The TypeAccess/CSS Manual

Version 3.0 (2 May 2003)

Chadley K. Dawson, Randall C. O'Reilly

Welcome to the TypeAccess/CSS Manual

This describes version 3.0 of the software.

Manual revision date: 2 May 2003.

1 Copyright Information

2 Introduction to TypeAccess/CSS

TypeAccess (TA) and CSS were developed in order to automate the interface to a neural-network simulation package called PDP++. This manual is simply a subset of the PDP++ software manual, and it has a number of references to PDP++ specific objects and issues. Just ignore these.

The TA/CSS system was designed to provide an automatic interface to C++ objects. The members and functions on the objects define what the user can do, and the GUI (graphical user interface) and script-level interface simply provides a way of accessing the members and methods of these objects. Also, a generic, automatic way of saving and loading objects and structures of objects to disk is provided.

The TypeAccess system makes this possible by automatically parsing the header files in an application directory and creating a record of the critical type information needed to make the GUI and script interfaces to these objects. This type information is also available for the programmer, as with all RTTI (run-time type information) systems.

Unlike most RTTI systems, TA also records the comments from the source code, which provides the prompts and "comment directives" for the automatic interface. The comment directives control how the member or function is presented to the user, making the interface quite versatile.

CSS is an interpreted script language that uses the C/C++ syntax. It implements most of C and much of the basics of C++. See Section 4.3.1 [css-c++-diff], page 27 and Section 4.3.2 [css-c-diff], page 27 for details on the discrepancies. It has a built-in command system for controlling the execution and debugging of programs.

CSS is implemented in C++ as a set of objects that basically know how to "run" themselves, convert themselves into other types, etc. TypeAccess can generate a set of "stub" functions for class member functions which take CSS object arguments, and return a CSS object. These stub functions provide the basis for the CSS interface to hard-coded classes, and for GUI buttons and pull-down menus which call member functions.

The same kind of automatic GUI is also available for classes defined in CSS. Thus, it is possible to create entire applications with automatic GUI interfaces without compiling a line of code. This same code can then be compiled with C++ when the application is stable enough to warrant compiling.

CSS can also be used as a stand-alone program like Perl or TCL. This provides the ability to write shell-like scripts in C/C++ instead of learning a complicated new language (use the complicated language you already know..). The basic version of CSS has the GNU libg++ String object as a basic type, as well as standard stream-based I/O, and all of the usual functions from '`unistd.h`' and '`math.h`', plus some additional special math functions. It would be fairly simple to parse a good C++ matrix library under TA/CSS (this is on the 'todo' list), or some classes that provide faster I/O primitives for doing specific file-processing tasks, network protocols, etc.

The distribution includes a number of low-level objects which provide transparent access to the TA type information, and basic container functions like List, Group, and Array, as well as a host of other object types that were used to implement the PDP++ software.

3 Installation Guide

This chapter provides a guide to installing the PDP++ software. There are two basic forms in which the software is distributed — the executable files only for use by an "end user", and the complete source code, for use by a "programmer" who will be compiling new additions to PDP++. We will refer to the executable files only distribution as the "end user's version", and the source code distribution as the "programmer's version", where the version refers to the manner in which the software is distributed, not to the software itself (it's all the same code).

For most systems, the end user's distribution is obtained in two parts, a tar file containing pre-compiled binaries for a particular system, and another which contains the manual, demos, default files, and other miscellaneous things. This is for the end user who will not need to compile new versions of the software to add new functionality to it. The relevant tar files are:

```
pdp++_version_bin_CPU.tar.gz
pdp++_version_ext.tar.gz
```

where *version* is the version number of the software release, and *CPU* is the cpu-type of the system you will be running on (see below).

For LINUX systems under RedHat or compatible distributions, an rpm version (with the '`.rpm`' extension) is available.

For MS Windows, a standard `setup` install file (with the '`.exe`' extension) is available.

For Mac OS-X, a standard "package" file (with the '`.pkg.sit`' extension) is available.

The currently supported *CPU* types (listed in rough order of level of support) are:

`LINUX` An Intel 386-Pentium machine running a modern glibc version of Linux (e.g., RedHat 7+).

`CYGWIN` An Intel 386-Pentium machine running the Windows operating system (using the Cygnus CygWin system as a compilation environment).

`DARWIN` A Mac running OS-X (aka Darwin), which is based on FreeBSD. Binaries are for the power PC (PPC) architecture.

`SUN4` A SUN sparc-station system running a modern 5.x Solaris version of the operating system.

`SGI` A Silicon Graphics workstation running a recent Irix 6.x release.

`HP800` A Hewlett Packard workstation running HP-UX version 10.x.

`IBMaix` An IBM RS/6000 machine running AIX v4.1.4 (4.1.x shouldwork)

If you don't have one of these machines, then you will have to compile the software from the source code using the programmers distribution.

The programmer's distribution is contained in one tar file that contains the source code along with the manual and other supporting files:

```
pdp++_version_src.tar.gz
```

These and any other files mentioned below can be obtained from our anonymous FTP servers:

```
CMU FTP Site:            ftp://cnbc.cmu.edu/pub/pdp++/
Colorado FTP Site:   ftp://grey.colorado.edu/pub/oreilly/pdp++/
European (UK) Mirror:    ftp://unix.hensa.ac.uk/mirrors/pdp++/
```

The Colorado site is updated most frequently.

3.1 Installing the End User's Version

After downloading the two tar files, '`pdp++_version_bin_CPU.tar.gz`' and '`pdp++_version_ext.tar.gz`', you need to decide where to locate the files. It is recommended that you put them in '`/usr/local/pdp++`', but they can be put anywhere. However, the `PDPDIR` environmental variable must then be set for all users to the location it is actually installed in. In addition if your CPU supports shared libraries (all unix versions, including LINUX, IBMaix, SUN4, HP800, SGI, but not DARWIN), you will need to insure that the `LD_LIBRARY_PATH` environment variable includes the path `PDPDIR/lib/CPU` where PDPDIR is the location of the pdp++ distribution, and CPU is your system type as described above (more details on this below). The following will assume that you are installing in '`/usr/local/pdp++`'.

Note: all of the PDP++ software is distributed in the gnu '`gzip`' format, and it also uses gzip to automatically compress and decompress the network, project, and environment files so that they take up less space on your disk. Thus, your system must have '`gzip`' installed before proceeding. It can be obtained from the GNU ftp server ('`gnudist.gnu.org`') or one of its mirrors, and is typically installed on most modern systems anyway.

Go to the '`/usr/local`' directory, and issue the following command:

```
gzip -dc <tarfile> | tar -xf -
```

or, on Linux or other systems having a gnutar program

```
tar -xzf <tarfile>
```

where <*tarfile*> is the name of the tar archive file. Note that the tar files will create the pdp++ directory, or load into it if it already exists. Thus, if you have an old version of the software, be sure to rename its directory something else before loading the new files.

LINUX users: There is a special '`.rpm`' file that will install the LINUX binaries and ext tar contents, including making links to the binaries in /usr/local/bin, and installing the libIV.so library (and links) in /usr/local/lib, and all of the ext-tra stuff in /usr/local/pdp++. To install this file, you need to be super-user, and then execute the following command:

```
rpm -Uvh pdp++-binext-VERSION.i386.rpm
```

Note that the PDP++ specific libraires are still installed in PDPDIR/lib/LINUX, so you still need to set the LD_LIBRARY_PATH to include this path.

Windows users (CYGWIN): There is a special '`.exe`' file that is an auto-installing executable distribution of both the bin and ext tar files described above. This should be used to install under windows. If you should install it in a location other than the default '`C:\PDP++`' directory, you should add a `set PDPDIR=path` in the '`C:\autoexec.bat`' file.

OS-X users: There is a special '`.pkg.sit`' file that is an auto-installing package file distribution of both the bin and ext tar files described above.

All further references to file names, unless otherwise stated, assume that you are in the PDPDIR directory (e.g., '`/usr/local/pdp++`').

The files will get loaded into the following directories:

'`bin`' binaries (executable files) go here

'`config`' configuration (for Makefile) and some standard init files are found here

'`css`' contains include files for commonly-used css scripts and some additional documentation, plus some demo script files

'`defaults`'
contains default configuration files for the various executables (see the manual for more information).

'`demo`' contains demonstrations of various aspects of the PDP++ software.

'`manual`' contains the manual, which is in texinfo format and has been made into a .ps, emacs .info, and html files.

'`src`' contains the source code for the software.

'`lib`' libraries (for dynamically linked executables) go here

'`interviews/lib`'
InterViews toolkit libraries (for dynamically linked executables) go here.

The binaries will get unloaded into '`bin/CPU`', where `CPU` is the system name as described above. The binaries are:

'`bp++`' The backpropagation executable (see ⟨undefined⟩ [bp], page ⟨undefined⟩).

'`cs++`' The constraint satisfaction executable (see ⟨undefined⟩ [cs], page ⟨undefined⟩).

'`so++`' The self-organizing learning executable (see ⟨undefined⟩ [so], page ⟨undefined⟩).

'`bpso++`' A combination of backpropagation and self-organization algorithms, so hybrid networks can be built.

'`leabra++`'
The Leabra algorithm developed by O'Reilly, which incorporates Hebbian and error-driven learning, together with a k-Winners-Take-All competitive activation function, into a single coherent framework, which is biologically based. See "Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain", by O'Reilly and Munakata, MIT Press, 2000 (September) and associated simulations Chapter 2 [intro], page 3 for details.

'`lstm++`' The long-short-term-memory algorithm by Hochreiter, Schmidhuber et al.

'`rns++`' The real-time neural simultation program developed by Josh Brown.

'`maketa`' The type-scanner used for programming the software. You can read about it in Chapter 5 [prog], page 50.

'`css`' A stand-alone version of the CSS script language system. It can be used as an interpreted C++ language system for any number of tasks.

You should either add the path to these binaries to your standard path, or make symbolic links to these files in '`/usr/local/bin`' or some similar place which most user's will have on their path already. For example, in a csh-like shell (e.g., in the '`~/.cshrc`' file that initializes this shell), add (for the LINUX CPU):

```
set path = (/usr/local/pdp++/bin/LINUX $path)
```

or to make the symbolic links, do:

```
cd /usr/local/bin
ln -s /usr/local/pdp++/bin/LINUX/* .
```

Configuring the Libraries

IMPORTANT: Most of the binaries are dynamically linked, which means that the '`pdp++_version_bin_CPU.tar.gz`' file installed some dynamic libraries in the '`PDPDIR/lib/CPU`' directory and in the '`PDPDIR/interviews/lib/CPU`' directory. When one of the PDP++ programs is run, it will need to know where to find these dynamic libraries. Thus you must set the `LD_LIBRARY_PATH` environmental variable (using `setenv` under csh/tcsh) to point to both of these locations. For example, under LINUX with the standard PDPDIR:

```
setenv LD_LIBRARY_PATH /usr/local/pdp++/lib/LINUX:/usr/local/pdp++/interviews/lib/LINU
```

It is a good idea to put this setting in your initialization file for your shell (i.e. '`~/.cshrc`').

It might be easier, especially if you want to use idraw or other programs available under interviews, to copy the '`PDPDIR/interviews/lib/CPU/libIVhines.so*`' (or .sl for HP800) file into your '`/usr/local/lib`' or somewhere else that is already on your dynamic linker's path.

ADDITIONAL STEPS FOR libIV: First, note that we are now using (as of version 3.0) the version of InterViews maintained by Michael Hines as part of the NEURON detailed neural simulation package. This library is now called '`libIVhines`'. The latest version should always be available at '`ftp://www.neuron.yale.edu/neuron/unix/`', with the current version being '`iv-15.tar.gz`' (also available on the PDP++ ftp sites).

The interviews library in '`PDPDIR/interviews/lib/CPU`' is called libIVhines.so.3.0.3 (or possibly other numbers instead of 3.0.3), but on several unix systems (including LINUX, SGI, SUN4) the linker also wants to see a libIVhines.so.3 and a libIVhines.so as different names for this same file. Therefore, you need to do the following in whatever directory you end up installing libIVhines.so.3.0.3 (even if you keep it in the original location, you need to do this extra step):

```
ln -s libIVhines.so.3.0.3 libIVhines.so.3
ln -s libIVhines.so.3.0.3 libIVhines.so
```

(replace 3.0.3 and .3 with the appropriate numbers for the libIVhines file you actually have). Note that the .rpm install under LINUX does this automatically.

Manual

The manual is distributed in several versions, including a postscript file that can be printed out for hard-copy, a set of "info" files that can be installed in your standard info file location and added to your '`dir`' file for reading info files in gnu emacs and other programs, and a directory called '`html`' which contains a large number of '`.html`' files that can be read with '`Netscape`', '`Mosaic`' or some other WWW program. Point your program at '`pdp-user_1.html`' for the chapter-level summary, or '`pdp-user_toc.html`' for the detailed table of contents.

Help Viewer Configuration

There is now a `Help` menu item on all of the objects (under the `Object` menu), which automatically pulls up the appropriate section of the '`html`' version of the manual, using '`netscape`' by default.

Under Windows (CYGWIN), the default help command is setup to use '`C:/Program Files/Internet Explorer/iexplorer.exe`' — if you prefer netscape or any other HTML browser, this should be changed to the full path to that executable.

Under Mac (DARWIN), the default help command is '`open -a \"Internet Explorer\" %s &`' – this should work for most systems, although you might want to use the new browser whatever it is called.

These defaults can be changed in the `Settings` menu of the `PDP++Root` object, see ⟨undefined⟩ [proj-defaults], page ⟨undefined⟩ and Section 6.6 [gui-settings], page 87 for details.

The latest version of the manual is also available on-line from:

`http://psych.colorado.edu/~oreilly/PDP++/PDP++.html`

MS Windows Configuration

Memory Configuration

The PDP++ executables are compiled using an environment called cygwin developed by Cygnus Solutions (now owned by Red Hat Software). The default cygwin configuration has an upper limit of 128MB, which should work for most simulations. However, you might want to increase this limit if you are exploring larger simulations.

Open regedit (or regedt32) and find the key HKEY_CURRENT_USER\Software\Cygnus Solutions\Cygwin\

If this does not exist, you must create a new Key called Cygnus Solutions, and then another within it called Cygwin.

Then, create a new DWORD value under this key called "heap_chunk_in_mb" that contains the maximum amount of memory (in Mb) your application needs (watch the hex/decimal toggle – you'll probably want to set it to decimal). For example if you wanted to set the memory limit to 256Mb, just enter 256. Exit and restart all cygwin applications (e.g., pdp++).

Taskbar Configuration

Because the simulator uses many windows, the windows taskbar often does not adequetly display the names of the windows. This can be remediated by dragging the top of the bar up, allowing more room for each icon. Another approach is to grab the taskbar, drag it to the right edge of the screen, drag the left edge to widen it, and then set it to auto hide (right click on the taskbar, and select Properties to expose this option, or go to the Start menu/Settings/Taskbar). When all this has been done, the window list can be exposed bringing the pointer to the right edge of the screen.

Mac OS-X Configuration

PDP++ depends on having an XWindows (X11) server running on your mac. Apple now has their own version of the XFree86 X11 server, which runs very smoothly under the standard OSX window manager. This is the recommended solution. Read more about it and download from:

`http://www.apple.com/macosx/x11/`

Basically, everything works just as under unix because this is a fairly standard unix setup after all is said and done. The manual should therefore provide all the info you need.

Other Configuration

See ⟨undefined⟩ [proj-defaults], page ⟨undefined⟩ for instructions on how to setup customized startup files if you want to change some of the default properties of the system.

Happy simulating!

3.2 Installing the Programmers Version

Read over the instructions for installing the end-user's version first. This assumes that you have unloaded the '`pdp++_version_src.tar.gz`' file in something like '`/usr/local/pdp++`'.

IMPORTANT: Whenever you are compiling, you need to have the environmental variable `CPU` set to reflect your machine type (see above). Other machine types can be found in the interviews '`config/InterViews/arch.def`' directory. These are (in addition to the above): `VAX`, `MIPSEL`, `SUN3`, `SUNi386`, `SUN`, `HP300`, `HP200`, `HP500`, `HP`, `ATT`, `APOLLO`, `SONY68`, `SONYmips`, `SONY`, `PEGASUS`, `M4330`, `MACII`, `CRAY`, `STELLAR`, `IBMi386`, `IBMrt`, `IBMr2`, `LUNA68`, `LUNA88`, `MIPSEB`, `MOTOROLA`, `X386`, `DGUX`, `CONVEX`, `stratus`, `ALPHA`.

The C++ compiler types that are supported are modern gnu g++/egcs compilers (anything released after 1999 seems to work), and proprietary system compilers, which we refer to as '`CC`' compilers.

There are a couple of other libraries that the PDP++ software depends on. These need to be made before PDP++ itself can be compiled. Please ensure that all of the following are installed properly:

1) The '`readline`' library, which will have already been installed if '`gdb`' or perhaps other gnu programs have been installed on your system (or if you are using Linux). Look for '`/usr/lib/libreadline.so`' or '`/usr/local/lib/libreadline.so`'. If it isn't there, then download a version of it from one of the gnu ftp server sites (e.g., '`gnudist.gnu.org`'), and compile and install the library.

2) If using g++, and not on a Linux-based system, you need to make the the '`libstdc++`' library in the libg++ distribution. **NOTE:** PDP++ now requires the `sstream` header file, which defines the `stringstream` class, which is a much improved replacement for the `strstream` class. **For g++ version 2.9x, this header file might be missing or broken (e.g., under RedHat 7.3, it is present but broken!).** A good version of this file is present in `PDPDIR/include/new_sstream_for_gcc_2.9x.h`, and should be installed in the appropriate location (e.g., for RH 7.3, do the following as root: `cp include/new_sstream_for_gcc_2.9x.h /usr/include/g++-3/sstream`.

I haven't done this install in a while so the following is likely out of date:. It seems that in the latest distribution of libg++ both of these are installed in '`/usr/local/lib`' automatically, but if they are not there, '`libiostream.so`' is made in the '`libio`' directory in the libg++ distribution (do a `make install` to get the properly installed or copy it yourself), and '`libstdc++.so`' is made in the '`libstdc++.so`' directory. CC/cfront users should have their iostream code linked in automatically via the standard C++ library that comes with the compiler.

3) Install the InterViews library, which provides the graphics toolkit used by PDP++. We have collaborated with Michael Hines, developer of the Neuron simulation system, in developing an improved version of the InterViews library – **You must install the Hines version of InterViews!**. The source code for this version is:

```
iv-15.tar.tz
```

available on our ftp servers, and the latest version should be available from Michael Hines' ftp site at: '`ftp://www.neuron.yale.edu/neuron/unix/`'. Some miscellaneous information about interviews can be found in the '`PDPDIR/lib/interviews`' directory.

An alternative to compiling interviews yourself is to download pre-compiled interviews libraries from us. These are available for the dominant form of compiler (CC or g++) for the platforms on which the binary distribution is available (see list above in Chapter 3 [inst], page 4) and are provided as '`pdp++_version_ivlib_CPU_CC.tar.gz`' for the CC compiler and '`..._g++.tar.gz`' for g++. The include files, which are necessary to use the libraries to compile PDP++, are in '`pdp++_version_ivinc.tar.gz`'. These create a directory called '`interviews`' when extracted, which means this should be done in the '`/usr/local`' directory so that the interviews directory is '`/usr/local/interviews`'. Alternatively, these can be installed elsewhere and the `IDIRS_EXTRA` and `LDIRS_EXTRA` makefile variables set to point to this directory (see below). We install ours in '`/usr/local/lib`' – follow the directions in the End Users's install version described above to do this.

4) Once you have the above libraries installed, the next step is to configure the makefiles for the type of compiler and system you have. These makefiles are located in '`PDPDIR/config`'. The actual makefiles in a given directory (e.g., '`src/bp/Makefile`' and '`src/bp/CPU/Makefile`') are made by combining several makefile components. '`config/Makefile.std`' has the standard rules for making various

things, '`config/Makefile.defs`' has the standard definitions for everything, and then '`config/Makefile.CPU`' overrides any of these definitions that need to be set differently for a different CPU type (i.e., "local" definitions). There are several sub-steps to this process as labeled with letters below:

a) If you have installed the software in a location other than '`/usr/local/pdp++`': You need to change the definition of `PDPDIR` in **both** the top-level Makefile ('`PDPDIR/Makefile`') and in the definitions makefile '`PDPDIR/config/Makefile.defs`'.

There are makefiles in '`PDPDIR/config`' for the supported CPU types listed above, and the two different supported compilers (g++, CC). The makefiles are named '`Makefile.CPU.cmplr`', where cmplr is either '`g++`' or '`CC`'. The actual makefile used during compiling for a given machine is the one called '`Makefile.CPU`', where CPU is the type of system you are compiling on (e.g., LINUX, SUN4, HP800, SGI, etc.). If you are compiling on a machine for which a standard makefile does not exist, copy one from a supported machine for the same type of compiler. Also, see the notes below about porting to a new type of machine.

b) Copy the appropriate '`Makefile.CPU.cmplr`' makefile (where cmplr is either g++ or CC depending on which compiler you are using), to '`Makefile.CPU`' (again, CPU is your machine type, not 'CPU'). For some architectures there is just one '`Makefile.CPU`' since only one type of compilier is currently supported. In this case you can just leave it as is.

c) You should put any specific "local" definitions or modifications to the makefiles in the '`config/Makefile.CPU`'. This will be included last in the actual makefiles, and any definitions appearing here will override the standard definitions. To see the various definitions that might affect compiling, look at '`Makefile.defs`', which contains all the "standard" definitions, along with descriptive comments. The following are items that you will typically have to pay attention to:

i) `IOS_INCLUDES` and `IOS_LIB`: In order to be able to access via the CSS script language the functions associated with the standard C++ iostream classes, the type-scanning program '`maketa`' needs to process the iostream header files: '`streambuf.h`', '`iostream.h`', '`fstream.h`', and '`strstream.h`'. These files are scanned in the '`src/ta`' directory, as part of the building of the type access library '`libtypea.a`'. These header files are different depending on the compiler being used. For CC compilers, the `IOS_INCLUDES` variable should be set to `CC-3.1`.

These header files are typically located in '`/usr/include/CC`', which is where the `CC-3.1` versions of these files in '`src/ta`' point to via symbolic links. Thus, if your headers are located elsewhere, you will need to change these symbolic links, or just copy the header files directly into the '`ios-CC-3.1`' subdirectory in '`src/ta`'. For g++ users, `IOS_INCLUDES` variable should be set to `g++-2.8.1` (for g++ 2.9x) or `g++-3.1` (for g++ 3.x). As of version 3.0, these g++ iostream headers are never actually included in the compile process itself, and are only scanned via the maketa program. Therefore, they have been dramatically edited to expose only the relevant interface components. As such g++-3.1 should work for all subsequent releases of g++ (hopefully!).

Note that you can use the `make force_ta` action to force a re-scan of the header files. A `make opt_lib` is then neccessary to compile this type information into the library. Finally,

the IOS_LIB variable should be blank by default for both g++ and C++ users (for newer g++), but for older g++ configurations it was necessary to set it to -liostream.

ii) IDIRS_EXTRA and LDIRS_EXTRA can be used to specify locations for other include and library files, respectively (for example, the cfront compiler may need to be told to look in -I/usr/include/CC for include files). Use these if you have installed any libraries (e.g., InterViews) in a non-standard location.

iii) MAKETA_FLAGS should be set to -hx -css -instances by default, but it is also often necessary to include the include path for the iostream and other C++ library files. For example, the LINUX makefile has the following: LIBG++_INCLUDE_DIR = -I/usr/include/linux -I/usr/include/g++-3, and then MAKETA_FLAGS = -hx -css -instances $(LIBG++_INCLUDE_DIR).

iv) cppC: this is the c-pre-processor, needed for the 'maketa' program to process header files. Although the system default preprocessor, usually installed in '/usr/lib/cpp', should work, 'maketa' was developed around the gnu cpp program, and so if you run into difficulties using the system cpp, install the gnu one (included as part of the gcc compiler). Note that make install of gcc/g++ does not apparently install this program by default, so you have to manually copy it from either your gcc compile directory or '/usr/lib/gcc-lib/<machine>/cpp'. You often need to include a define for the system architecture (e.g., -D__i386__ for Linux on intel chips, or -Dsparc for suns) in the cppC command.

v) VT_xxx and TI_xxx: these specify the virtual-table instantiation and template-instantiation (respectively) files, which are needed by different compilers. CC typically requires the VT_xxx files along with the +e[01] flags to only make one copy of virtual tables, while g++ requires the TI_xxx files to only make one copy of the templates. The TI_xxx files are included by default, so you will need to define them to be empty to override this default: TI_INST_SRC = , TI_INST_OBJ = , TI_INST_DEP = .

vi) Porting to a non-supported machine: There are a small set of system-dependent definitions contained in 'src/ta/ta_stdef.h', which are triggered by defines set up in the makefiles. NO_BUILTIN_BOOL should be defined if the c++ compiler does not have a bulitin bool type, which is the case with most cfront-based CC compilers, but not g++. CONST_48_ARGS determines if the seed48 and lcong48 functions take const arguments or not. In addition, different platforms may require different defines than those that are flagged in 'ta_stdef.h'. In this case, you will have to edit 'ta_stdef.h' directly. Please send any such additions, and the corresponding 'config/Makefile.CPU' along with any notes to us ('pdpadmin@crab.psy.cmu.edu') so we can put them on our web page for others to use, and incorporate them into subsequent releases.

vii) For some more information about the makefiles, see Section 5.1 [prog-make], page 50.

