch between read and skip mode, and for files smaller than a certain size, when skipping forward to the next file, it will be faster to read the remainder of the file than to close the file and do a file skip forward. The **fe** parameter is provided for such devices, to define the "file equivalent" in kilobytes of file data, which can be read in the time that it takes to complete a short file positioning operation and resume reading. Use of this device parameter in a tape scanning application such as *rfits* can make a factor of 5-10 difference in the time required to execute a tape scan of a tape containing many small files.

On a device such as most cartridge tape devices where backspacing is not permitted or does not work reliably it may be necessary to set the **nf** parameter to tell the driver to rewind and space forward when backspacing to a file.

Lastly, when configuring a new low-level generic entry for the device it is sometimes necessary to change the various size parameters for the drive. These include:

bs	**device block size (0 if variable)**
fb	**default FITS blocking factor (recsize=fb*2880)**
fe	**time to FSF equivalent in file Kb**
mr	**maximum record size**
or	**optimum record size**
fs	**approximate filemark size (bytes)**
ts	**tape capacity (Mb)**
dn	**density**

All but the last three fields are used either by the driver or a task when reading or writing a tape, the **:fs**, **ts** and **:dn** fields are used by tape monitoring tasks such as *xtapemon* to compute the approximate amount of tape used and do not affect tape operation. For devices which are capable of variable block size I/O (i.e. almost anything but a cartridge tape) it is best to leave the **bs** field at zero. The maximum and optimum record sizes, the **mr** and **or** fields, are usually determined by the host tape driver used. Values for these can either be found in the host driver man page or it's system include file.

2.2.5. 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. It is the IRAF name of the device, as given in files such as termcap, graphcap, and tapecap, which should appear in this help file followed by a brief description of the device, see the distributed file as an example. Starting with V2.10 this file in no longer used to configure tape devices, it is informational only.

2.2.6. 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 device-specific entry, called `DD`. This consists of three fields: the device name, e.g. "node!device", the template for the temporary spoolfile, and the UNIX command to be used to dispose of the file to the printer. On most UNIX systems it is not necessary to make use of the node name and IRAF networking to access a remote device since UNIX *lpr* already provides this capability, however it might still be useful if the desired device does not have a local *lpr* entry for some reason. Printer devices named in this file may be used for text hardcopy output such as you get from the LPRINT task, graphics hardcopy devices are configured by editing the `graphcap` file discussed in the next section.

As an example, assume we have a printer known to the sun as 'lw5', the termcap entry would look something like:

```
lw5|lp5|                               :tc=sapple5:

sapple5|sapple|Apple laser writer NT on Orion:\\
    :co#80:li#66:os:pt:ta^I:\\
    :DD=lpnode!apple,/tmp/asfXXXXXX,!{ lpr -Plw5 $F; rm $F; }:
```

To then create an entry for a new device named 'lw16' simply copy this entry and change the '5' to '16' in the device and termcap entry names, and especially in the *lpr* command of the `DD` string. The `$F` denotes the name of the file to be printed, specifically the temp file created so it should be removed to avoid filling up the disk. Note that the `DD` string can contain any valid unix command to print a file to a specific device, we use various local print commands, Enscript, etc.

If you have a local terminal which has no entry in the IRAF termcap file, you probably already have an entry in the UNIX termcap file. Simply copy it into the IRAF file; both systems use the same termcap database format and terminal device capabilities. However, if the terminal in question is a graphics terminal with a device entry in the graphcap file, you should add a '`:gd`' capability to the termcap entry. If the graphcap entry has a different name from the termcap entry, make it '`:gd=`*gname*'.

2.2.7. The GRAPHCAP file

There must be an entry in the graphcap file for all graphics terminals, batch plotters, and image displays accessed by IRAF programs. We will discuss each briefly since the setup is slightly different in each case. Help preparing new graphcap device entries is available from iraf site support if needed, but with the exception of new graphics terminals creating an entry for a new device is usually just a matter of editing an existing entry. 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.

2.2.7.1. Graphics hardcopy devices

Graphics hardcopy devices nowadays are typically Postscript printers, but support is included in the system for various pen and raster plotters, and non-PostScript printers such as HP LaserJet, Imagen, QMS, etc. We will concentrate here on PostScript devices since they are the most common. The typical graphcap entry will look something like

```
lp5|lw5|               :tc=uapl5:

uapl5|UNIX generic interface to 300dpi printer on Orion:\\
    :xs#0.269:ys#0.210:ar#0.781:\\
    :DD=apl,tmp$sgk,!{ sgidispatch sgi2uapl $F -l$(XO) -w$(XW) \\
    -b$(YO) -h$(YW) -p$(PW) | lpr -Plw5; rm $F; }&:tc=sgi_apl:
```

where the device is known to the system as *lw5* or *lp5*. The entry is very similar in form to the `termcap` entry discussed above, and changing it for a new device is primarily a matter of changing the device names. The exception however is in the `DD` string: here instead of a simple print command we invoke an SGI translator via the *sgidispatch* command (in this case the *sgi2uapl* translator) which is used to the convert the graphics kernel metacode to PostScript for the final printing. The arguments to the *sgi2uapl* translator are the device resolution and offset parameters obtained from the *sgi_apl* entry linked by the :tc field at the end of the graphcap entry. The output from the translator is piped to a printer and the temp file is removed.

