RTF Tools
Translator Architecture

Paul DuBois
dubois@primate.wisc.edu

Wisconsin Regional Primate Research Center
Revision date: 5 April 1994

Introduction

The RTF translators included in the RTF Tools distribution were revised for release 1.10 to provide a more flexible architecture that makes it easier to port the writer end of a translator to different systems. This document describes some of the problems the architecture is intended to solve and how it tries to solve them. A companion document, *RTF Tools Macintosh Driver Architecture*, contains notes specific to the Macintosh translator drivers.

The Original Translator Architecture

The translators in the RTF Tools distribution have always used a design that factors the reader and writer code into separate entities. The reader and writer are linked together by a driver which sets up the input and output files, installs writer functions into the reader to be called when tokens of particular classes occur, and then calls the reader to process the input stream and pass tokens to the appropriate writer routines automatically.

Here is a simple example of a driver for a translator that can read an input file from the standard input or from a named file on a UNIX system:

```
int
main (argc, argv)
int     argc;
char    *argv[];
{
    if (argc > 1)   /* if a file was named, open it */
    {
        if (freopen (argv[1], "r", stdin) == (FILE *) NULL)
        {
            fprintf (stderr, "cannot open %s\n", argv[1]);
            exit (1);   /* terminate abnormally */
        }
        fprintf (stderr, "processing %s...\n", argv[1]);
    }

    /*
     * Initialize the reader, install a callback to catch text
     * tokens, and read the input stream.
     */

    RTFInit ();
    RTFSetClassCallback (rtfText, TextClass);
    RTFRead ();
```

```
        exit (0);   /* terminate normally */
    }
```

This driver installs a callback *TextClass()* into the reader to be called when text tokens occur in the input stream. *TextClass()* could be written like this to make the translator a simple program for extracting text from RTF files:

```
void
TextClass ()
{
    putchar (rtfMajor); /* character is in token major number */
}
```

Note that *TextClass()* is the entire text of the writer code for the translator. Tokens other than text tokens will be ignored. Most translators ramify the writer code considerably beyond this minimal effort.

Limitations of the Original Architecture

The translator just described reflects one of the primary characteristics exhibited by most of the translators included in the RTF Tools distribution as they were originally written. That is, they were built as single-file converters conceived for command-line use in a UNIX environment. They read a file and wrote the translated output to the standard output.

Such a translation model has limited applicability to other environments. For instance, under System 7 on a Macintosh it's desirable to take advantage of the drag-and-drop facility to allow the user to drop an arbitrary number of documents onto the translator with the expectation that each of them will be translated in sequence. For that to work, it must be possible to ‘‘restart’’ the translator for each file to be processed, without carrying over from file to file any file-specific state. Unfortunately, although the RTF reader has always been written to be restartable, the writer code for many of the translators has not been.

The example also demonstrates other system dependencies that have characterized the RTF Tools translators:

- Diagnostic messages are written to *stderr*.
- When a fatal error occurs, the process is terminated by calling *exit()* after writing a message to *stderr*.

These actions are appropriate for UNIX, but may not be for other environments. For instance, under System 7 it may be preferable to write diagnostic messages to a window rather than to *stderr*. But if you exit immediately after writing a message to a window when a fatal error occurs, the message disappears. This is not helpful to the user. It might be preferable to present the message in an alert that persists until the user dismisses it.

Another OS-dependent operation that's not shown in the example translator is the mechanism for locating library files such as the charset maps. Under UNIX these files are assumed to be located in a well-known directory such as */usr/local/lib/rtf-tools*. Under System 7 they might be best located inside the *Preferences* folder, or perhaps in the same folder as that in which the application itself is located.

The translators included with the RTF Tools distribution were revised for release 1.10 to alleviate these problems. Specifically:

- Reader and writer code is more system-independent.
- Writers are restartable so they can process multiple files.

Handling Operating System Dependencies

In order to accommodate system dependencies, a given translator is compiled on different systems with the same reader and writer code, but a separate driver is used for each system on which the translator is built. When it starts up, the driver provides callback functions for the OS-specific operations and plugs them into the RTF library:

```
RTFSetMsgProc (Msg);
RTFSetPanicProc (Panic);
RTFSetOpenLibFileProc (OpenLibFile);
```

Thereafter, the translator calls generic routines that map onto the OS-specific functions. For example:

```
/* display message */
RTFMsg ("processing input file %s\n", inputName);

/* abort */
RTFPanic ("cannot open %s", outputName);

/* open library file */
f = RTFOpenLibFile ("ansi-gen", "r);
```

This mechanism localizes the system dependencies into the driver, allowing the reader and writer to refer to standard function names without regard to the type of system on which the program is running.