5) The standard makefiles use gnu's 'bison' instead of 'yacc' for making parsers. If you don't touch any of the .y files in the distribution, you won't need either. If you plan on messing around with the guts of the maketa type scanner or CSS, then you will probably want to install the latest version of 'bison'.

6) The dependency information, which is essential if you are going to be editing the main body of PDP++ code, but not necessary for a one-pass make of the system, is not made by the default make world action. If you want to make this dependency information, do it with

a `make depend` after a successful `make world`. Also, note that the automatic dependencies are made by calling '`gcc`' in the standard configuration. If your local C compiler supports the `-M` flag for generating dependency information, then this can be used instead. Just change the definition for `CC` in your '`Makefile.CPU`'. If you don't have gcc and your local C compiler doesn't support this, you can edit the end of the '`Makefile.std`' and change it to use the '`makedepend`' program, which we have not found to work as well, but it is an option.

7) On some systems, the standard '`make`' program is broken and will not work with our complex makfile system. This is true of the SUN4 system and IBMaix, and may be true of others. In this case, you will have to install the GNU make program, and use it to compile the software. If you get inexplicable errors about not being able to make certain things (seems to be the .d dependency files in particular that cause a problem), then try using GNU make (again, available at '`gnudist.gnu.org`' or mirrors).

8) If your CPU supports shared libraries (most do now), you will need to insure that the `LD_LIBRARY_PATH` environment variable includes the path `PDPDIR/lib/CPU` nad `PDPDIR/interviews/LIB/CPU` where PDPDIR is the location of the pdp++ distribution, and CPU is your system type as described above. **This is important even during the compile process, because the maketa program will need to access shared libs**.

After setting appropriate definitions, go back up to the PDPDIR and just do a:

```
make world
```

this should compile everything. This will make makefiles in each directory based on your CPU type, and then compile the various libraries and then the executables.

Most likely, the make will at least proceed past all the basic directory initialization stuff that is part of '`make world`'. Thus, if the compile stops after making the makefiles and after making the '`maketa`' program, you can fix the problem and re-start it by doing '`make all`' instead of '`make world`'.

If you run into difficulties during the compile process, the programming guide might contain some useful information for debugging what is going wrong: Chapter 5 [prog], page 50.

4 Guide to the Script Language (CSS)

CSS is the script language for the PDP++ software. It is a general-purpose language that allows one to do virtually anything that could be done by writing programs in C or C++ and compiling them. It provides full access to all the types and objects defined in PDP++, and allows you to call the "member functions" of these objects and set the values of their members variables.

There are two principal uses for the script language in running simulations. One is to automate the procedure of setting up and running a simulation. In this role, the script replaces commands that would otherwise be given through the graphical user interface (gui). The other is to extend the software without having to re-compile and re-link. Thus, one can write new kinds of procedures, statistics, etc. that are particular to a given simulation in the script language, and run them as if they were hard coded. This gives the PDP++ software more flexibility.

4.1 Introduction to CSS

CSS is a "C" language interpreter and script language (C Super-Script). The name derives from the fact that the language is C, written in C, so it is C to the C, or C^c, while at the same time being an excellent, even superlative scripting language. The actual syntax for CSS is somewhere in between C and C++, and the language is written entirely in C++. The major differences between CSS and C++ are listed in Section 4.3.1 [css-c++-diff], page 27, and the few discrepancies between CSS and C are noted in Section 4.3.2 [css-c-diff], page 27. Some of the convenient features of C++ that are incorporated into CSS are enumerated in Section 4.3.4 [css-c++-intro], page 29, for those unfamiliar with C++.

Being an interpreter, CSS presents the user with a prompt: `css>`. At this prompt, the user can enter any C expression, which will be evaluated immediately upon pressing Return, or enter a command. The commands operate much like a debugger (e.g., gdb, dbx), and allow control over the running, listing, and debugging of programs. In addition to the default "immediate mode" behavior of the system, it can behave like a compiler. For example, the command `define` will cause the system to enter a compiling mode (the prompt changes to `css\#`), where the C code that is subsequently entered is compiled into an intermediate machine code, but not run directly. Instead, it becomes a stored program that can be run many times.

Typically, one works with a program file (use the extension .css to indicate a css file) that is loaded and then run, instead of manually typing programs into the system using define mode. The command `load "filename.css"` will load a file and compile it, at which point it can be run. The command `run` will run the program from the beginning. `reload` will remove the existing compiled program and load (and re-compile) the previously loaded one from the disk file, which is handy when you are editing the program file and testing it out.

One can view the source code within CSS by using the `list` command, which takes (optional) line number and length arguments. Unlike C functions, which need to be called with the usual syntax of parentheses around the arguments, which are themselves comma

separated, and the whole thing is terminated by a semi-colon, the arguments to commands don't require the parentheses or the semi-colon, but do require thte commas.

One of the principle uses of CSS is as an interface for hard-coded C/C++ programs. There is a "TypeAccess" program that reads the header files for a given application and generates type information that can be used by CSS to automatically interface with the objects and types in the hard-coded world. CSS comes initially with some relevant hard-coded types from the standard C/C++ library, including a String class and the stream I/O classes, and various math and posix library functions.

4.2 Tutorial Example of Using CSS

This section provides a tutorial-style introduction to some of the features of the CSS language environment.

4.2.1 Running an Example Program in CSS

The following example will be used to illustrate the use of the CSS system:

```
// a function that decodes the number
String Decode_Number(float number) {
  String rval = "The number, " + number + " is: ";
  if(number < 0)
    rval += "negative";
  else if(number < .5)
    rval += "less than .5";
  else if(number < 1)
    rval += "less than 1";
  else
    rval += "greater than 1";
  return rval;
}

// decodes n_numbers randomly generated numbers
void Decode_Random(int n_numbers) {
  int i;
  for(i=0; i<n_numbers; i++) {
    float number = 4.0 * (drand48()-.5);
    String decode = Decode_Number(number);  // call our decoding function
    cout << i << "\t" << decode << "\n";    // c++ style output
  }
}
```

You can enter this code using a text editor (e.g., emacs or vi), or use the example code in '`css_example.css`' included with the PDP++ software in the '`css/include`' directory. Then, run the PDP++ software. At the prompt (which will vary depending on which executable you run — the example will use `css>`), type:

```
css> load "css_example.css"
```

and the prompt should return. If you made any typos, they might show up as a Syntax Error, and should be corrected in the text file, which can be re-loaded with:

```
css> reload
```

Loading the text file translates it into an internal "machine code" that actually implements CSS. This is like compiling the code in traditional C, but it does not write out an executable file. Instead, it keeps the machine code in memory, so that it can be run interactively by the user.

To ensure that CSS has loaded the text, you can list the program:

```
css> list

Listing of Program: css_example.css
1
2       // a function that decodes the number
3       String Decode_Number(float number) {
4         String rval = "The number, " + number + " is: ";
5         if(number < 0)
6           rval += "negative";
7         else if(number < .5)
8           rval += "less than .5";
9         else if(number < 1)
10          rval += "less than 1";
11        else
12          rval += "greater than 1";
13        return rval;
14      }
15
16      // decodes n_numbers randomly generated numbers
17      void Decode_Random(int n_numbers) {
18        int i;
19        for(i=0; i<n_numbers; i++) {
20          float number = 4.0 * (drand48()-.5);
21          String decode = Decode_Number(number);  // call decoder
css> list

Listing of Program: css_example.css
21          String decode = Decode_Number(number);  // call decoder
22          cout << i << "\t" << decode << "\n";    // c++ style output
23        }
24      }
26      list
```

Note that you have to type `list` in twice in order to see the whole program (by default, `list` only shows 20 lines of code). Also, notice that the list command itself shows up at the end of the program—this is because commands that are entered on the command line are actually compiled, run, and then deleted from the end of the program. Because `list`, when run, shows the current state of the code (i.e., before it has itself been deleted), it shows up at the bottom of the listing.

Now, let's try running the example:

```
css> run
css>
```

Nothing happens! This is because the code as written only defines functions, it does not actually call them. The program would have to have had some statements (i.e., function calls, etc) at the *top level* (i.e., not within the definition of a function) in order to do something when the `run` command is issued.

However, we can call the functions directly:

```
css> Decode_Random(5);
0       The number, -0.414141 is: negative
1       The number, 1.36194 is: greater than 1
2       The number, -0.586656 is: negative
3       The number, -0.213666 is: negative
4       The number, -0.725229 is: negative
css>
```

(of course, your output will vary depending on the random number generator). This illustrates the interactive nature of CSS — you can call any function with any argument, and it will execute it for you right then and there. This is especially useful for debugging functions individually. Thus, we can call the `Decode_Number` function directly with different numbers to make sure it handles the cases appropriately:

```
css> Decode_Number(.25);
css>
```

Notice, however, that there was no output. This is because the function simply returns a string, but does not print it out. There are several ways to print out results, but the easiest is probably to use the `print` command:

```
css> print Decode_Number(.25);
(String)  = The number, 0.25 is: less than .5
```

`print` gives you the type name, the variable name (which is blank in this case), and the value. To illustrate this, you can just declare a variable directly:

```
css> String foo = "a string";
css> print foo
(String) foo = a string
```

Compare this with the print *function* (not command) `printf`:

```
css> printf(Decode_Number(.25));
The number, 0.25 is: less than .5css>
```

which just gives you the value of the string (and does not automatically append a '\n' return at the end of the line). Finally, we can use C++ stream-based printing, which directs the return value from the function to print itself to the default current output `cout`:

```
css> cout << Decode_Number(.75) << "\n";
The number, 0.75 is: less than 1
css>
```

Note also that you can put any expression in the arguments to the function:

```
css> print Decode_Number(exp(cos(tan(.2) + .5) * PI)/ 20);
```

```
(String)  = The number, 0.549689 is: less than 1
```

In order to actually be able to run a script, we can add the following lines to the code by switching to `define` mode instead of the default interactive mode:

```
css> define
css# cout << Decode_Number(.75) << "\n";
css# Decode_Random(5);
css# exit
```

Note that we use `exit` to exit the `define` mode. You could also use the `EOF` character (*`ctrl-D`* on most systems) to exit this mode. To see that we have added to the program, list it from line 20 onwards:

```
css> list 20

Listing of Program: css_example.css
20          float number = 4.0 * (drand48()-.5);
21          String decode = Decode_Number(number);  // call decoder
22          cout << i << "\t" << decode << "\n";    // c++ style output
23        }
24      }
26      cout << Decode_Number(.75) << "\n";
27      Decode_Random(5);
28      exit
29      list 20
```

Now, when we `run` the program, we get:

```
css> run
The number, 0.75 is: less than 1
0       The number, 1.54571 is: greater than 1
1       The number, -1.93767 is: negative
2       The number, 0.336361 is: less than .5
3       The number, -1.36253 is: negative
4       The number, -0.465137 is: negative
css>
```

All of the C language rules about declaring or defining functions before they are called apply in CSS as well (in general, CSS obeys most of the same rules as C), so we could not have put those two extra lines of code in the example program before the functions themselves were defined. In general, this leads to a program layout consisting of various different functions, followed at the very end by one or two lines of code at the top-level which call the relevant function to start things off. If you are feeling traditional, you can call this function `main`, and it will look like a regular C/C++ program, except for the last line which calls the main function.

4.2.2 Debugging an Example Program in CSS

This section illustrates the use of the debugging facilities within CSS. We will use the example program from the previous section, with the addition of the two extra lines of code that were added in `define` mode. It is important to note that some important aspects of debugging like breakpoints can only be used when a program was started with the `run`

command. Thus, do not set breakpoints if you are going to be simply calling a function on the command line.

There are three primary debugging tools in CSS: **breakpoints**, **execution tracing**, and **single-stepping**. Because CSS is basically an interpreted system, all of the facilities for examining variables, the stack, source code, etc. which must be added into a real debugger come for "free" in CSS.

To illustrate, we will set a breakpoint in the `Decode_Number` function (assuming you have loaded this code already):

```
css> list Decode_Number
(String) Decode_Number((Real) number) {
4         String rval = "The number, " + number + " is: ";
5         if(number < 0)
6           rval += "negative";
7         else if(number < .5)
8           rval += "less than .5";
9         else if(number < 1)
10          rval += "less than 1";
11        else
12          rval += "greater than 1";
13        return rval;
14      }
css> setbp 5
css> showbp
Decode_Number   5         if(number < 0)
css>
```

Note that we first listed the function (referring to it by name), and then set a breakpoint with the `setbp` command at line number 5. We then confirmed this breakpoint with the `showbp` command.

Now, when we `run` the program, execution will stop at line 5, and we will be returned to the `css>` prompt:

```
css> run
2 css>
```

which has changed to `2 css>`, indicating that we are 2 levels deep into the execution of the program. To see where we are, `list` and `status` can be used:

```
2 css> list

Listing of Program: css_example.css
4         String rval = "The number, " + number + " is: ";
5         if(number < 0)
6           rval += "negative";
7         else if(number < .5)
8           rval += "less than .5";
9         else if(number < 1)
10          rval += "less than 1";
11        else
12          rval += "greater than 1";
```

```
13        return rval;
14      }
28      list
2 css> status

        Status of Program: css_example.css
curnt:  Decode_Number   src_ln: 5       pc:     9
debug:  0       step:   0       depth:  2       conts:  2
lines:  29      list:   28
State: shell:   1   run: 0  cont:   0  defn:   0  runlast: 0
run status:     Waiting
shell cmd:      None
2 css>
```

The `src_ln` of the status output tells us which source-code line we are at (and what function we are in). Here, we can use the interactive mode of css to determine the values of our variables:

```
2 css> print number
(Real) number = 0.75
2 css> print rval
(String) rval = The number, 0.75 is:
```

The values of all of the variables for the current "frame" (there is one frame for every call to a given function, or any expression appearing between the curly-brackets) can be viewed at once with the `frame` command:

```
2 css> frame

Elements of Spaces For Program: Decode_Number (frame = 1)

Elements of Space: Autos (3)
(String) _retv_this =                   (Real) number = 0.75
(String) rval = The number, 0.75 is:

Elements of Space: Stack (0)


Elements of Space: css_example.css.Static (3)
(String) Decode_Number((Real) number)   (void) Decode_Random((Int) n_numbers)
(String) foo = a string
```

Included in the frame information are the "automatic" variables (`Autos`), and the contents of the stack. To get information on previous frames in the execution sequence, use the command `trace`, which gives both a trace of processing and can give a dump of the entire stack up to this point if given a higher "trace level", which is an optional argument to the `trace` command. The default trace level of 0 just shows the frames, 1 also shows the stack, 2 also shows the auto variables, and 3 shows all variables. In addition, just the contents of the current stack can be viewed with the `stack` command (note that the saved stack is what is actually used by the program during execution).

In addition to viewing variables, it is possible to change their values:

```
2 css> number = 200;
2 css> print number
(Real) number = 200
```

Finally, to continue the program from where it was stopped by the breakpoint, use the `cont` command:

```
2 css> cont
The number, 0.75 is: greater than 1
5 css>
```

Since we changed the number after it was turned into a string for the `rval`, but before the `if` statements, we got the contradictory result printed above. Also, because the breakpoint is still in effect, the program has stopped due to the `Decode_Random` function calling the `Decode_Number` function. We can continue again, or we can disable the breakpoint first, and then continue.

```
5 css> cont
0       The number, 0.764017 is: less than 1
5 css> unsetbp 5
5 css> showbp
5 css> cont
1       The number, -1.76456 is: negative
2       The number, 1.59942 is: greater than 1
3       The number, -1.34582 is: negative
4       The number, -1.36371 is: negative
css>
```

Note that the breakpoint is referred to by the source-code line number.

Another way to debug is to get a trace of the running program. This can be done by setting the `debug` level, which can have a value from 0 (normal, quiet operation) through 4, with higher levels giving more detail than lower levels. For most users, levels 0 and 1 are the only ones needed, since the higher levels depend on understanding the guts of the CSS machine code. Debug level 1 shows the source-code line corresponding to the currently-executing code:

```
css> debug 1
css> run
31      run
3       String Decode_Number(float number) {
17      void Decode_Random(int n_numbers) {
26      cout << Decode_Number(.75) << "\n";
4         String rval = "The number, " + number + " is: ";
5         if(number < 0)
5         if(number < 0)
5         if(number < 0)
7         else if(number < .5)
7         else if(number < .5)
7         else if(number < .5)
9         else if(number < 1)
9         else if(number < 1)
9         else if(number < 1)
```

```
10          rval += "less than 1";
13        return rval;
26      cout << Decode_Number(.75) << "\n";
The number, 0.75 is: less than 1
27      Decode_Random(5);
18        int i;
19        for(i=0; i<n_numbers; i++) {
.
.
.
```

Since running proceeds from the top to the bottom, the definitions of the functions appear in the trace even though they do not really do anything. Also, some constructs like `if` and `for` result in multiple copies of the same source-code line being printed. This kind of trace is useful for seeing what branches of the code are being taken, etc.

The final kind of debugging is single-stepping, which is like having a breakpoint after every line of source code. Execution continues after each point by simply entering a blank line:

```
css> debug 0
31      debug 0
css> step 1
css> run
31      run
3       String Decode_Number(float number) {
1 css>
17      void Decode_Random(int n_numbers) {
1 css>
26      cout << Decode_Number(.75) << "\n";
4         String rval = "The number, " + number + " is: ";
2 css> print number
31      print number
(Real) number = 0.75
2 css> print rval
(String) rval = The number, 0.75 is:
5         if(number < 0)
3 css>
7         else if(number < .5)
4 css>
9         else if(number < 1)
5 css>
10          rval += "less than 1";
4 css>
13        return rval;
26      cout << Decode_Number(.75) << "\n";
The number, 0.75 is: less than 1
1 css>
27      Decode_Random(5);
18        int i;
2 css>
```

```
19        for(i=0; i<n_numbers; i++) {
19        for(i=0; i<n_numbers; i++) {
20          float number = 4.0 * (drand48()-.5);
4 css>
```

(note that we turned debugging off, since it is redundant with single-stepping). The `step` command takes an argument, which is the number of lines to step over (typically 1). Then, when we `run` the program again, it stops after every line. If you simply want to continue running, you can just hit return and it will continue to the next line.

If at any point during debugging you want to stop the execution of the program and return to the top-level (i.e., get rid of that number in front of the prompt), use the `restart` command.

```
4 css> restart
34      restart
css>
```

Be sure not to confuse `restart` with `reset`, as the latter will erase the current program from memory (you can just reload it with `reload`, so its not so bad if you do).

4.2.3 Accessing Hard-Coded Objects in CSS

In general, hard-coded objects (and their members and member functions) are treated just as they would be in C or C++. Objects that have been created (i.e., Networks, Units, etc) can be referred to by their *path* names, which start with the *root* object (PDPRoot). While you could type `root.projects[0]`, for example, to refer to the first project, it is easier to use the abbreviation of a preceding period to stand for `root`, resulting in: `.projects[0]`.

The following examples were performed on the XOR example project in Bp++. In order to examine the project in CSS, one could simply use the `print` command on the path of this object:

```
bp++> print .projects[0]
.projects[0] Proj (refn=1) {
  ta_Base*        owner           = .projects;
  String          name            = Proj;
  WinBase*        win_owner       = root;
  WinGeometry     win_pos         = {lft=4: bot=74: wd=535: ht=24: };
  WinGeometry     root_win_pos    = {lft=9: bot=79: wd=161: ht=23: };
  TypeDefault_MGroup   defaults          =  Size: 5 (TypeDefault);
  BaseSpec_MGroup   specs         =  Size: 3 (BaseSpec);
  Network_MGroup   networks       =  Size: 1 (Network);
  Environment_MGroup   environments       =  Size: 1 (Environment);
  Process_MGroup   processes      =  Size: 5 (SchedProcess);
  PDPLog_MGroup   logs            =  Size: 2 (TextLog);
  Script_MGroup   scripts         =  Size: 1 (Script);
}
```

The first network within this project would then be referred to as `.projects[0].networks[0]`:

```
bp++> print .projects[0].networks[0]
.projects[0].networks[0] XOR (refn=15) {
```

```
  ta_Base*        owner           = .projects[0].networks;
  String          name            = XOR;
  WinBase*        win_owner       = .projects[0];
  WinGeometry     win_pos         = {lft=4: bot=3: wd=536: ht=390: };
  WinView_MGroup   views          =  Size: 1 (NetView);
  Layer_MGroup    layers          =  Size: 3 (Layer);
  Project*        proj            = .projects[0];
  TDGeometry      pos             = {x=0: y=0: z=0: };
  TDGeometry      max_size        = {x=2: y=2: z=3: };
  int             epoch           = 0;
  Network::Layer_Layout   lay_layout      = THREE_D;
}
```

You can also use a shortcut by just typing `.networks[0]`, which finds the first member with the name `networks` in a search starting at the root object and scanning down the *first branch* of the tree of objects (i.e., looking in the first element of every group along the way).

Scoped types such as `Network::Layer_Layout` which appear in the above class are referred to just as they would be in C++:

```
bp++> .networks[0].lay_layout = Network::TWO_D;
```

As you can see, setting the values of hard-coded object variables simply amounts to typing in the appropriate C/C++ statement.

Type information (obtained via the TypeAccess system) about hard-coded objects can be obtained with the `type` command:

```
bp++> type Network
class Network : PDPWinMgr : WinMgr : WinBase : ta_NBase : ta_Base {
// The Network

  // sub-types
  enum Layer_Layout {   // Visual mode of layer position/view
    TWO_D            = 0;       // all z = 0, no skew
    THREE_D          = 1;       // z = layer index, default skew
  }

  // members
  ta_Base*        owner;          //   pointer to owner
  String          name;           //   name of the object
  .
  .
  TDGeometry      max_size;       //   max size in each dim
  int             epoch;          //   epoch counter
  Network::Layer_Layout   lay_layout;     // Visual mode of layer

  // functions
  void            UnSafeCopy(ta_Base* na);
  ta_Base*        GetOwner(TypeDef* tp);
  .
  .
  .
```

```
  void            InitWtState();        // Initialize the weights
  .
  .
  void            Compute_dWt();        // update weights for whole net
  void            Copy_Weights(const Network* src);
  void            Enforce_Layout(Network::Layer_Layout layout_type);
}
```

This shows the inheritance of this object, any sub-types that are defined within it, and all of its members and functions (including those it inherits from other classes).

In addition, there is Tab-completion for path names and types in the CSS prompt-level script interface. Thus, as you are typing a path, if you hit the Tab key, it will try to complete the path. If there are multiple completions, hitting Tab twice will display them.

In order to call member functions of hard-coded classes, simply give the path to the object, followed by the member function, with any arguments that it might require (or none).

```
bp++> .networks[0].InitWtState();
bp++>
```

It is possible to create pointers to hard-coded objects. Simply declare a pointer variable with the appropriate type, and assign it to the given object by referring to its path:

```
bp++> Unit* un;
bp++> un = .networks[0].layers[1].units[0];
bp++> print un
.projects[0].networks[0].layers[1].units[0] hid_1 (refn=6) {
  ta_Base*        owner           = .projects[0].networks[0].layers[1].units;
  String          name            = hid_1;
  UnitSpec_SPtr   spec            = {type=BpUnitSpec: spec=.specs[0]: };
  TDGeometry      pos             = {x=0: y=0: z=0: };
  Unit::ExtType   ext_flag        = NO_EXTERNAL;
  float           targ            = 0;
  float           ext             = 0;
  float           act             = 0;
  float           net             = 0;
  Con_Group       recv            =  Size: 0.1.2 (BpCon);
  Con_Group       send            =  Size: 0.1.1 (BpCon);
  BpCon           bias            = BpCon;
  float           err             = 0;
  float           dEdA            = 0;
  float           dEdNet          = 0;
}
```

There are two ways to create new hard-coded objects. The preferred way is to call one of the `New` functions on the group-like objects (`List` or `Group`, see Section 7.2 [obj-group], page 97), which will create the object and add it to the group, so that it can be referred to by its path as just described.

```
bp++> .layers[1].units.List();

Elements of List:  (2)
```

```
hid_1   hid_2
bp++> .layers[1].units.New(1);
bp++> .layers[1].units.List();

Elements of List:  (3)
hid_1   hid_2
bp++> .layers[1].units[2].name = "new_guy";
bp++> .layers[1].units.List();

Elements of List:  (3)
hid_1   hid_2   new_guy
```

Finally, it is possible to create new instances of hard-coded object types through the C++ `new` operator, which is especially useful in order to take advantage of some of the handy built-in types like arrays (see Section 7.3 [obj-array], page 102):

```
bp++> float_RArray* ar = new float_RArray;
bp++> print ar
[0];

bp++> ar.Add(.25);
bp++> ar.Add(.55);
bp++> ar.Add(.11);
bp++> print ar
[3] 0.25 0.55 0.11;

bp++> print ar.Mean();
(Real)  = 0.303333
bp++> print ar.Var();
(Real)  = 0.101067
bp++> ar.Sort();
bp++> print ar
[3] 0.11 0.25 0.55;
```

Remember to `delete` those objects which you have created in this fashion:

```
bp++> delete ar;
bp++> print ar
(float_RArray) ar = 0
```

(the `ar = 0` means that it is a null pointer). Be sure *not* to use the `delete` operator on those objects which were created with the group's `New` function, which should be `Removed` from the group, not deleted directly (see Section 7.2 [obj-group], page 97).

4.3 CSS For C/C++ Programmers

This section outlines the ways in which CSS differs from C and C++. These differences have been kept as small as possible, but nonetheless the fact that CSS is an interactive scripting language means that it will inevitably differ from compiled versions of the language in some respects.