If we wish to convert this entry for a different type of printer, aside from the changing the name in the graphcap entries and the print command, the DD string may have to be changed to call a new SGI translator with the appropriate arguments, and the final :tc field would have to link to a new entry appropriate for that device. In V2.11 the following SGI translators are available:

```
sgi2uapl.c  - PostScript for LaserWriters and PS plotters
sgi2ueps.c  - Encapsulated PostScript, PS-Adobe-3.0, EPSF-3.0
sgi2uhpgl.c - HP Graphics Language for HP 7550A and others
sgi2uimp.c  - Impress language for Imagen printers
sgi2uqms.c  - QMS Vector Graphics (Talaris Lasergrafix)
sgi2uhplj.c - HP Printer Command Language (LaserJet Series)
sgi2uptx.c  - Printronix plotter
```

In addition, Versatec plotters are supported (no SGI translator needed).

2.2.7.2. Image display frame buffers

Graphcap entries are required to configure the available `stdimage` devices for the system. These are basically just frame buffer configurations describing the size of the image display being used (whether it's an actual frame buffer such as an IIS mode 70 or a display server such as XImtool or SAOimage). A typical entry for a 512x512 frame buffer looks like:

```
imt1|imt512|imtool|Imtool display server:\\
    :cn#1:LC:BS@:z0#1:zr#200:DD=node!imtool,,512,512:tc=iism70:
```

Here the `:cn` field is the configuration number and the frame buffer size is given in the `DD` field. For display servers such as XImtool the configuration number is passed to the server which then uses that as an index to the *imtoolrc* file (normally installed by the system as a link to `dev$imtoolrc`) it uses to determine the frame buffer size to be used. When adding a new frame buffer you need to be sure the :cn field is unique and the size in the graphcap file agrees with the size in the imtoolrc file for that config, *both* files must be edited for the new size to be recognized correctly. Note that SAOimage has a limit of 64 possible frame buffers that will be recognized, XImtool and SAOtng recognize up to 128 possible configurations.

2.2.7.3. Graphics Terminals

New graphics terminals will need a new entry in the graphcap file if one does not already exist. The IRAF file gio$doc/gio.hlp contains documentation describing how to prepare graphcap device entries. A printed copy of this document is available from the iraf/docs directory in the IRAF network archive. However, once IRAF is up you may find it easier to generate your own copy using *help*, as follows:

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

This will print the document on the default IRAF printer device which will be the default printer for your machine or the one named by your UNIX `PRINTER` environment variable (use the "device=" hidden parameter to specify a different device). Alternatively, to view the file on the terminal,

```
cl> phelp gio$doc/gio.hlp fi+
```

The help pages for the IRAF tasks *showcap* and *stty* should also be reviewed as these utilities are useful for generating new graphcap entries. The i/o logging feature of *stty* is useful for determining exactly what characters your graphcap device entry is generating. The *gdevices* task is useful for printing summary information about the available graphics devices.

2.2.8. Configuring IRAF networking

The dev directory contains several files (`hosts`, `irafhosts`, and `uhosts`) used by the IRAF network interface. IRAF networking is used to access remote image displays, printers, magtape devices, files, images, etc. via the 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 "hosts" file. Make an entry for each logical node, in the format

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

following the examples given in the hosts file supplied with the distribution (which is the NOAO/Tucson hosts file). Note that there may be multiple logical entries for a single physical node.

The "uhosts" file is not used by UNIX/IRAF systems hence does not need to be modified (it used by VMS/IRAF). The "irafhosts" file is the template file used to create user .irafhosts files. It does not have to be modified, although you can do so if you wish to change the default parameter values given in the file.

To enable IRAF networking on a particular IRAF host, the host OS **hostname** (i.e. the output of the unix *hostname* command) must appear as a primary name or alias somewhere in the IRAF hosts table. On systems where this is the fully qualified host name (FQHN) the node name may exceed a limit 16-character limit on a node name so at least one alias should include a truncated version of the FQHN, the entire FQHN should appear on the right side of the ':' in the irafks.e pathname. 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. Note that it may be best to simply delete the NOAO host table entries since any duplicate with a local host entry will will cause the IRAF "cd" command to fail and may have other consequences.

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. Read the output carefully to see if any problems are reported.

Alternatively users can set up a private hosts table by copying the system version and making any additions. To then make use of this define a CL environment variable *irafhnt* which is the path to the private hosts file. For example,

```
cl> copy dev$hosts home$myhosts         # make private copy
cl> edit home$myhosts                   # edit any changes
cl> reset irafhnt = home$myhosts        # reset hosts table to be used
cl> flpr 0                              # reinitialize system to use it
```

You can also define a UNIX *irafhnt* variable in the same way prior to logging into the CL to accomplish the same thing.

