Classing Engine 5

5.1 Overview

The Classing Engine (CE) is an OpenWindows database used to identify the characteristics, or *attributes*, of files. The CE specifies attributes such as print method, icons, and opening commands for specific *file types*. File type is defined by a file's format, its parent application, or the application executable itself. Examples of file format are:

- ASCII
- PostScript
- Sun raster files

Examples of data files created by a parent application are:

- FrameMaker® and
- Lotus 1-2-3® data files

Examples of application executables are:

- File Manager
- Mail Tool, or
- Wingz® executable files

The CE consists of two parts: a database that stores file type names and attributes, and a collection of routines that query the database. Some of the more common file attributes are:

- A filename pattern or content string to identify the file type
- Directory location of a file type icon

- Foreground and background colors of a file type icon
- Print command of a file type, if applicable
- Load and launch command for the application associated with a data file
- Edit or display command of a file

Other attributes, such as data exchange filters and text compression procedures, can be associated with a file type as well; the CE is completely extensible. In addition, it is possible to add custom databases for other data objects to the CE.

The CE acts as a central repository for all file types and their attributes. The CE also provides applications with a set of routines for determining a file's type and retrieving its attributes.

This chapter describes the CE technology and how one DeskSet program, File Manager, uses it. (File Manager is a graphical file and directory tool shipped with OpenWindows.) The CE can be used in the same manner by any Desktop application.

Note – The CE can be used by the File Manager. This chapter only discusses the CE as used by File Manager.

5.2 *File Type Registration*

For an application to obtain the operating attributes of a file, the file's identifier and attributes must be stored in the CE database. This requires that the file's originators, typically the vendor whose application created the file, incorporate the file's type and its attributes into the CE database. File types can be incorporated into the CE database in the following ways:

1. Software vendors may register file types and their attributes with SunSoft through the Developer Integration Format Registration program (DIFR). The new file types will be incorporated into the CE database and distributed in subsequent CE releases. Refer to Appendix C, "Vendor Data Type Registration" for detailed registration instructions.

2. Software vendors may use the CE utilities in their software installation process to update their user's CE databases with new file type information. Thus, a vendor's application can, as part of the installation process, enter its file types and attributes into the CE database.

3. Users can use CE utilities or Binder, a DeskSet application, to enter new file type information into the CE database.

5.3 Classing Engine Usage

File Manager, displayed in Figure 5-1, provides an example of how the CE can be used. File Manager is a DeskSet application that graphically displays a UNIX file system. Users may move, copy and delete files by dragging and dropping file icons onto directory icons, or onto a wastebasket icon. In addition, File Manager allows users to double-click on a data file icon to open the file in its parent application (file opening commands are stored in the CE database). For example, double-clicking on a spreadsheet data file could start the spreadsheet application program and open the data file. Double-clicking on an ASCII file will open the file with the Text Editor. File Manager also lets users print a data file by simply dropping the file's icon on the Print Tool.

Another feature of the File Manager is that different file types are represented by different icons. Thus, one application's files will have one icon, and the files of another application will have a different icon. Unique icons allow users to identify a file without opening it. File Manager retrieves the icon location from the CE. Refer to the *Solaris User's Guide* for details on how to use File Manager.

5

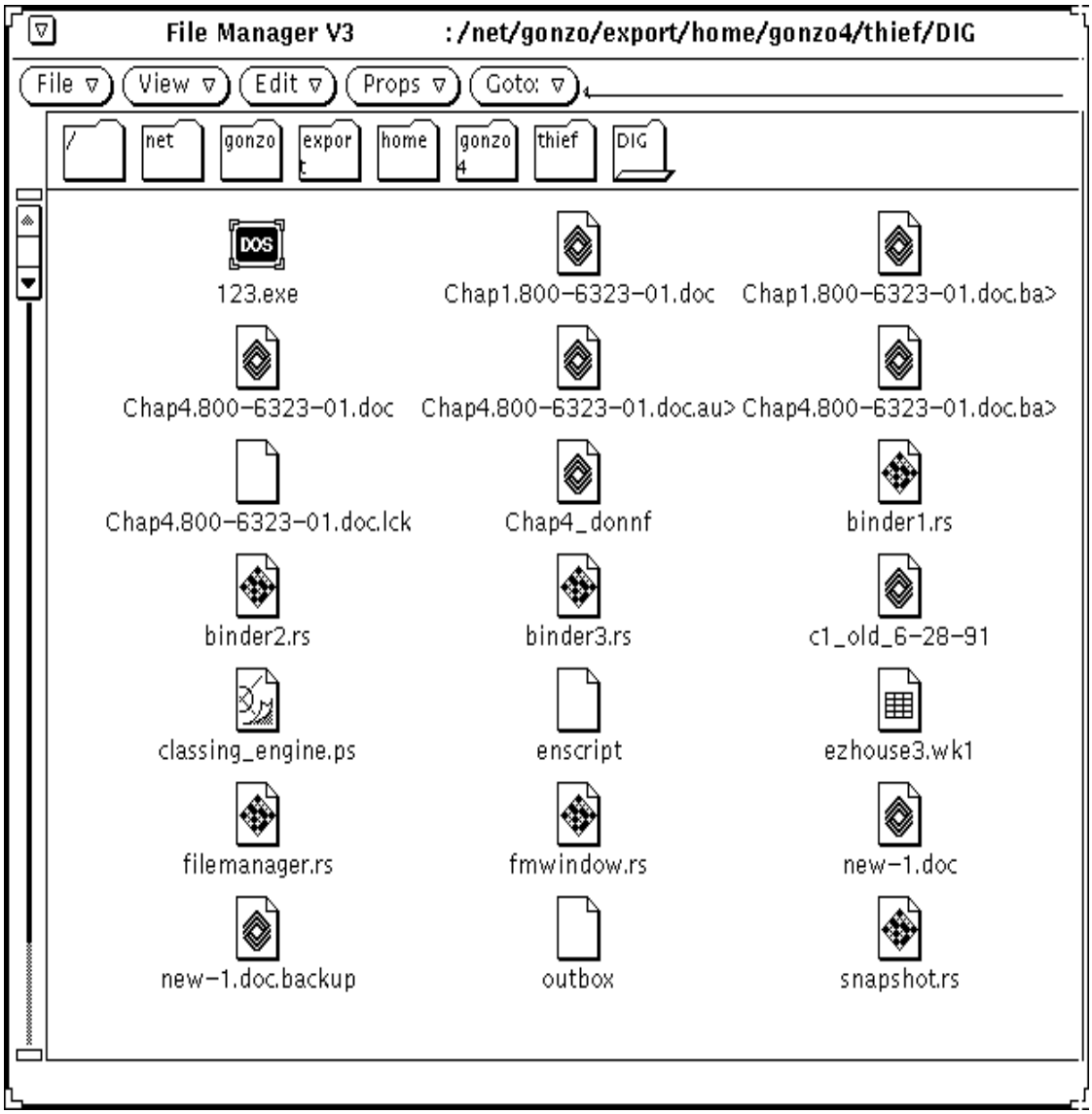

Figure 5-1 File Manager

5.4 Adding and Changing Classing Engine File Types and Attributes

Adding or changing file types and attributes in the CE consists of changing the CE database to reflect these new filetypes and attributes. Before discussing how to do this, it is necessary to discuss the structure of the CE database.