Note that this manual does not contain an exhaustive description of the CSS language — only differences from standard C/C++. Thus, it is expected that the user who is unfamiliar with these languages will purchase one of the hundreds of books on these languages, and then consult this section to find out where CSS differs.

4.3.1 Differences Between CSS and C++

The primary difference between CSS and C++ is that CSS does not support the complex set of rules which determine which of a set of functions with the same name should be called given a particular set of arguments. Thus, CSS does not allow multiple functions with the same name. By avoiding the function-call resolution problems, CSS is much faster and smaller than it otherwise would be.

One consequence of the lack of name resolution is that only default (no argument) constructors can be defined. This also obviates the need for the special parent-constructor calling syntax.

Also, at the present time, CSS does not support the definition of `operator` member functions that redefine the operation of the various arithmetic operators. Further, it does not provide access control via the `private`, `public`, and `protected` keywords, or the `const` type control, though these are parsed (and ignored). Thus, it also does not deal with the `friend` constructs either. While these language features could be added, they do not make a great deal of sense for the interactive script-level programming that CSS is designed to handle.

CSS *does* support multiple inheritance, and the overloading of derived member functions. It does not support the inlining of functions, which is a compiler-level optimization anyway. The keyword `inline` will be parsed, but ignored.

Also, note that CSS gives one access to hard-coded classes and types (via TypeAccess), in addition to those defined in the script.

Templates are not supported for script-based classes, but are supported in hard-coded classes (via TypeAccess).

Exception handling is not supported, and probably will not be.

Since the primary use of CSS is for relatively simple pieces of code that glue together more complex hard-coded objects, and not implementing large programs, the present limitations of CSS will probably not affect most users.

4.3.2 Differences Between CSS and ANSI C

The design specification for CSS should be the same as ANSI C. However, at the present time, some features have not been implemented, including:

The full functionality of `#define` pre-processor macros is not supported. At this point, things are either defined or undefined and the `#ifdef` and `#ifndef` functions can be used to selectively include or not different parts of code.

The promotion of types in arithmetic expressions is not standard. In CSS, the result of an expression is determined by the left-most (first) value in the expression. Thus, `20 / 3.0` would result in an integer value of 6, whereas `20.0 / 3` gives a real value of 6.6667.

The guarantee that only as much of a logical expression will be evaluated as is necessary is not implemented in CSS. Thus, something like '`if((a == NULL) || (a->memb == "whatever"))`' will (unfortunately) not work as it would in C, since the second expression will be evaluated even when `a` is `NULL`.

Initialization of multiple variables of the same type in the same statement, is not supported, for example:

```
int var1 = 20, var2 = 10, var3 = 15;
```

and initialization of an array must occur after it has been declared (see example below).

There are a limited number of primitive types in CSS, with the others defined in C being equivalent to one of these basic CSS types (see Section 4.4.5 [css-types], page 34). This is because everything in CSS is actually represented by an object, so the storage-level differences between many different C types are irrelevant in the script language.

Pointers in CSS are restricted to point to an object which lies in an array somewhere, or is a single entity. The pointer is "smart", and it is impossible to increment a pointer that points to a single entity, which makes all pointer arithmatic safe:

```
css> int xxx;
css> int* xp = &xxx;
css> print xp
(Int)* xp --> (Int) xxx = 0
css> xp++;
Cannot modify a NULL or non-array pointer value
>1      xp++;
css> print *(xp +1);
Cannot modify a NULL or non-array pointer value
>1      print *(xp +1);
```

but it is possible to increment and perform arithmetic on a pointer when it points to an array:

```
css> int xar[10];
css> xp = xar;
css> xar = {0,1,2,3,4,5,6,7,8,9};
css> print xar
(Int) xar[10] {
0       1       2       3       4       5       6       7       8       9
}
css> print xp
(Int)* xp --> (Int) xar[10] {
0       1       2       3       4       5       6       7       8       9
}
css> print *(xp+2);
(Int)  = 2
css> print *(xp+3);
(Int)  = 3
css> xp++;
css> print *(xp+3);
(Int)  = 4
css> print *(xp+9);
```

```
Array bounds exceeded
>1      p *(xp+9);
(void) Void
css> print *(xp+8);
(Int)  = 9
```

The error that occurred when the `print *(xp+9)` command was issued (after `xp` already points to `xar[1]`), illustrates the kind of "smart pointers" that are built into CSS, which prevent crashes by preventing access to the wrong memory areas.

4.3.3 Extensions Available in CSS

There are several "extensions" of the C/C++ language available in CSS. The most obvious one is the ability to shorten the path to refer to a particular hard-coded object by skipping those elements that are the first of a given group. Thus, if referring to a unit in the first layer of a network, in the first project, one can say: `.units[x]` instead of `.projects[0].networks[0].layers[0].units[x]`.

Also, one can often avoid the use of a type specifier when initializing a new variable, because CSS can figure out the type of the variable from the type of the initializing expression:

```
var1 = "a string initializer";          // type is inferred from initializer
//        instead of
String var2 = "a string initializer";   // instead of explicitly declared
```

When referring to a hard-coded type, CSS will automatically use the actual type of the object if the object derives from the `ta_Base` class which is aware of its own type information.

CSS does not pay attention to the distinction between `ptr->mbr` and `obj.mbr`. It knows if the object in question is an object itself or a pointer to an object, and can figure out how to access the members appropriately.

All of the basic CSS types (see Section 4.4.5 [css-types], page 34) know how to convert themselves into the other types automatically, without even casting them. However, you can use an explicit cast if you want the code to compile properly in standard C/C++.

Also, all script variables for hard-coded objects are implicitly pointers, even if not declared as such. This is because script objects are not the same thing as hard-coded ones, and can only act at best as reference variables for them (i.e., essentially as pointers, but you can use the `obj.mbr` notation, see previous paragraph). Thus, while you can write CSS code that would also compile as C++ code by using `ptr->mbr` notation to refer to hard-coded objects, you can also cheat and use the simpler `obj.mbr` notation even when `obj` is a pointer.

4.3.4 Features of C++ for C Programmers

Since more users are likely to be familiar with C than C++, this section provides a very brief introduction to the essentials of C++.

The central feature of C++ is that it extends the `struct` construct of C into a full-fledged object oriented programming (OOP) language based around a `class`. Thus, a class

or object has data members, like a `struct` in C, but it also has functions associated with it. These *member functions* or *methods* perform various functions associated with the data members, and together the whole thing ends up encapsulating a set of related tasks or operations together in a single entity.

The object-oriented notion is very intuitive for neural-network entities like units:

```
class Unit {    // this defines an object of type Unit
public:         // public means that any other object can access the following

  // these are the data members, they are stored on the object
  float         net_input;      // this gets computed by the weights, etc.
  float         activation;     // this is computed by the function below
  float         target;         // this is set by the environment before hand
  float         error;          // this gets computed by the function below

  // these are the member functions, they can easily access the data members
  // associated with this object
  virtual void  Compute_Activation()
  { activation = 1.0 / (1.0 + exp(net_input)); }
  virtual void  Compute_Error()
  { error = target - activation; error *= error; }
};
```

The member functions encapsulate some of the functionality of the unit by allowing the unit to update itself by calling its various member functions. This is convenient because different types of units might have different *definitions* of these functions, but they all would have a `Compute_Activation()` function that would perform this same basic operation. Thus, if you have an object which is an instance or token of the type `Unit`, you can set its data members and call its member functions as follows:

```
Unit un;
un.net_input = 2.5;
un.target = 1.0;
un.Compute_Activation();
un.Compute_Error();
```

Thus, you use the same basic notation to access data members as member functions (i.e., the `obj.mbr` syntax). To illustrate why this encapsulation of functions with objects is useful, we can imagine defining a new object type that is just like a unit but has a different activation function.

```
class TanhUnit : public Unit {          // we're going to inherit from a Unit
public:
  void          Compute_Activation()    { activation = tanh(net_input); }
};
```

This notation means that the new type, `TanhUnit`, is just like a `Unit`, except that it has a different version of the `Compute_Activation()` function. This is an example of *inheritance*, which is central to C++ and OOP in general. Thus, if we have something which we know to be a unit of some type (either a `Unit` or a `TanhUnit`, or maybe something else derived from a `Unit`), we can still call the `Compute_Activation()` function and expect it to do the right thing:

```
Unit* un;
un->Compute_Activation();
```

This is an example of a *virtual member function*, because the actual function called depends on what actual type of unit you have. This is why the original definition of the `Compute_Activation()` function has the `virtual` keyword in front of it. Virtual functions are an essential part of C++, as they make it possible to have many different definitions or "flavors" of a given operation. Since these differences are all encapsulated within a standard set of virtual functions, other objects do not need to have special-case code to deal with these differences. Thus, a very general purpose routine can be written which calls all of the `Compute_Activation()` functions on all of the units in a network, and this code will work regardless of what actual type of units are in the network, as long as they derive from the *base type* of `Unit`.

While there are a number of other features of C++, the PDP++ software mainly makes use of the basics just described. There are a couple of basic object types that are used in the software for doing file input/output, and for representing character-string values.

The C++ way of doing file input/output is via the *stream* concept. There is an object that represents the file, called a stream object. There are different flavors of stream objects depending on whether you are doing input (`istream`), output (`ostream`) or both (`iostream`). To actually open and close a file on a disk, there is a version of the `iostream` called an `fstream` that has functions allowing you to open and close files. To send stuff to or read stuff from a file, you use something like the "pipe" or i/o redirection concept from the standard Unix shells. The following example illustrates these concepts:

```
fstream fstrm;     // fstrm is a file stream, which can do input or output

// we are opening the file by calling a member function on the
// fstream object. the enumerated type ios::out means 'output'
// also available are ios::in, ios::app, etc.

fstrm.open("file.name", ios::out);

// we "pipe" stuff to the fstrm with the << operator, which can deal
// with all different types of things (ints, floats, strings, etc.)

fstrm << "this is some text " << 10 << 3.1415 << "\n";

fstrm.close();     // again, the file is closed with a member fun
```

The mode in which the file is opened is specified by an *enumerated type* or an `enum`. This provides a way of giving a descriptive name to particular integer values, and it replaces the use of string-valued arguments like "r" and "w" that were used in the `open()` function of the standard C library. The base class of all the stream types is something called `ios`, and the enum for the different modes a file can be opened in are defined in that type, which is why they are *scoped* to the `ios` class by the `ios::` syntax. The definition of this enum in ios is as follows:

```
enum open_mode  { in=1, out=2, ate=4, app=010, trunc=020,
                        nocreate=040, noreplace=0100 };
```

which shows that each enum value defines a bit which can be combined with others to affect how the file is opened.

The following example illustrates how file input works with streams. One simply uses the >> operator instead of <<. Note that the fstream has to be opened in `ios::in` mode as well:

```
fstream fstrm;     // fstrm is a file stream, input or output

// we are opening the file by calling a member function on the
// fstream object. the enumerated type ios::out means 'output'
// also available are ios::in, ios::app, etc.

fstrm.open("file.name", ios::in);

// these variables will hold stuff that is sucked in from the stream
String words[4];
int number;
float pi;
// this assumes that the stream was written by the output example
// given previously
fstrm >> words[0] >> words[1] >> words[2] >> words[3] >> number >> pi;

fstrm.close();     // again, the file is closed with a member fun
```

4.4 CSS Reference Information

This section of the manual contains reference material on many aspects of CSS (excluding the basic C/C++ language itself, for which many excellent reference sources exist).

4.4.1 Name and Storage Spaces in CSS

Like C and C++, CSS has several different places that it can put variables and functions. Some of these amount to ways of organizing things that have a similar role in the same list, so that they can all be viewed together. These are essentially transparent to the user, and are described in the context of the various commands that print out lists of various types of functions and variables. The remaining distinctions have consequences for the programmer, and are described below.

The initial program space in CSS is different than that of C or C++, since it can actually contain executable code, whereas the compiled languages restrict the basic top-level space to consist of definitions only. Any definitions or variables declared in the top-level space are by default considered to be `static`, which means that they are visible only to other things within that program space. Other program spaces (such as those in a `Script` object or a `ScriptProcess`, `ScriptEnv`, etc.) do not have access to these variables or functions. However, the `extern` declaration will make a variable or function visible across any other program spaces.

Within a function, all variables declared as arguments and local variables are known as "autos", as they are automatically-allocated. These are stored locally on a kind of

stack within the same object that holds the code for the function, and a new set of them are allocated for each invocation of the function. The exception is for variables declared `static`, which are also stored on the function object, but the same one is used for all invocations of the function.

All new types that are declared (including enums and classes) are put in the global typespace, which is available to all program spaces.

4.4.2 Graphical Editing of CSS Classes

A class object defined within CSS can be edited using the `edit` command or the `EditObj` function (described below). Classes offer the ability to customize the edit dialog through the use of comment directives, which are the same as those used in the hard-coded C++ classes with the TypeAccess system. These allow class member functions to be associated with a button (using the #BUTTON directive) which, when pressed, calls the member function. By default functions are added to an "Actions" menu, but the #MENU_ON_menuname directive puts that function on a menu named "menuname". The full list of directives is given in Section 5.3 [prog-comdir], page 63.

4.4.3 CSS Startup options

The following startup arguments are interpreted by CSS:

`[-f|-file] <file>`
: Compile and execute the given file upon startup. The default is to then exit after execution, but this can be overridden by the `-i` flag.

`[-e|-exec] <code>`
: Compile and execute the given code upon startup. The code is passed as a single string argument, and should contain CSS code separated by semicolons. The default is to then exit after execution, but this can be overridden by the `-i` flag.

`[-i|-interactive]`
: If using `-f` or `-e`, CSS will go into interactive (prompt) mode after startup execution.

`-v[<number>]`
: Run CSS with the initial debug level set to given number (default 1)

`[-b|-bp] <line>`
: Set an initial breakpoint at the given line of code (only if using `-f`).

`[-gui]` Enable the graphical-user-interface to CSS class objects. This is not activated by default, since most uses of CSS don't involve the gui.

Any other arguments can be accessed by user script programs by the global variables `argv` (an array of strings) and `argc` (an int).

Note that it is possible to make a self-executing css program by putting the following on the very first line of the file:

```
#!/usr/local/pdp++/bin/SUN4/css -f
```

In addition to these arguments, CSS looks for a file named '`.cssinitrc`' in the user's home directory, which contains CSS code that is run when CSS is started. This is useful for setting various command `aliases`, and defining commonly-used `extern` functions. An example '`.cssinitrc`' file is located in '`config/std.cssinitrc`'.

4.4.4 The CSS Command Shell

The primary interface to CSS is via the command shell, which provides a prompt and allows the user to enter commands, etc. This shell has some useful features, including *editing*, *history*, and *completion*, all of which are provided by the GNU readline library.

The current line can be edited using a subset of the emacs editing commands, including Ctrl-f, Ctrl-b, Ctrl-a, Ctrl-e, etc.

A running history of everything that has been entered is kept, and can be accessed by using the Ctrl-p and Ctrl-n commands (previous and next, respectively). This makes it easy to repeat previous commands, especially when combined with the editing facility.

Completion occurs when the TAB key is pressed after some partially-entered expression, causing the shell to suggest a completion to the expression based on the part that has already been entered. If there is more than one possible completion, then the shell will beep, and pressing TAB a second time will produce a list of all of the possible completions. Completion is based on all of the keywords currently defined (including types, commands, functions, user-defined variables and functions, etc), except in the following two cases: If the expression starts with a ".", it is interpreted as a path, and the next segment of the path to either an object or a member function of an object is suggested by the shell. If the expression contains a scoping operator "::", then the completion will interpret whatever is in front of the scoping operator as a type name, and will suggest the possible members, subtypes, etc. of that type as completions.

4.4.5 Basic Types in CSS

There are six basic "primitive" types in CSS: `Int`, `Real`, `char`, `String`, `bool`, and `enum`. An `Int` is essentially an `int`, a `Real` is essentially a `double`, and a `String` is a C++, dynamically-allocating string object. A `char` is just an int object that will convert into its equivalen ASCII character instead of its numerical value when converted into a String. A `bool` has two values: `true` and `false`. An `enum` has defined enumeration values, and will convert a string representation of one of those values into the corresponding symbolic/numeric value.

All of the other flavors of `int` in C, including `short`, `long`, `unsigned`, etc, are all just defined to be the same as an `Int`. Similarly, a `float` and `double` are defined to be the same as a `Real`. The `String` object is the same as the one that comes with the GNU libg++ library (slightly modified), which provides all of the common string manipulation functions as member functions of the `String` object, and handles the allocation of memory for the string dynamically, etc.

In addition to the primitive types, there are reference, pointer, and array types. Also, there are both script-defined and TypeAccess-derived hard-coded `class` objects. Finally, there are types that point to hard-coded members of class objects, which correspond to all

of the basic C types, and, unlike the basic script types, are sensitive to the actual size of a `short` vs a `long`, etc.

Finally there is a special `SubShell` type, which allows one to have multiple compile spaces (cssProgSpace's) within css. Thus, one could `load` one program into the current shell, and use a SubShell to load another one without getting rid of the first. The two shells can communicate via `extern` objects, and they share the same type space. Typical usage is:

```
css> extern SubShell sub_shell;
css> chsh sub_shell
```

The first line defines the object (`extern` is recommended else it will be removed when the parent shell is recompiled), and the second switches to it. To exit from the sub shell, just use `exit` or *`ctrl-D`* (and have faith that when you answer 'y' you will be returned to the parent shell).

4.4.6 CSS Commands

The following are the commands available in CSS. Commands are a little bit "special" in that they are typically executed immediately, they can be called with arguments without using the parentheses required in normal C functions (though they can be used if desired), and they do not need to be terminated with a semicolon (note that this also means that commands cannot extend across more than one line). Also, they do not generate return values. In general, commands provide debugging and program management (loading, listing, etc) kinds of facilities.

`alias <cmd> <new_nm>`
: Gives a new name to an existing command. This is useful for defining shortcuts (e.g., `alias list ls`), but does not allow more complex functionality. For that, either define a new function, or use a pre-processor `#define` macro.

`chsh <script_path>`
: Switches the CSS interface to access the CSS script object pointed to by the given path. This is for hard-coded objects that have CSS script objects in them (of type `cssProgSpace`).

`clearall`
: Clears out everything from the current program space. This is like restarting the CSS shell, compared to `reset` which does not remove any variables defined at the top-level.

`commands`
: Shows a list of the currently available commands (including any aliases that have been defined, which will appear at the end of the list).

`constants`
: Shows a list of the pre-defined constants that have been defined in CSS. These are just like globally-defined `Int` and `Real` values, and thus they can be assigned to different values (though this is obviously not recommended).

`cont`
: Continues the execution of a program that was stopped either by a breakpoint or by single-stepping. To continue at a particular line in the code, use the `goto` command.

`debug <level>`
: Sets the debug level. Level 1 provides a trace of the source lines executed. Level 2 provides a more detailed, machine-level trace, and causes `list` to show the program at the machine level instead of at the usual source level. Levels greater than 2 provide increasing amounts of detail about the inner workings of CSS, which should not be relevant to most users.

`define`
: Toggles the mode where statements that are typed in become part of the current program to be executed later (define mode), as opposed the default (run mode) where statements are executed immediately after entering them.

`defines`
: Shows a list of all of the current `#define` pre-processor macros.

`edit <object> [<wait>]`
: If the GUI (graphical user interface) is active (i.e., by using `-gui` to start up CSS), `edit` will bring up a graphical edit dialog for the given object, which must be either a script-defined or hard-coded `class` object. The optional second argument, if `true`, will cause the system to wait for the user to close the edit dialog before continuing execution of the script.

`enums`
: Shows a list of all the current `enum` types. Note that most `enum` types are defined within a `class` scope, and can be found there by using the `type` command on the class type.

`exit`
: Exits from the program (CSS), or from another program space if `chsh` (or its GUI equivalent) was called.

`frame [<back>]`
: Shows the variables and their values associated with the current block or frame of processing. The optional argument gives the number of frames back from the current one to display. This is most relevant for debugging at a breakpoint, since otherwise there will only be a single, top-level frame to display.

`functions`
: Shows a list of all of the currently defined functions.

`globals`
: Shows a list of all of the currently defined global variables, including those in the script and hard-coded ones.

`goto <src_ln>`
: Continues execution at the given source line.

`help [<expr>]`
: Shows a short help message, including lists of commands and functions available. When passed argument (command, function, class, etc), provides help information for it.

`inherit <object_type>`
: Shows the inheritance path for the given object type.

`list [<start_ln> [<n_lns>]] [<function>]`
: Lists the program source (or machine code, if `debug` is 2 or greater), optionally starting at the given source line number, and continuing for either 20 lines (the initial default) or the number given by the second argument (which then

becomes the new default). Alternatively, a function name can be given, which will start the listing at the beginning of that function (even if the function is `extern`al and does not appear in a line-number based list). `list` with no arguments will resume where the last one left off.

`load <program_file>`
: Loads and compiles a new program from the given file.

`mallinfo`
: Generates a listing of the current `malloc` memory allocation statistics, including changes from the last time the command was called.

`print <expr>`
: Prints the results of the given expression (which can be any valid CSS expression), giving some type information and following with a new line (`\n`). This is useful for debugging, but not for printing values as part of an executing program.

`printr <object>`
: Prints an object and any of its sub-objects in a indented style output. This can be very long for objects near the top of the object hierarchy (i.e., the root object), so be careful!

`reload`
: Reloads the current program from the last file that was `load`ed. This is useful because you do not have to specify the program file when making a series of changes to a program.

`remove <var_name>`
: Removes given variable from whatever space it was defined in. This can be useful if a variable was defined accidentally or given the wrong name during interactive use.

`reset`
: Reset is like `clearall`, except that it does not remove any top-level variables that might have been defined. Neither of these commands will remove anything declared `extern`.

`restart`
: Resets the script to start at the beginning. This is useful if you want to stop execution of the program after a break point.

`run`
: Runs the script from the start (as opposed to `cont` which continues execution from the current location).

`setbp <src_ln>`
: Sets a breakpoint at the given source-code line. Execution of the program will break when it gets to that line, and you will be able to examine variables, etc.

`setout <ostream>`
: Sets the default output of CSS commands to the given stream. This can be used to redirect listings or program tracing output to a file.

`settings`
: Shows the current values of various system-level settings or parameters. These settings are all static members of the class `ta_Misc`, and can be set by using the scoped member name, for example: `ta_Misc::display_width = 90;`

`shell <"shell_cmd">`
: Executes the given Unix shell command (i.e., `shell "ls -lt"`).

`showbp` Shows a list of all currently defined breakpoints, and the source code line they point to.

`source <cmd_file>`
Loads a file which contains a series of commands or statements, which are executed exactly as if they were entered from the keyboard. Note that this is different than `load`ing a program, which merely compiles the program but does not execute it immediately thereafter. `source` uses run mode, while `load` uses define mode.

`stack` Displays the current contents of the stack. This can be useful for debugging.

`status` Displays a brief listing of various status parameters, such as current source line, depth, etc.

`step <step_n>`
Sets the single-step mode for program execution. The parameter is the number of lines to step through before pausing. A value of 0 turns off single stepping.

`tokens <obj_type>`
Lists the instances of the given object type which are known to have been created. Many object types do not register tokens, which will be indicated in the results of this command if applicable. It is possible to refer to the objects by their position in this list with the `Token` function, which can be a useful shortcut to using the object's path.

`trace [<level>]`
Displays a trace of the functions called up to the current one (i.e., as called from within a breakpoint). A trace level of 0 (the default) just gives function names, line numbers, and the source code for the function call, while level 1 adds stack information, level 2 adds stack and auto variable state information, and level 3 gives a complete dump of all available information.

`type <type_name>`
Gives type information about the given type. This includes full information about classes (both hard-coded and script-defined), including members, functions, scoped types (enums), etc.

`undo` This undoes the previous statement, when in `define` mode.

`unsetbp <src_ln>`
Removes a breakpoint associated with the given source-code line number.

4.4.7 CSS Functions

The following functions are built into CSS, and provide some of the basic functionality found in the standard C library. Note that the `String` and `stream` objects encapsulate many commonly-used C library functions, which have not in general been reproduced in CSS (with the exception of some of the file functions).

`Int access(String fname, int ac_type)`
This POSIX command determines if the given file name is accessible according to the ac_type argument, which should be some bitwise OR of the enums `R_OK W_OK X_OK F_OK`. Returns success and sets errno flag on failure.

`Real acos(Real x)`
: The arc-cosine (inverse cosine) – takes an X coordinate and returns the angle (in radians) such that cos(angle)=X.

`Real acosh(Real x)`
: The hyperbolic arc-cosine.

`Int alarm(int seconds)`
: Generate an alarm signal in the given number of seconds. Returns success and sets errno flag on failure.

`Real asin(Real x)`
: The arc-sine (inverse sine) – takes a Y coordinate and returns the angle (in radians) such that sin(angle)=Y.

`Real asinh(Real x)`
: The hyperbolic arc-sine.

`Real atan(Real x)`
: The arc-tangent (inverse tangent) – takes a Y/X slope and returns angle (in radians) such that tan(angle)=Y/X.

`Real atan2(Real y, Real x)`
: The arc-tangent (inverse tangent) – takes a Y/X slope and returns angle (in radians) such that tan(angle)=Y/X.

`Real atanh(Real x)`
: The hyperbolic arc-tangent.

`Real beta(Real z, Real w)`
: The Beta function.

`Real beta_i(Real a, Real b, Real x)`
: The incomplete Beta function.