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. It is not necessary to have a separate copy of IRAF on each node, i.e., a single copy of IRAF may be NFS mounted on all nodes (you will need to run the IRAF *install* script on each client node). 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 systems currently support only TCP/IP based networking. Networking between any heterogeneous collection of systems is possible provided they support TCP/IP based networking (virtually all UNIX-based systems do). The situation with networking between UNIX and VMS systems is more complex. Contact the IRAF project for further information on networking between UNIX and VMS systems.

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. For example, if *foo* is a network node,

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

In a network of "trusted hosts" the network connection will be made automatically, without a password prompt using the *rsh* protocol. A password prompt will be generated if the user does not have permission to access the remote node with UNIX commands such as *rsh*. Hosts are made "trusted" in a network by listing them in the system `/etc/hosts.equiv` file, most often when rsh fails it's because this file hasn't been configured, usually for security reasons. User's can configure a `.rhosts` file in their UNIX login directories (see the rhosts(5) man page) to make the hosts trusted for their account and bypass the passwd prompt. Each user also has a .irafhosts file in their UNIX login directory which can be used to exercise more control over how the system connect to remote hosts. See the discussion of IRAF networking in the *IRAF Version 2.10 Revisions Summary* (in iraf$doc/v210revs.ms), or in the V2.10 system notes file, for a more in-depth discussion of how IRAF networking works.

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. The network pathname allows an IRAF process running on any node to access an object regardless of where it is located on the network.

Inefficiencies can result when image pixel files are stored on disks which are cross-mounted using NFS. The typical problem arises when imdir (the pixel file storage directory) is set to a path such as "/data/iraf/user/", where /data is a NFS mounted directory. Since NFS is transparent to applications like IRAF, IRAF thinks that /data is a local disk and the network pathname for a pixel file will be something like "foo!/data/iraf" where "foo" is the hostname of the machine on which the file is written. If the image is then accessed from a different network node the image data will be accessed via an IRAF networking connection to node "foo", followed by an NFS connection to the node on which the disk is physically mounted, causing the data to traverse the network twice, slowing access and unnecessarily loading the network.

A simple way to avoid this sort of problem is to include the server name in the imdir, e.g.,

```
cl> set imdir = "server!/data/iraf/user/"
```

This also has the advantage of avoiding NFS for pixel file access - NFS is fine for small files but can load the server excessively when used to access bulk image data.

Alternatively, one can set imdir to a value such as "HDR$pixels/", or disable IRAF networking for disk file access. In both cases NFS will be used for image file access.

2.2.9. 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 (user 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 both a `.login` file defining the environment for the IRAF account, and a number of other "dot" files used to setup the IRAF system manager's working environment.

Most site managers will probably want to customize these files according to their personal preferences. In doing this please use caution to avoid losing environment definitions, etc., which are essential to the correct operation of IRAF, including IRAF software development and maintainence.

The default login.cl file supplied in the IRAF login directory uses machine independent pathnames and should work as-is (no need to do a *mkiraf* - in fact *mkiraf* has safeguards against inadvertent use within the IRAF directories and may not work in iraf/local). It may be necessary to edit the .login file to modify the way the environment variable `IRAFARCH` is defined. This variable, required for software development but optional for merely using IRAF, must be set to the name of the desired machine architecture, e.g., sparc, vax, rs6000, ddec, 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, so be careful. The alias *setarch*, defined in the iraf account .login, is convenient for setting the desired architecture for IRAF execution and software development.

2.2.10. 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 user will be doing any IRAF based software development (including IMFORT). Programmers doing IRAF software development may wish to source hlib$irafuser.csh in their .login file as well.

2.3. Tuning Considerations

2.3.1. Stripping the system to reduce disk usage

If the system is to be installed on multiple CPUs, or if a production version is to be installed on a workstation, it may be necessary or desirable to strip the system of all non-runtime files to save disk space. This equates to 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 use "mkpkg strip" as in the following example (this may be executed from either the host system or from within IRAF).

```
% 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 and be sure to include the "-p *pkg*" switch to mkpkg so the proper environment is loaded. For example, to strip the NOAO package:

```
% cd $iraf/noao
% mkpkg -p noao strip
```

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

Often the computing facilities at a site consist of a heterogeneous network of workstations and servers. These machines will often have quite different architectures or operating systems. Since IRAF is a large system it is undesirable to have to maintain a separate copy of IRAF for each machine architecture 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. It should be noted however that it is not always possible to use the multiple architecture support within the core system to maintain a single IRAF source tree for the entire heterogeneous network. The Host System Interface (HSI) for IRAF ports is different for platforms as diverse as Sun and SGI so there should be a separate installation for each system to minimize difficulties (the update schedules usually differ as well so maintaining the same version is also more difficult). In the case of supporting SunOS and Solaris systems, or a single platform with multiple compiler options (e.g. DECStation Ultrix with the MIPS and/or DEC compilers), one installation is sufficient since it's generally only the binaries that will differ (i.e. the OS is still the same). Almost any combination of architectures may be supported by a single copy of an external package.

Multiple architecture support is implemented by separating the IRAF sources and binaries into different directory trees. The 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 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 software architectures are shown below.

Architecture	*System*	*Description*
generic	any	no binaries
ssun	Sun-4	Sun SPARC under Solaris (RISC) architecture, integral fpu
sparc	Sun-4	Sun SPARC (RISC) architecture, integral fpu
pg	Sun-4	Sun/IRAF compiled for profiling
linux	PC	PC platforms running Linux
freebsd	PC	PC platforms running FreeBSD
alpha	Dec Alpha	DEC Alpha running Digital Unix
ddec	Decstation	DEC Fortran version of DSUX/IRAF
dmip	Decstation	MIPS Risc Fortran version of DSUX/IRAF
rs6000	IBM	IBM RS/6000 running AIX
hp700	HP	HP 700 series running HPUX 10
irix	SGI	SGI IRIX, MIPS cpu

Most of these correspond to hardware architectures or operating system options. 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 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 alpha
```

would cause IRAF to run using the alpha architecture, corresponding to the BIN directory bin.alpha. 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
alpha
```

or

```
cl> show arch
.alpha
```

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 doc$notes.v28.

3.2. Shared libraries

Among the UNIX based versions of IRAF, currently only Sun/IRAF and the Digital Unix (DEC Alpha) version of IRAF support shared libraries, although we may add shared library support to additional versions of IRAF in the future. Years ago SunOS was one of the few UNIX systems supporting shared, mapped access to files, and hence shared libraries. This is no longer the case however, and today most or all versions of UNIX provide some sort of shared library facility. Shared libraries result in a considerable savings in disk space, so eventually we will implement shared library support for additional platforms. In the meanwhile, if you are running IRAF on a system other than a Sun or DEC Alpha this section can be skipped.

So long as everything is working properly, the existence and use of a shared library should be transparent to the user and to the site manager. This section gives an overview of the shared library facility to point the reader in the right direction in case questions should arise.

What the shared library facility does is take most of the IRAF system software (currently the contents of the `ex`, `sys`, `vops`, and `os` libraries) and link it together into a special sharable image, the file `S`*n*`.e` in each core system BIN directory (*n* is the shared image version number, e.g. "S7.e"). This file is mapped into the virtual memory of each IRAF process at process startup time. Since the shared image is shared by all IRAF processes, each process uses less physical memory, and the process pagein time is reduced, speeding process execution. Likewise, since the subroutines forming the shared image are no longer linked into each individual process executable, substantial disk space is saved for the BIN directories. Link time is correspondingly reduced, speeding software development.

The shared library facility consists of the **shared image** itself, which is an actual executable image (though not runnable on all systems), and the **shared library**, contained in the library lib$libshare.a, which defines each VOS symbol (subroutine), and which is what is linked into each IRAF program. The shared library object module does not consume any space in the applications program, rather it consists entirely of symbols pointing to **transfer vector** slots in the header area of the shared image. The transfer vector slots point to the actual subroutines.

When an IRAF program is linked with *xc*, one has the option of linking with either the shared library or the individual system libraries. Linking with the shared library is the default; the `-z` flag disables linking with the shared library. In the final stages of linking *xc* runs the HSI utility *edsym* to edit the symbol table of the output executable, modifying the shared library (VOS) symbols to point directly into the shared image (to facilitate symbolic debugging), optionally deleting all shared library symbols, or performing some other operation upon the shared library symbols, depending upon the *xc* link flags given.

At process startup time, upon entry to the process main (a C main for Sun/IRAF) the shared image will not yet have been mapped into the address space of the process, hence any attempted references to VOS symbols would result in a segmentation violation. The *zzstrt* procedure, called by the process main during process startup, opens the shared image file and maps it into the virtual space of the IRAF program. Once the IRAF main prompt appears (when running an IRAF process standalone), all initialization will have completed.

Each BIN, if linked with the shared library, will have its own shared image file `S`*n*`.e`. If the shared image is relinked this file will be moved to `S`*n*`.e.1` and the new shared image will take its place; any old shared image files should eventually be deleted to save disk space, once any IRAF processes using them have terminated. Normally when the shared image is rebuilt it is not necessary to relink applications programs, since the transfer vector causes the linked application to be unaffected by relocation of the shared image functions.

If the shared image is rebuilt and its version number is incremented (the *n* in `S`*n*`.e`), the transfer vector is rebuilt the new shared image cannot be used with previously linked applications. These old applications will still continue to run, however, so long as the older shared image is still available. It is not unusual to have several shared image versions installed in a BIN directory.

Further information on the Sun/IRAF shared library facility in given in the IRAF V2.8 system notes file. In particular, anyone doing extensive IRAF based software development should review this material, e.g., to learn how to debug processes that are linked with the shared image.

3.3. Layered software support

An IRAF installation consists of the core IRAF 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 requires the facilities of the core system to run, and is portable to any computer which already runs IRAF. Any number of layered products can be installed in IRAF to produce the IRAF system seen by the user at a given site.

The support provided by IRAF for layered software is essentially the same as that provided for maintaining the core IRAF system itself (the core system is a special case of a layered package). Each layered package (usually this refers to a suite of subpackages) 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 is rooted in the IRAF directories, NOAO is equivalent to any other layered product, e.g., STSDAS, TABLES, XRAY, CTIO, NSO, ICE, GRASP, NLOCAL, STEWARD, and so on. In general, layered products should be rooted somewhere outside the IRAF directory tree to simplify updates.

3.4. 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*). About the only reasons to use these with the UNIX versions of 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 VMS/IRAF system, 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 and the shared library, knows how to read 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 for the *vi* editor, a help database examine tool, and other tasks. Further information on these tasks is available in the online help pages.

3.5. Modifying and updating a package

IRAF applications development is most conveniently performed from within the IRAF environment, since testing must be done from within the environment. The usual edit-compile-test development cycle is illustrated below. This takes place within the *package directory* containing all the files specific to a given 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`	The *mkpkg* command with no arguments does the default mkpkg operation; for a subpackage this is usually the same as *mkpkg relink* below. For the root mkpkg in a layered package it udpates the entire layered package.
`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_*pkgname*.e in the package directory, where *pkgname* 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 layered package to which the subpackage *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 giving its name to 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 parameter queries during process execution. The LOGIPC debugging facility introduced in V2.10 is also useful for debugging subprocesses. 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 IRAF 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 subpackages of the NOAO layered package. If the package being updated references any libraries or include files in *other* layered packages, those packages must be indicated with a "-p pkgname" flag as well, to cause the external package to be searched.

The CL process cache can complicate debugging and testing if one forgets that it is there. 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 CL will automatically run the new executable (the CL checks the modify date on the executable file every time a task is run). If however 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 debugger breakpoints.

The IRAF shared image can also complicate debugging, although for most applications-level debugging the shared library is transparent. By default the shared image symbols are included in the symbol table of an output executable following a link, so in a debug session the shared image will appear to be part of the applications program. When debugging a program linked with the shared library, the process must be run with the `-w` flag to cause the shared image to be mapped with write permission, allowing breakpoints to be set in the shared image (that is, you type something like ":r -w" when running the process under the debugger). Linking with the `-z` flag will prevent use of the shared image entirely.

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
```