5.4.1 Classing Engine Database

The CE database contains file type names, identification patterns, and attributes. The CE database is one logical database that is the composite of three physical databases called the *user*, *system*, *a*nd *network* databases. Multiple databases allow users to personalize their environment while still having access to global data.

The user database is unique to each user and resides in the user's directory structure, the system portion is common to all users on that specific machine, and the network portion is available to everyone on the network. The CE treats these three portions as overlays. When an application queries the CE database for information, the CE will first read the entry in the user database. If an entry is not found in the user database, the CE tries the system database, and finally the network database. This assures that any CE database information customized by the user (or the system) will be used if it exists; otherwise, network information is used. The following discussion treats the three databases as a single aggregate database.

Default Location of Classing Engine Databases

Each of the three Classing Engine databases has a default location, as shown in the table below. These files are in a non-readable format. To convert these files into an ASCII-readable format, use the `ce_db_build` utility as follows:

```
ce_db_build <user | system | network> -to_ascii <file name>
```

Table 5-1 Default Classing Engine Database Locations

database	default location
user	`~/.cetables/cetables`
system	`/etc/cetables/cetables`
network	`$OPENWINHOME/lib/cetables/cetables`

5.4.2 Namespace Tables

Each CE database file consists of two *namespace tables*, which are data bases of file entries:

- A *files namespace table*, contains file type names and identifiers
- A *types namespace table*, which stores file type attributes

Both of these namespace tables are resident in the same file. To view the namespace tables use the `ce_db_build` command described in the previous section.

Each namespace table has an accompanying *namespace manager*, a collection of routines used to query that namespace table.

5.4.3 File Type Identification

Before an application can use a file's attributes, the application must identify, or *derive*, the file type. In other words, it must determine whether a file is an ASCII file, Mail Tool executable file, PostScript file, an so forth. Two methods are used to determine file types: type-by-*pattern* or type-by-*content*.

Typing by pattern involves matching the filename with a filename pattern. For example, all files whose names end in `.c` are C source files, all files that end in `.exe` are DOS executable files, and all files that end in `.ps` are PostScript files.

Typing by *content* involves matching the contents of a file to a pre-defined string or number. For example, files that have the string `WNGZWZSS` as their first characters are Wingz worksheet files. Files that contain `<Framemaker` as its first characters are FrameMaker files. This is similar to the procedure that is used by the standard UNIX `file` command that uses the `/etc/magic` file.

5.4.3.1 Files Namespace Table

The files namespace table contains entries that are used to derive file types. An excerpt of a files namespace table is shown below.

```
NS_NAME=Files      # Beginning of Files namespace table
NS_ATTR=((NS_MANAGER,junk,<$CEPATH/fns_mgr.so>))# The Files namespace manager
NS_ENTRIES=(
  ( . . .
    (FNS_TYPE,refto-Types,<filemgr-prog>)# File type = File Manager
    (FNS_FILENAME,str,<filemgr>)# File pattern = filemgr
  )( . . .
    (FNS_TYPE,refto-Types,<mailtool-prog>)# File type = Mailtool program
    (FNS_FILENAME,str,<mailtool>)# File pattern = mailtool
  )( . . .
    (FNS_TYPE,refto-Types,<lotus-spreadsheet>) # File type = lotus spreadsheet
    (FNS_FILENAME,str,<*.wk?>)# File pattern = *.wk?
  )( . . .
    (FNS_TYPE,refto-Types,<msdos-executable>)# File type = MS DOS Application
    (FNS_FILENAME,str,<*.exe>)# File pattern = *.exe
   )( . . .
    (FNS_TYPE,refto-Types,<c-file>)# File type = C source file
    (FNS_FILENAME,str,<*.c>)# File pattern = *.c
   )( . . .
    (FNS_TYPE,refto-Types,<sun-raster)# File type = Sun Raster
    (FNS_MAGIC_OFFSET,str,<0>)# Offset = 0 bytes
    (FNS_MAGIC_MATCH,str,<0x4d4d002a>)# Content Pattern = 0x4d4d002a
    (FNS_MAGIC_TYPE,str,<long>)# Content Type = long int
  )( . . .
    (FNS_TYPE,refto-Types,<framemaker-document>)#File type = Framemaker Document
    (FNS_MAGIC_OFFSET,str,<0>)# Offset = 0 bytes
    (FNS_MAGIC_MATCH,str,<<MakerFile>)# Content Pattern = <Makefile
    (FNS_MAGIC_TYPE,str,<string>)# Content Type = string
  )( . . .
    (FNS_TYPE,refto-Types,<sunwrite-document>)# File Type = SunWrite Document
    (FNS_MAGIC_OFFSET,str,<3>)# Offset = 3 bytes
    (FNS_MAGIC_MATCH,str,<pgscriptver>)# Content pattern = pgscriptver
    (FNS_MAGIC_TYPE,str,<string>)# Content Type = String
  )( . . .
    (FNS_TYPE,refto-Types,<postscript-file>)# File type = Postscript file
    (FNS_FILENAME,str,<*.ps>)# File pattern = *.ps
  )(
    (FNS_TYPE,refto-Types,<postscript-file>)# File type = Postscript file
    (FNS_MAGIC_OFFSET,str,<0>)# Offset = 0 bytes
    (FNS_MAGIC_MATCH,str,<%!>)# Content Pattern = %!
    (FNS_MAGIC_TYPE,str,<string>)# Content Type = String
  )( . . .
```

Entries in the files namespace table consist of the following arguments:

`FNS_TYPE`, or *file type name*, is the name (identifier) assigned to a file type. In the following example, the file type name for the File Manager program is `filemgr-prog`. The file type name for Lotus 1-2-3 spreadsheet files is `lotus-spreadsheet`.

`FNS_FILENAME` is the file name pattern that identifies a file's type. The file name pattern is used to match a file name to its type. For example, a file ending with `.c` is a C source file. A file ending with `.exe` is a DOS executable file.

If a file type is derived with the type-by-content method, the file type entry requires these arguments:

`FNS_MAGIC_MATCH` or *magic match*, is a string contained in all files of the type specified by `FNS_TYPE`. Thus, all FrameMaker document files contain the string `<MakerFile`. All PostScript files contain the string `%!`.

