Chapter 1

User Interface

During the prototype stage a basic command line user interface was build, with which tasks have been constructed. Some work was spend in showing that both the AIPS interpreter and the Graphical User Interface (GUI) are plug-in compatible user interfaces. For example, a functional GUI for Khoros[1] is available for demo purposes. The AIPS shell interpreter can be thought of in terms of the Miriad[2] shell interpreter.

1.1 Astronomers vs. Programmers

The basic (command line) user interface is a series of "*keyword=value*" pairs, which we call **program parameters**[3].

The `class Param` (see `Param.h`) implements one single such **parameter**. In addition to a name and a value, a parameter has a variety of other attributes, such as a one-line help string (useful when being prompted etc.), a type, a range and optional units. All of these are character strings; parsing and error checking is done at a different level. The programmer however will never interact with a parameter through it's class interface. This is done with the `class Input`, which is some kind of container of `Param`'s, with a variety of user interface attributes (help-level, message/debug-level etc).

Although the programmer must supply the user interface with a number of predefined **program parameters**, the user interface itself will create a small number of **system parameters** (help=, debug=). The purpose of these is to tell the task how to communicate with the user and it's environment, and give the user control over these items. For example, the user may want to be prompted, with error recovery, and see (debug) messages above a certain threshold level.

[1](c) University of New Mexico
[2](c) BIMA
[3]The name **parameter** and **keyword** are sometimes used both

For the benefit of the Programmer, the user interface also defines a number of standard parameters ("templates"), which can be copied and bound to a program parameter.

Parameter names are to be found by minimum match, if so requested by the user.

Most programs are probably happy with a simple set of parameters, like a linear list. We have discussed hierarchical keywords and in Section 1.3 a few thoughts are expressed.

All input as well as output is controlled by the user interface. The Astronomer has a varying degree of control over how and where input and output occurs. In the command line interface system control occurs through a small number of system parameters, which can be preset by environment variables, supplied as if they were parameters on the command line, or both.

For example, a interactive UNIX shell session may look like:

```
1% setenv DEBUG 1
2% setenv HELP prompt,aipsenv
3% program key1=val1 key3=val3
4% program val1 val2 key4=val4 key5=val5 debug=0
5% unsetenv HELP DEBUG
6% program help=pane > prog.pane
```

After having preset the DEBUG and HELP modes in commands `1%` and `2%`, commands `3%` and `4%` will act accordingly: the user is prompted, and parameter default values are restored and saved from an AIPS environment file before and after invocation. In addition, in command `4%` the user decided not to see any messages. Command `6%` gives an example of the self-describing mode of programs, where a pane description file for Khoros has been constructed.

1.2 Programmers: Where is my `main`?

No, we don't want you to use `main(int argc, char **argv)` anywhere in your code. Instead, use `aips_input()`, `aips_main()` and `aips_output()`.

To summary, your section of code could then look something like:

```
 //aips++
 //  Hypothetical Silly Interactive Contour Plotter
 //

 #include <Main.h>        // Standard declarations needed for an AIPS++ main program
 #include <SillyImage.h>

aips_input(Input &inputs) // Definition of the allowed Program Parameters
 {
   inputs.Version("19-mar-92 PJT");
   inputs.Usage("Hypothetical Silly Interactive Contour Plotter");

   inputs.Create(              "in",      "",      "Input file",        "InFile",  "r!");
   inputs.Create(              "levels",  "",      "Contour levels",    "RealArray");
   inputs.StdCreate("lstyle",  "lstyle",  "solid", "My Contour line type");
   inputs.StdCreate("lwidth");
   inputs.Create(              "annotate","full",  "What annotation?",  "String",  "full|brief|none|publication");
   inputs.StdCreate("device");
 }

 aips_main(Input &inputs) // Computation box - this could be spawned to various machines
 {
     String    dname     = inputs.GetString("device");
     Device device(dname);

     do {

         File      f         = inputs.GetFile("in");
         RealArray contours  = inputs.GetRealArray("levels");
         String    lstyle    = inputs.GetString("lstyle");
         Int       lwidth    = inputs.GetInt("lwidth");

         contours.Sort();       // Make sure this array is sorted

         if(contours.Count() > 20) cwarning << "A lot of contours buddy\n"
         if(countour.Count() == 0) break;

         cdebug.Level(1);
         cdebug << "Plotting " << contours.count() << " contours\n"
                << Level(2) << contours << "\n";

         SillyImageContour(f.name(),contours.Count(),contours.Value(),
                           lstyle, lwidth, dname);

     } while (inputs.More());

     device.Close();
 }
```

Comments:

- In `aips_input`, the **program parameters** are defined through the `Create` member function. In addition, a `Version` and `Usage` string should be supplied to the user interface.
- The `aips_input` routine could be automatally made by a code generator from a description section encoded in the source code of the program itself, much like Mark Calabretta's proposal discussed last fall. The advantage of this is that we can generate more elaborate online context and level dependant help. It should not be too hard to create readable documents in page description languages like man, latex or texinfo. The Andrew Toolkit, which has been considered too, is a different story.
- A number of standard `ostream`'s (`cwarning`, `cerror` and `cdebug`) are to be provided for[4], acting much like `cerr`; they handle warning messages, fatal error messages and a (Astronomer controlable message level) debug output. After a fatal error the program will exit gracefully. A specified number of fatal errors can be overridden by a system parameter (error=). The Programmer can also define a cleanup function, say `aips_cleanup`, which is called before the program really quits. Even a recover function could be supplied with which Programmers can recover from a known localized fatal error.
- Alternatively, variable argument (`<stdarg.h>`) versions of the above output could be made available under the names `error`, `warning` and `debug`:

  ```
  #include <stdarg.h>

  void error(char *fmt ...);
  void warning(char *fmt ...);
  void debug(int level, char *fmt ...);
  ```

- The `aips_main` function acts as a replacement for where C/C++ programmers commonly define their `main`. A true `main(int argc, char **argv)` is present in the AIPS library (See `Main.C`), and gets automatically linked in when you `#include <Main.h>`.
- An `Output` object has not been defined yet.
-

[4]Not present in this prototype

1.3 Heirarchical parameters

A hierarchical parameter would be set using the format

```
key.class1.class2.class3=value
```

(e.g. "*xaxis.grid.style=dotted*") we will use a notation where the hierarchical level is given by a the appropriate number of dots that the keyname starts with. To start with an example, a somewhat elaborate program which would clearly benefit from hierarchical keywords

```
<Key>           <StdKey>        <Type>      <Range>
====            ========        ======      =======

in              infile          InFile      r|w|w!|rw|.....
.region         xyzselect       String
contour                         bool        t|f
.levels                         RealArray   sort($0,$N)
.style          lstyle          String      solid|dotted|dashed|....
.thickness      lwidth          int         0:5
.color          color           String      cyan|red|green|0x134|....
greyscale                       bool        t|f
.minmax                         Real[2]     $1<$2
.gamma                          Real        >=0
.invert                         bool        t|f
.colormap       colormap        InFile      bw|rainbow|..
xaxis
.ticks                          Real[2]
.grid                           Real
..style         lstyle
..thickness     lwidth
.label                          String
..font          font            InFile      (calcomp|helvetica|roman)(10,12,15,20)
yaxis
.ticks                          Real[2]
.grid                           Real
..style         lstyle
..thickness     lwidth
.label                          String
..font          font            InFile      (helvetica|roman)(10,12,15,20)
annotate                        String      none|brief|full|publication
```

Comments/Problems:

- The order in which keywords are "created"[5] is still important, not only to properly define their hierarchy, but foremost to allow shortcuts with nameless specification of parameters on the command line. E.g. "`ccdplot ngc1365u 'box(10,10,20,20)' t 10:20:2 grey=t ann=full`" would be interpreted as `in=ngc1565u` etc. Obviously once a parameter was named, all subsequent ones need to be too (assuming the command line is parsed left to right).

[5]See `Input::Create()`

- `Range` must contain a boolean expression, where `$0` is the name of an array, `$N` the number of elements, `$1`, `$2`, `$3`, ... `$($N)` the array elements, `&` and `|` the boolean operators, `:` to denote an implied do-loop (with optional second `:` followed by the stride). A fairly rich syntax will be made available.
- `File` could be the same as a `String` but could also be usefull class (InFile and OutFile) in itself, with name, file pointer? and appropriate wildcard expansion of the string into the full filename.
- `xaxis,yaxis`: these two keywords are clearly related. In prompt mode it would be annoying if the Astronomer sat through the whole `xaxis` family, and then wants to do the `yaxis` tree with the defaults now inherited from the `xaxis` tree. (perhaps only the label name would be different (though the most appropriate default would be the one from the image header, if available). The programmer must leave the defaults in `yaxis` blank, and take the `xaxis` equivalent if none supplied in the `yaxis` equivalent.

1.4 Terminology/Glossary

program	Executable within the Unix environment, that has the AIPS user interface.
task	– same as above?
parameter	Has a name, value, help and all that other good stuff. They come as **program parameters** and **system parameters**, though a third kind, the **standard parameters**[6] are internally defined by the user interface. Programmers can bind **standard parameters** to `program parameters` at compile time.
keyword	The name of a parameter.
default	The value of a parameter as defined by `aips_input`, though possibly overriden by previous settings of the Astronomer if the user interface was told to (aipsenv file, commandline)

1.5 KHOROS User Interface

A small excersize to let Khoros talk to a simple user interface is presented here. The essential information needed by Khoros to run 'foreign' programs, is a User Interface Specification (UIS), also referred to as a **pane** file (each basic program in Khoros typically comes with a `program.pane` file).