`Real bico_ln(Int n, Int j)`
: The natural logarithm of the binomial coefficient "n choose j". The number of ways of choosing j items out of a set containing n elements:

```
/ n \        n!
|   |  = ------------
\ j /      k! (n-k)!
```

`Real binom_cum(Int n, Int j, Int p)`
: The cumulative binomial probability of getting j *or more* in n trials of probability p.

`Real binom_den(Int n, Int j, Real p)`
: The binomial probability density function for j "successes" in n trials, each with probability p of success.

```
                / n \   j       (n-j)
P(n,j,p) =      |   |  p  (1-p)
                \ j /
```

`Real binom_dev(Int n, Real p)`
: The binomial random deviate: produces an integer number of successes for a binomial distribution with p probability over n trials.

`Int CancelEditObj(obj)`
: Cancels the edit dialog for the given object that would have been opened by `EditObj`.

`Real ceil(Real x)`
: Rounds up the value to the next-highest integral value.

`int chdir(String dir_name)`
: Change the current directory to given argument. Returns success and sets errno flag on failure.

`Real chisq_p(Real X, Real v)`
: Gives the chi-squared statistic P(X^2 | v).

`Real chisq_q(Real X, Real v)`
: Gives the complement of the chi-squared statistic Q(X^2 | v).

`Int chown(String fname, int user, int group)`
: Changes the ownership of the given file to the given user and group numbers. Returns success and sets errno flag on failure.

`Real clock()`
: Returns processor time used by current process in seconds (with fractions expressed in decimals).

`Real cos(Real x)`
: The cosine of angle x (given in radians). Use `cos(x / DEG)` if x is in degrees.

`Real cosh(Real x)`
: The hyperbolic cosine of angle x.

`String ctermid()`
: Returns the character-id of the current terminal.

`String cuserid()`
: Returns the character-id of the current user.

`String_Array& Dir([String& dir_nm])`
: Fills an array with the names of all the files in the given directory (defaults to "." if no directory name is passed). The user should copy the array if they want to keep it around, since the one returned is just a pointer to an internal array object.

`Real drand48()`
: Returns a uniformly-distributed random number between 0 and 1.

`Int EditObj(<object>, [Int wait])`
: This is the function version of the `edit` command. If the GUI (graphical user interface) is active (i.e., by using `-gui` to start up CSS), edit will bring up a graphical edit dialog for the given object, which must be either a script-defined or hard-coded `class` object. The optional second argument, if TRUE, will

cause the system to wait for the user to close the edit dialog before continuing execution of the script.

`Real erf(Real x)`
: The error function, which provides an approximation to the integral of the normal distribution.

`Real erf_c(Real x)`
: The complement of the error function.

`Real exp(Real x)`
: The natural exponential (e to the power x).

`css* Extern(String& name)`
: Returns the object with the given name on the 'extern' variable list. This provides a mechanism for passing arbitrary (i.e., class objects) data across different name spaces (i.e., across different instances of the css program space), since you can pass the name of the extern class object that contains data relevant to another script, and use this function to get that object from its name.

`Real fabs(Real x)`
: The absolute value of x.

`Real fact_ln(Int x)`
: The natural logarithm of the factorial of x (x!).

`void fclose(FILE fh)`
: Closes the file, which was opened by `fopen`. The FILE type is not actually a standard C FILE, but actually a `fstream` type, so stream operations can be performed on it.

`Real floor(Real x)`
: Rounds the value down to the next lowest integral value.

`Real fmod(Real x, Real y)`
: Returns the value of x modulo y (i.e., `x % y`) for floating-point values.

`FILE fopen(String& file_nm, String& mode)`
: Opens given file name in the given mode, where the modes are `"r"`, `"w"`, and `"a"` for read, write and append. The FILE type is not actually a standard C FILE, but actually a `fstream` type, so stream operations can be performed on it.

`void fprintf(FILE strm, v1 [,v2...])`
: Prints the given arguments (which must be comma separated) to the stream. Values to be printed can be of any type, and are actually printed with the `<<` operator of the stream classes. Unlike the standard C function, there is no provision for specifying formatting information. Instead, the formatting must be specified by changing the parameters of the stream object. The FILE type is not actually a standard C FILE, but actually a `fstream` type, so stream operations can be performed on it.

`Real Ftest_q(Real F, Real v1, Real v2)`

Gives the F probability distribution for P(F | (v1 < v2)). Useful for performing statistical significance tests. The _q suffix means that this is the complement distribution.

`Real gamma_cum(Int i, Real l, Real t)`

The cumulative gamma distribution for event i with parameters l=lambda and t=time, which is the same as `gamma_p(j, l * t)`.

`Real gamma_den(Int j, Real l, Real t)`

The gamma probability density function for j events, l=lambda, and t=time.

$$P(j,l,t) = \frac{l^j \, t^{(j-1)}}{j!} e^{-lt} \qquad (t > 0)$$

`Real gamma_dev(Int j)`

A random gamma deviate: how long it takes to wait until j events occur with a unit lambda (l=1).

`Real gamma_ln(Real z)`

The natural logarithm of the gamma function, which is a generalization of (n-1)! to real-valued arguments. Note that this is not the gamma probability distribution.

$$Gamma(z) = \int_0^x t^{z-1} e^{-t} \, dt$$

`Real gamma_p(Real a, Real x)`

The incomplete gamma function:

$$P(a,x) = \frac{1}{Gamma(a)} \int_0^x t^{a-1} e^{-t} \, dt \qquad (a > 0)$$

`Real gamma_q(Real a, Real x)`

The incomplete gamma function as the complement of `gamma_p`

$$P(a,x) = \frac{1}{Gamma(a)} \int_x^{inf} t^{a-1} e^{-t} \, dt \qquad (a > 0)$$

`Real gauss_cum(Real x)`

The cumulative of the Gaussian or normal distribution up to given x (sigma = 1, mean = 0).

`Real gauss_den(Real x)`
: The Gaussian or normal probability density function at x with sigma = 1 and mean = 0.

`Real gauss_inv(Real p)`
: Inverse of the cumulative for p: returns z value for given p.

`Real gauss_dev()`
: Returns a Gaussian random deviate with unit variance and 0 mean.

`String getcwd()`
: Returns the current working directory path.

`String getenv(String var)`
: Returns the environment variable definition for varable var.

`Int getegid()`
: Returns the current effective group id number for this process.

`Int geteuid()`
: Returns the current effective user id number for this process.

`Int getgid()`
: Returns the current group id number for this process.

`Int getuid()`
: Returns the current user id number for this process.

`String getlogin()`
: Returns the name the current user logged in as.

`Int getpgrp()`
: Returns the process group id for current process.

`Int getpid()`
: Returns the process id for current process.

`Int getppid()`
: Returns the parent process id for current process.

`Int gettimesec()`
: Returns current time of day in seconds.

`Int gettimmesec()`
: Returns current time of day in microseconds.

`Real hyperg(Int j, Int s, Int t, Int n)`
: The hypergeometric probability function for getting j number of the "target" items in an environment of size "n", where there are "t" targets and a sample (without replacement) of this environment of size "s" is taken.

`Int isatty()`
: Returns true if the current input terminal is a tty (as opposed to a file or a pipe or something else).

`Int link(String from, String to)`
: Creates a hard link from given file to other file. (see also symlink). Returns success and sets errno flag on failure.

`Real log(Real x)`
: The natural logarithm of x.

`Real log10(Real x)`
: The logarithm base 10 of x.

`Int lrand48()`
: Returns a uniformly-distributed random number on the range of the integers.

`MAX(<v1>, <v2>) or max(<v1>, <v2>)`
: Works like the commonly-used `#define` macro that gives the maximum of the two given arguments. The return type is that of the maximum-valued argument.

`MIN(<v1>, <v2>) or min(<vi>, <v2>)`
: Just like `MAX`, except it returns the minimum of the two given arguments.

`Int pause()`
: Pause (wait) until an alarm or other signal is received. Returns success and sets errno flag on failure.

`void perror(String prompt)`
: Prints out the current error message to stderr (cerr). The prompt argument is printed before the error message. Also, the global variable `errno` can be checked. Further, there is an include file in css/include called errno.css that defines an enumerated type for the defined values of errno.

`Real poisson_cum(Int j, Real l)`
: The cumulative Poisson distribution for getting 0 to j-1 events with an exected number of events of l (lambda).

`Real poisson_den(Int j, Real l)`
: The Poisson probability density function for j events given an expected number of events of l (lambda).

$$P(j, l) = \frac{l^j}{j!} e^{-l}$$

`Real poisson_dev(Real l)`
: A random Poisson deviate with a mean of l (lambda).

`Real pow(Real x, Real y)`
: Returns x to the y power. This can also be expressed in CSS as x ^ y.

`void PrintR(<object>)`
: This is the function version of the `printr` command. Prints an object and any of its sub-objects in a indented style output. This can be very long for objects near the top of the object hierarchy (i.e., the root object), so be careful!

`void printf(v1 [,v2...])`
: Prints the given arguments (which must be comma separated) to the standard output stream. Values to be printed can be of any type, and are actually printed with the `<<` operator of the stream classes. Unlike the standard C function, there is no provision for specifying formatting information. Instead,

the formatting must be specified by changing the parameters of the standard stream output object, `cout`. The FILE type is not actually a standard C FILE, but actually a `fstream` type, so stream operations can be performed on it.

`Int putenv(String env_val)`
: Put the environment value into the list of environment values (avail through getenv).

`Int random()`
: Returns a uniformly-distributed random number on the range of the integers. CSS actually uses the `lrand48` function to generate the number given the limitations of the standard `random` generator.

`String_Array& ReadLine(istream& strm)`
: Reads a line of data from the given stream, and returns a reference to an internal array (which is reused upon a subsequent call to ReadLine) of strings with elements containing the whitespace-delimited columns of the line. The size of the array gives the number of columns, etc. This allows one to easily implement much of the functionality of awk. See the file '`css_awk.css`' in '`css/include`' for an example.

`Int rename(String from, String to)`
: Renames given file. Returns success and sets errno flag on failure.

`Int rmdir(String dir_name)`
: Removes given directory. Returns success and sets errno flag on failure.

`Int setgid(Int id)`
: Sets group id for given process to that given. Note that only the super-user can in general do this. Returns success and sets errno flag on failure.

`Int setpgid(Int id)`
: Sets process group id for given process to that given. Note that only the super-user can in general do this. Returns success and sets errno flag on failure.

`Int setuid(Int id)`
: Sets user id for given process to that given. Note that only the super-user can in general do this. Returns success and sets errno flag on failure.

`Real sin(Real x)`
: The sine of angle x (given in radians). Use `sin(x / DEG)` if x is in degrees.

`Real sinh(Real x)`
: The hyperbolic sine of x.

`Real sqrt(Real x)`
: The square-root of x.

`void srand48(Int seed)`
: Provides a new random seed for the random number generator.

`Int sleep(Int seconds)`
: Causes the process to wait for given number of seconds. Returns success and sets errno flag on failure.

`Real students_cum(Real t, Real v)`
: Gives the cumulative Student's distribution for v degrees of freedom t test.

`Real students_den(Real t, Real v)`
: Gives the Student's distribution density function for v degrees of freedom t test.

`Int symlink(String from, String to)`
: Creates a symbolic link from given file to other file. (see also link). Returns success and sets errno flag on failure.

`void system(String& cmd)`
: Executes the given command in the Unix shell.

`Real tan(Real x)`
: The tangent of angle x (given in radians). Use `tan(x / DEG)` if x is in degrees.

`Real tanh(Real x)`
: The hyperbolic tangent of x.

`Int tcgetpgrp(Int file_no)`
: Gets the process group associated with the given file descriptor. Returns success and sets errno flag on failure.

`Int tcsetpgrp(Int file_no)`
: Sets the process group associated with the given file descriptor. Returns success and sets errno flag on failure.

`String ttyname(Int file_no)`
: Returns the terminal name associated with the given file descriptor.

`Token(<obj_type>, Int tok_no)`
: Returns the token of the given type of object at index `tok_no` in the list of tokens. Use the `tokens` command to obtain a listing of the tokens of a given type of object.

`TypeDef Type(String& typ_nm | <obj_type>)`
: Returns a type descriptor object (generated by TypeAccess), for the given type name or type object (the type object can be used directly in some situations, but not all).

`Int unlink(String fname)`
: Unlinks (removes) the given file name.

4.4.8 Parameters affecting CSS Behavior

All of the settings that control the behavior of CSS are contained in the global object called `cssSettings`. This is actually just a reference to the `taMisc` class, which is part of the TypeAccess system. The members of this class that can be set by the user are listed below (see also see Section 6.6 [gui-settings], page 87):

`int display_width`
: Width of the shell display in characters.

`int sep_tabs`
: Number of tabs to separate items by in listings.

`int search_depth`
: The recursive depth at which css stops searching for an object's path.

`TypeInfo type_info`
: The amount of information about a class type that is reported when the "type" command is used in CSS.

  Type_info has one of the following values:

  `MEMB_OFFSETS` shows the byte offset of members

  `All_INFO` shows all type info except memb_offsets

  `NO_OPTIONS` shows all info except type options

  `NO_LISTS` shows all info but lists

  `NO_OPTIONS_LISTS` shows all info but options and lists

  The default is `NO_OPTIONS_LISTS`

4.5 Common User Errors

The following are common user errors, which you can anticipate and avoid by reading about them in advance.

Forgetting the semicolon: CSS, being essentially like C or C++, requires most statements to end with a semicolon (;). This allows one to spread statements over multiple lines, since the semicolon and not the newline indicates the end of a statement. However, it is easy to forget it when typing stuff in interactively. The consequences of this are that the following command or statement will be treated as if it was part of the one where the semicolon was forgotten, usually resulting in a `Syntax Error`. Note that commands are exempt from the semicolon requirement, and, as a corollary, can not be extened across multiple lines.

Delayed impact of syntax error: This happens when the user types in something erroneous (i.e., something that will result in a Syntax Error), without following it with a semicolon (usually because it was supposed to be a command, which does not require a semicolon). However, because the entry was neither a command nor followed by a semicolon, it treats the following material as being on the same line, so that only after the second line has been entered (typically), is the first syntax error caught. The solution is to simply press enter a couple of times (or hit the semicolon and press enter), which will clear out the preceding line and let you continue on.

Trying to `print` or do something else with a `void`: If an expression cannot be evaluated (resulting in a `void` value), or a function is called which returns a type of `void` (i.e., nothing is returned), and the result of this expression is then printed or passed to some other function, the following error will result: `Incomplete argument list for: <function_name>, Should have: 1 Got: 0`. Since the void does not get passed to the function or command which is expecting an argument, the function/command (typically `print`) complains with the above error.

4.6 Compiling CSS files as C++ Hard Code

Because they use the standard C++ syntax, CSS script files can be compiled into "hard" code that runs (fast) as a stand-alone application. Of course, the files must not use any of the CSS shortcuts, and must otherwise be standard C++ code (i.e., no executable code outside of functions, using the correct `.` or `->` operator, etc).

There are three main steps that are needed to compile your CSS code. The first, which need only be done once, is the creation of the appropriate libraries that will be linked with the C++ compiled code to produce an executable. The second is formatting your file so that it can be both run by CSS and compiled by C++. The third is creating a Makefile which will allow C++ to compile your file. An example of this is provided in the directory '`demo/css`'.

There are two special libraries that are linked into your C++ executable, one in the '`src/ta`' directory, and one in the '`src/css`' directory. Both can be made using top-level Makefile commands, or by going into the directories separately. '`make LibMin`' makes a library which contains the minimal type-access stuff from the '`src/ta`' directory of the distribution, and it makes the '`libtypea_min`' library. '`make hard_css`' makes a library of special functions in '`src/css`' (e.g., the "special math" functions which have been added into CSS and are not part of the standard C library, and the Dir and ReadLine functions). This library is '`libhard_css`'. Both of these libraries will be visible to the C++ compiler using the makefiles as described below.

The CSS file needs to have a couple of conditially-included elements that resolve the basic differences between CSS and C++. Basically, this amounts to including a header file that estabilshes some defines and includes some commonly-used standard library headers, which are automatically present in CSS. This is the '`css/hard_of_css.h`' file. It is only included when compiling by making it conditional on the pre-processor define `__CSS__`, which is automatically defined by CSS. Also, '`hard_of_css.h`' defines a `main` function which calls a function called `s_main`, which is the actual main function that should be defined in your script.

The following example illustrates these elements, and can be used as a template for making your own CSS files compilable (see '`demo/css`' for a larger example):

```
#ifndef __CSS__
#include <css/hard_of_css.h>
#endif

void s_main(int ac, String* av) {
  // do stuff here..
}

// in css, call our main function as the thing we actually run..
#ifdef __CSS__
s_main(argc, argv);
#endif
```

In order to make the C++ compiling environment as similar to CSS as possible, a variant of the same Makefile can be used. This assumes that the makefiles for your CPU type are correct (i.e., those used in installing the PDP++/CSS source-code distribution (see Chapter 3

[inst], page 4, Section 3.2 [inst-prog], page 9)). The following steps will result in a Makefile that will enable you to compile your CSS code.

1) Copy the sample makefile in '`config/Makefile.hard_of_css`' into the directory where your CSS file is to be compiled, and name it '`Makefile.in`'.

2) Edit this file and ensure that the PDPDIR path is pointing to the installed pdp++ distribution.

3) Then, do a `make -f Makefile.in InitMakefile`, which will make a '`Makefile`' in the current directory that can be used to compile your file.

4) To compile, just type `make <filenm>`, where `<filenm>` is the CSS file without any extension (i.e., the name of the executable that will be produced. Some C++ compilers will complain if the file does not end in a "standard" C++ extension like .cc or .C, so you may have to rename it or create a symbolic link from your .css file (CSS does not care about using a non .css extension, as long as you specify the entire file name).

5 Programming with TypeAccess

This chapter contains some useful information for those who want to add new functionality to the PDP++ software by compiling their own executable. By creating new subclasses of existing classes, and using these new classes in your simulations, it should be possible to make PDP++ do exactly what you want it to.

Before taking this step, you should be reasonably comfortable with the CSS language and using it to access objects in the simulator. Further, you will need to know (or learn about) C++ in a bit more detail than is covered in the CSS section of this manual. There are a number of good books on this subject available in most bookstores.

This chapter describes how to set up the makefiles in your own directory where you will compile your executable. It then describes various coding conventions and extensions to the basic C++ language that we have added to facilitate programming in PDP++. We have established a standard way of dealing with creating, copying, and deleting objects. In addition, each object has special functions that allow groups to manage them. All of these "coding conventions" are described in this chapter.

We have developed a run-time-type-information (RTTI) system called TypeAccess, which provides type information about most classes at run-time. This can be used to determine what kind of unit a Unit* object *really* is, for example (i.e., is it a BpUnit or a MyWackyBpUnit?).

The TypeAccess system requires a more complicated than normal set of makefiles. Fortunately, it is reasonably straightforward to use the makefiles we have developed, so you won't have to deal with much of this complexity.

Most of the graphical interface (i.e., edit dialogs, menus, etc) is generated automatically from the information provided by TypeAccess. The same is true for the way you can transparently access hard-coded types and objects through CSS. Thus, you don't need to do anything special to be able to use your newly defined classes exactly in the way that you use the ones that come with the software.

There are some special keywords that you can put in the comments for your classes and class members and methods called "comment directives". These comment directives allow you to control various aspects of how the GUI and CSS treat your objects. These comment directives are described in this chapter.

5.1 Makefiles and Directory Organization

The PDP++ code should be installed in '`/usr/local/pdp++`', or some such similar place. This path will be referred to from here on out as PDPDIR. This directory contains a set of sub-directories, like '`demo`' and '`manual`', etc. which contain different pieces of the distribution. See Section 3.2 [inst-prog], page 9 for instructions on how to install and compile the source distribution of the PDP++ software. In order compile your own additions to the software, you must install and compile the source code distribution!

The critical directories from a programmer's perspective are the '`src`', which contains the source code, '`config`', which contains the Makefile configuration stuff, '`include`' which has links to the header files, and '`lib`' which has links to the various libraries.

Each sub-directory within the 'src' directory contains code relevant to some sub-component of the PDP++ software. These directories are as follows:

'ta_string'
: The basic String class used throughout the software. It is a slightly modified version of the GNU String class that is distributed with the libg++ distribution (version 2.6.2).

'iv_misc'
: Contains a number of extra pieces of code that supplement the InterViews GUI toolkit.

'ta'
: Contains the TypeAccess system, which gives classes the ability to know all about themselves and other classes at run time. The use of this software is what makes the largely automatic interface used in PDP++ possible. It is described further in Section 5.2 [prog-typea], page 54. This directory also contains a lot of basic objects, like **Array** (Section 7.3 [obj-array], page 102), **List** and **Group** (Section 7.2 [obj-group], page 97) objects.

'css'
: Contains the code for the CSS script language (see Chapter 4 [css], page 14).

'iv_graphic'
: Contains a set of objects which implement a graphical object manipulation environment that is used in the network viewer (see ⟨undefined⟩ [net-view], page ⟨undefined⟩) and the graph log (see ⟨undefined⟩ [log-views-graph], page ⟨undefined⟩).

'ta_misc'
: Contains a smorgasbord of various objects that might have general applicability, and are not specifically PDP objects.

'pdp'
: Where all of the specific pdp code is.

'bp'
: Implements bp and rbp.

'cs'
: Implements cs.

'so'
: Implements so.

'bpso'
: Implements the combined bp and so executable (just links the libraries).

'leabra'
: Implements the leabra algorithm.

Each directory has a set of include files which can be accessed as <xxx/yyy.h>, where xxx is one of the directory names given above. In addition, each directory has its own library, which is just 'libxxx.a', where xxx is the name of the directory (without any underbars). The bp, cs and so directories have a library name of libpdp_xx.a, to indicate that they are part of the pdp software.

All of the compilation results (e.g. object files) go in a subdirectory named after the CPU type being used. The user must set the CPU environmental variable appropriately, as per the definitions used in the InterViews system. The ones that everything has been tested and compiled on are listed in the installation guide (see Chapter 3 [inst], page 4), and the INSTALL file.

Other possibilities are listed in Section 3.2 [inst-prog], page 9. This should be the same as when the system was first installed.

The include files and library are made in two stages. The first stage involves compiling the object files into the CPU subdirectory. Then, if everything goes ok, the library is made, which is then copied into a further subdirectory of the CPU subdirectory called '`lib_include`'. Also, all of the header files are compared with those already in the '`lib_include`' subdirectory (if any), and those ones that are different are copied over. It is these header files in '`lib_include`' that the '`PDPDIR/include`' directory makes links to, and thus these are the ones that are included by other programs. This setup allows one to test out a set of code by making an executable in a given directory and getting things working before installing the new header files and library for the rest of the system to use.

In order to add functionality to the software, one needs to create a new directory, and then include various files from the above directories, and link in their respective libraries. This directory can be located in the same master directory as the main distribution, or it can be located in your own home directory somewhere. This latter option is the preferred one.

We have developed a shell file that does all of the things necessary to create your own directory. The first step is to make a master directory off of your home directory, typically called '`pdp++`'.

Then, run the '`PDPDIR/bin/mknewpdp`' command from this new '`home/pdp++`' directory with an argument which is the name of the directory/executable that you want to make. This will give you step-by-step instructions. In the end, you will end up with a directory that contains some sample code in the form of a .h and .cc file with the same name as the directory.

The script will have installed a '`Makefile`' in your directory which is the same as the one's found in the main PDP++ directories. These makefiles are constructed by concatenating together a bunch of pieces of makefiles, some of which contain standard make actions, and others which contain specific defaults for particular machines. All of the pieces are found in the '`PDPDIR/config`' directory.

The makefiles named '`Makefile.CPU.[CC|g++]`' are the machine-specific files that you should edit to make sure they have all the right settings for your system. This should have been done already during the installation of the PDP++ source code distribution, see Section 3.2 [inst-prog], page 9 for details.

To these pieces is added the specific list of files that need to be made in your directory. This is specified in the '`Makefile.in`' file. This is the only makefile you should edit. It can be used to override any of the settings in the standard makefiles, simply by redefining definitions or actions. If you add files to your directory, follow the example for the one already in your default '`Makefile.in`' that was installed with the '`mknewpdp`' command.

Note that there are a couple of compiler-specific "extra" files in the directory. These have the name of the directory plus a '`_vt.cc`' or '`_it.cc`' suffix. The '`_vt`' file is for virtual table instantiation, which is controlled in cfront with the +e0/+e1 arguments. It simply includes most of the header files in the software. We have found that by compiling everything except the '`_vt`' file with +e0 that the executables are much smaller. This is even true in cfront versions where they had "fixed" this problem. You can try doing the other way by leaving out the +e args and not using the '`_vt`' file (see the definitions in

'PDPDIR/config/Makefile.defs' for how to do this: change your 'Makefile.CPU' file and recompile the entire distribution first..).

The '_ti.cc' is the template instantiation file needed by gnu g++ version 2.6.3 (reportedly, it won't be needed in 2.7). It contains explicit instantiations of all of the templates used in each library. For user directories, this probably isn't needed, but its there if you do declare any templates and encounter link problems with g++. Also, the 'Makefile.CPU.g++' show how this file gets included in the making of a given project.

New for 2.0: All of the makefile actions, as shown below, are now available using a consistent syntax structure: all lower case, with underbars separating different words. This makes it much easier to remember what command to type. The old eclectic combinations of upper and lower case words, etc are still available if you already know them.

The commonly-used actions that are defined in the makefile are as follows:

'make bin, make opt_bin, make dbg_bin'
: Makes the binary from the files in this directory. Bin makes the default form specified in the make file, while opt and dbg make optimized and debug versions, respectively.

