RTF Tools
Macintosh Driver Architecture

Paul DuBois
dubois@primate.wisc.edu

Wisconsin Regional Primate Research Center
Revision date: 6 April 1994

Introduction

This document accompanies *RTF Tools Translator Architecture*, which you are assumed to be familiar with. It describes how the Macintosh translator drivers are set up.

The AppInfo Structure

The Macintosh versions of the RTF translators included in the RTF Tools distribution provide a standard Macintosh interface (windows, menus, etc.). They all use pretty much the same driver code, so there is a reasonable amount of consistency from translator to translator. In fact, most of the driver is absorbed into a wrapper that's provided as a library. When a translator starts up, the driver calls the wrapper and passes it a pointer to an *AppInfo* structure that contains information instructing the wrapper how to behave:

```
int
main (void)
{
    MacintoshWrapper (&myAppInfo);
}
```

The structure looks like this:

```
typedef struct AppInfo  AppInfo;

struct AppInfo
{
    void    (*pMenuInit) ();        /* application-specific menu setup */
    void    (*pMenuAdjust) ();      /* application-specific menu adjustment */

    void    (*pWriterInit) ();      /* writer one-time initialization */
    void    (*pWriterEnd) ();       /* writer one-time cleanup */

    int     (*pWriterBeginFile) (); /* per-file initialization */
    void    (*pWriterEndFile) ();   /* per-file cleanup */

    void    (*pPrefs) ();           /* function to handle Preferences dialog */

    Str31   inputSuffix;            /* input file suffix to strip if present */
    Str31   outputSuffix;           /* output file suffix for autonaming */
    Str31   errorSuffix;            /* error file suffix for autonaming */

    long    outputCreator;          /* creator of output files */
    long    errorCreator;           /* creator of error files */

    int     flags;                  /* flags to control wrapper behavior */
};
```

This structure contains several function pointers. If the translator has no need for a given function, specify `nil` in the *AppInfo* structure.

The structure members are used as follows:

`pMenuInit`

A pointer to a function to call that allows the driver to add its own menus to the menu bar. It's defined like this:

```
void
MyMenuInit (void)
{
}
```

The wrapper provides Apple, File, and Edit menus. If the driver needs to add other menus, it should do so in this function. The function doesn't need to call *DrawMenuBar()* because the wrapper calls it after calling the function.

`pMenuAdjust`

A pointer to a function to call when a mouse click occurs in the menu bar. It's defined like this:

```
void
MyMenuAdjust (void)
{
}
```

If the driver provides any of its own menus and the state of any items in them need to be adjusted (enabled, disabled, checked, etc.) before the user sees them, use this function to do so.

`pWriterInit`

A pointer to a function that performs one-time writer initialization before any input files are processed. It's defined like this:

```
void
MyWriterInit (void)
{
}
```

`pWriterEnd`

A pointer to a function that performs one-time writer cleanup after all input files have been processed. It's defined like this:

```
void
MyWriterEnd (void)
{
}
```

`pWriterBeginFile`

A pointer to a function to call to tell the writer to get ready to generate a new output file. It's defined like this:

```
int
MyWriterBeginFile (void)
{
}
```

When this function is called, the wrapper will already have opened the input file and called *RTFInit()*, installed the input file name into the RTF library by calling *RTFSetInputName()*, and opened the error file if there is one. If there is an output file, the wrapper will have installed its name by calling *RTFSetOutputName()*, and, if the `autoOpenOutputFile` flag is set (see below), will have opened it on *stdout*.

If the wrapper doesn't open the output file and one is to be generated, this function must open it.

The function should return non-zero for success (it was able to set up for a new file), zero otherwise (setup failed).

`pWriterEndFile`

A pointer to a function to call to tell the writer to clean up after an input file has been read. It's defined like this:

```
void
MyWriterEndFile (void)
{
}
```

The function should generate any final output that may be necessary. If the wrapper opened the output file, the wrapper will also close it after this function returns. Otherwise the function should close the file itself.