Since an AIPS++ task is self-describing and can supply the caller with internal information about it's knowledge of keywords, it is relatively straightforward to automate this process and have each program create a `pane` file from itself.

For example,

[6]The name **template parameters** is perhaps more appropriate, but confusing in the C++ environment

```
    program help=pane > program.pane
```

where the `program.pane` could be something like:

```
-F 4.2 1 0 170x7+10+20 +35+1 'CANTATA for KHOROS' cantata
 -M 1 0 100x40+10+20 +23+1 'An AIPS++ program' aips++
  -P 1 0 80x38+22+2 +0+0 'View an image' view
  -I 1 0 0 1 0 1 50x1+2+2 +0+0 '???' 'in' 'Input Image' in
  -H 1 13x2+1+4 'Help' 'Help for view' aips.help
  -R 1 0 1 13x2+39+4 'Run' 'RunMe' khoros2aips view
  -E
 -E
-E
```

Currently most parameters are string parameters, except input and output files, which need to be labeled as such in the creation phase.

Comments/Problems:

- Array or composity parameters can only be implemented as strings, since Khoros cannot handle those (as far as I can see at this moment).
- Khoros does not allow for mixed (structured) keywords, unless structured via their mutually exclusive or inclusive tags.
- Though AIPS programs lend themselves to the dataflow based model, they may not like their existing files to be overwritten. By default, interconnecting Khoros programs output with another input creates a temporary (assumed re-usable) filename. It is up to the user to define a fixed name. This turned out to be very laborious and annoying for the user. The user can also decide to use shared memory or sockets, in which case this difference disappears. This will cause problems if datasets are implemented as directories (e.g. Miriad).
- Programs that are sending output to *stdout* or *stderr*, or for that matter, reading from *stdin* will get stuck. If one sticks to the load-and-go principle, like AIPS programs normally are, there is no problem with input. Output is another story. The *stdout* goes to the screen (whichever cantata was started up from), and *stderr* will appear in the error window that the glyph will pop up.
- The pane/subform files should be organized in the KHOROS tree, `$KHOROS_HOME/<toolbox>/repos/cantata/`
- The on-line documentation (referenced by the -H line on a pane file) comes in two options: single file and directory. If the name, referenced on the -H help line, is a directory, within that directory there can be several files which will be displayed. Khoros/Cantata comes with a program `formatdoc`, which converts an nroff-like document in the format that this help system wants (more or less ascii).

1.6 The KHOROS demo

```
cd /home/saturn/pteuben/bima
demo
```

This will start up `cantata`, with the demo predefined for you. The setup was saved in a file called `demo.workspace.Z`.

You can also manually load in new programs as follows:

- pull down the `Workspace` menu, and select the `File Utilities` option.
- Fill in the name of the relevant pane file in the `UIS Filename` entry and hit RETURN to import them into cantata's drawing space: drag the cursor into the drawing space of `cantata` to "drop" the glyph. The demo incoorporates pane files for the following programs: `observe.pane, calib.pane, invert.pane`, `clean.pane` and `view.pane`.

Some comments:

- programs are represented by a glyph, the white rectangles, with on the top a red (bomb), blue (form) and green (on/off) button. They represent a Quit, Edit and Run functionality. Typically a user selects the programs to run, opens a glyph by clicking on the middle/ blue (form) button, and fills out the necessary parameters.
- Input file's parameters inside the pane (visible after you've clicked on the blue button) come with a file browser to easy filename insertion.
- Parameters can use the program default by unsetting the option button. It normally comes as a filled black square, meaning the entered value will be the one used by the program.
- dataflow lines (the green lines) can be created by first clicking on an **out** (OutFile) arrow, followed by an **in** (InFile) arrow. An OutFile can be split (i.e. goto various InFile's), but of course one cannot merge various OutFile's into one InFile. Any previous connections would then be broken.
- dataflow lines can be files (default), sockets or shared memory. Intermediate results are typically lost, after you exit from Khoros, though can always be stored into a file at demand. Click the mouse while pointing at a dataflow line, and you'll see what they mean with this.
- With the RESET button (on the left) you can force the flow to be run all the way through for the next RUN. Ne sure all required inputs and outputs are connected, otherwise the flow will be interrupted. An inactive in/out can be recognized as a grey arrow, as opposed to a black arrow, inside the yellow rectangles.
- Saving a workspace can be done within the WORKSPACE/File Utilities menu option. The default demo is in `demo.workspace.Z`.

Chapter 2

documentation

2.1 hypertext

Available systems, mostly public domain:

- texinfo; various programs available are
  - info: screen (curses) based
  - xinfo: x-windows based
  - ivinfo: interviews based
  - info.el: emacs based
  - hyperbole (but see also below) - epoch / emacs.
  - hyperman
- xman:
- hman: needs Motif1.1.2
- hyperbole