'make re_bin, make opt_re_bin, make dbg_re_bin'
: Same as above, except it first removes the executable before making. This is useful if a library has changed but no header files from that library were changed.

'make lib, make opt_lib, make dbg_lib'
: Like the above, except it makes a library containing the relevant .o files.

'make new_makefile'
: This makes a new version of the 'Makefile' file in the current directory. This concatenates all of the different parts that together make up a single 'Makefile'. However, it does not make a CPU directory, which is necessary to actually compile (see cpu_dir next).

'make cpu_dir, make local_cpu_dir'
: This makes and configures a directory with the same name as the CPU environmental variable (reflecting the CPU type of the machine) suitable for compiling the object files into. If local_cpu_dir is made first, then this directory is actually a symbolic link to a directory created on a disk local to the current machine, so that compilation will be faster than if the directory where the source is located is a networked (slow) directory (i.e., NFS). The cpu_dir action copies the current 'Makefile' into the directory, and configures the directory for compiling. Note that these actions remove any existing dependency information, so that a depend action should be made following either of them.

'make depend'
: This automatically adds dependency information for the files in this directory onto the 'CPU/Makefile' file. This allows the make command to know when to compile these files after something they depend on has been touched (edited).

'`make makefiles, make make_depend, make new_make_depend`'
: These actions simply combine some of the above steps together into one action. `makefiles` does a `new_makefile` and then a `cpu_dir`, `make_depend` does a `cpu_dir` and then a `depend`, and `new_make_depend` does all three of the necessary steps: `new_makefiles`, `cpu_dir`, and `depend`. The only reason you should not use the latter all the time is if your `make` program has trouble using a new '`Makefile`' (i.e., as created by the `make new_makefiles` action) for calling the subsequent actions. In this case, you have to first do a `make new_makefiles` and then you can do a `make make_depend`.

'`make force_ta`'
: Forces a call to the TypeAccess scanning program '`maketa`'.

5.2 The TypeAccess System

The TypeAccess system consists of a set of objects that can hold type information about class objects. This type information includes the names and types of all the members and methods in a class, the parents of the class, etc. This information can be used by classes to get information about themselves at run time. In addition, the TypeAccess system provides a set of type-aware base classes and macros for defining derived versions of these that can be used to easily incorporate run-time type information into any C++ system.

In addition to being type-aware, the base classes can use their own type information to save and load themselves automatically to and from ASCII format text files. Further, there is an extensible graphical interface based on InterViews which can automatically build editing dialogs for filling in member values and calling member functions on arbitrary objects. Finally, the type information can be used to provide a transparent script-level interface to the objects from the CSS script language. This provides the benefits of compiled C++ for fast execution, and the ability to perform arbitrary interactive processing in an interpreter using the C++ language supported by CSS.

Many features of the interface and script level interface, as well as various options that affect the way objects are saved and loaded, can be specified in comments that follow the declaration of classes, members, and methods. These *comment directives* constitute a secondary programming language of sorts, and they greatly increase the flexibility of the interface. They are documented in Section 5.3 [prog-comdir], page 63.

Thus, the PDP++ software gets much of its functionality from the TypeAccess system. It provides all of the basic interface and file-level functionality so that the programmer only needs to worry about defining classes that perform specific tasks. These classes can then be flexibly used and manipulated by the end user with the generic TypeAccess based interface.

5.2.1 Scanning Type Information using '`maketa`'

Type information for TypeAccess is scanned from the header files using a program called '`maketa`', which looks for `class` and `typedef` definitions, and records what it finds. It operates on all the header files in a given directory at the same time, and it produces three output files: '`xxx_TA_type.h`', '`xxx_TA_inst.h`', and '`xxx_TA.cc`', where xxx is given by a "project name" argument. The first file contains a list of `extern` declarations of instances of the **TypeDef** type, which is the basic object that records information about types. Each

type that was defined with a `class` or `typedef`, or ones that are modifications of basic types, such as reference or pointer types, are given their own **TypeDef** object, which is named with the name of the type with a leading `TA_` prefix. Thus, a class named *MyClass* would have corresponding **TypeDef** object named *TA_MyClass*, which can be used directly in programs to obtain type information about the *MyClass* object. a Pointers have a `_ptr` suffix, and references have a `_ref` suffix. Template instances are represented by replacing the angle brackets with underbars. The '`xxx_TA_type.h`' file must be included in any header files which reference their own type information.

The '`xxx_TA_inst.h`' file contains declarations of "instance" objects, which are pointers to a token of each of the classes for which type information is available. These instances are named `TAI_` with the rest the same as the corresponding `TA_` name. The `-instances` argument to '`maketa`' determines if instances are made, and this can be overridden with the `#NO_INSTANCE` and `#INSTANCE` comment directives (see Section 5.3 [prog-comdir], page 63). The **TypeDef** object can use an instance object of one of the type-aware base classes to make a new token of that object given only the name of the type to be created. This gives the system the power to create and delete objects at will, which is necessary for the file saving and loading system to work.

Finally, the '`xxx_TA.cc`' file contains the actual definitions of all the type information. It must be compiled and linked in with the project, and its `ta_Init_xxx` function must be called at the start of the execution of the program before any type information is used.

Note that while '`maketa`' does process complexities like `template` and multiply inherited classes properly, it does not deal with multiple versions of the same function which differ only in argument type in the same way that C++ does. Instead, the scanner just keeps the last version of a given method defined on the class. This makes the type information compatible with the limitations of CSS in this respect, since it does not know how to use argument types to select the proper function to be called (see Section 4.3.1 [css-c++-diff], page 27). This limitation greatly simplifies the way that functions are called by CSS. It is recommended that you create methods which have some hint as to what kinds of arguments they expect, in order to get around this limitation. The **taList** and **taGroup** classes, for example, contain both overloaded and specific versions of the `Find` function, so the C++ programmer can call `Find` with any of a number of different argument types, while the CSS programmer can use the `FindName` or `FindType` versions of the function.

5.2.2 Startup Arguments for '`maketa`'

The type-scanning program '`maketa`' takes the following arguments:

`[-v<level>]`
: Verbosity level, 1-5, 1=results,2=more detail,3=trace,4=source,5=parse.

`[-hx | -nohx]`
: Generate .hx, .ccx files instead of .h, .cc. This is used in conjunction with a makefile that compares the .hx with the .h version of a file and only updates the .h if it actually differs from the .hx version. This prevents lots of needless recompiling when the type-scanned information is not actually different when a header file was touched.

`[-css]` Generate CSS stub functions. The stub functions take **cssEl*** arguments, and call member functions on classes. These must be present to use CSS to call member functions on classes, or to call functions from the edit dialog menus and buttons.

`[-instances]`
Generate instance tokens of types. Instances are needed to make tokens of class objects.

`[-class_only | -struct_union]`
Only scan for `class` types (else `struct` and `union` too). The default is to only scan for `class` types because they are always used in the definition of a class object. `struct` and `union` can be used to modify the type name in old-style C code, which can throw off the scanner since these don't amount to class definitions.

`[-I<include>]...`
Path to include files (one path per -I).

`[-D<define>]...`
Define a pre-processor macro.

`[-cpp=<cpp command>]`
Explicit path for c-pre-processor. The default is to use '`/usr/lib/cpp`', which doesn't work very well on C++ code, but its there. It is recommended that you use cccp, which is the gnu preprocessor that comes with gcc.

`[-hash<size>]`
Size of hash tables (default 2000), use -v1 to see actual sizes after parsing all the types.

`project` This is the stub project name (generates project_TA[.cc|_type.h|_inst.h]).

`files...` These are the header files to be processed.

5.2.3 Structure of TypeAccess Type Data

The classes used in storing type information in the TypeAccess system are all defined in the '`ta/typea.h`' header file. Basically, there are a set of **Space** objects, which all derive from a basic form of the `List` object (defined in '`ta/ta_list.h`', which represent type spaces, member spaces, method spaces, etc. These are just containers of objects. The spaces are: **TypeSpace**, **MemberSpace**, **MethodSpace**, **EnumSpace**, **TokenSpace**. Note that they contain functions for finding, printing, and generally manipulating the objects they contain.

There are corresponding **TypeDef**, **MemberDef**, **MethodDef**, and **EnumDef** objects which hold specific information about the corresponding aspect of type information. The **TypeDef** contains the following fields:

`String name`
Holds the name of the type.

`String desc`
: A description which is obtained from the user's comment following the declaration of the type.

`uint size` The size of the object in bytes.

`int ptr` The number of pointers this type is from a basic non-pointer type.

`bool ref` True if this is a reference type.

`bool internal`
: True if this type information was automatically or internally generated. This typically refers to pointer and reference types which were created when the scanner encountered their use in arguments or members of other classes that were being scanned.

`bool formal`
: True for basic level objects like `TA_class` and `TA_template` which are are formal parents (`par_formal`) of types that users declare. These provide a way of determining some basic features of the type. Formal type objects are declared and installed automatically by the type scanning system.

`bool pre_parsed`
: True if this type was registered as previously parsed by the type scanning system (i.e., it encountered an '`extern TypeDef TA_xxx`' for this type, where xxx is the name of the type). These types don't get included in the list of types for this directory. This makes it possible to do type scanning on a complex set of nested libraries.

`String_PArray inh_opts`
: These are the options (comment directives) that are inherited by this type (i.e., those declared with a `##` instead of a `#`).

`String_PArray opts`
: These are all of the options (comment directives) for this type, including inherited and non-inherited ones.

`String_PArray lists`
: A list of the `#LIST_xxx` values declared for this type.

`TypeSpace parents`
: A list of parents of this type. There are multiple parents for multiple-inheritance `class` types, and for `internal` types which are the combination of basic types, such as `unsigned long`, etc.

`int_PArray par_off`
: A list of offsets from the start of memory occupied by this class where the parent object begins. These are used for multiply inherited class types. They are in a one-to-one correspondence with the `parents` entries.

`TypeSpace par_formal`
: A list of the formal parents of this type, including `TA_class`, etc.

`TypeSpace par_cache`
: A special cache of frequently-queried type parents. Currently if a type derives from **taBase**, then `TA_taBase` shows up here (because a lot of the TypeAccess code checks if something is derived from the basic type-aware type **taBase**).

`TypeSpace children`
: A list of all the types that are derived from this one.

`void** instance`
: A pointer to a pointer of an instance of this type, if it is kept. The `GetInstance` function should be used to get the actual instance pointer.

`TokenSpace tokens`
: A list of the actual instances or tokens of this type that have been created by the user (the `TAI_xxx` instance object is not registered here). These are not kept if the type does not record tokens (see the `#NO_TOKENS` comment directive, Section 5.3.1 [comdir-objs], page 64).

`taivType* iv`
: A pointer to an object which defines how a token of this type appears in a GUI edit dialog. There is a "bidding" procedure which assigns these objects, allowing for the user to add new specialized representations which out-bid the standard ones. This bidding takes place when the gui stuff is initialized, and the results are stored here.

`taivEdit* ive`
: This is like the `iv` pointer, except it is the object which is used to generate the entire edit dialog for this object. It also is the result of a bidding procedure.

`taBase_Group* defaults`
: These are pointers to different **TypeDefault** objects for this type. Each **TypeDefault** object is for a different scope where these types can be created (i.e., a different **Project** in the PDP++ software).

`EnumSpace enum_vals`
: Contains the enum objects contained within a given `enum` declaration.

`TypeSpace sub_types`
: These are the sub-types declared with a `typedef`, `enum`, or as part of a template instantiation within a `class` object.

`MemberSpace members`
: These are the members of a `class` object.

`MethodSpace methods`
: These are the methods of a `class` object.

`TypeSpace templ_pars`
: These are the template parameters for template objects. In the `template` itself, they are the formal parameters (i.e., `T`), but in the template instance they point to the actual types with which the template was instantiated.

The most important functions on the **TypeDef** object are as follows:

`bool HasOption(const char* op)`
: Checks to see if the given option (comment directive) (don't include the `#`) is present on this type.

`String OptionAfter(const char* op)`
: Returns the portion of the option (comment directive) after the given part. This is used for things like `#MENU_ON_xxx` to obtain the xxx part. If option is not present, an empty string is returned.

`InheritsFrom(TypeDef* tp)`
: Checks if this type inherits from the given one (versions that take a string and a reference to a **TypeDef** are also defined). Inheritance is defined only for classes, not for a pointer to a given class, for example. Thus, both the argument and the type this is called on must be non-pointer, non-reference types.

`DerivesFrom(TypeDef* tp)`
: Simply checks if the given type appears anywhere in the list of parents for this type. Thus, a pointer to a class derives from that class, but it does not inherit from it.

`String GetValStr(void* base, void* par=NULL, MemberDef* memb_def=NULL)`
: Uses the type-scanned information to obtain a string representation of the value of an instance of this type. `base` is a pointer to the start of a token of this type, and `par` and `member_def` can be passed if it is known that this token is in a parent class at a particular member def. This and the following function are used widely, including for saving and loading of objects, etc.

`SetValStr(const char* val, void* base, void* par=NULL, MemberDef* memb_def=NULL)`
: Takes a string representation of a type instance, and sets the value of the token accordingly (it is the inverse of `GetValStr`).

`CopyFromSameType(void* trg_base, void* src_base, MemberDef* memb_def=NULL)`
: Uses the type-scanned information to copy from one type instance to the next. Any class objects that are members are copied using that object's copy operator if one is defined (this is only known for derivatives of the **taBase** base class).

`Dump_Save(ostream& strm, void* base, void* par=NULL, int indent=0)`
: This will save the given type object to a file. Files are saved in an ASCII format, and are capable of saving pointers to other objects when these objects derive from the **taBase** object. Special code is present for dealing with groups of objects stored in the **taList** or **taGroup** classes. See Section 5.2.5 [prog-typea-dump], page 63 for more details.

`Dump_Load(istream& strm, void* base, void* par=NULL)`
: This will load a file saved by the `Dump_Save` command.

The other **Def** objects are fairly straightforward. Each includes a `name` and `desc` field, and a list of `opts` (comment directives) and `lists`. Also, each contains an `iv` field which represents the item in the GUI edit dialog, and is the result of a bidding process (see the `iv` field in the **TypeDef** object above). They all have the `HasOption` and `OptionAfter` functions plus a number of other useful functions (see the '`ta/typea.h`' for details).

MemberDef objects contain the following additional fields. Note that derived classes contain links (*not copies*) of the members and methods they inherit from their parent, except when the class has multiple parents, in which case copies are made for the derived class because the offset information will no longer be the same for the derived class.

`TypeDef* type`
: The type of the member.

`ta_memb_ptr off`
: The address or offset of this member relative to the start of the memory allocated for the class in which this member was declared.

`int base_off`
: The offset to add to the base address (address of the start of the class object) to obtain the start of the class this member was declared in. This is for members of parents of multiply-inherited derived classes.

`bool is_static`
: True if the member was declared `static`. Thus, it can be accessed without a `this` pointer. The `addr` field contains its absolute address.

`void* addr`
: The absolute address (not relative to the class object) of a static member.

`bool fun_ptr`
: True if the member is actually a pointer to a function.

The **MethodDef** object contains the following additional variables:

`TypeDef* type`
: The type of the method.

`bool is_static`
: True if the method was declared `static`.

`ta_void_fun addr`
: The address of a `static` method. Non-static methods do not have their addresses recorded. Methods are called via the `stubp` function, if the `-css` option was used during scanning.

`int fun_overld`
: The number of times this function was overloaded (i.e., a function of the same name was declared in the class or its parents). TypeAccess does not perform name mangling on functions, so only one instance of a given method is recorded. It is the last one that the scanner encounters that is kept.

`int fun_argc`
: The number of arguments for this function.

`int fun_argd`
: The index where the arguments start having default values. Thus, the function can be called with a variable number of arguments from `fun_argd` to `fun_argc`.

`TypeSpace arg_types`
: These are the types of the arguments.

`String_PArray arg_names`
: These are the names of the arguements (in one-to-one correspondence with the types).

`css_fun_stub_ptr stubp`
: A pointer to a "stub" function which calls this method using **cssEl** objects as arguments. This function is defined in the '`xxx_TA.cc`' file if the `-css` argument is given to '`maketa`'. The **cssEl** objects have conversion functions for most types of arguments, so that the function is called by casting the arguments into the types expected by the function. Pointers to class objects are handled by `cssTA` objects which have a pointer and a corresponding `TypeDef` pointer, so they know what kind of object they point to, making conversion type-safe. These stubs return a cssEl object. They also take a `void*` for the `this` object. These stubs are used both by CSS and to call methods from the edit dialogs from menus and buttons.

5.2.4 The Type-Aware Base Class taBase

There is a basic class type called **taBase** that uses the TypeAccess type information to perform a number of special functions automatically. This object is aware of its own type information, and can thus save and load itself, etc. Special code has been written in both the TypeAccess system and in CSS that takes advantage of the interface provided by the **taBase** type. Thus, it is recommended that user's derive all of their types from this base type, and use special macros to provide derived types with the hooks necessary to get their own type information and use it effectively. The type **TAPtr** is a `typedef` for a pointer to a **taBase** object. The definition of a **taBase** object and the macros that are used with it are all in '`ta/ta_base.h`'.

All **taBase** objects have only one member, which is a reference counter. This provides a mechanism for determining when it is safe to delete an object when the object is being referenced or pointed to in various different places. **taBase** provides a set of referencing and pointer-management functions that simplify the use of a reference-count based memory management system. `Ref` increments the reference count, `unRef` decrements it, `Done` checks if the refcount is zero, and deletes the object if it is, and `unRefDone` does both. `Own` both Ref's an object and sets its owner. For pointers, `SetPointer` unrefs any existing object that the pointer points to, and sets it to point to the new object. `DelPointer` does an `unRefDone` on the object pointed to, and sets the pointer to NULL. `OwnPointer` is like SetPointer except it also sets the owner of the pointed-to object to the one given by the argument. See Section 5.4.2 [coding-funs], page 72 and '`ta/ta_base.h`' for more details.

The one essential function that **taBase** provides is `GetTypeDef()`, which is a virtual function that returns a pointer to the `TypeDef` type descriptor for this object. This function is defined as part of the basic macro `TA_BASEFUNS`, which must be included in all classes derived from **taBase**. This function makes it possible for a generic pointer to a taBase object to find out what type of object is really being pointed to.

There are a number of functions defined on the taBase type that simply call the corresponding function on the **TypeDef** pointer. These can be found in the '`ta/ta_base.h`' header file. They just make it easier to call these commonly-used functions, instead of requiring the user to put in a `GetTypeDef` function in between.

taBase also provides a simplified way of managing the construction, deletion, and copying of an object. Basically, construction is broken down into a set of functions that `Initialize` the member variables, `Register` the new token with the type if it is keeping track of tokens, and it sets the default name of the object based on its type name using `SetDefaultName`. The `TA_BASEFUNS` macro defines a default constructor that calls these three functions in that order. The user thus needs to provide a `Initialize` function for every class defined, which does the appropriate member initialization. Note that if this function is not defined, the one on the parent class will be called twice, so its more efficient to include a blank Initialize function when there are no members that need to be initialized.

The destructor function is similar to the constructor. A default destructor is defined in `TA_BASEFUNS`, which simply calls `unRegister`, and `Destroy`. Thus, the user needs to provide a `Destroy` function which frees any additional resources allocated by the object, etc. Like `Initialize`, a blank `Destroy` should be defined when there is nothing that needs to be done to prevent the parent function from being called twice.

Copying, cloning, and making a new token of the given type are also supported in the **taBase** class. The `Copy` function performs the basic copy operations for both the copy constructor and the `=` operator. This should replace the values of this class and any of its existing sub-objects with those of the object passed to it, as it is intended for assignment between two existing objects. In general, the `=` operator should be used for copying all members, except for the case of `LINK_GROUP` groups and lists, which should use the `BorrowUnique` function (since they do not own the items in the list, just link them). `Copy` must call the parent's `Copy` function as well. As a minor simplification of calling the parent (and to provide a copy function for just the items in a given class), it is conventional to define a `Copy_` function, which does everything except for calling the parent copy function. The macro `COPY_FUNS` can be used to define a `Copy` function which calls the parent function and then `Copy_`. The macro `SIMPLE_COPY` defines a `Copy_` function which uses the type-scanned information to do the copying. It is slower than hand-coding things, so it probably shouldn't be used on types which will have a lot of tokens or be copied often.

A `Clone` function which returns a `TAPtr` to a new duplicate of this object is defined in `TA_BASEFUNS`, as well as an "unsafe" version of `Copy` (`UnSafeCopy`), which takes a generic `TAPtr` argument and casts it into their type. The argument's type should thus be tested before calling this function. A safe interface to this function is provided by the `CopyFrom` function, which does the type checking. Finally, the `MakeToken` function will create a new token of the type.

The **taBase** class also contains functions for creating and manipulating a structure hierarchy of objects. This is where certain objects contain groups of other objects, which contain other objects, etc. For example, the PDP++ software has a structure hierarchy built around a root object, which contains projects, which contain lots of other objects like networks, projects, environments, etc. Special container objects like **taList** and **taGroup** play an important role in representing and manipulating this structure (note that it is possible to write other types of container objects which could play the same role simply by overloading the same functions that these objects do).

When an object is "linked" into the object hierarchy, a function called `InitLinks` is called. This function should perform any kind of initialization that depends on the object being situated in the hierarchy, like being able to know what object "owns" this one. **taBase**

has functions for getting and setting the owner of an object. For example, when a group (**taList** or **taGroup** creates a new object and links it into its list of objects, it calls the `SetOwner` function with a pointer to itself on this new object, and then it calls `InitLinks`. Similarly, when the object is removed from the group, the `CutLinks` function is called, which should cut any links that the object has with other objects.

An object's location in the object hierarchy can be represented by a *path* to that object from a global root object. A given application is assumed to have a root object, which contains all other objects. A pointer to that object is kept in **tabMisc::root**, which is used to anchor the path to any given object. An object can find its path with the `GetPath` function, and an object can be found from a path with the `FindFromPath` function.

Finally, a function for allowing an object to set the values of certain members based on changes that might have been made in other members after a user edits the object, called `UpdateAfterEdit`, is provided. This function is called on most objects after they are loaded from a save file (except those with the `#NO_UPDATE_AFTER` comment directive), and on all objects after the user hits *Apply* or *Ok* in an edit dialog, and after any member is set through a CSS assign statement. While the object does not know which members were changed when `UpdateAfterEdit` is called, the object can buffer previous state on its own to figure this out if it is needed.

For a step-by-step guide to making a new class that derives from **taBase**, see Section 5.4 [prog-coding], page 70.

5.2.5 The Dump-file Format for Saving/Loading

The format used for dumping objects to files an loading them back in involves two passes. The first pass lists all of the objects to be saved (i.e., the object that the `Save` function was called on, and any sub-objects it owns. This is done so that during loading, all objects will have been created before pointers to these objects attempt to be cashed out. The second pass then saves all of the values associated with the members in the object. The format is a name-value based one, so that files can be loaded back into objects whose definition has changed. It skips member names it can't find, etc, so you can continue to modify your software and still load old data.

Paths (i.e., the `GetPath` function) figure heavily into the saving of objects, especially pointers. Pointers are saved by giving the path to the object. These saved paths are automatically corrected if the objects are loaded into a different location than the one they were saved in. All pointers that are saved are assumed to be reference-counter based. Thus, the `SetPointer` function is used to set the pointer. Also note that it is impossible to save a pointer to a non-taBase derived object, since there is no way to get the path of such an object.

5.3 Standard TypeAccess Comment Directives

The following sections document comment directives that are recognized by the standard TypeAccess GUI and script-language interfaces. These must be placed in comments immediately following the definition of that which the apply to. Thus, an object directive should appear as

```
class whatever : public something {  // #IGNORE comment goes here
```

for members and methods, it should be as follows:

```
class whatever : public something {  // #IGNORE comment goes here
  int         member_1;       // #HIDDEN comment goes here
  float       member_2;
  // #READ_ONLY or here
  float       get_real();     /* #USE_RVAL note that multi-line
     old-fashioned c-style comments are legal too! */
```

5.3.1 Object Directives

If you add an extra "#" to the beginning of the comment directive, it will automatically be inherited by any sub-classes of the given object. Otherwise, it only applies to the object on which it was given.

`#IGNORE` Do not register this object in the list of types.

`#NO_TOKENS`
: Do not keep a record of the tokens of this object type. Types can keep pointers to all instances or tokens of themselves. This can be expensive in terms of memory, but the interface uses "token menus" for arguements or methods which are pointers to objects of the given type.

`#NO_INSTANCE`
: Do not create a global instance (`TAI_xxx`) of this object. This will prevent tokens of this object from being made.

`#INSTANCE`
: If default is not to create instances, then create one anyway for this object.

`#NO_MEMBERS`
: Do not store the members (including member functions) of this class. Only the type name will be registered.

`#NO_CSS` Do not create CSS stub functions for the member functions on this object.

`#INLINE` Causes this item to be edited in a single line in a dialog box (e.g. for geometry x,y,z) and affects saving/loading, etc.

`#EDIT_INLINE`
: Only causes this item to be edited in a single line in a dialog box, but in all other respects it is treated as a normal included class. This is useful for certain complex objects such as arrays and lists that do not otherwise save/load well as INLINES.

`#BUTROWS_x`
: Set the number of button rows to be x, useful if default allocation of number of rows of buttons for edit dialog is not correct

`#EXT_xxx` Sets the default extension for saving/loading this type to xxx.

`#COMPRESS`
: store dump file's of this object compressed. Since the save files are text, they can be large, so it is a good idea to auto-compress dump files for large objects.