or

```
cl> mkpkg update
```

to update the package.

3.6. 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 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. One sets up several BIN directories, one for each architecture, named `bin.`*arch*, where *arch* is "sparc", "ddec", "rs6000", 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 layered package LIB directory 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 update >& spool &
```

where *foo* is the name of the IRAF package 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 uninstalled 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.

3.7. 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 or loginuser.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 may contain non-portable or site specific software).

To make a custom local you make a copy of the "template local" package (iraf$local) somewhere outside the IRAF directory tree, change the name to whatever you wish to call the new layered package, and install it as outlined in §3.5. 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.

3.8. Updating the full IRAF system

This section will describe how to recompile or relink IRAF. Before we get into this however, it should be emphasized that *most users will never need to recompile or relink IRAF*. In fact, this is not something that one should attempt lightly - don't do it unless you have some special circumstance which requires a custom build of the system (such as a port). Even then you might want to set up a second copy of IRAF to be used for the experiment, keeping the production system around as the standard system. If you change the system it is a good idea to make sure that you can undo the change.

While the procedure for building IRAF is straightforward, it is easy to make a mistake and without considerable knowledge of IRAF it may be difficult to recover from such a a mistake (for example, running out of disk space during a build, or an architecture mismatch resulting in a corrupted library or shared image build failure). More seriously, the software - the host operating system, the host Fortran compiler, the local system configuration, and IRAF - is changing constantly. A build of IRAF brings all these things together at one time, and every build needs to be independently and carefully tested. An OS upgrade or a new version of the Fortran compiler may not yet be supported by the version of IRAF you have locally. Any problems with the host system configuration can cause a build to fail, or introduce bugs. For example, systems which support multiple Fortran compilers or which require the user to install and configure the compiler are a common source of problems.

The precompiled binaries we ship with IRAF have been carefully prepared and tested, usually over a period of months prior to a major release. They are the same as are used at NOAO and at most IRAF sites, so even if there are bugs they will likely have already been seen elsewhere and a workaround determined. If the bugs are new then since we have the exact same IRAF system we are more likely to be able to reproduce and fix the bug. Often the bug is not in the IRAF software at all but in the host system or IRAF configuration. As soon as an executable is rebuilt (even something as simple as a relink) you have new, untested, software.

3.8.1. The BOOTSTRAP

To fully build IRAF from the sources is a three step process. First the system is "bootstrapped", which builds the host system interface (HSI) executables. A "sysgen" of the core system is then performed; this compiles all the system libraries and builds the core system applications. Finally, the bootstrap is then repeated, to make use of some of the functions from the IRAF libraries compiled in step two.

To bootstrap IRAF, login as IRAF and enter the commands shown below. This takes a while and generates a lot of output, so the output should be spooled in a file. Here, *arch* refers to the IRAF architecture you wish to build for.

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

There are two types of bootstrap, the initial bootstrap starting from a source only system, called the NOVOS bootstrap, and the final or VOS bootstrap, performed once the IRAF system libraries `libsys.a` and `libvops.a` exist. The bootstrap script *reboot* will automatically determine whether or not the VOS libraries are available and will perform a NOVOS bootstrap if the libraries cannot be found. It is important to restore the desired architecture before attempting a bootstrap, as otherwise a NOVOS bootstrap will be performed.

3.8.2. The SYSGEN

By sysgen we refer to an update of the core IRAF system - all of the files comprising the runtime system, excluding the HSI which is generated by the bootstrap. On a source only system, the sysgen will fully recompile the core system, build all libraries and applications, and link and install the shared image and executables. On an already built system, the sysgen scans the full IRAF directory tree to see if anything is out of date, recompiles any files that need it, then relinks and installs new executables.

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. Each sysgen builds or updates a single BIN directory. 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 ddec binaries on a Decstation:

```
% cd $iraf
% mkpkg ddec
% 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. Modules that have already been compiled should not need to be recompiled. 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 fully recompiled. If the system libraries already exist they will merely be updated. Once the system libraries are up to date the sysgen will rebuild the shared library if any of the system libraries involved were modified, then the core system executables will be relinked.

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 sparc binaries for the NOAO package:

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

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

To force a full recompile of the core system or a layered package, one can use *rmbin* to delete the objects, libraries, etc. scattered throughout the system, or do a "mkpkg generic" and then delete the `OBJS.arc.Z` file in the BIN one wishes to regenerate (the latter approach is probably safest).

A full IRAF core system sysgen currently takes anywhere from 3 to 30 hours, depending upon the system (e.g. from 30 hours on a VAX 11/750, to 3 hours on a big modern server). On most systems a full sysgen is a good job to run overnight.

3.8.3. Localized software changes

The bootstrap and the sysgen are unusual in that they update the entire HSI, core IRAF system, or layered package. Many software changes are more localized. If only a few files are changed a sysgen will pick up the changes and update whatever needs to be updated, but for localized changes a sysgen really does more than it needs to (if the changes are scattered all over the system an incremental sysgen-relink will still be best).

To make a localized change to a core system VOS library and update the linked applications to reflect the change all one really needs to do is change the desired source files, run *mkpkg* in the library source directory to compile the modules and update the affected libraries, and then build a new IRAF shared image (this assumes that the changes affect only the libraries used to make the shared image, i.e., libsys, libex, libvops, and libos). Updating only the shared image, without relinking all the applications, has the advantage that you can put the runtime system back the way it was by just swapping the old shared image back in - a single file.

For example, assume we want to make a minor change to some files in the VOS interface IMIO, compiling for the sparc architecture on SunOS, which uses a shared library. We could do this as follows (this assumes that one is logged in as IRAF and that the usual IRAF environment is defined).

```
        % whoami
        iraf
        % cd $iraf
        % mkpkg sparc
        % cd imio
          (edit the files)
        % mkpkg                         # update IMIO libraries (libex)
        %
        % cd $iraf/bin.sparc            # save copy of old shared image
        % cp S6.e S6.e.V210
        %
        % cd shlib
        % tar -cf ~/shlib.tar .         # backup shlib just in case
        % mkpkg update                  # make and install new shared image
```

If IRAF is not configured with shared libraries, one must relink the full IRAF system and all layered packages for the change to take effect. This is done by running *mkpkg* at the root of the core system and each layered package. For example, on an IBM RS/6000,

```
% whoami
iraf
% cd $iraf
% mkpkg rs6000
% cd imio
  (edit the files)
% cd iraf
% mkpkg                          # update the core system
%
% cd noao
% mkpkg rs6000
% mkpkg -p noao                  # update the NOAO packages
```

and so on, for each layered package.

Changing applications is even easier. Ensure that the system architecture is set correctly (i.e. "mkpkg *arch*" at the iraf or layered package root), edit the affected files in the package source directory, and type "mkpkg -p <pkgname> update" in the root directory of the package being edited. This will compile any modified files, and link and install a new executable. You can do this from within the CL and immediately run the revised program.

We should emphasize again that, although we document the procedures for making changes to the software here, to avoid introducing bugs we do not recommend changing any of the IRAF software except in unusual (or at least carefully controlled) circumstances. To make custom changes to an application, it is best to make a local copy of the full package somewhere outside the standard IRAF system. If changes are made to the IRAF system software it is best to set up an entire new copy of IRAF on a machine separate from the normal production installation, so that one can experiment at will without affecting the standard system. An alternative which does not require duplicating the full system is to use the `IRAFULIB` environment variable. This can be used to safely experiment with custom changes to the IRAF system software outside the main system; IRAFULIB lets you define a private directory to be searched for IRAF global include files, libraries, executables, etc., allowing you to have your own private versions of any of these. See the system notes files for further information on how to use IRAFULIB.