`FNS_MAGIC_TYPE` specifies the data type of the magic march. In the example, all type-by-content entries match with strings, except for sun-raster files, which use a long integer.

`FNS_MAGIC_OFFSET` specifies the number of bytes preceding the magic match. As shown in the following example, `<MakerFile` starts at the first byte in a FrameMaker document file. pgscriptver starts after the third byte in a SunWrite document file.

If both a file name pattern and a magic match are are defined like as shown in the PostScript example, a file must pass both tests before it is typed.

5.4.4 Types Namespace Table

The types namespace table contains the attribute values of the file types. Once a file type is derived, the CE can retrieve the files attributes from the types namespace table. An excerpt of a types namespace table is shown below.

```
NS_NAME=Types                                # The namespace named "Types"
NS_ATTR= ((NS_MANAGER,string, <$CEPATH/tns_mgr.so>))# The Types namespace
manager
NS_ENTRIES= (...
    (TYPE_NAME,type-id,<filemgr-prog>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/filemgr.icon>)
    (TYPE_BGCOLOR,color,<79 241 255>)
    (TYPE_PRINT,string,<lpr -Plp>)
   )( . . .
    (TYPE_NAME,type-id,<lotus-spreadsheet>)
    (TYPE_OPEN,call,<dos -c 123>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/spreadsheet.icon>)
    (TYPE_ICON_MASK,icon-file,<$OPENWINHOME/include/images/doc.mask.icon>)
    (TYPE_BGCOLOR,color,<255 225 255>)
    (TYPE_FILE_TEMPLATE,string,<lotus%t.wks>)
   )( . . .
    (TYPE_NAME,type-id,<compress>)
    (TYPE_OPEN,call,<uncompress>)
    (TYPE_ENCODE_PROG,call,<compress>)
    (TYPE_ENCODE_ARGS,string,<-c>)
    (TYPE_DECODE_PROG,call,<uncompress>)
    (TYPE_DECODE_ARGS,string,<-c>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/compress.icon>)
    (TYPE_ICON_MASK,icon-file,<$OPENWINHOME/include/images/doc.mask.icon>)
    (TYPE_BGCOLOR,color,<255 0 0>)
    (TYPE_FILE_TEMPLATE,string,<data%t.Z>)
   )( . . .
    (TYPE_NAME,type-id,<default-app>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/application.icon>)
    (TYPE_FGCOLOR,color,<0 0 0>)
    (TYPE_BGCOLOR,color,<183 229 193>)
   )(
    (TYPE_NAME,type-id,<default-doc>)
    (TYPE_OPEN,call,<textedit>)
    (TYPE_OPEN_TT,tt,<textedit>)
    (TYPE_PRINT,string,<cat $FILE | mp -lo | lpr -h>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/document.icon>)
    (TYPE_ICON_MASK,icon-file,<$OPENWINHOME/include/images/doc.mask.icon>)
    (TYPE_FGCOLOR,color,<0 0 0>)
    (TYPE_BGCOLOR,color,<183 193 229>)
   )( . . .
```

Entries in the type namespace table consist of the following arguments:

TYPE_NAME is the name of the file type. TYPE_NAME matches FNS_TYPE in the files namespace table.

TYPE_ICON is the file containing the icon representation of the file type.

TYPE_ICON_MASK is the file containing the icon representation of file when it is selected.

TYPE_BGCOLOR specifies the background color of the file icon. Values are in red-green-blue (RGB) values ranging from 0 (lighter) to 255 (darker).

TYPE_FGCOLOR specifies the foreground color of the file icon in RGB values.

TYPE_OPEN specifies the command to open the file. (For File Manager this is triggered by a double mouse-click.)

TYPE_PRINT gives the print command for the file.

TYPE_FILE_TEMPLATE specifies a unique filename generated and used by the application as a filename identifier.

TYPE_OPEN_TT is the ToolTalk identifier used when starting applications requiring the obsolete Tooltalk protocol (as used in releases prior to Solaris 2.2).

TYPE_MEDIA is the ToolTalk identifier used when starting applications requiring the new Message Alliance/Media Exchange protocol (as used in releases starting with Solaris 2.2).

The attribute entry for compress demonstrates CE extensibility. In addition to the standard attributes, compress file types have four additional attributes: TYPE_ENCODE_PROG, TYPE_ENCODE_ARGS, TYPE_DECODE_PROG, and TYPE_DECODE_ARGS. A program designer has added these attributes to compress file types in order to provide automatic file compression/decompression. For example, these attributes can be used to link large files to a mail message. Instead of pasting the file into the message, the file could be automatically compressed (using the UNIX compress command) when the file glyph is selected and dropped on the mail tool. The compressed file appears as an file glyph. After the message is sent, the file is automatically decompressed when the file glyph is selected.

The last two entries of the code segment, `default-app` and `default-doc`, demonstrate two other interesting features. Each represents the attributes of undefined data and application files. If a file does not have a definition in the files namespace, it is given a set of generic attributes depending on whether it is an application or document.

5.4.5 *Adding a New File Type*

The basic steps for adding a new file type to the CE database are as follows:

1. Create an ASCII description file for the new file entry. Either extract the ASCII description file for the entire CE database using `ce_db_build` (the `man` page is in the back of this chapter), or create a new ASCII description file for the new file type entry. The process for creating a single entry ASCII description file is described in the section that follows.

2. Add a file type name and file type pattern to the files namespace table in the ASCII description file. You only need to add the file type pattern if the file type is derived using the type-by-pattern method. If the file is derived using the type-by-content method, add a magic match, magic match data type, and an offset.

3. After a new file type has been added to the files namespace table, add its attributes to the types namespace table.

4. Once the attributes are added, you can overwrite the old CE database file with the one you just created using the `ce_db_build` command. Use this command only if you are replacing the entire CE database file. If you created an ASCII description file for a subset of the entire CE database (this procedure is described in the next section), merge the file into the current CE database with the `ce_db_merge` command (the `man` page for this command is in the back of this chapter).

5.4.5.1 *Adding a New File Type to the Classing Engine—Example*

This section shows a step-by-step example of adding a new file type to the CE.

1. Define the file type name, its unique file name pattern or content string, and its attributes. For this example we'll use a hypothetical program called `Peakstool` that works on files of a type called `twin-peaks-type`:

```
Object Name = twin-peaks-type
```

```
    Content Pattern = Good Coffee!
    Offset = 0
    Content Type = string
    Open Command (program name) = peakstool
    Icon Location = $OPENWINHOME/include/images/laura.icon
    Icon Mask Location =
$OPENWINHOME/include/images/laura.mask.icon
    Foreground Color = r=91, g= 229, b= 229
    File Pattern = *.pks
```

Only file type name and either a file content pattern (with offset and type) or file pattern are necessary to add a valid entry in the CE database. All other parameters are optional.