`#MEMB_IN_GPMENU`
: This indicates that there is a group object as a member of this one who's objects should appear in menus where this object appears.

`#VIRT_BASE`
: This is a "virtual" base class: don't show in token menus for this object, etc.

`#NO_UPDATE_AFTER`
: Don't call UpdateAfterEdit when loading this object (and other places it might automatically get called). Since a list of objects which should be updated after loading is made, small or numerous objects should not be added to this list if not necessary.

`#IMMEDIATE_UPDATE`
: Perform an immediate UpdateAfterEdit on this object after loading (i.e., it creates other objects..). Normally, updating happens after all of the other objects have been loaded.

`#SCOPE_xxx`
: Type of object to use as a scope for this object. The scope restricts token menus and other things to only those things that share a common parent token of the given scope type.

`#ARRAY_ALLOC`
: Specific to taList_impl derivatives: this list or group should have saved items created all together during loading (ie., like an array). If actually using array-based memory allocation, this is essential, but otherwise it can only speed things up a little bit.

`#LINK_SAVE`
: Save the actual contents of this object even when it appears as a link in a list. Usually just the link pointer is saved, and the object is saved later in the group that actually owns it. This overrides this and saves the information in both places – can be useful if info from the linked object is needed during loading.

`#NO_OK` Do not present an OK button on the edit dialog for this object.

`#NO_CANCEL`
: Do not present a CANCEL button on the edit dialog for this object.

5.3.2 Member Directives

`#HIDDEN` Hides member from user's view in edit dialogs and CSS type information printouts.

`#HIDDEN_INLINE`
: Hides member when inlined in another object, but not when edited itself. This only applies to members of #INLINE objects.

`#SHOW` Always show this member in the edit dialog (i.e., even if it was marked #READ_ONLY).

`#IGNORE` Does not register this member in the type information for this class.

`#DETAIL` Flags this member as a level of detail that the user usually does not need to deal with — can be viewed by changing the Show setting in the edit dialog.

`#NO_SAVE` This member is not saved when dumping to a file.

`#NO_SAVE_PATH_R`
Don't create these objects in the 1st pass of the dump file (e.g., they will be created automatically by something else, usually an #IMMEDIATE_UPDATE UpdateAfterEdit function on a parent object). This can be used to speed up saving and loading of large numbers of repetitive objects which can be created instead.

`#READ_ONLY`
Allows the user to see but not edit this item. By default the gui edit dialog will not show these items. This prevents the member from being changed in CSS as well.

`#IV_READ_ONLY`
Like READ_ONLY, but user can modify the value via CSS (which is prevented by READ_ONLY).

`#LIST_xxx`
Sets the Lookup List for this element. This is used mainly for pointers to functions, where one wants the gui to show a list of top-level functions that have been scanned by maketa (see Section 5.3.4 [comdir-funs], page 69).

`#TYPE_xxx`
Sets the default type for members which are pointers to TypeDef objects. This also works for MemberDef pointers. If xxx is 'this', then the type of the current object is used.

`#TYPE_ON_xxx`
For object, TypeDef, or MemberDef pointers: use member xxx of this object to anchor the listing of possible types, tokens, or members.

`#FROM_GROUP_xxx`
For token pointers, use given member xxx as the group from which to select token options (xxx can be a pointer to a group).

`#GROUP_OPT_OK`
For FROM_GROUP_xxx mbrs, allows group itself as an option (else not allowed).

`#SUBTYPE_xxx`
Sets this token pointer member to be only subitems (objects owned by this one) of type xxx. A recursive scan of members on this object is performed to search for objects of the given type as possible values for this field.

`#NO_SUBTYPE`
Don't search this ptr for possible subitems (use if this ptr might point "up", causing a endless loop of searching for subitems).

`#NO_FIND` Don't search this member for the recursive FindMembeR function which searches recursively through objects (use if this ptr might point up in the hierarchy, which might cause an endless loop).

`#NO_SCOPE`
: Don't use scope for tokens for a token pointer member. See SCOPE object directive

`#LABEL_xxx`
: Set the label for item (or menu or button) to be xxx.

`#OWN_POINTER`
: For a pointer to an object, when loading, set the owner of the obj to be this object. Thus, this pointer is always created and owned by this object.

`#NULL_OK` A null value is ok as an option for the user (else not) for pointer to a type, and SUBTYPE tokens.

`#NO_NULL` A null value is not ok (for tokens) (else ok).

`#NO_EDIT` Don't include Edit as an option on a token pointer menu (else ok).

`#POS_ONLY`
: Only positive (non-negative) integers, this controls behavior of the stepper for integer types.

`#LINK_GROUP`
: This group member only has linked items (doesn't allow user to create new tokens in this group).

`#IN_GPMENU`
: This members' items should appear in the group menu. The member must be a **taGroup_impl** descendent type, and the class must have a MEMB_IN_GPMENU option set.

`#CONDEDIT_xxx`
: This makes editing a member conditional on the value of another member. For example: #CONDEDIT_OFF_type:NONE,LT_MAX specifies that this member is to be not editable (OFF) when the type enum variable is either NONE or LT_MAX. One alternatively specify ON for conditions when it should be editable. The comparison is based on the string representation of the member value – sub-paths to members within contained objects can also be used.

`#DEF_xxx` Specifies a default value for the member. If the field is set to a value other than this default value, it will be highlighted in yellow to indicate that the value is different from default. This should only be used where there are clear default values that are typically not changed.

`#AKA_xxx` This allows old project files etc to be loaded correctly after changing the name of a field or enum by matching xxx to the new field/enum.

5.3.3 Method Directives

`#MENU` Creates a Menu for this function item in an Edit dialog.

`#MENU_SEP_BEFORE`
: Create a separator before this item in the menu.

`#MENU_SEP_AFTER`
: Create a separator after this item in the menu.

`#MENU_ON_xxx`
: Puts this function on given menu. Creates menu if not already there. This does not replace the #MENU directive. Everything on the File and Edit menus will be on the edit button for this class in an edit dialog.

`#BUTTON` Creates a button for this function in the edit dialog.

`#LABEL_xxx`
: Sets the label for item (or menu or button) to be xxx.

`#USE_RVAL`
: Use (display) return value from this function. Otherwise return values are ignored.

`#USE_RVAL_RMB`
: Use (display) return value from this function only if the right mouse button was pressed on the Ok button. Otherwise return values are ignored.

`#NO_APPLY_BEFORE`
: Do not apply any changes to dialog before calling this function. The default is to apply the changes first.

`#NO_REVERT_AFTER`
: Do not update (revert) dialog after calling this function (and do not call the UpdateAfterEdit function either). The default is to do both.

`#UPDATE_MENUS`
: Update the global menus after calling this function (e.g., because altered the structure reflected by those menus).

`#ARGC_x` How many args to present to the user (if default args are available).

`#ARG_ON_OBJ`
: An argument to this function is an object within the base object (e.g., a member of the group).

`#TYPE_xxx`
: For TypeDef pointer args: use given type to anchor the listing of possible types. if xxx == 'this', then the type of the current object is used.

`#TYPE_ON_xxx`
: For a function with (any) TypeDef or Token args, uses the member xxx of this to anchor type selection or type of tokens to present.

`#FROM_GROUP_xxx`
: Performs selection of tokens for args from given group member xxx, which is a member of this object (like ARG_ON_OBJ). Can also specify which arg(s) this applies to by doing #FROM_GROUP_1_gp: 1 = this arg or below uses from_group, so put your from_group args first and specify the highest index as this.

`#NO_GROUP_OPT`
: For FROM_GROUP_xxx args, disallows group itself as an option.

`#NO_SCOPE`
: Don't scope the argument to this function. See SCOPE object directive

`#NO_SCRIPT`
: Do not generate script code to call this function, if script code recording is currently active. Section 5.3.1 [comdir-objs], page 64.

`#GHOST_ON_xxx`
: For BUTTON meths, ghosts the button based on the value of boolean member xxx of this class. If member == true, button is ghosted.

`#GHOST_OFF_xxx`
: Like above, except if member == false, button is ghosted.

`#CONFIRM` For functions with no args, put up a dialog for confirmation (shows function description too).

`#NEW_FUN` Give user the option to call this (void) function during New (in the "new" dialog).

`#NULL_OK` A null value is ok as an option for the user (else not). for all pointers as args.

`#EDIT_OK` Include Edit as an option on the token pointer menu (else not)

`#FILE_ARG_EDIT`
: For functions with one ostream arg, use the normal arg edit dialog, instead of a shortcut directly to the file chooser (arg edit allows user to choose open mode for saving).

`#QUICK_SAVE`
: For functions with one ostream arg, use existing file name if possible (default is to prompt).

`#APPEND_FILE`
: For functions with one ostream arg, use append as the file opening mode.

5.3.4 Top-Level Function Directives

In addition to class objects and typedef's, it is possible to scan information about certain top-level functions. These functions must be preceded by a `#REG_FUN` comment, and the comments that apply to the function must precede the trailing `;` that ends the function declaration.

```
// #REG_FUN
void Cs_Simple_WtDecay(CsConSpec* spec, CsCon* cn, Unit* ru, Unit* su)
// #LIST_CsConSpec_WtDecay Simple weight decay (subtract decay*wt)
     ; // term here so scanner picks up comment
```

These functions get registered as `static` functions of a mythical object with a **TypeDef** of `TA_taRegFun`. The purpose of registering functions in this way is to make them available for members of classes that are pointers to functions. These registered functions are shown in a menu in the edit dialog if the `#LIST_xxx` directive matches on the registered function and the pointer to a function.

5.3.5 PDP++ Specific Directives

`#NO_VIEW` For real-valued unit members, do not display this item in the net view display.

`#AGGOP_xxx`
(Process object only) sets the default aggregate operator for this process's statistics.

`#FINAL_STAT`
(Stat object only) indicates if this should be created as a final stat.

`#LOOP_STAT`
(Stat object only) indicates if this should be created as a loop stat.

`#COMPUTE_IN_xxx`
(Stat object only) level at which this stat should be computed, xxx is a process type.

`#NO_INHERIT`
In specs, makes member not-inherit from higher-ups.

5.4 Coding Conventions and Standards

This section describes the steps that need to be taken to define a new class. Every class based on the **taBase** type (i.e., all classes in the PDP++ software) needs to have a set of standard methods (member functions) that allow it to interface with the rest of the software. Also, many commonly occurring data types and tasks that a class needs to perform have been dealt with in a standardized way. This chapter familiarizes the programmer with these standards and interfaces.

Defining a new class is typically the first step a user will take in programming with the PDP++ software. This is because the software is designed to be extended, not revised, by the user. Fortunately, most everything that is done by the PDP++ library code can be overwritten by defining a new class that does something differently, or simply by adding on to what the existing code does. Both of these approaches require the definition of a new class.

The first step in defining a new class is figuring out which existing class to base the new one on. This requires a knowledge of the existing class structure, which is covered in this manual. Once this has been decided, the guidelines in this section should be followed as closely as possible. It is assumed that the reader knows C++ and the basic ideas about constructors, destructors, virtual vs. non-virtual functions, etc.

5.4.1 Naming Conventions

The basic tension in naming something is the length vs. descriptiveness tradeoff. In general, we try to avoid abbreviations, but really long words like "environment" inevitably get shortened to "Env", etc.

The bulk of the conventions we have established have to do with distinguishing between different categories of names: class type names, member function names, and member names

being the major categories. In addition, the way in which names composed of multiple words are formed is discussed.

Object Class Names

Class types are first-letter-capitalized with words separated by capitalization: e.g. `MyClass`. There are certain exceptions, where an underbar '_' is used to attach a high-frequency suffix, usually from a template, to the name:

Common Suffixes:

`_List` taList derivative.

`_Group` taGroup derivative.

`_MGroup` MenuGroup derivative.

`_Array` taArray derivative.

`_SPtr` Smart Spec object pointer (PDP++).

Also if the class name contains multiple words, words which are actually acronyms ending with a capital letter are separated from the following word by an '_', e.g., (CE_Stat).

Classes in lower-level libraries also have the name-space identifier prefixed to the name, which is lower case: e.g., `ta`, `taiv`, `css`.

Enums

`enum` type names follow the same naming convention as class types. `enum` members are all upper-case, words are separated by '_', e.g., `INIT_STATE`.

Member Names

Members are lower-case, words are separated by a '_', e.g., member_name. One exception is for names ending in spec or specs in which case there is no separation (e.g., viewspecs).

Method Names

Methods are first-letter-capitalized, words are separated by capitalization (e.g., `RunThis()`). However, there some special prefixes and suffixes that are exceptions to this rule, because they are "high frequency" and denote a whole class of methods:

Prefixes:

`Dump_` Saving and loading functions.

`Compute_` Network computation functions (PDP++).

`Send_` Network communication functions (PDP++).

`C_` C code versions of process functions (PDP++).

`Init_` Special initialize function for processes (PDP++).

Suffixes:

`_impl`	Implementation (guts) of some other function which is the one that should be called by the user.
`_xxxx`	Other _impl type functions that do specific aspects of the implementation (xxx is lower case). Examples in PDP++ include _flag, _force.
`_`	(just a trailing underbar) This is a short version of _impl, which is used extensively in InterViews, and sparingly in TA/PDP++.
`_post`	A function which is to be called after another one of the same name (for two-step processes).
`_Copy`	A function called after the Copy function (e.g., to clean up pointers).
`_gui`	A special GUI version of function call.
`_mc`	A special menu callback version of function call.

5.4.2 Basic Functions

These are the functions that must be either specified or considered in any new instance of a **taBase** class:

`void Initialize()`

- It is called in every constructor.
- Do not call `Parent::Initialize()`, as this is a constructor function and the parent's will be called for you by C++ as the object is constructed.
- Set the initial/default values of each member in the class.
- Set the default type for groups that you own (`SetBaseType()`).
- Call `taBase::InitPointer(ptr)` on every taBase object pointer in class, or just set all pointers to NULL.
- EVEN IF NOTHING NEEDS INITIALIZING: use '`void Initialize() { };`' to avoid multiple calls to the parent's Initialize.

`void Destroy()`

- It is called in every destructor.
- Do not call the parent, as C++ will automatically call the parent's destructor for you.
- Free any resources you might have allocated.
- Call `CutLinks()`, if defined, to sever links with other objects.
- EVEN IF NOTHING TO DESTROY: use '`void Destroy() { };`' to avoid multiple calls to parents Destroy.

`void InitLinks()`

- Called when an object is linked into some kind of ownership structure.
- Call the `Parent::InitLinks()`, since this is not a constructor function and the parent's links will not otherwise be set.
- Own any classes contained as members: '`taBase::Own(recv, this);`'

- Set any pointers to objects with default values (e.g., '`spec->SetDefaultSpec(this));`', etc.
- Be sure to use '`taBase::SetPointer(ptr, new_val);`' for setting pointers.
- Or use '`taBase::OwnPointer(ptr, new_val);`' for those you own.
- If you do not need to do any of these InitLinks actions, then you do not need to define an InitLinks function.

`void CutLinks()`

- Called when an object is removed from its owner, or as part of the `Destroy` function when an object is actually deleted, or explicitly by the user when the object is a member of another object.
- At end of `CutLinks()`, call `Parent::CutLinks()`, since this is not always used as a destructor function, and parent's might not be called. Note, however, that when it is called in the destructor, it will be repeatedly called, so it should be robust to this (i.e., SET ANY POINTERS YOU DELETE TO NULL SO YOU DON'T DELETE THEM AGAIN!).
- Should sever all links to other objects, allowing them to be freed too.
- Call `CutLinks()` on any owned members, especially groups!
- Use `taBase::DelPointer()` on any pointers.
- If you have a spec, call `CutLinks()` on it.
- If you have group members, call `CutLinks()` on those groups.

`void Copy_(const T& cp), Copy(const T& cp)`

- Used to duplicate the class, Copy is the = oper and copy constructor
- Call `Parent::Copy` since this will not be called otherwise.
- `Copy_(const T& cp)` is an "implementation" that does the copying for just this class, and does not call the parent `Copy`.
- Use `COPY_FUNS(T, P);` (type and parent-type) to define the default macros for doing this:

```
void Copy(const T& cp)      { P::Copy(cp); Copy_(cp); }
```

- Use `SIMPLE_COPY(T);` to define a `Copy_` function that automatically copies the members unique to this class in a member-by-member (using TypeDef) way. This is less optimal, but easy when members are just simple floats and ints, etc.
- Be sure to use `taBase::SetPointer(&ptr, cp.ptr)` for copying pointers.

`TA_BASEFUNS(T);`

This defines the actual "basic" functions like constructors, destructors, `GetTypeDef()` etc. for taBase classes. These default constructors and destructors call the other functions like `Initialize()` and `Destroy()`.

`TA_CONST_BASEFUNS(T);`

This defines the actual "basic" functions like constructors, etc. for **taBase** classes which have `const` members defined in the class. These need to be initialized as part of the constructor, so this macro leaves out the default constructors and destructor, which should contain the following code:

```
MyClass() { Register(); Initialize(); SetDefaultName(); }
MyClass(const MyClass& cp)
 { Register(); Initialize(); Copy(cp); }
~MyClass() { unRegister(); Destroy(); }
```

`TA_TMPLT_BASEFUNS(y,T);`

Defines the actual "basic" functions like constructors, etc. for **taBase** classes which are also templates. `y` is the template class, `T` is the template class parameter.

`void UpdateAfterEdit()`

- Called after class members change via edit dialogs, loading from a file, or and assign operator in CSS.
- Maintain consistency of member values.
- Update links, etc.

When you add/remove/change any class members:

Check and add/remove/change initialization, copying, of this member in:

`Initialize()`

`Copy_()`

`Copy()`

For classes with Specs:

A pointer to a spec is encapsulated in a SpecPtr template class, which is declared once immediately after a new class of spec types is defined as follows (this will not typically done by the user):

```
SpecPtr_of(UnitSpec);  // this defines the template class
  (only for base spec type)
```

This pointer is then included in the class with the following:

```
UnitSpec_SPtr  spec;  // this puts a spec pointer in the class
```

Also, InitLinks() should have:

```
spec.SetDefaultSpec(this);
```

So that the spec pointer will set its pointer to a default instance of the correct spec type (the `this` pointer is because this also "owns" the spec pointer object.

For classes with taBase members:

All **taBase** members which appear as members of another class should be owned by the parent class. This increments their ref counter, so that if they are ever pointed to by something else (e.g., during loading this happens), and then unref'd, they won't then be deleted.

`InitLinks()` should own the object member as follows:

```
ta_Base::Own(obj_memb, this);
```

For members that derive from **taList** or **taGroup**, `Initialize()` should set the default type of object that goes in the group:

```
gp_obj.SetDefaultType(&TA_typename);
```

Referring to other objects via pointers:

If a class contains a pointer to another object, it should typically refer to that object whenever the pointer is set. The interface assumes that this is the case, and any pointer member that it sets will use the `SetPointer` function described below, which does the referencing of the new value and the dereferencing of the current one.

HOWEVER, when the pointer is to a physical PARENT of the object (or just higher in the deletion hierarchy) then it should not be referenced, as this will prevent the parent from being deleted, which will then prevent the child from being deleted.

In this case, and in general when the pointer is just for "internal use" of the class, and is not to be set by the user, the following comment directives should always be used: `#READ_ONLY #NO_SAVE` as this will prevent the user from overwriting the pointer, and the loading code automatically does a reference when setting a pointer, so these should not be saved. DO NOT COPY SUCH POINTERS, since they typically are set by the `InitLinks` based on the owner, which is usually different for different tokens.

When managing a pointer that the user can set, there are a set of convenient functions in taBase that manage this process (note that the argument is a *pointer* to the pointer):

`taBase::InitPointer(TAPtr* ptr)`
: initializes the pointer (or just set the ptr to NULL yourself) in `Initialize()`

`taBase::SetPointer(TAPtr* ptr, TAPtr new_val)`
: unRef's *ptr obj if non-null, refs the new one.

`taBase::OwnPointer(TAPtr* ptr, TAPtr new_val, TAPtr ownr)`
: like set, but owns the pointer too with the given owner.

`taBase::DelPointer(ptr)`
: unRefDone the object pointed to, sets pointer to NULL.

Using these functions will ensure correct refcounts on objects pointed to, etc.

If you `Own` the object at the pointer, then you should either mark the member as `#NO_SAVE` if it is automatically created, or `#OWN_POINTER` if it is not. This is because saving and loading, being generic, use `SetPointer` unless this comment is present, in which case they use `OwnPointer`.

Using Group Iterators:

There are special iterator functions which iterate through the members of a group. One method is to iterate through those sub-groups (including the 'this' group) which contain actual terminal elements ("leaves"). This is leaf-group iteration. Then, the elements of each group can be traversed simply using the `El` or `FastEl` functions.

- for leaf-group iteration, using macros (preferred method):

```
Con_Group* recv_gp; // the current group
int g;
FOR_ITR_GP(Con_Group, recv_gp, u->recv., g)
  recv_gp->UpdateWeights();
```

- for leaf-group iteration without macros:

```
Con_Group* recv_gp; // the current group
int g;
for(recv_gp = (Con_Group*)u->recv.FirstGp(g); recv_gp;
    recv_gp = (Con_Group*)u->recv.NextGp(g))
  recv_gp->UpdateWeights();
```

When all you care about are the leaf elements themselves, you can iterate over them directly using leaf-iteration:

- for leaf-iteration, using macros (preferred method):

```
Connection* con; // the current leaf
taLeafItr i; // the iterator data
FOR_ITR_EL(Connection, con, u->recv., i)
  con->UpdateWeights();
```

- for leaf-iteration without macros:

```
Connection* con; // the current leaf
taLeafItr i; // the iterator data
for(con = (Connection*)u->recv.FirstEl(i); con;
    con = (Connection*)u->recv.NextEl(i))
  con->UpdateWeights();
```

6 Guide to the Graphical User Interface (GUI)

This chapter provides a general guide and reference for using the graphical user interface to the PDP++ software. This covers all of the generic aspects of the interface—details about specific parts of the interface like the network viewer are found in the the section of the manual that covers the object in question (e.g., ⟨undefined⟩ [net-view], page ⟨undefined⟩).

6.1 Window Concepts and Operation

In PDP++ the hierarchy of objects provides the basis for most of the gui (Graphical User Interface) interaction. To access a sub-object of a class, the best place to start is with the gui window for the parent class instance and work your way down to the sub-object. The higher levels of the hierarchy have windows which are mapped to the screen when the object is created. These windowing objects inherit from the base class **WinBase**. When PDP++ starts up, only one of these classes has been created and thus there is only one window on the screen. The initial object is an instance of the class PDPRoot, and is the top of the PDP hierarchy.

6.1.1 How to operate Windows

PDP++ relies on your window manager for positioning, and iconifying the graphical windows of the program. Please refer to your window manager's manual for more information on the mouse movements and button presses needed to accomplish these tasks. For all the window objects in PDP++ there are CSS commands which ask the window manager to position or iconify the windows associated with the object. It is up to the window manager to provide the correct behavior for these "hints". PDP++ WinBase Window's position and iconification status can be manipulated with the following commands:

`GetWinPos()`
: Stores the window's current position and size on the object. When the object is saved the position and size of its window will be saved as well so that the window has the correct geometry when the object is loaded at a later time.

`ScriptWinPos()`
: Generates css script code for positioning the window at its current location and prints the code to the output window or to a recording script.

`SetWinPos(float left, float bottom, float width, float height)`
: Asks the window manager to resize and move the window to the specified parameters. If no parameters are given, this functions uses the parameters stored on the object.

`Resize (float width, float height)`
: Asks the window manager to resize the window to the specified parameters. If no parameters are given, this functions uses the parameters stored on the object.

`Move (float left, float bottom)`
: Asks the window manager to move the window to the specified parameters. If no parameters are given, this functions uses the parameters stored on the object.

`Iconify()`
: Asks the window manager to iconify the window.

`DeIconify()`
: Asks the window manager to deiconify the window.

6.1.2 How to operate Menus

In all WinBase windows, there is a horizontal menubar along the top. In this menubar is a "Object" menu (see Section 6.2 [gui-object], page 79), and also "Subgroup" menus (see Section 6.4 [gui-subgroup], page 81) for access to the sub-objects in this class.

To access the menus press and hold button-1 (left button) on the mouse while the mouse pointer is over the menu name. A smaller vertical menu window will pop up under the mouse pointer. Moving the mouse vertically while the button is still pressed will highlight the different choices in a the menu. Some of the items in the menu may have three dots '...' after them. When highlighting these menu items a cascaded menu will popup to the right of the selected menu item. The cascaded submenu may be traversed by moving the mouse pointer horizontally into the submenu, and then moving the mouse vertically as before. Again, button-1 must be pressed and held down during this operation. To select an item, release the mouse button while the mouse pointer is over the highlighted selection. To cancel the menu (to select nothing) move the mouse out of the menu and release the mouse button. Menu items which spawn submenus may not be selected. A menu may be pinned (it will stay on the screen after the mouse is released) by releasing the mouse while the pointer is on the menu name or on a submenu's name. A selection can be made from a pinned window by clicking (pressing and then releasing a button) on the desired selection. This will also cause the pinned menu to become unpinned and it will disappear.

Note: This applies specifically to the motif mode in InterViews. Open-look or other modes may be slightly different.

6.1.3 Window Views

Some objects may have multiple windows. These multiple windows are called views, and provide alternative methods of interacting with the same object and its data. **In a view window then menubar at the top of the window is split into a left menubar and a right menubar.** The left menubar contains menus whose actions pertain the the base object of the view. The right menubar contains menus whose actions pertain to this particular view only. For instance, a network object may have multiple netviews. In each netview there will be the left and right menubars along the top of the window. In the left menubar one would find menus with function that are particular to the network, such as menus for adding new layers, or removing all connections. In the right menubar on would find menus with functions that are particular to the view, such as setting the colors or shape of the units in the view.