4. Graphics and Image Display

IRAF is device and window system independent, and can be used with any windowing system such as X11, or with hardware graphics and display devices. Today most people will run IRAF on a UNIX workstation under some X11-based desktop such as CDE or *fvwm95*. The X11IRAF support package, which includes the *xgterm* and *ximtool* programs for graphics and imaging, is system independent and is distributed separately from IRAF. IRAF can also be used with other graphics and image display servers, e.g. *xterm*, *SAOtng*, and *SAOimage*. The *x11iraf* utilities are available from the IRAF network archives or by contacting IRAF site support.

Most people will prefer to use *xgterm* and *ximtool* (or a similar display tool such as *saoimage*) for IRAF graphics and imaging. *xgterm* is based on *xterm*, providing an equivalent vt100 (text window) capability but a much enhanced graphics capability. *ximtool* provides a general interactive image display capability, including support for multiple image frame buffers and frame blinking, independent zoom, pan, and color enhancement for each frame, and many other features. Both programs are implemented at the host level as general purpose window system tools, and are useful independently of IRAF. Detailed documentation on the basic operation and use of these programs is available with the X11IRAF distribution. Our concern in this document is with the use of these programs with IRAF.

4.1. The X11 environment

The graphics and image display tools provided with IRAF operate within the X11 windowing environment much like the standard tools provided with X11. To help illustrate the use of these tools, IRAF is distributed with a sample X11 environment already configured for the IRAF account, the exact nature of these files depends on the platform. This consists, for example on a Sun/IRAF system, of the following files in the IRAF account login directory, iraf$local.

`.Xdefaults`	Sets up the defaults for how the window system looks, e.g., defines the X resources controlling window colors, fonts, etc.
`.openwin-menu`	An example of a simple custom rootmenu for the OpenLook window manager, including entries for xgterm and ximtool. Other window managers will rely on a different configuration file, e.g. ".mwmrc" for Motif, ".twmrc" for the *twm* window manager, etc.
`.xinitrc`	Executed at window system start up time to create all the windows, some systems require that this file be named *.xsession*.

No one screen layout will suit all users or all applications. Everyone will wish to customize the workstation screen to suit their preferences and the type of work they are doing. However, the configuration provided works and should be useful as an example of how to make things function correctly.

4.2. Vector graphics capabilities

The standard graphics terminal emulator for IRAF under X11 is *xgterm*, which emulates a conventional dual plane text/graphics terminal. On systems to which xgterm has yet to be ported, such as VMS, *xterm* is typically used, this is an equivalent terminal emulator but the graphics support isn't quite as nice. This software terminal is driven via an ASCII datastream like a conventional hard terminal (except that the effective baud rate is much higher). The text window behaves like the system console and the graphics window behaves like a Tektronix 4012, plus some IRAF oriented extensions. Since xgterm emulates standard text and graphics devices non-IRAF programs can easily be run as well as IRAF programs.

Configuring IRAF to use xgterm is very simple. The following command does the job. This is normally executed by the login.cl or loginuser.cl file at login time.

```
cl> stty xgterm
```

Further information on xgterm may be found in the *xgterm.info* file in the IRAF network archive with the xgterm binaries or by contacting site support.

Xterm users can define the window type similarly, i.e.

```
cl> stty xterm            # or
cl> stty xtermjh
```

Since xterm lacks a true status line users may prefer the second example which puts status output on the text window instead of overwriting the graphics window.

4.3. Image Display capabilities

Image display for IRAF running in the X11 environment is provided by *XImtool* or a comparable IRAF-compatible display server (e.g. *SAOimage*). The current *XImtool* program provides a basic display capability, including programmed access from the IRAF environment to load images, interactive windowing of the display, pseudocolor, an interactive image cursor readback capability, zoom and pan, a variety of frame buffer sizes, independent frame buffer and display window sizing, up to four frames, each with its own state, and programmable frame blink. *ximtool* runs as a display server, meaning that it sits idle most of the time, waiting for some client, e.g., IRAF, to send it an image to be displayed via some form of interprocess communication.

To use ximtool from within IRAF one must define the logical device and enable image cursor input. For example,

```
cl> reset stdimage = imt512
```

would configure IRAF and ximtool for use with a 512 pixel square frame buffer (image display image memory). A variety of frame buffer sizes are predefined; see the `imtoolrc` file (normally in /usr/local/lib) for a complete list of possible configurations or use the IRAF *gdevices* command.

The image cursor is enabled by

```
    cl> reset stdimcur = stdimage
```

This is the default for Unix/IRAF. Setting `stdimcur` to "text" disables the image cursor, allowing cursor values to be typed in interactively in the terminal window. This is useful, for example, when running image oriented programs from a simple terminal.