2. Create a CE database definition file in ASCII and give it a name. The file illustrated below, `newtype.ascii`, corresponds to our twin-peaks file. Note that the attributes go in the types namespace table, and the file/content patterns go in the files namespace table.

```
# newtype.ascii: A sample ASCII CE database description file
{
NS_NAME=Types
NS_ATTR= ((NS_MANAGER,string, <$CEPATH/tns_mgr.so>))
NS_ENTRIES= (
     (
    (TYPE_NAME,type-id,<twin-peaks-type>)
    (TYPE_OPEN,call,<peakstool>)
    (TYPE_ICON,icon-file,<$OPENWINHOME/include/images/laura.icon>)
    (TYPE_ICON_MASK,icon-file,<$OPENWINHOME/include/images/laura.mask.icon>)
    (TYPE_FGCOLOR,color,<91 229 229>)
    (TYPE_BGCOLOR,color,<91 126 229>)
    (TYPE_FILE_TEMPLATE,string,<peaks.%t>)
     )
    )
}
# Tell CE how to match files of your type. If the file begins with Good_coffee!,
# it's of type twin-peaks-type. The string begins at offset 0 in the file.
{
NS_NAME=Files
NS_ATTR=((NS_MANAGER,junk,<$CEPATH/fns_mgr.so>))
NS_ENTRIES=(
     (
     (FNS_TYPE,refto-Types,<twin-peaks-type>)
     (FNS_MAGIC_OFFSET,str,<0>)
     (FNS_MAGIC_MATCH,str,<Good_coffee!>)
     (FNS_MAGIC_TYPE,str,<string>)
     )
    )
}
```

3. After creating the ASCII description file, execute the `ce_db_merge` command to add the new file type to one of the three CE databases. The *network* database is used in this example, with an ASCII description file called `newtype.ascii`.

```
% ce_db_merge network -from_ascii newtype.ascii
```

4. Note that you can also use `cd_db_build` to add a new file type. Refer to the `man` page for details.

5.4.6 *Syntax of ASCII Database Description File*

The grammar that describes the Database Description File is given here in Backus-Naur Form (BNF):

```
database::=name_space
      | database name_space
name_space::={ name ns_attrs entries }
name::= NS_NAME = variable
ns_attrs::=NS_ATTR = (av_list)
av_list::=av
        |av_list av
av  ::= (av_name, av_type, av_val)
entries::=NS_ENTRIES = (entry_info_list)
entry_info_list::=entry_ent
      | entry_info_list entry_ent
entry_ent::=( av_list )
av_name::=variable
av_type::=variable
variable::=Id
av_val::=av_token
```

The terminals are:

```
Id = a-z, A-Z, 0-9, _, -.
NS_NAME, NS_ATTR, NS_ENTRIES,
"{", "}", "(", ")", "," "=", Id, and av_token.
```

`av_token` can come in two forms:

- It can begin with a "<" and end with a ">" and can have any ASCII character (except a ">") within it.
- It can begin with one or more digits (which represent a number *n*) followed by zero or more spaces followed by a "<" followed by any *n* characters closed off by a ">." This is the escape mechanism to allow for arbitrary byte string attributes that could have ">" characters within them.

5.4.7 Binder

Attributes can be added or changed by editing the types namespace file, or by using Binder shown in Figure 5-2. Binder is a DeskSet tool that provides an interactive display of the CE database (refer to the *"OpenWindows Reference Manual"* for operating instructions). With Binder, an advanced user can bind together a file type, its application, a print method, and an icon by setting the desired attributes.

The Binder is also helpful in understanding the Classing Engine, since it interacts directly with the CE database. When you open the Binder, you are given a selection of file types shown in icon form. These correspond to the file types contained in the files namespace table. Once you select a file type, you can view and the attributes in either the *icon properties sheet* or *files property sheet*. Binder allows you to change attributes or create new file types interactively.