Diagnostic Output

To handle diagnostic output, the driver should provide a function that takes a single null-terminated string argument and displays it to the user:

```
void
Msg (s)
char    *s;
{
    /* print s */
}
```

When the translator starts up, install the routine like this:

```
RTFSetMsgProc (Msg);
```

To write a diagnostic message from your translator, call *RTFMsg()*. *RTFMsg()* takes *printf()*-like arguments, but it formats the message into a single string before passing it to your message function so your function doesn't have to understand a variable number of arguments. Newlines in the string indicate linebreaks; your callback may need to convert them to something else if newlines are inappropriate linebreak characters for your system.

The default message function writes the message to *stderr*. If this is suitable, your translator doesn't need to provide a message function.

Process Termination

To terminate the translator when a fatal error occurs, the driver should provide a function that takes a single null-terminated string argument, displays it to the user, and exits:

```
void
Panic (s)
char    *s;
{
    /* print s */
    /* terminate process */
}
```

Your panic function *must* terminate the process, in whatever way is appropriate for your system. (e.g., by calling *exit()* under UNIX or by calling *ExitToShell()* on a Macintosh).

When the translator starts up, install the routine like this:

```
RTFSetPanicProc (Panic);
```

To terminate your translator, call *RTFPanic()*. Like *RTFMsg()*, *RTFPanic()* takes *printf()*-like arguments and formats them into a single string which it passes to the OS-specific routine. Unlike *RTFMsg()*, *RTFPanic()* adds a newline to the end of the message and indicates what token was last read from the input stream. Newlines in the string indicate linebreaks; your callback may need to convert them to something else.

RTFPanic() doesn't pass your panic function any exit status; your function can supply one if that's desired.

The default panic function writes the message to *stderr* and calls *exit()* with a status of 1. If this is suitable, your translator doesn't need to provide a panic function.

Library File Access

For opening library files, the driver should provide an OS-specific routine that takes a basename and access mode, locates the file and opens it, and returns a `FILE` pointer:

```
FILE *
OpenLibFile (name, mode)
char    *name;
char    *mode;
{
    /* find and open the file */
    /* return FILE pointer, or NULL if the file cannot be opened */
}
```

When the translator starts up, install the routine like this:

```
RTFSetOpenLibFileProc (OpenLibFile);
```

The translator opens library files by calling *RTFOpenLibFile()*. Library files should be specified by basename rather than by full pathname because pathnames vary in form among systems.

There is no default library file opening function. If your translator doesn't provide one, *RTFOpenLibFile()* always returns `NULL`.

The Driver/Writer Interface

In addition to providing OS-specific callbacks so the reader and writer can perform OS-dependent tasks in an OS-independent way, the driver must also translate information from its operating environment into a form the reader and writer can use:

- The driver handles the details of the file-delivery mechanism provided by the OS to determine which files are to be translated. For UNIX, these are named on the command line and are available in *argv[]*. For System 7, they're delivered through Apple Events.
- For each input file, the driver takes care of opening it, initializing the reader, and telling the writer when to begin and end output file processing.
- The driver transmits user preference information to the writer. For UNIX, these are named on the command line and are available in *argv[]*. For System 7, they can be specified in preferences files or via dialogs.

All of these are issues involving the interface between the driver and the writer. The next few sections describe the interface used by the translators in the RTF Tools distribution. Note that it's one I find convenient; it's not the only interface that's possible.

Structuring the Writer

This section describes how the writer code for a translator can be structured to allow for multiple file processing. As described below, there are four primary writer interface calls, although they need not all be applicable to any given translator. When present, the functions are written as shown below. None of them take any arguments and none except *WriterBeginFile()* return a value.

```
void
WriterInit ()
{
    /* perform any once-only initializations */
}

void
WriterEnd ()
{
    /* perform any once-only cleanup */
}

int
WriterBeginFile ()
{
    /* prepare to generate new output file */
    /* return value indicating success or failure of setup */
}

void
WriterEndFile()
{
    /* write any final per-file output */
}
```

In general, a driver uses the writer functions in a sequence something like this:

```
WriterInit ();  /* do once-only writer initialization */

for (/* each file to be translated */)
{
    freopen (..., stdin);   /* open input and output files */
    freopen (..., stdout);
    RTFInit();              /* initialize reader */
    if (WriterBeginFile ()) /* do per-file writer initialization */
    {
        RTFRead ();         /* read the input file */
        WriterEndFile ();   /* do per-file writer cleanup */
    }
}

WriterEnd ();   /* do once-only writer cleanup */
```

Writer Initialization and Termination

If your writer must perform one-time initialization before any input files are processed, or one-time cleanup after all files have been processed, provide the functions *WriterInit()* and *WriterEnd()*.