`pPrefs`

A pointer to a function to call to process a preferences dialog. It's defined like this:

```
void
MyPrefsDialog (void)
{
}
```

This function is ignored by the wrapper unless the `prefsInEditMenu` flags is set (see below). If the flag is set, the wrapper adds a gray line and a "Preferences..." item to the bottom of the Edit menu and calls this function when the user selects the item.

`inputSuffix`

`outputSuffix`

`errorSuffix`

These suffixes are used in constructing default output and error file names. They should be specified as Pascal strings and must be non-`nil`.

The names are constructed by stripping the input suffix off the input file name if it's present and then adding the output or error suffix. The input and error suffixes are typically `.rtf` and `.err`. The output suffix varies by translator.

Example: If the input, output, and error suffixes are `.rtf`, `.txt`, and `.err`, the default names for the input file *testfile.rtf* will be *testfile.txt* and *testfile.err*. Since the input suffix is only stripped if it's present, the default output and error file names for an input file named *testfile* would be the same. If the input file were *testfile.doc*, the default names would be *testfile.doc.txt* and *testfile.doc.err*.

The output and error suffixes are not used if the translator doesn't generate output and error files, but if it does, the suffixes should not be empty.

`outputCreator`

`errorCreator`

Creator signatures of the applications that should own output and error files created by the translator. A default of `'ttxt'` (TeachText) is used if the creator is given as zero in the *AppInfo* structure.

Output and error files, if created by the wrapper, are always `'TEXT'` files. If the translator wants to create output files with a type other than `'TEXT'`, it should tell the wrapper not to open output files and should open them itself. However, since file type is a Macintosh-specific thing and since the translator's writer code should be written to be system-independent, the writer cannot portably be written to set the file type. Once way to deal with this problem is to have the driver provide a function for opening output files which is called by the *pWriterBeginFile* function and which is appropriate for the type of system for which the driver is written.

`flags`

Flags controlling how the wrapper behaves. The value of this member should be zero or constructed as a combination of the following flags, or'ed together:

`persistentStatus`

If `persistentStatus` is set, the status window is always visible. Otherwise, it's visible only during file translations.

`prefsInEditMenu`

If `prefsInEditMenu` is set, the wrapper adds a gray line and a "Preferences..." item to the bottom of the Edit menu, and calls the `pPrefs` function in the *AppInfo* structure to process the preferences dialog when the user selects the item.

`usesOutputFile`

If `usesOutputFile` is set, it means the translator expects to write to an output file. If the flag is set, the `autoNameOutputFile` and `autoOpenOutputFile` flags are also examined (they're ignored if `usesOutputFile` isn't set).

`autoNameOutputFile`

If `autoNameOutputFile` is set, the output filename is automatically generated, and the user can't override it. If the flag isn't set, the name is still generated and used as the suggested default. The user can override it by means of a standard file dialog.

`autoOpenOutputFile`

If `autoOpenOutputFile` is set, the output file is opened by the wrapper on *stdout.* Otherwise, the writer is expected to open the output file itself.

`usesErrorFile`

`autoNameErrorFile`

`usesErrorFile` and `autoNameErrorFile` are similar to `autoNameOutputFile` and `autoOpenOutputFile`, but they apply to the error file. There is no `autoOpenErrorFile`; if an error file name is used, the wrapper always opens it (on *stderr*).

The writer can call *RTFMsg()* and *RTFPanic()* to generate diagnostic output even if no error file is created, because such output goes to the log window anyway.

Subverting pWriterInit

If the driver needs to do its own initialization (e.g., to read a preferences file), it can do this by specifying a driver function instead of a writer function for the *pWriterInit* member of the *AppInfo* structure. You can use this function to initialize the driver and then call any writer initialization function that may be necessary. When the driver initialization function is called, the *pMenuInit* function will already have been called, but none of the rest of the *AppInfo* structure will have been used. This means you can read a preferences file to get suffix settings and so forth and install them in the *AppInfo* structure with the expectation that the wrapper will use them properly.