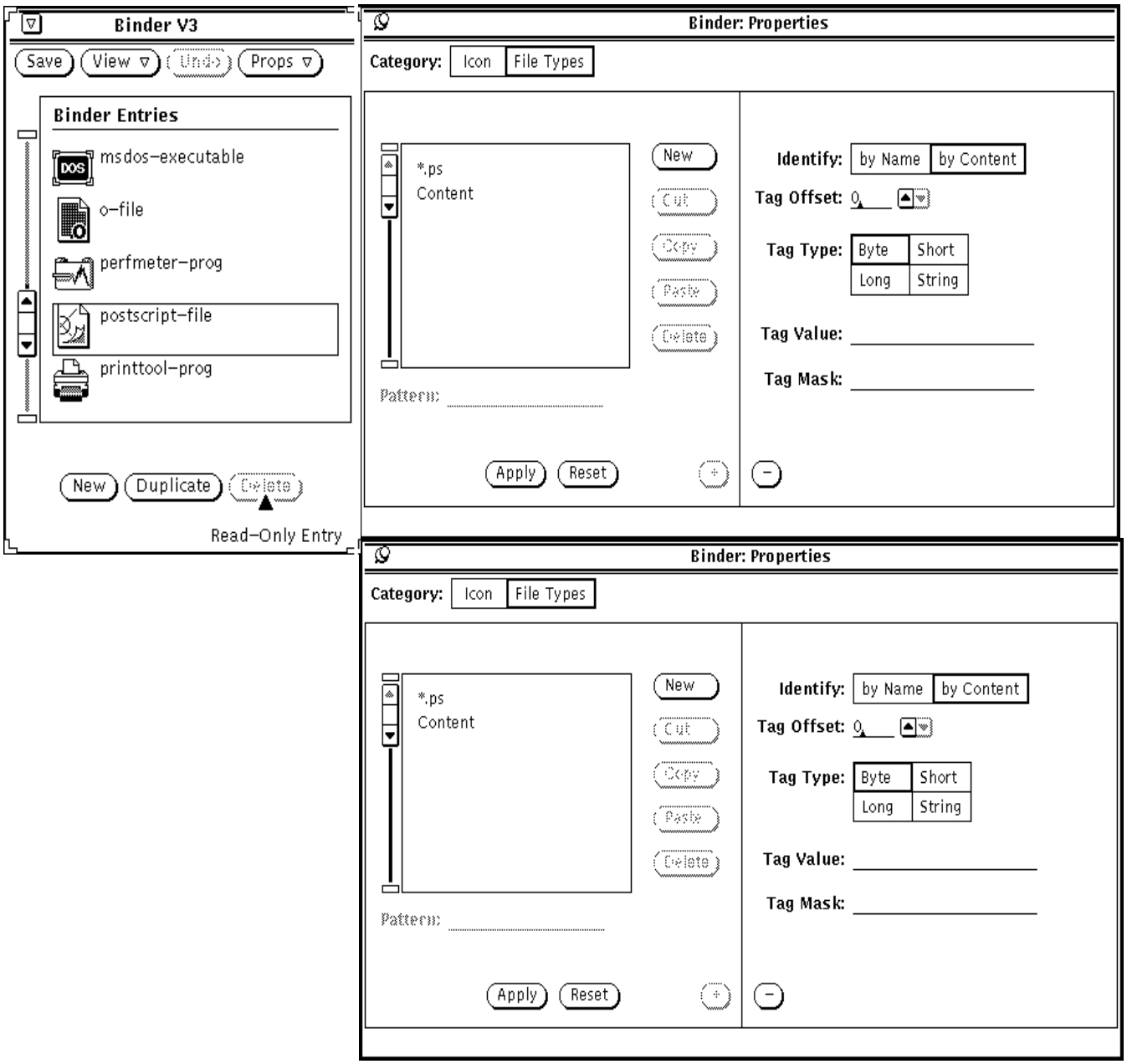

Figure 5-2 Binder—Icon and File Types Property Sheet

5.5 Accessing the Classing Engine Database

The namespace manager, a collection of routines used to query the CE database and perform other database chores provides access to the CE database. These routines are described in Section 5.6, "The Classing Engine API." It may, however, be helpful to first see two simple programs that use the CE. In addition, there are two Classing Engine examples in `$OPENWINHOME/share/src/dig_samples/ce1` and `ce2`.

5.5.1 Example Program—Querying the Classing Engine Database

The program illustrated below shows how the CE database is queried (`$OPENWINHOME/share/src/dig_samples/ce2/ce_simple.c`). When compiled and executed, the program prompts the user to enter the name of an object (a filename). The program accesses the CE database then displays the file type and the location of its icon file on the screen. The user types "quit" to exit the program. This program must be executed on OpenWindows Version 3.1 or later.

The program is divided into code segments with a detailed explanation of how each code segment works. Table 5-2 shows the variable definitions for the sample program.

Table 5-2 Variable Definitions for ce_simple.c

Type	Variable	Comment
`CE_NAMESPAC`	`f_name_space` `t_name_space`	file namespace table handle types namespace table handle
`CE_ENTRY`	`ftype_ent` `ttype_ent`	file namespace table entry handle types namespace table entry handle

`CE_ATTRIBUTE`	`fns_type`
	`tns_icon`
	`fns_attr`
	`tns_attr`
		file type
		icon filename
		file namespace tbl: file type attr. handle
		types namespace tbl: icon attr. handle

`int`	`argcount` `fd;`	`ce_get_entry` arg counter file descriptor for file to be typed

Table 5-2 Variable Definitions for ce_simple.c

Type	Variable	Comment
`char`	`filename`[81] `buf[256]`	buffer for file name buffer for contents of file
`int`	`bufsize` `status`	return values

5.5.1.1 *Preliminary Setup*

This first segment includes a short program description and the compile statement. Loading the program requires the Classing Engine and dynamic linking libraries. The include files and variable definitions are:

`<stdio.h>`for standard io to get input and output
`<desktop/ce.h>`Needed for the Classing Engine variables
`<desktop/ce_err.h>`Error return codes from Classing Engine (not used in this program, except for `ce_begin`)

```
/* ce_simple.c - Simple Classing Engine Example that types a
 * file and determines its icon.
 *
 * cc -g -o ce_sample -I$OPENWINHOME/include -L$OPENWINHOME/lib
 *  ce_simple.c -lce -ldl */

#include <stdio.h>
#include <desktop/ce.h>
#include <desktop/ce_err.h>

/* variable definitions */
CE_NAMESPACEf_name_space, t_name_space;
CE_ENTRY ftype_ent, ttype_ent;
CE_ATTRIBUTEfns_type, tns_icon, fns_attr, tns_attr;
int     argcount;
```

5.5.1.2 *Open the CE Database*

After declaring the global variables, the program declares the variable definitions for the file to be typed: the file descriptor (fd), the file name (sufficiently long to include any likely path name), and a 256-byte buffer to hold the first characters of the file.

The CE is initialized by `ce_begin()` using the mandatory NULL argument (see the API section for details). The call returns zero if successful; otherwise, it returns a positive integer representing an error code, which is printed to standard error, after which the program exits.

```
main(argc, argv)
int argc;
char *argv[];
{
    int  fd;
    char filename[81];
    char buf[256];
    int  bufsize, status;

    /*  Initialize the Classing Engine.  */

    status = ce_begin( NULL );
    if ( status ) {
            fprintf(stderr, "Error Initializing Classing Engine
                  Database - Error no: %d.\n", status );
            exit( 0 );
    }
```

5.5.1.3 *Setting the Namespace Pointers*

The code segment below sets up the pointers in anticipation of reading the namespace entries for both the files and the types namespaces.

`ce_get_namespace_id("Files")` returns a handle to the files namespace table in `f_name_space`. If the either the file namespace table is not found, or the file namespace manager is not found, the call returns `NULL` and the program exits.

A similar `ce_get_namespace_id()` call and error routine is used for the types namespace table. These calls only need to be done once.

```
/* Read in Namespace Entries. */
f_name_space = ce_get_namespace_id( "Files" );
if ( !f_name_space ) {
        fprintf( stderr, "Cannot find File Namespace\n" );
        ce_end();
        exit(0);
}

t_name_space = ce_get_namespace_id( "Types" );
if ( !t_name_space ) {
        fprintf( stderr, "Cannot find Types namespace\n" );
        ce_end();
        exit(0);
}
```

5.5.1.4 *Retrieve Desired Attribute IDs*

`ce_get_attribute_id(f_name_space, "FNS_TYPE")` returns a handle to the file type attribute in the files namespace table and assigns it to the object ID `fns_attr`. Similarly, the second `ce_get_attribute_id()` returns a handle to the icon filename attribute in the types namespace table and assigns it to the object ID `tns_attr`.

```
/*  Get the attribute ID's that we're interested in  */

    fns_attr = ce_get_attribute_id ( f_name_space, "FNS_TYPE" );

    if (!fns_attr){
            fprintf (stderr, "Cannot find FNS_ATTR in Files\n");
            ce_end();
            exit(0);
    }

    tns_attr = ce_get_attribute_id (t_name_space, "TYPE_ICON");

    if (!tns_attr){
            fprintf (stderr, "Cannot find TYPE_ICON in Types\n");
            ce_end();
            exit(0);
    }
```

5.5.1.5 *Loop to Read File Names*

The next segment starts the loop to read in file names and derive their types. A `while` loop prompts the user for the name of the file that will be tested in the CE. If the user types "quit" the loop is exited (break) and CE database is closed (shown in next segment).

The second `if` statement attempts to open the file. If the file is found, but cannot be opened, an error message is printed and the loop starts again.