Each view has an associated list of Schedule processes (see ⟨undefined⟩ [proc-sched], page ⟨undefined⟩) which update the view in the course of their processing. By adding or removing these "updaters", one can control the grain at which changing data is displayed. For instance, in the netview the colors of the units change to reflect their current values whenever the view is updated. If the updater was a trial level process then the view would display new values for all the units after every trial.

WinView Functions

`AddUpdater(SchedProcess* updater)`
: Add the schedproc to the list of updaters for this view. (and vice-versa).

`RemoveUpdater(SchedProcess* updater)`
: Removes the schedproc to the list of updaters for this view. (and vice-versa).

`InitDisplay()`
: Initializes and graphically rebuilds the view's display.

`UpdateDisplay()`
: Refreshes the winview's display to reflect changes in the base object's values. This is the function that is called by the updater processes.

6.2 The "Object" Menu

The "Object" menu always appears as the left most menu in a WinBase's menubar. The "Object" menu for each object is used to perform file based actions on the object itself. Many of these actions involve the use of a file requester dialog. See Section 6.8 [gui-file-requester], page 93. These actions on the object can also be called through css (e.g., the Print action can be called from css as '`object.Print()`'). The "Object" menu has the following menu actions:

`Load`
: Load a text object dump of an object of the same class as this object on top of this object, replacing the values of fields of this object with the values of the saved object's fields. The saved object file is selected with the file requester.

`Save`
: Save a text object dump of this object in a file created with the file requester, or with the object's most recently used filename for saving.

`SaveAs`
: Save a text object dump of this object in a -new- file created with the file requester.

`Edit`
: The Edit menu action brings up an Edit Dialog on the object. See Section 6.5 [gui-edit], page 82.

`Close`
: The Close menu action will attempt to close/delete the object. If the object is referenced or pointed to by other objects, then it will not actually be deleted, only the windows which display it will be removed. The user must confirm the deletion if it is possible to safely delete the object. **NOTE THAT CLOSE IS NOT ICONIFY!, it really does delete the object, not just close the menu.**

`Copy From`
: Copies from another object of the same or related type — replace all of the current data in the object with those in another. In the menu, only the same

or subtypes of this object will be shown, but in the script, any type of related object can be passed to this function.

`Copy To` Copies the data in this object to another object. This can be useful if you want to copy from a more basic type of object (e.g., Environment) to a derived type (e.g., FreqEnv) – CopyFrom won't show the more basic type of object to copy from, but CopyTo will show the derived type.

`DuplicateMe`
Makes another copy of this object – creates a new object and then copies from this current object to that new object. Note this is DuplicateMe in the script code.

`ChangeMyType`
Changes the type of this object to be another related type (e.g., change to a FreqEnv from an Environment). Will usually do a good job of updating the various links to this object if changed. Not good for objects within a network, or generally for Stat objects that are aggregated. This is ChangeMyType in the script code.

`SelectForEdit`
Allows you to select a field (member) of this object to be edited in a **SelectEdit** object, which consolidates parameters and functions across multiple objects into a single edit dialog (see ⟨undefined⟩ [proj-seledit], page ⟨undefined⟩).

`SelectFunForEdit`
Allows you to select a function (method) of this object to be accessible from a **SelectEdit** object, which consolidates parameters and functions across multiple objects into a single edit dialog (see ⟨undefined⟩ [proj-seledit], page ⟨undefined⟩).

`Help` Will automatically pull up a help browser for information relevant to this object. Depends on the browser actually running on your system, as specified in the Settings on the root object.

`Print` The Print menu action will save a snapshot of the entire object's window to a file in Postscript format using a file requester. Note that this printout file uses structured graphics so it will scale well if resized, etc.

`Print Data`
The Print data menu action will save a snapshot of the window's data, not including the menubars and window decoration, to a file in Postscript format using a file requester.

`Update Menus`
If objects are created or deleted, sometimes the menus of their parent objects can become out of date. If this appears to be the case then use the "update menus" menu action on the parent object to fix the menus. The "update menus" menu action recursively traverses the menus of the object and its subobjects, adding and deleting menu items appropriately.

`Iconify` This will iconify the window (shrink down to an iconic representation, to get it out of your way).

6.3 The "Actions" Menu

Some objects will have an "Actions" menu following their "Object" menu. In the Action menu are functions which are apply specifically to the object and are not the common file based functions found on the Object menu. If you wish to perform a function directly on the object you are viewing in the window, the "Actions" menu is a good place to look. For more specific information concerning an object's "Action" menu please refer to the section of this manual which pertains to the object itself.

6.4 The SubGroup Menu(s)

A subgroup menu appears in the menubar of a WinBase object for each of the group members of the object. The name of the menu corresponds to the name of the group member (e.g., The Layer object has a group of units and a group of projections. It would have two subgroup menus, one labeled `.units` and a second labeled `.projections`. The "." before the name of the subgroup is used to indicate that the subgroups are sub-objects of the WinBase object. In the CSS script language one would access objects in these subgroups using the "." operator. (e.g., To access the first network in the project one would type '`.projects.networks[0]`'). In addition the subgroup menus appear in a non-italicized font to distinguish them from the "Object" and "Action" menus of the WinBase. The subgroup menu's have the following menu choices. Occasionally subgroup menus may add additional menu choices as well (e.g., The Processes submenu of the Project has the menu choice *Control Panel* which opens a control panel dialog for one of the processes in the subgroup). Some of the group operations require the use of a file requester. See Section 6.8 [gui-file-requester], page 93.

`Edit` The Edit action brings up a Group Edit Dialog for the group or an Edit Dialog for an individual object (Section 6.5 [gui-edit], page 82).

`New` The New action allows the use to create new objects in the group or in a subgroup of the group. The user can chose to create objects of the base object type, objects of a subclass of the base object type, or a subgroup object. A popup dialog appears which enables the user to select the number of objects to create, the type of objects, and where to place them. Sometimes the popup dialog may have additional fields and toggles which are particular to the item being created. If the auto_edit flag in the global settings (see Section 6.6 [gui-settings], page 87) is turned on, an edit dialog will be created for the newly created objects when button-1 (left button) is pressed on the OK button of the popup dialog. If button-2 (middle button) is pressed the edit dialog will be created only if the auto_edit variable is off. When button-3 (right button) is pressed an edit dialog will always be created.

`Open In` The "Open in" action allows the user to open a previously-saved object file and add the data into the group. See Section 7.1.3 [obj-basics-files], page 96.

`Load Over` The "Load Over" action allows the user to open a previously-saved object file and overwrite the objects in the group with the data in the file. See Section 7.1.3 [obj-basics-files], page 96.

`Save` The Save action allows the user to save the group or a group element as a PDP++ object file using the file requester or the object's most recently used name for saving. See Section 7.1.3 [obj-basics-files], page 96.

`Save As` The Save As action allows the user to save the group or a group element in a new PDP++ object file using the file requester. See Section 7.1.3 [obj-basics-files], page 96.

`Remove` The Remove action allows the user to remove the group or a group element. The user is prompted with a confirmation dialog to confirm the choice. However, If the chosen object is referenced by other objects then it will not be deleted.

`Duplicate`
The Duplicate action allows the user to add a duplicate of one of the objects in the group to the group or its subgroups.

`Move Within`
This allows the user to move objects to new positions within the group (e.g., for rearranging the order of layers in the network, which is important for the feedforward Bp algorithm).

`View Window`
View Window brings the window associated with selected object to the front, deiconifies it, or creates it if a window does not exist.

6.5 The Edit Dialog

The Edit Dialog allows the user to both visually inspect and modify the values of an object's fields. As an inspection tool, the user can use and edit dialog to check the values of an object's fields, and as an editing tool the user can use the edit dialog to change the values of some or all of those fields. For some objects, the edit dialog is the only representation available for inspecting, or accessing it members. Other objects may have extended edit dialogs or even multiple views which may also allow modification of the object's fields. An edit dialog also presents the user with easy access to the substructures of an object including editing of its subgroups and arrays. In addition the edit dialog may allow access to certain member functions on an object. Edit Dialogs are created by choosing "Edit" from an object's menu or by the `EditObj` command in css (see Section 4.4.6 [css-commands], page 35).

The basic layout of the edit dialog includes a list of object member names which are listed in a vertical column along the left hand side of the display. Clicking on a member name will popup a short description of the member. Certain members may not appear in the edit dialog. This is determined by the `show_iv` field of the global settings (see Section 6.6 [gui-settings], page 87). Typically the unshown members will not be of interest to the average user, or will contain values which are potentially dangerous to change.

The *Show* menu on the right-hand side of the menu bar at the top of the edit dialog will allow you to control how much stuff to view on a case-by-case basis. Typically, you'll only want to switch to viewing *Detail*, which is not viewed by default, but all levels can be controlled in this menu. If you suspect that something is not there which should be, try selecting *Show/Detail*.

Fields that are highlighted in bright yellow indicate that the field value is at variance with the default value for that field.

The values associated with the member names appear to the right. There are a number of different graphical representations of the content of these fields, which are described in the following section.

In addition, there can be special buttons and menu actions available on the dialog. These are described in subsequent sections.

6.5.1 Member Fields

The member fields in an edit dialog can be found to the right of the member name. These member fields can appear in many different forms. Sometimes there may even be more than one on a line.

Field Editor

The Field Editor is used for editing strings (e.g., an object's name), and number values. It appears as a box with text in it. If you click on the box an Ibeam cursor appears at the click location. When the mouse is in the box, the characters you type will be entered at the Ibeam location point as you would expect.

You must place mouse pointer in field to start editing! However, once you have started typing, the mouse pointer need only remain in the overall window.

Some Emacs style editing keys are also recognized, in addition to the standard keypad arrow and related keys:

Ctrl-f moves Ibeam forward (also right arrow key)

Ctrl-b moves Ibeam backward (also left arrow key)

Ctrl-a moves Ibeam to beginning of line (also Home)

Ctrl-e moves Ibeam to end of line (also End)

Ctrl-d deletes character to right of Ibeam (also Del)

Ctrl-u selects the entire line – subsequent typing replaces contents

Ctrl-n, TAB, or RETURN moves the Ibeam to the next field editor

Ctrl-p or Shift-TAB moves the Ibeam to the previous field editor

Alt-n moves to next object in list editor (also PageDn)

Alt-p moves to previous object in list editor (also PageUp)

You can also hit shift-mousebutton2 (middle button) to scroll the text with the hand cursor.

If the item to be edited is an integer then increment/decrement arrow buttons will appear to the right of the field editor. Pressing on these buttons will increase or decrease the value in the field editor. Pressing button-1 (left button) changes the value by 1, button-2 (middle button) changes the value by 10, and button-3 (right button) changes the value by 100. If the mouse button is pressed and held, the action will auto repeat.

Read Only Member Field

The read only member field is like the FieldEditor except the user cannot edit the field. It is used for display purposes only.

Boolean CheckBox

The checkbox is used for boolean values that are either on or off. Clicking on the checkbox changes its values. A checkmark or solid block indicates an ON, TRUE or 1 value, while no checkmark or an empty box indicates an OFF, FALSE or 0 value.

Enum Menus

The Enum menu is used for enumerated types. These types may have one of a number of symbolic values. The menu is shown with the member's current value. by clicking on the menu the other value choices appear in a popup menu. Selecting an alternate value from the menu sets the menu's value to the new value.

SubObject Menus

A SubObject menu is used when an instance of a class is part of the edited object and that instance is not edited inline (see Section 6.5.1 [gui-edit-fields], page 83). The menu has the name "Type: Actions" where Type is the class name of the subobject. The menu contains the following functions as well as object specific functions that are unique to the subobject class.

Load: Load an instance of the subobject class

Save: Save the subobject using a file requester

SaveAs: Save the subobject with its last saved name

Edit: Edit the subobject

Object Pointer Menus

Object pointer menus are used when the member is a pointer to another object. The current value of the pointer is shown in the menubar as the pathname of the object that is pointed to. If the pointer is not set, the name on the menubar will be set to NULL. Clicking on the menubar brings up a popup menu of other objects of similar type. Selecting an alternate object sets the menubar to the name of that object. The popup menu also allows the user to set the pointer to NULL and to Edit the object currently pointed to.

Object Type Menus

The object type menus are used when a member specifies a type class of object. This menu allows the user to select from classes which inherit from it base class. The current selection is displayed in the menubar.

SubGroup Menus

The subgroup menu allows the user to interact with a group member of the object. The name on the menubar is one of the two forms depending upon whether or not the group has subgroups. (see Section 7.2 [obj-group], page 97).

Clicking on the menubar gives the user the following choices:

New Create new object in the group

Load Load a group from a file using a file requester. See Section 7.1.3 [obj-basics-files], page 96.

Save Save the group using a file requester. See Section 7.1.3 [obj-basics-files], page 96.

SaveAs Save the group with its last saved name. See Section 7.1.3 [obj-basics-files], page 96.

Edit Bring up a Group Edit Dialog of the subgroup

RemoveAll
Remove all the elements of the group

Remove Remove an individual element of the group

Link Link an object from another group into this group

Move Rearrange the order of the elements in this group

Transfer Transfer objects from another group into this one

EditEl Edit and individual element of the group

Find Find and edit an object with a given name

Each of these selections will bring up a dialog for the user to specify the parameters to the operation.

SubArray Menus

The subarray menu allows the user to interact with a array member of the object. It appears as an "Edit" menubar. Clicking on the menubar gives the user the following choices:

Load Load an array from a file using a file requester. See Section 7.1.3 [obj-basics-files], page 96.

Save Save the array using a file requester. See Section 7.1.3 [obj-basics-files], page 96.

SaveAs Save the array with its last saved name. See Section 7.1.3 [obj-basics-files], page 96.

Edit Bring up an Array Edit Dialog for the array

Remove Remove a number of elements from the array at a given index

Permute Permute the order of the elements in the array

Sort Sort the order of the elements in the array

El Show the value of a member of the array at given index

Add Add elements onto the end the array

Insert Add a number of given values at a certain Array index

Find Find and Show the index of a member of the given value

ColorEdit Bring up a Color Array Editor for number arrays

Inline Fields

Inline Fields are used for small substructures within an object (e.g., an object's position, which is only three values: x,y,z). Instead of the expected SubObject menubar, the object is laid out horizontally across on line in the Edit Dialog. The names of each of the fields are often truncated so that the line will not take up too much space. This allows quicker access and visualization without having to traverse through multiple dialogs. The fields within an inline field behave as do the rest of the fields in the dialog.

6.5.2 Edit Dialog Buttons

At the bottom of the Edit Dialog is a row of buttons which have actions pertaining to the Edit Dialog's fields. Some of the buttons may be appropriately inactive due the state of the edit dialog. They will become active when their action is applicable. The following buttons will be available:

`Ok` The Ok button stores the information in the dialog's fields in the object's fields and closes the dialog window.

`Apply` The Apply button stores the information in the dialogs fields but does not close the window. If the apply button is highlighted the edit dialog may contain changes which have not been applied yet. The apply action can also be generated by pressing the Return key in the body of the editor, unless the editor has multiple field editors within it (see Section 6.5.1 [gui-edit-fields], page 83).

`Revert` The Revert button reloads the information from the object into the dialog's fields, clearing any changes the user might have made. If the revert button is highlighted the object's fields might have been changed "behind the scenes" by another part of the program in which case the information displayed in the edit dialog may be outdated. The revert action can also be generated by pressing the Escape key in the body of the editor.

`Cancel` The Cancel button dismisses the dialog without applying the user's changes.

`Other Member Buttons`
Some objects will add an additional row of member buttons to the Edit Dialog.(e.g., The Process object adds a row of Run,Step,Init buttons). These buttons perform additional actions on the object.

6.5.3 Edit Dialog Menus

At the top of the edit dialog is a menubar with at least two menus. The first menu entitled Object is similar to the Object menu on a WinBase window. If the edited object

is not a WinBase, the "Object" menu my only provide the "Load", "Save, "Save As", and "Close" menu choices. (see Section 6.2 [gui-object], page 79). The second menu called "Actions" may contain member functions particular to the edited object's type which will be called on the edited object.

6.6 Settings Affecting GUI Behavior

There are two primary ways of setting parameters which affect the look of the GUI. There is an object called **taMisc** which contains miscellaneous parameters and settings that affect various aspects of the PDP++ system, but mostly the graphical interface. This object can be edited by using the *Settings* menu option on the **PDPRoot** object.

In addition to the taMisc settings, there are a number of XWindow resources or Xdefaults that can be set. Also, since PDP++ uses the InterViews graphics toolkit a number of command line arguments and InterViews Xresources can also be customized (see ⟨undefined⟩ [proj-startup], page ⟨undefined⟩).

6.6.1 Settings in taMisc

The following parameters can be set:

`int display_width`
: Width of the shell display in characters.

`int sep_tabs`
: Number of tabs to separate items by in listings.

`int max_menu`
: The maximum number of elements in a menu — if there are more than this number of instances (tokens) of a given type of object, then the menu for that type of object will say "<Over max, Select>", meaning that each item could not be listed separately in the menu, so you select this option to pull up a object chooser (see Section 6.9 [gui-obj-chooser], page 94), which allows you to choose the object from a longer list.

`int search_depth`
: The recursive depth at which css stops searching for an object's path.

`int color_scale_size`
: This determines how many colors are in a color scale for a color monitor (see Section 6.7 [gui-colors], page 91).

`int mono_scale_size`
: This determines how many colors are in a color scale for a monochrome monitor (see Section 6.7 [gui-colors], page 91).

`ShowMembs show`
: Type of members to show in css.

`ShowMembs show_iv`
: Type of members to show in edit dialogs The previous two variables can have one of four "ShowMembs" values:

`ALL_MEMBS`
: Shows all members.

`NO_READ_ONLY`
: Shows all but read_only members – stuff that can't be changed.

`NO_HIDDEN`
: Shows all but hidden members – stuff that is hidden because it is not typically relevant.

`NO_HID_RO`
: Shows all but hidden and r/o members.

`NO_DETAIL`
: Shows all but "detail" members – stuff that might be relevant, but is not often accessed.

`NO_HID_DET, etc.`
: Further combinations of RO, HID, and DET options.

The default is `NO_HID_RO_DET`.

`TypeInfo type_info`
: The amount of information about a class type that is reported when the "type" command is used in CSS. Type_info has one of the following values:

`MEMB_OFFSETS`
: Shows the byte offset of members.

`All_INFO` Shows all type info except memb_offsets

`NO_OPTIONS`
: Shows all info except type options

`NO_LISTS` Shows all info but lists

`NO_OPTIONS_LISTS`
: Shows all info but options and lists

The default is `NO_OPTIONS_LISTS`

`KeepTokens keep_tok`
: This sets the Default for keeping tokens (lists of members) for object types. This can have one of three "KeepTokens" values:

`Tokens` Keep tokens as specified in the type.

`NoTokens` Do not keep any tokens lists.

`ForceTokens`
: Force the keeping of all tokens lists.

The default is `Tokens`.

`verbose_load`
: Indicates the amount of information reported in the shell window when objects are loaded

`iv_verbose_load`
: Indicate the amount of information reported in the Loading window when objects are loaded The previous two members may have one of the following values:

  `QUIET` No information is given

  `MESSAGES` General messages are reported for each object

  `TRACE` Each line of the file is printed as it is loaded

  `SOURCE` Intricate debugging details are reported

`bool auto_edit`
: If this value is on, an edit dialog will appear each time an object is created with the New dialog. (see Section 6.4 [The Subgroup Menu(New)], page 81).

`AutoRevert auto_revert`
: In some cases a dialog may be reverted to its old values by a PDP++ process. This variable controls the behavior of the edit dialog when this situation occurs. The behavior is determined by one of three values:

  `Auto_Apply`
  : Automatically apply changes before auto-reverting

  `Auto_Revert`
  : Automatically revert the edit dialog loosing changes

  `Confirm_Revert`
  : Popup a confirmation dialog to okay reverting

`String include_paths`
: Directory Paths used for finding files. This is an array of string values that represent paths to find CSS, defaults files, and Help files in (see Section 4.4.8 [css-settings], page 46, see ⟨undefined⟩ [proj-settings], page ⟨undefined⟩, see ⟨undefined⟩ [proj-objdef], page ⟨undefined⟩).

`String tmp_dir`
: The directory to use for temporary files

`String compress_cmd`
: Command to use for compressing files

`String uncompress_cmd`
: Command to use for uncompressing files

`String compress_sfx`
: Suffix to append to filenames when compressed

`String help_file_tmplt`
: Template for converting the type name of an object into a help file – %t is replaced with the type name. Include any leading paths to help files relative to the basic root paths listed on include_paths.

`String help_cmd`
: Comand for bringing up a help browser to read help file (typically html, netscape by default). %s is subsituted with the help file produced from `help_file_tmplt`.

6.6.2 XWindow Resources (Xdefaults)

The following XWindow resources can be set. These are typically placed in the user's '`.Xdefaults`' file in their home directory.

`PDP+++*flat`
: The color of the buttons. It is typically some kind of greyish color. The default has a blueish tint: `#c0c4d3`

`PDP+++*background`
: The color of various non-view background regions. The default is an aquamarine color: `#70c0d8`.

`PDP+++*name*flat`
: If you change the valueof background, you should change this one to match. It makes the member names in the edit dialogs (see Section 6.5 [gui-edit], page 82) the same color as the background.

`PDP+++*apply_button*flat`
: The color of the apply button (and other buttons) when they are the "suggested" choice. It is a dusty-red by default: `#c090b0`.

`PDP+++*FieldEditor*background`
: The color of the edit fields, default is white.

`PDP+++*font`
: The generic font to use for buttons, etc: `*-helvetica-medium-r-*-*-10*`.

`PDP+++*name*font`
: The font to use for member names: `*-helvetica-medium-r-*-*-10*`.

`PDP+++*title*font`
: The font to use for titles of edit dialogs: `*-helvetica-bold-r-*-*-10*`.

`PDP+++*small_menu*font`
: The font to use for small menus (e.g., those in edit dialogs): `*-helvetica-medium-r-*-10*`.

`PDP+++*small_submenu*font`
: This font is used for menu items that contain sub-menus (these are also indicated by three dots after the name) `*-helvetica-medium-r-*-*-10*`.

`PDP+++*big_menu*font`
: This font is used for big menus like those on the permanent object windows, default is `*-helvetica-medium-r-*-12*`.

`PDP+++*big_submenu*font`
: For sub menus on big menus: `*-helvetica-medium-r-*-*-12*`.

`PDP+++*big_menubar*font`
: For big menus, the name of the menu itself: `*-helvetica-bold-r-*-*-14*`.

`PDP+++*big_italic_menubar*font`
: For big menus, name of italicized menu items (i.e. those that apply to the object and not to its sub-objects): `*-helvetica-bold-o-*-*-14*`.

`PDP++*double_buffered:`
: Double buffering makes the display smoother, but can use lots of display memory. Default is "on".

`PDP++*FileChooser.width`
: The width of the file-chooser in pixels. Default is 100.

`PDP++*FileChooser.rows`
: The number of items to list in the file chooser, default is 20.

`PDP++*clickDelay`
: This is the number of milliseconds to count two clicks as a double-click. The default is 250.

For MS Windows Users:

To set XResources under MS Windows, you need to create an "application defaults" file that is formatted much like the XResources. There are two steps for creating this application defaults file:

1. Edit '`C:\WINDOWS\WIN.INI`', and add the following two lines:

```
[InterViews]
location = C:\PDP++
```

Where the location should be the actual location where you installed the software.

2. Create a sub-directory under '`C:\PDP++`' (again, use the actual location) called '`app-defaults`', and then create a file called '`InterViews`' in that directory. This file should contain resource values you want to set. For example, to change the amount of time to detect a double-click, you would enter:

```
*clickDelay: 400
```

To change the overall size of the PDP++ windows (scaling):

```
*mswin_scale: 1.25
```

6.7 Color Scale Specifications

Color scales are used in PDP++ to display the values of variables graphically in various types of displays. The choice of color scale depends on personal preferences as well as what type of display the user has available. There are a number of different types of color scales that come with the software, and the user can create their own custom colorscales.

A color scale is specified by creating a set of different color points. The actual scale is just the linear interpolation between each of these points, where the points are distributed evenly through the range of values covered by the scale. Thus, if there were three such points in a color specification that goes from -1 to 1, the first point would represent the value -1, the middle one would represent 0, and the last one would represent 1. Values in between would be represented by intermediate colors between these points. The actual number of colors created in a given color scale is determined by the `color_scale_size` setting parameter for color displays, and `mono_scale_size` for monochrome displays (see Section 6.6 [gui-settings], page 87).

The **PDPRoot** object contains a group of color specifications called `.colorspecs`, which is where the default color scales and any new ones the user creates are located. The default element (see Section 7.2 [obj-group], page 97) of this group represents the default color scale to use when creating a new display that uses color scales. The defaults are as follows:

`C_ColdHot`
: interpolates from violet->blue->grey>->red->yellow

`C_BlueBlackRed`
: interpolates from blue->black-red

`C_BlueGreyRed`
: interpolates from blue->grey->red

`C_BlueWhiteRed`
: interpolates from blue->white->red

`C_BlueGreenRed`
: interpolates from blue->green->red

`C_Rainbow`
: interpolates from violet->blue->green->yellow->red

`C_ROYGBIV`
: interpolates from violet->indigo->blue->green->yellow->red

`C_DarkLight`
: interpolates from black->white

`C_LightDark`
: interpolates from white->black

`M_DarkLight`
: dithers from black->white

`M_LightDark`
: dithers from white->black

`M_LightDarkLight`
: dithers from white->black->white

`P_DarkLight`
: dithers from black->white with a white background for printing

`P_DarkLight_bright`
: dithers from black->white with a white background for printing, having a brighter overall tone than the basic one (a lighter zero value).