The standard IRAF interface to the display server is the *display* program in the TV package. Automatic determination of the optimum intensity mapping to the 200 ximtool greylevels is provided. Entire frames can be displayed, or one can write to subregions of the display. Other programs useful with the image display include *imexamine*, used to interactively examine images under image cursor control, *imedit*, used to edit images using the display, and *tvmark*, used to write color graphics into a display frame.

The display server has the capability of displaying the cursor (mouse) position and pixel value in image pixel units as the mouse is moved about in the window. In addition, text file cursor lists can be generated and displayed, or the image cursor can be read interactively from within IRAF. The image cursor may be called up at any time by typing

```
    cl> =imcur
```

into the CL. Applications programs which read the interactive image cursor will do this automatically during program execution.

4.4. Using the workstation with a remote compute server

A common mode of operation with a workstation is to run IRAF under X11 directly on the workstation which runs IRAF, accessing files either on a local disk, or on a remote disk via a network interface (NFS, IRAFKS, etc.). It is also possible, however, to run X11 with xgterm and ximtool on the workstation, but run IRAF on a remote node, e.g., some powerful compute server such as a large Sun server, a large VAX, or a vector minisupercomputer or supercomputer, possibly quite some distance away. This is done by logging onto the workstation, starting up X11 and a *xgterm* window, logging onto the remote machine with *rlogin*, *telnet*, or whatever, and starting up IRAF on the remote node.

After IRAF comes up one need only type

```
    cl> stty xgterm
    cl> reset node = hostname!
```

to tell the remote IRAF that it is talking to a xgterm window and that the image display is on the network node *hostname*. The trailing exclamation point is required in V2.10.4 and later versions of IRAF to avoid interpretation of general environment variables as network logical node names. For this to work IRAF networking must be enabled between the two hosts (see §2.2.7). Alternatively, an inet socket may be used to connect to the ximtool directly by defining an `IMTDEV` environment variable. For example, suppose you are running IRAF on remote node but wish to display to an ximtool running on your workstation which is in a different network domain, to do this define something like

```
    % setenv IMTDEV inet:5137:foo.bar.edu
```

prior to logging into the CL. This overrides the normal display connection selection and tells IRAF to display to inet socket 5137 running on node "foo.bar.edu" (5137 is the default inet socket for ximtool). The advantage here is that one doesn't need to enable iraf networking for a host that may only temporarily be used.

In this mode one is effectively using the workstation as a sort of super terminal with powerful graphics and image display capabilities. One gets the best of both worlds, i.e., a state of the art user interface, and the compute power of a large machine. It matters little what operating system is used on the remote machine, so long as it also runs IRAF. Except for the details of the login sequence, operation is completely transparent; xgterm does not care whether the process it is talking to is on a local or remote node. Performance, e.g,. for image loads, is often *better* than when everything is run directly on the local node, due to the more powerful server.

5. 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. Useful documentation for writing graphcap entries is the GIO reference manual and the HELP pages for the *showcap* and *stty* tasks, information on creating new `graphcap` entries for each type of device is covered in §2.2.6.

5.1. 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. Assistance with interfacing new graphics terminals is available via the IRAF Hotline.

5.2. Graphics plotters

The current IRAF system comes with several graphics kernels used to drive graphics plotters. The standard plotter interface 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 or in our network archive /iraf/doc directory as "sgi.ms".

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. Many 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.

5.3. Image display devices

The standard image display facility for a Sun workstation running the MIT X or OpenWindows window system is the *ximtool* or *saoimage* display server. XImtool is available from the /iraf/x11iraf directory of the iraf.noao.edu ftp archives, SAOimage was developed for IRAF by SAO; distribution kits are available from the IRAF network archive /contrib directory.

Some interfaces for hardware image display devices are also available, although a general display interface is not yet included in the system. Only the IIS model 70 and 75 are current supported 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., ximtool and saoimage, 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.

6. Host System Requirements

Any modern host system capable of running UNIX should be capable of running IRAF as well. IRAF is supported on all the more popular UNIX platforms, as well as on other operating systems such as VMS.

A typical small system is a single workstation with a local disk. In a typical large installation there will be one or more large central compute servers, each with several Gb of disk and many Mb of RAM, networked to a number of personal or public workstations. For scientific use, a megapixel color screen is desirable.

6.1. Memory requirements

The windowing systems used in these workstations tend to be very memory intensive; the typical screen with ten or so windows uses a lot of memory. Interactive performance will suffer greatly if the system pages a lot. Fortunately, memory is becoming relatively cheap. No system, including personal diskless nodes, should be configured with less than 32 Mb of main memory; 48 Mb or more is recommended if you plan to do a lot of image processing. On servers, 64, 128 or even 256 Mb is not an unreasonable amount of memory to try to configure on the system.

6.2. 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, no disk 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.

6.3. Diskless nodes

For an application such as programming or word processing, 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 is desirable but not essential. For most IRAF applications however, where serious image processing is planned, 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

The main branches of the IRAF directory tree are summarized below. Beneath the directories shown are some 400 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:

`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 (SunView)

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.