If the open is successful, an attempt is made to read the beginning of the file into the 256-byte buffer (to be used later by the CE). If the file is empty or a directory, an error message is printed and the loop starts again.

```
/*  Start loop to read in filenames  */

    while(1) {
        fprintf(stdout, "Filename: ");
        gets(filename);
        if ((strcmp(filename, "quit")) == 0)
            break;

        if ((fd = open (filename, 0)) == -1) {
            fprintf(stderr, "Cannot open: %s\n", filename);
            continue;
        }

        bufsize = read (fd, buf, sizeof (buf));
        if (bufsize <= 0) {
            fprintf(stderr, "Empty file or Directory: %s\n",
                    filename);
            close (fd);
            continue;
        }
```

5.5.1.6 *Get Entry in the Files Namespace*

This next code segment searches through the files namespace table for the file name and/or file content obtained in the previous segment. If a match is found, the file type is returned.

The program calls `ce_get_entry()` to search the files namespace table and return the handle for the matching files namespace table entry. `ce_get_entry()` requires the files namespace ID (`f_name_space`), the number of arguments used to match entries in the files namespace table (`3`), and the three arguments themselves (the file name entered by the user, the buffer that contains the contents of the previous read, and the length of the buffer).

`ce_get_entry()` returns a handle for the files namespace table entry that matches the filename pattern, contents of the file, or both if both are present. The handle to the entry is assigned to `ftype_ent`. If no entry is found in the files namespace table, a `NULL` is returned, and the `while` loop resumes.

The program then gets the requested attribute value (file type) by calling `ce_get_attribute()`. `ce_get_attribute()` requires the files namespace handle (`f_name_space`), the handle to the entry (`ftype_ent`), and the file type attribute handle (`fns_attr`). After the attribute value is obtained the value is printed.

```
    /*  Get a matching entry in the files namespace    */
        argcount = 3;
        ftype_ent = ce_get_entry (f_name_space, argcount,
                                  filename, buf, bufsize);
        if ( !ftype_ent ) {
            fprintf(stderr, "No match in Files Namespace\n" );
            continue;
        }


fns_type=ce_get_attribute(f_name_space,ftype_ent,fns_attr);
        if (!fns_type)  {
            fprintf(stderr,"No FNS_TYPE for entry in Files
                                  Namespace\n");
          continue;
}
else{o
        fprintf(stdout, "FNS_TYPE = %s\n", fns_type);
```

5.5.1.7 *Get Entry in the Types Namespace*

The final segment of this program retrieves the icon information from the types namespace table. Use `ce_get_entry()` to retrieve a handle for the desired entry. `ce_get_entry()` is passed the types namespace handle (`t_name_space`), the number of arguments used to match entries in the types namespace table (1), and the argument itself (`fns_type`). If a matching entry is not found, an error message is printed and the `while` loop is resumed. If a correct entry is found, the program calls `ce_get_attribute()` with `t_name_space` (types namespace), the handle to the entry (`ttype_ent`), and the icon handle (`tns_attr`) to return the icon filename.

Finally, the icon name (path and name) is printed, the Classing Engine is closed, and the program exits normally.

```
        /* Get a matching entry in the types namespace found from
         * getting type from the files namespace and find icon
         */
        argcount = 1;
        ttype_ent = ce_get_entry ( t_name_space, argcount,
                                fns_type );

        if ( !ttype_ent ) {
            fprintf( stderr, "No match in Types namespace\n" );
            continue;
        }
            tns_icon = ce_get_attribute (t_name_space, ttype_ent,
                                        tns_attr);

        if (!fns_icon)  {
           fprintf(stderr,"No TYPE_ICON in Types Namespace\n");
           continue;
        }
        else
            fprintf(stdout, "TYPE_ICON = %s\n", tns_icon);
    }
}
    cd_end ( );
    exit (0);
}
```

5.5.2 *Example Program—CE Mapping Functions*

This program, `ce_map1.c`, is located online at `$OPENWINHOME/share/src/dig_samples/ce1`. It demonstrates the use of the CE mapping functions. Refer to the API section that follows for further details.

```
/* ce_map1.c - Classing Engine example that print all the types
 * in the Files and Types namespaces.
 *
 * cc -g -o ce_map1 -I$OPENWINHOME/include -L$OPENWINHOME/lib
 * ce_map1.c -lce -ldl  */

#include <stdio.h>
#include <desktop/ce.h>
#include <desktop/ce_err.h>

/* variable definitions */
CE_NAMESPACE f_name_space, t_name_space;
CE_ENTRYttype_ent;
CE_ATTRIBUTE fns_attr, fns_type;

main(argc, argv)
int argc;
char *argv[];
{
    int  status;
    void *map_func(), *type_map_func();

/* Initialize the Classing Engine. */

    status = ce_begin( NULL );
    if ( status ) {
        fprintf( stderr, "Error Initializing Classing Engine
                    Database - Error no: %d.\n", status );
        exit( 0 );
    }

/* Get Files and Types Entries. */

    f_name_space = ce_get_namespace_id( "Files" );
    if ( !f_name_space ) {
        fprintf( stderr, "Cannot find File Namespace\n" );
        exit( 0 );
```

```
    }
    t_name_space = ce_get_namespace_id( "Types" );
    if ( !t_name_space ) {
        fprintf( stderr, "Cannot find Type Namespace\n" );
        exit( 0 );
    }

    /* Get the FNS_TYPE attribute ID */
    fns_attr = ce_get_attribute_id (f_name_space, "FNS_TYPE");

    if (!fns_attr){
        fprintf (stderr, "No FNS_TYPE in Files Namespace\n");
        ce_end();
        exit (0);
    }

    /* ce_map_through_entries() passes each entry handle and
     * namespace handle to the map_func()
     */
    ce_map_through_entries (f_name_space, map_func, NULL);
    ce_end ();
    exit(0);

}

/* Function to handle each entry as it is passed from the mapping
 * function  */
void
*map_func (fns_handle, ent_handle)
CE_NAMESPACE fns_handle;
CE_ENTRY ent_handle;
{
    int argcount = 1;

    /* Get File type value (FNS_TYPE) and print out */
    fns_type = ce_get_attribute (f_name_space, ent_handle,
                                    fns_attr);

    if (!fns_type)
        return (NULL);
    else
        fprintf (stdout, "FNS_TYPE = %s\n", fns_type);

    /* Get matching entry in the Type namespace */
```

```
    ttype_ent = ce_get_entry (t_name_space, argcount, fns_type);
    if (!ttype_ent){
        fprintf (stderr, "No match in Type namespace\n");
        return (NULL);
    }

    /* Map through all the attributes of the entry and send to
     * type_map_func()
     */
    ce_map_through_attrs (t_name_space, ttype_ent, type_map_func,
                NULL);
    fprintf (stdout, "\n");
    return (NULL);
}
/*
 * Function to print all the Type attributes associated with the File
 * type
 */
void
*type_map_func (tattr_handle, tattr_value, args)
CE_ATTRIBUTE tattr_handle;
char    *tattr_value;
void    *args;
{
    char    *attr_name;

    attr_name = ce_get_attribute_name (tattr_handle);

    if (attr_name)
        fprintf (stdout, "%s = %s\n", attr_name, tattr_value);

    return (NULL);
}
```

5.6 The Classing Engine API

The CE API can be called from C, C++, or ANSI C programs. All CE calls have names that begin with `ce_`, with each session begun with a `ce_begin()` and ending with `ce_end()`.