`P_LightDark`
: dithers from white->black with a white background for printing

`P_DarkLightDark`
: same as M_DarkLightDark with a white background for printing

`P_LightDarkLight`
: same as M_LightDarkLight with a white background for printing

The **ColorScaleSpec** is the object that specifies the color scale. It contains a group of RGBA objects, each of which is used to specify a point on the colorscale range based on the Red, Green, and Blue values, plus a "transparency" parameter Alpha. The ColorScaleSpec object has one primary function.

`GenRanges (ColorGroup* cl, int nper)`
: This function creates a range of colors in the ColorGroup by linearly interpolating nper colors for each RGBA set point value in the ColorScaleSpec.

The **RGBA** object has the following fields:

`String name`
: The name of the color

`float r` Amount of red in the color (0.0 - 1.0)

`float g` Amount of green in the color (0.0 - 1.0)

`float b` Amount of blue in the color (0.0 - 1.0)

`float a` Alpha intensity value (ratio of foreground to background) This is used primarily for monochrome displays.

If the name field of an RGBA object is set, then it will try to lookup the name to find the r,g, and b values for that color.

Thus, to create your own color scale specification, just create a new ColorScaleSpec object, and then create some number of RGBA objects in it. Then, edit your views (e.g., the network view, ⟨undefined⟩ [net-view], page ⟨undefined⟩), and set their `colorspec` to point to your new specification. In order to see changes you make to your color spec, you need to switch the `colorspec` pointer to a different one and then back to yours after making the changes.

6.8 File Requester

The File Requester Dialog is used for choosing filename for reading and writing to files. The directions for the file manipulation are listed at the top of the dialog. Typically the directions will ask the user to "Select a File for xxx" where xxx is one of "Opening", "Appending", "Writing", etc. Under the directions is the prompt "Enter Filename", and a FieldEditor where the user can type in the name of the file.

Below the FieldEditor is a vertical scrollbox of filenames in the current directory. If there are more names than will fit in the window, the scrollbar on the right edge of the scrollbox can be used to scroll through the full listing. A filename can be chosen by clicking on the name in the scrollbox. Subdirectories are listed with a slash ('/') following the name, and can be read in by double-clicking on their name. The "../" directory can be used to navigate up a directory level.

Below the filename scrollbox is a Field Editor for the filename filter. The String listed here is a unix csh filename completion string. Wildcards can be specified using the '*' character. See the csh man page for more details. Typically this field will be set by PDP++ to limit the range of filenames available to those which correspond to the type of file the dialog is to act upon. (e.g., If projects are being loaded, then the filter will be set

to "*.proj.*" to limit the selection to files with ".proj" in the filename.) In most cases, compressed files can be loaded and saved as well.

Below the filter Field Editor are the action buttons. The leftmost button will perform the action specified in the directions at the top of the dialog. The rightmost button is the "Cancel" button and will exit the dialog without performing the action. Double-clicking on a filename, or pressing the "return" key, also causes the dialog's action to be taken.

6.9 Object Chooser

The Object Chooser Dialog is much like the file requester described previously, and is used primarily for choosing objects when there are too many such objects to fit within a menu. It can also be used to browse the entire hierarchy of objects. In the simple choosing function, just scroll and pick the object – double clicking to select, or using the Select button at the bottom. A name can also be typed into the field at the top of the browser.

By selecting *PDP++Root/Browse*, the object chooser acts more like a file browser. When you select an object, the default is to descend into the sub-objects on that object. Pressing .. will go back up a level (as in the file system). When you select an object, an edit dialog will show up for that object. This can be used for editing things that may not show up in a useful way in standard edit dialogs (e.g., the StatVal's in Stats, see ⟨undefined⟩ [proc-stat], page ⟨undefined⟩).

7 Object Basics and Basic Objects

PDP++ is written in the computer language C++. C++ extends the C language in many ways, but its primary addition is the concept of objects and object oriented programming (OOP). Many PDP concepts such as Units, Layers, and Networks easily fall into the object paradigm, while some other concepts such as Processes and Environments are initially difficult to grasp in this framework. The following chapters explain the particulars of the PDP concepts in the object-based paradigm in greater detail. This chapter establishes a number of conventions in PDP++ used for organizing and structuring its objects. It is really only relevant if you will be programming with PDP++, either in the script language or directly in C++ — the end user generally need not be concerned with this level of detail.

In addition to these object basics, this chapter also describes a number of basic types of objects that are used in many different places within the software. Examples of these include lists, groups, arrays, and specifications.

7.1 Object Basics

The following sections describe some basic ideas about objects and general things you can do with them.

7.1.1 What is an Object?

An object has both fields with data and functions (aka methods) which it can perform. A PDP Unit as an object might have the fields `activation` and `netinput`, and some functions like `ComputeActivation()` and `ClearNetInput()`. In C++ this might look like:

```
class Unit {
  float         activation;     // this is a member holding activation
  float         netinput;       // this is a member holding net input

  virtual void  ComputeActivation(); // this is a function
  void          ClearNetInput();        // this is a function
}
```

C++ also provides the mechanisms for object inheritance in which a new object can be defined as having the same fields and functions as its "parent" object with the addition of a few fields and/or functions of its own. In PDP++ the **BpUnit** object class inherits from the parent **Unit** class and adds fields such as `bias weight` and a BackProp version of `ComputeActivation`. In C++ this might look like:

```
class BpUnit : public Unit {
  float         biasweight;     // a new member in addition to others

  void          ComputeActivation(); // redefining this function
  void          ClearBiasWeight();  // adding a new function
}
```

By *overloading* the ComputeActivation function the BpUnit redefines the way in which the activation is computed. Since it doesn't overload the ClearNetInput function, the BPUnit clears its net input in the exact same way as the standard unit would. The new

function ClearBiasWeight exists only on the BpUnit and is not available on the base Unit class. Through this type of class inheritance, PDP++ provides a hierarchical class structure for its objects.

For a slightly more detailed treatment of some of the basic ideas in OOP and C++, see Section 4.3.4 [css-c++-intro], page 29.

7.1.2 Object Names

In PDP++, many objects have a `name` field. This gives the particular object an identifier that can be used to refer to it in the CSS script language (see Section 4.2.3 [css-tut-access], page 23), and it makes it easier to identify when you need to point to it one of the GUI operations.

All objects when created are given default names that consist of the their type followed by a number which increases for each subsequent object of that type which is created. Thus, the first **Layer** created will be called "Layer_0", and so on.

Note that the name given to an object is not necessarily the same name that object will have when saved to a file. It is simply a value of a particular member of the object.

7.1.3 Saving and Loading Objects

All PDP++ objects can also be saved to a file and loaded back in from that file. Note that when you save an object that has sub-objects in it (i.e. a Layer, which has Units in it), those sub-objects are also saved in the file, and loaded back in as well. Thus, to save all the elements of a project, one only has to save the project object, which will automatically save all of its sub-objects.

There are two ways in which objects can be loaded back in: in one case, you load the file *over* an existing object. This corresponds to the *Load Over* menu item that you will frequently see in PDP++. This will simply replace the values of the existing object with those saved in the file. This also applies to any sub-objects. Note if the existing object has *fewer* of some kind of sub-object than the saved file, new sub-objects will be created as needed. However, the same is not also true if the existing object has *more* of these sub-objects—they are not removed to make the existing object exactly the same as that which was saved in the file. If you want to make sure the object is exactly as it was saved, open a new one, do not open over an existing one.

To open a new object from a saved file, use the *Open in* menu item (instead of *Load Over*). This is equivalent to opening a new item into a parent group object. For example, the project has a group that contains its networks. If you open a saved network file in this group, as opposed to over an existing network in the group, a new network will automatically be created (along with all of its saved substructure).

Some types of objects have default "extensions" that are automatically added on to the file name to identify the type of file. For example, projects are saved with a '`.proj`' extension. Further, some objects automatically compress the file after saving, (and automatically uncompress it before loading) in order to save disk space. This is because the file format objects are saved in is actually text-based and human readable. Thus, it is much more efficient to save large files in a compressed form. These files automatically get the

compression suffix associated with the type of compression used (.gz for gzip, which is the default).

7.2 Groups

Groups are one of the major workhorse objects in the PDP++ software. Virtually every object is contained in a group object. Thus, groups provide a way of managing other objects, creating them, ordering them, and iterating through them.

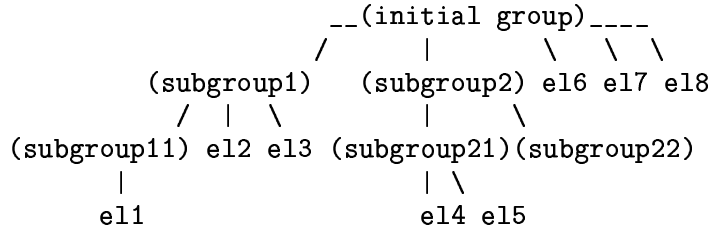
A particular group typically manages objects of the same type. Thus, a group of units always contains units of one form or another. Groups have a *default type* of object that they expect to contain, and this is what they will create if asked to create a new object.

Groups also have a notion of a *default element*, which can be useful. For example, the default element of the group of color specifications on the root object is the default color spec that will be used by all windows that need a color spec.

There are two basic types of groups. The simpler form of group is actually just a **List**. A **List** object manages a single list of elements, and does not allow for any sub-grouping of these elements. Lists are used to hold simple objects that probably don't need to have such subgroups.

A **Group** may also contain subgroups, which are similar in type to the group itself. The nested structure of subgroups within subgroups within groups can be organized into a conceptual "tree" of groups and elements. The initial group comprises the base of the tree with each of its subgroups representing a branch. The subgroup's subgroups continually branch off until in the end a group is reached without any further subgroups. The actual elements of the group are conceptually represented as "leaves", and may occur within any group at any level of the group tree.

`Ex:`



In this example there are six groups and eight "leaf" elements. The leaves are numbered in their Depth First Search retrieval order which is used by the `Leaf()` function described in the group functions section See Section 7.2.3 [obj-group-functions], page 99.

In some cases it useful for the same group element to occur in more than one group at time. The parent groups share the element, however, one group is assigned to be the owner (usually the initial group the element was created within). Usually a group will own all of its elements, but sometimes it may contain elements from other groups as well. Those elements which are owned by other groups are known as *linked* elements, and are added to a group by using the `Link()` function described in the group functions section. When

a group is deleted, it breaks its link to the linked elements, but does not delete them as it does the other elements it owns. Subgroups can also be linked.

```
Ex:
        (group1)                  (group2)
        /   |   \                /    |  \
(subgroup1) el1  \.............el2    el3 \
  /  \                                    \
el4   \.................................(subgroup2)
```

In this example group1 has a link to element el2 in group2. El2 is owned by group2 and is a link element of group1. In addition, subgroup1 has a link to subgroup2, which is owned by group2.

The following sections document the various operations and variables of the group types. These are probably of most interest to users who will be manipulating groups in the CSS language, or in programming in PDP++. However, some of the group functions are available in the *Actions* menu in the gui, and their function might not be fully clear from their name and arguments.

7.2.1 Iterating Through Group Elements

The elements in a group with nested sub-structure can be accessed as though this structure were "flattened out". This can be done more efficiently using an iterator scheme as opposed to repeatedly calling the `Leaf()` function. The iterator for going through leaves is an object that remembers information about where it is in the group structure. This iterator is of type **taLeafItr**, and a variable should be declared of this type before beginning a for loop to iterate. The `FirstEl` and `NextEl` functions use this iterator to traverse through the elements of the group.

There is a macro which makes group traversal relatively painless. The following example illustrates its use:

```
Unit* u;                        // a pointer to a unit object
taLeafItr i;                    // the iterator object
FOR_ITR_EL(Unit, u, layer->units., i) {
  u->DoSomething();
}
```

The FOR_ITR_EL macro iterates though the leaves of a group. It accepts four arguments. The first is the type of object expected at the leaf level of the group. The second is a pointer to an object of that type. The third is group followed by its access method. In the example, the group is "layer->units", and the access method is the "." . If the group was instead a pointer to group called "mygroup" then the third argument would be "mygroup->". The fourth argument is the iterator object.

To iterate through all the leaf groups (groups which contain leaf elements) the FOR_ITR_GP macro may be used. Its first argument is the group type (not the leaf type). The second is a pointer to group of that type. The third is pointer group and its access method. And again, the fourth argument is the iterator object.

7.2.2 Group Variables

The Group class has the following variables:

`String name`
: The (optional) name of the group

`int size`
: Number of elements in the group.

`TypeDef* el_typ`
: The "minimum" type of an element in the Group. When objects are created, transferred, or linked in the group, only objects of this type, or objects of a type which inherits from this type are considered for inclusion.

`int el_def`
: The index of the default element of the group. This element is returned when the `Default_El()` function is called on the group.

`int leaves`
: The number of leaves of the Group. This is the total sum of all the elements in this group plus all the elements of its subgroups, their subgroups and so on.

`Group gp`
: This is the group of subgroups.

`Geometry pos`
: The Group's position. In some cases, the group is represented graphically and the position provides a relative offset for the elements of the group

`Geometry geom`
: The Group's geometry. Although the group is accessed linearly, it can be represented graphically in more than one dimension. The geometry specifies the layout of the group's elements.

7.2.3 Group Functions

Groups have many functions for operating on the elements of the group.

These functions return an element of the group. The element is an instance of the group's element type.

`El (int i)`
: Returns the element of the group at index `i`. If the index is out of range, an out of rangeerror is reported, and `NULL` is returned.

`DefaultEl()`
: Returns the element indexed by the `default_el` variable of the group.

`Leaf (int n)`
: Performs a depth first search to find the nth leaf of the group tree.

`FirstEl()`
: Utilizes sets the internal leaf index to zero and return the first leaf.

`NextEl();`
: Increments the internal leaf index and returns the leaf of that index.

`Pop()` Returns the last element of the group, and removes it from the group as well.

`Peek()` This function is similar to the `Pop()` function, but it does not remove the element.

These functions return an integer indicating the index of the element in the group. In all these functions if no match is found, a value of "-1" is returned.

`FindEl (inst el)`
Returns the index of the element of the group which matches `el`.

`FindName (char* name)`
Returns the index of the element which has a name field which matches `name`.

`Find (TypeDef t)`
Returns the index of the first element of type `T`.

`FindLeaf (char* name)`
Returns the leaf index of the leaf element which has a name field which matches `name`.

`FindLeafEl (inst el)`
Returns the leaf index of the leaf element which matches `el`.

These functions are used to add elements to the group.

`AddEl (inst el)`
Adds element `el` to the group

`AddUniqueName (inst el)`
Adds an element `el` to the group and adds an instance index to its name if another element in the group has the same name.

`Push (inst el)`
Adds element `el` to the end of the group.

`PushUnique (inst el)`
Adds element `el` to the end of group only if it is not in the group already.

`PushUniqueName (inst el)`
Adds element `el` to the end of the group only if there are no other elements of the group with the same name.

`Insert (inst el, int i)`
Inserts element `el` at position `i` in the group. If the position `i` is out of range, it is added to the beginning or end of the group accordingly.

`Replace (int i ,inst el)`
Replaces the element at location `i` with the element `el`.

`ReplaceEl (inst rel, inst el)`
Replaces the element `rel` of the group with element `el`. If no match is found, the element `el` is not inserted.

`ReplaceName (char* name, inst el)`
Replaces the element with the name `name` with the element `el`. If no match is found, the element `el` is not inserted.

`Transfer (inst el)`
: Removes `el` from the group or owner it is currently in and adds it to this group.

`Link (inst el)`
: Adds a link to the object `el` to this group. There are also corresponding functions (InsertLink, ReplaceLink, etc) which perform as the do the Add functions except they link instead of add (see Section 7.2 [obj-group], page 97).

`New (int i, typedef t)`
: Creates `i` new objects of type `t` in the group.

These functions are used for removing elements from the group.

`Remove (int i)`
: Removes the element at position i. If the element is owned by the group it deletes the element. If the element is a linked element, it deletes the link.

`Move (int from, int to)`
: Removes an element from position `from` and inserts at position `to`.

`RemoveName (char* name)`
: Removes the element with a name matching `name`.

`RemoveLeafName (char* name)`
: Removes the leaf with a name matching `name`.

`RemoveAll()`
: Removes all the elements of the group.

These functions return a subgroup of the group.

`Gp (int i)`
: Returns the subgroup at index(i)

`LeafGp (int n)`
: Returns the nth subgroup which contains leaves.

`FirstGp()`
: Sets the group index to zero and returns the first subgroup with leaf elements.

`NextGp()` Increments the group index and return the corresponding subgroup with leaf elements.

7.2.4 Group Edit Dialog

The group edit dialog (GED) is in many ways like the the Edit dialog for other objects (see Section 6.5 [gui-edit], page 82). In the GED however, all the elements of the group are editable at one time. The elements are represented in a horizontal scrollbox. If there are more elements than can be represented in the dialog, dragging the scrollbar under the scrollbox will allow access to the other members. The members are ordered sequentially from left to right.

In some cases the elements of the groups may be of different types. When this occurs, all the members of all the element types are listed in the member names section of the dialog.

Elements which do not contain a certain member in the member names section will have a blank field where the edit field of that member would normally appear. The dialog buttons (Ok, Apply, Revert, Cancel) apply to all the elements of the group.

In addition to the standard editing control keys in the Edit Dialog, the following keys are are mapped to special functions.

Meta-f moves the Ibeam forward to the same field in the next group member

Meta-b moves the Ibeam backward to the same field in the previous group member

Many of the functions described in the previous section (see Section 7.2.3 [obj-group-functions], page 99) are available in the menus of the group edit dialog. Note that if you want to operate on the subgroups (e.g., to move them around), you can do EditSubGps which will pull up a dialog of these subgroups where the move and other actions apply to them.

7.3 Arrays

Arrays are the data objects for related sequences of simple data structures like integers, floating point numbers, or strings. Each of these types has its own corresponding array type (ex. floating point numbers (floats) are arranged in a float_Array). All array types however have the the same structure of variables and access functions.

The following sections document the various operations and variables of the array type. These are probably of most interest to users who will be manipulating arrays in the CSS language, or in programming in PDP++. However, some of the array functions are available in the *Actions* menu in the gui, and their function might not be fully clear from their name and arguments.

7.3.1 Array Variables

The array classes have two important variables:

`int size` Indicates the number of elements in the array. Although the array may have allocated additional space, this is the number of elements in use.

`item err` Indicates the value to return when the array is accessed out of range.(ex. If the array had a size of 5 and was asked for element 6 or element -1 then the err value would be returned).

7.3.2 Array Functions

`Alloc (int x)`
Allocates an array with space for x items.

`Reset()` Sets the size of the array to zero, but does not change or free the amount to allocated space.

`Remove(int x)`
Removes the array element at index `x`.

`Permute()`
: Permutes the elements of the array into a random order.

`Sort()`
: Sorts the array in ascending order.

`ShiftLeft (int x)`
: Shifts all the elements in the array `x` positions to the left.

`ShiftLeftPct (float f)`
: Shifts the array to the left by `f` percent.

`El (int x)`
: Returns the element at index `x`, or err if out of range. Indexing starts at zero, therefore an array with five elements would have valid indices for zero to four.

`FastEl (int x)`
: Fast element return. Returns the element at index `x` with no error checking. Caution: PDP++ may behave unexpectedly if this function is called with an index that is out of range.

`Pop()`
: Returns and removes the last element in the array.

`Peek()`
: Returns the last element in the array without removing it.

`Add (item i)`
: Adds `i` to the array.

`Push (item i)`
: Pushes (adds) `i` to the the end of the array

`Insert (item i, int num, int loc)`
: Inserts `num` copies of item `i` at location `loc` in the array.

`Find (item i, int loc)`
: Returns the index of the first element in the array matching item `i` starting at location `loc`.

`RemoveEl (item i)`
: Removes the first element matching item `i`. Returns `TRUE` if a match is found and `FALSE` otherwise.

7.3.3 Array Editing

Arrays are editing using an enhanced version of the standard Edit Dialog See Section 6.5 [The Edit Dialog], page 82. The Array Edit Dialog arranges all of the elements of the array horizontally and allows the user to scroll though the array elements using a horizontal scroll bar at the bottom of the dialog. In addition, number arrays (floats, ints, etc..) can be edited using a Color Array Edit Dialog which is similar in layout to the Array Edit Dialog, but adds a color palette for painting the values See Section 6.7 [gui-colors], page 91.

7.4 Specifications

One of the important design considerations for the PDP++ software was the idea that one should separate state variables from specifications and parameters (see ⟨undefined⟩

[over-spec], page ⟨undefined⟩). The attributes of an object can often be divided into two types–the first type of attributes represent the object's state. These change over time and are usually distinct within each instance of an object. The second group of attributes represent parameters of an object that tend to remain fixed over time, and often have values that are common among instances of the object class. This second group can be thought of as a **specification** for the object class.

For example: The car object class might have two attributes: color, and current-speed. The color attribute would be set when the car was built, and would (hopefully) not be changing very much. It would be classified as a specification parameter. The current-speed attribute is likely to be constantly changing as the car accelerates and decelerates. It is representative of the car's current state and would be classified in the first group, the car's state space. If you took a group of cars, chances are that some of them would share the same color, but they would probably be moving around at different speeds. Rather than have each car carry around an attribute for its color, the specification attributes are split off from the car and put into a special class called a car-specification or carspec. In this way cars with identical colors can share the same specification, while still having their own state attributes like current-speed. By changing the color attribute in the specification, all the cars sharing that specification would have their color changed. This allows easy access to common parameters of an object class in one location. Rather than individually setting the color parameter for each instance of a car, the attribute can be set in just once in the spec.

This idea is instantiated in a particular class of objects known as **Spec**s. Specs also have special "smart pointers" that objects use to refer to them. These spec pointers or **SPtr** objects ensure that a given object always has a corresponding spec, and that this spec is of an appropriate type.

While specs are basically just classes that have parameters and functions that control other object's behavior, there are a couple of special properties that specs have which make them more powerful.

Often when one wants to use two different specs of the same type, it is because one spec has one parameter different than the other. For example, one spec might specify a learning rate of .01, and the other a learning rate of .001. However, these specs might very well share several other parameters.

To simplify the ability of specs to share some parameters and not others, a special system of *spec inheritance* was developed. Basically, each spec has a group on it called `children`, in which "child" specs can be created. These child specs inherit all of their parameters from the parent spec, except those specifically marked as unique to the child. These fields appear with a check in the left-hand check-box when edited in the GUI. Thus, whenever you change a value in the parent, the children automatically get this changed value, except if they have a unique value for this field, in which case they keep their current value. For a tutorial demonstration of how this works, see (see ⟨undefined⟩ [tut-config-running], page ⟨undefined⟩).

There are a couple of things to know about the "smart" spec pointers. These pointers have both a type field and the actual pointer to the spec. When you change only the `type` field, it will automatically find a spec of that type, and set the pointer to that. If one does not yet exist, one will be created automatically and the pointer set to it. If however you

change the pointer directly to point to a different spec, and this spec is of a different type than that shown in the `type` field, then the type will prevail over the pointer you set. Thus you have to change both the `type` and `spec` fields if you change the latter to point to a different type.

The reason for this is that the spec pointer object does not know which field you actually changed, and for the nice automatic properties associated with changing the type field to work, the need to update both the type and the spec pointer is an unfortunate consequence.

7.5 Random Distributions

Random distributions in PDP++ are handled by an instance of the class **Random**. This class has a number of functions which return a random number from the distribution named by the function. Alternately the distribution can be specified in the object itself in which case the `Gen()` function returns a value from the specified distribution. Many of the distributions require parameters which are again either passed to the specific functions or set on the Random object itself.

Random class Variables:

`Type type` Indicates the type of random variable to generate when the `Gen()` function is called. It can have one of the following values:

- `UNIFORM` A uniform distribution with `var` = half-range
- `BINOMIAL` A binomial distribution with `var` = p, `par` = n
- `POISSON` A Poisson distribution with `var` = lambda
- `GAMMA` A gamma distribution with `var` and `par` = stages
- `GAUSSIAN` A normal gaussian distribution with `var`
- `NONE` Returns the `mean` value

`float mean`
The mean of a random distribution

`float var` The "variance" or rough equivalent (half-range)

`float par` An extra parameter used for some distributions

Random Class Functions

`float ZeroOne()`
Returns a uniform random number between zero and one

`float Range(float rng)`
Returns a uniform random number with a given range centered at 0

`float Uniform(float half_rng)`
Returns a uniform random number with given half-range centered at 0

`float Binom(int n, float p)`
Returns a random number from a binomial distribution with `n` trials each of probability `p`

`float Poisson(float l)`
: Returns a random number from a Poisson distribution with parameter `l`

`float Gamma(float var, int j)`
: Returns a random number from a Gamma distribution with variance `var` and `par` number of exponential stages

`float Gauss(float var)`
: Returns a gaussian (normal) random number with a variance `var`

`float Gen()`
: Returns a random number using the distributions type `type` and the `mean`, `var`, and `par` variables on the Random object itself.

Concept Index

R

S

T

V

W

X

Class Type Index

Variable Index

R

S

T

U

V

Function Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

Z

Short Contents

Table of Contents