The arguments manipulated by the API are either Classing Engine object handles or client-decipherable argument values and return values. Classing Engine object handles are of type `CE_NAMESPACE`, `CE_ENTRY` and `CE_ATTRIBUTE` and are returned when a client successfully accesses a namespace, an entry, or an attribute. Client-decipherable argument values and return values are expected to be of type void *, if they are pointers, or of type int.

5.6.1 Mapping Functions

The `ce_map_through_*` functions loop through namespace, entry, and attribute lists, applying a client-supplied function to each member of a list. The previous example shows how the mapping functions work in detail.

5.6.2 Error Reporting

`ce_begin` returns 0 if it succeeds, otherwise it returns an error number. All Classing Engine `ce_get_*` calls return `NULL` if they fail, otherwise they return a valid handle or return value.

The `ce_map_through_*` calls map through namespaces, entries, or attributes and terminate if they encounter a non-null return value from the map function, and return the non-null value. If the map function returns `NULL` in every instance, the `ce_map_through_*` function returns `NULL`.

5.6.3 Location of Namespace Managers

Every namespace manager library file should be named as the `NS_MANAGER` namespace attribute. This should be a full pathname with both environment variables and the 'arch' command allowed.

If a namespace manager library name is preceded by a `$CEPATH`, the search rules implied by `$CEPATH` will be used to search for the namespace manager library.

5.7 Reading from the Classing Engine Database

5.7.1 Initializing the Classing Engine

```
int
ce_begin(void * args);
```

Reads in the CE database and makes CE internal structures suitable for subsequent CE API calls (except for another `ce_begin()`). Subsequent calls to `ce_begin()` will re-read the CE databases. `args`, which is reserved for future use, must be `NULL`.

This call returns 0 if successful. Otherwise, the return codes from this call have the following meanings:

CE_ERROR_READING_DB

This message indicates that an unrecoverable error occurred while reading a CE database. Note that the non-existence of a particular CE database file is not considered an error.

5.7.2 Determining if the Classing Engine Databases Changed

```
int
ce_db_changed();
```

Returns 0 if CE databases have not been changed since the last call to `ce_begin()`. It will return 1 if the databases have been changed.

5.7.3 Closing the Classing Engine

```
int
ce_end();
```

Frees all resources being used by the CE. All CE returned handles and values are invalid after this call. `ce_end()` returns 0 in all cases.

5.7.4 Determining Which Databases are Available

```
int
ce_get_dbs(
    int *num_db,
    char ***db_names
    char ***db_pathnames);
```

Returns a count of the databases in `*num_db`. The names of the databases read in is returned in `db_names`. The pathnames of the databases is returned in `db_pathnames`. There are three possible database names:

```
user the user-level database
systemthe system-level database
networkthe network level database
```

Returns database names and pathnames even if there was no database at a particular pathname. That is, it provides the caller information about where the CE databases would be even if one or more CE databases do not exist.

5.7.5 Accessing a Namespace

```
CE_NAMESPACE
    ce_get_namespace_id(
    char *namespace_name);
```

Returns a handle to a namespace. The namespace handle can be used in all subsequent calls to the CE in this process. This call returns `NULL` if the namespace was not found. This call also returns `NULL` if the namespace manager for the given namespace was not found.

5.7.6 Accessing an Entry in a Namespace Table

```
CE_ENTRY
    ce_get_entry(
    CE_NAMESPACE namespace,
    int argcount,
    void *arg1,
    void *arg2,...,
    void *argN);
```

Searches through a specified namespace table and returns an entry that contains a matching argument. This call requires a handle to a namespace, the number of arguments used to match entries, and the arguments themselves.

5.7.7 Getting an Attribute Handle

```
CE_ATTRIBUTE
    ce_get_attribute_id(
    CE_NAMESPACE namespace,
    char *attr_name);
```

Retrieves a handle to an attribute type within a namespace table. All attributes with the same name, within a namespace, can be retrieved using the same attribute handle. This handle is retrieved with this call.

For example, all attributes named ICON will have the same attribute handle within a single namespace. This call returns `NULL` if the named attribute was not found in this namespace.

5.7.8 Getting an Attribute

```
char
*ce_get_attribute(
    CE_NAMESPACE namespace,
    CE_ENTRY entry,
    CE_ATTRIBUTE attribute);
```

Retrieves the value of an individual attribute. This call returns `NULL` if the attribute could not be found in this entry. It requires a handle for the namespace table (`ce_get_namespace_id()`), entry (`ce_get_entry()`), and attribute (`ce_get_attribute_id()`).

5.7.9 Getting the Size of an Attribute

```
int
ce_get_attribute_size(
    CE_NAMESPACE namespace,
    CE_ENTRY entry,
    CE_ATTRIBUTE attribute);
```

Returns the size (in bytes) of an attribute value. Returns 0 if the attribute was not found in this entry.

5.7.10 Getting an Attribute's Type String

```
char
*ce_get_attribute_type(
    CE_NAMESPACE namespace,
    CE_ENTRY, entry
    CE_ATTRIBUTE attribute);
```

Returns the character string denoting the type of an attribute. Attribute types are not enforced nor understood by the CE. Returns `NULL` if the attribute was not found in this entry.

5.7.11 Getting a Namespace Entry

```
CE_ENTRY
    ce_get_ns_entry(
    CE_NAMESPACE namespace);
```

Returns the namespace entry handle for the specified namespace. Namespaces can have attributes of their own; for example, a range of bytes to read for magic number information in the case of files. Namespace attributes are stored in a namespace entry. This call returns a handle to a namespace's entry. All calls that apply to entries can be made using the returned entry handle. Returns `NULL` if the namespace entry was not found.

5.7.12 Mapping Through Namespaces

```
void
*ce_map_through_namespaces(
    void *(*map_func)(),
    void *args);
```

Maps through all installed namespaces, calls `map_func()` for each namespace, and passes each namespace handle as the first argument to `map_func()` and any other `args` as subsequent arguments. `map_func()` is a user defined function. `args` are optional additional arguments for `map_func()`. If no arguments are to be passed, use `NULL`.

The map will be stopped either when there are no more namespaces or when `map_func` returns a non-`null` value, which will be returned to the caller.

5.7.13 Mapping Through Entries

```
void
*ce_map_through_entries(
    CE_NAMESPACE namespace,
    void *(*map_func)(),
    void *args);
```

Maps through all the entries in a namespace, calls `map_func()` for each entry, and passes the namespace handle as the first argument to `map_func()`, entry handle as the second argument, and any other `map_func()` `args` as subsequent arguments. `map_func()` is a user defined function. `args` are optional additional arguments for `map_func`. If no arguments are to be passed, use `NULL`.

The map will be stopped either when there are no more entries or when `map_func` returns a non-`NULL` value, which will be returned to the caller.

5.7.14 Mapping Through Attributes

```
void
*ce_map_through_attrs(
    CE_NAMESPACE namespace,
    CE_ENTRY entry,
    void *(*map_func)(),
    void *args);
```

Maps through all the attributes in an entry, calls `map_func()` for each attribute, and passes the attribute handle as the first argument to `map_func()`, each attribute value as the second argument, and `args` as the subsequent arguments to `map_func`. `map_func()` is a user defined function. `args` are optional additional arguments for `map_func`. If no arguments are to be passed, use `NULL`.

The function will be stopped either when there are no more attributes or when `map_func` returns a non-null value, which will be returned to the caller.

5.7.15 Mapping Through the Attributes of a Namespace

```
void
*ce_map_through_ns_attrs(
    CE_NAMESPACE namespace,
    void *(*map_func)(),
    void *args);
```

Maps through all the attributes of a namespace, calls `map_func()` for each attribute, and passes each attribute handle as the first argument to `map_func()`, each attribute value as the second argument, and `args` as subsequent arguments. `map_func()` is a user defined function. `args` are optional additional arguments for `map_func`. If no arguments are to be passed, use `NULL`.

The map will be stopped either when there are no more attributes or when `map_func` returns a non-null value, which will be returned to the caller.

5.7.16 Getting the Name of a Namespace

```
char
*ce_get_namespace_name(CE_NAMESPACE namespace);
```

We envision some namespace mapping functions requiring to know the name of a namespace, given a namespace handle. This function will return a namespace name, when passed a namespace handle.

5.7.17 Getting the Name of an Attribute

```
char
*ce_get_attribute_name(CE_ATTRIBUTE attribute);
```

There may be some attribute mapping functions that need to know the name of an attribute when passed a handle to it. This function will return an attribute name, when passed an attribute handle.

5.7.18 Determining Which Database Contains an Entry

```
int
ce_get_entry_db_info(
    CE_NAMESPACE namespace,
```

```
CE_ENTRY entry,
char **name_ptr,
char **path_ptr);
```

Returns the name of the database (either user, system, or network) in which an entry is stored. The name is returned in `*name_ptr` and the pathname of the database in `*path_ptr`. This call returns 0 if it is successful, otherwise it returns `CE_ERR_WRONG_ARGUMENTS`.

5.8 Classing Engine Utility Programs

Two utilities that enable reading and writing the Classing Engine database files to and from an ASCII form are available to allow developers to view the database. The `man` pages for these utilities follows on the next pages.

5.8.1 ce_db_build

The build utility, `ce_db_build`, will generate a readable ASCII file from the CE database, if given the `-from_ascii` argument. The user must also indicate the desired database (user, system, or network) and the filename where the file should be written. This allows a developer to print and peruse a hard copy of the database for familiarization or troubleshooting.

Caution – The `ce_db_build` utility will overwrite an existing CE database if given the `-from_ascii` argument. This will overwrite the existing CE database and replace it with the information from an ASCII file.

An optional argument, `-db_file` *filename*, can be given to generate a CE database file without disturbing the existing CE database files.

5.8.2 cd_db_merge

The merge utility, `ce_db_merge`, permits the merging of an ASCII database description file with an existing CE database file. This utility permits the merging of custom CE entries to the database.

NAME

ce_db_build - build an entire CE database

5

SYNOPSIS

ce_db_build **user**|**system**|**network** −**from_ascii**|−**to_ascii** *filename* \

[−**db_file** *db-filename*]

DESCRIPTION

cd_db_build reads from/writes to the Classing Engine databases and an ASCII description file.

user|**system**|**network** indicates which CE database is to be used, ether the user, the system, or the network database.

−**from_ascii** *filename* indicates that the user wishes to write to the stated CE database from the ASCII file *filename*. The entire CE database will be re-written. This is an all or nothing update of the CE database; that is, effectively the old database is erased and a new one is created based solely on the contents of the ASCII file.

−**to_ascii** *filename* indicates that the file named *filename* should be written with the ASCII description of the stated CE database. This ASCII description may then be modified and supplied as input to an invocation of **ce_db_build** with the −**from_ascii** argument.

OPTIONS

−**db_file** should be used in the case that a particular database is to be read from/written to using *db-filename* as the pathname of the CE database, instead of the default database files noted below.

FILES

The Classing Engine uses the following default database files:

```
user~/.cetables/cetables
system/etc/cetables/cetables
network$OPENWINHOME/lib/cetables/cetables
```

EXAMPLES

Create an ascii definition file `newdef` from the existing **user** CE database.

```
ce_db_build user -to_ascii newdef
```

Create the **user** CE database from file `new_db`.

```
ce_db_build user -from_ascii new_db
```

NAME

ce_db_merge - merge a Classing Engine ASCII database description file into the CE database

SYNOPSIS

ce_db_merge **user**|**system**|**network** **–from_ascii** *filename* \
[**–db_file** *db-filename*]

DESCRIPTION

ce_db_merge will attempt to merge namespace and entry definitions from an ASCII description file into an existing CE database. It will overwrite namespace attributes; that is, namespace attributes from the ASCII file will replace existing namespace attributes.

user|**system**|**network** indicates whether the user wants to update the user, the system, or the network CE database.

–**from_ascii** *filename* indicates that the user wishes to write the stated CE database from the ASCII file *filename*. The named CE database will be updated based on the ASCII description file. Any existing entries that also exist in the ASCII description file will be updated. Any new ASCII descriptors will be entered in the database.

OPTIONS

–**db_file** should be used in the case that a particular CE database is to be written to, using *db-filename* as the pathname of the CE database, instead of the default database files noted below.

FILES

The Classing Engine uses the following default database files:

database:default location:

```
user~/.cetables/cetables
system/etc/cetables/cetables
network$OPENWINHOME/lib/cetables/cetables
```

EXAMPLES

Merge an ascii definition file `newdef` into the existing user CE database.

```
ce_db_merge user –from_ascii newdef
```

Merge the ascii file `newdef` into the Classing Engine system database at /foo/bar/sysfile.

```
cd_db_merge system –from_ascii newdef –db_file /foo/bar/sysfile
```

5