WriterInit() is called once before any files are to be translated, after the OS-specific callbacks have been installed. This allows the writer to perform any one-time initializations. One thing it might do is read the output map if there is one, since this generally need be done only once. (*rtf2troff* reads output maps on a per-input file basis, but that is the exception that proves the rule).

WriterEnd() is called after all the input files have been processed.

Processing the Input Files

Between the calls to *WriterInit()* and *WriterEnd()*, the translator processes the input files. For each one the driver should do the following:

- Open the input file on *stdin* and call *RTFInit()*.
- Open the output file if the writer doesn't do so. (Either the driver or the writer may open/close the output file, but of course they must agree between themselves which of them will do it.)
- If the writer needs to be called to begin each output file, call *WriterBeginFile()*. The responsibilities of this function will be described shortly.
- After calling *WriterBeginFile()*, call the RTF library to read the input file. This can be done two ways. You can call *RTFRead()* to read and process all the tokens in the input stream in one fell swoop. Alternatively, you can read and process them individually with a loop:

  ```
  while (RTFGetToken () != rtfEOF)    /* read and process one */
  {                                   /* token at a time */
      RTFRouteToken ();
  }
  ```

  RTFRead() is in fact a function consisting of the loop just shown. The reason you might want to use an explicit loop in the driver instead of calling *RTFRead()* is that you get control back each time through the loop. This allows you to take other actions in addition to calling *RTFRouteToken()*, which may be important on some systems.

  RTFRead() is sufficient for a system like UNIX where you can forcibly terminate a running process by typing the interrupt character. It's a considerably different undertaking to do the translation in a Macintosh event-loop user-driven environment where no external means is provided for terminating processes. The process itself must detect when the user wants an early quit, so it's desirable to make a translator responsive to the user during file processing, rather than just between files.

  RTFRead() doesn't work well under such circumstances. If it were to be called to process the input stream, control would disappear into the reader and writer code (which, being system independent, knows nothing of the Macintosh user interface code), until the file was entirely processed. This is bad because it locks out the user during the translation. To get around this, a normal event loop is run, and individual tokens are read from the input stream and processed when null events occur. This allows the user to select menu items (e.g., to request a quit, or to add new documents to the translation queue) while a translation is in progress. It also allows a progress bar and spinning cursor to be displayed.
- If the writer requires any per-file cleanup (e.g., to write any final output to the output stream) after the input stream has been exhausted, call *WriterEndFile()*.
- Close the input file.

Duties of WriterBeginFile()

In brief, *WriterBeginFile()* sets up the writer to begin producing a new output file and returns a status value to the driver.

WriterBeginFile() can assume that the driver has already opened the input file and called *RTFInit()*.

WriterBeginFile() sets the writer's state to what it should be at the beginning of each file. No state specific to the previous file should be retained. Typically this initialization involves the following:

- If the driver opens the file, it does so on *stdout*. If the driver doesn't open the output file, the writer should do so.

  If *WriterBeginFile()* needs to know the names of the input or output files, they may be obtained by calling *RTFGetInputName()* and *RTFGetOutputName()*. If this is to be done, the driver is responsible for calling *RTFSetInputName()* and *RTFSetOutputName()* beforehand. (The driver must do this after calling *RTFInit()* but before calling *WriterBeginFile()*.)

The driver must ensure that file pathnames are C strings in a form that can be *fopen()*-ed as is. Other than that, the writer cannot assume much about the contents of pathnames. For instance, pathname components are separated by "/" under UNIX, but by ":" on a Macintosh.

The input and/or output names may be `NULL`, which indicates that no file name is associated with the given stream. (This may happen, for example, if the translator is reading from or writing to a pipe under UNIX). One reason for wanting to know the name of the input file is if the writer wants to compute statistics on the input and report them in reference to the input name (for example, as *rtfwc* does to show the user which input file the output statistics are for). One reason the writer might want to know the output file name is when the writer writes multiple output files. In this case, it is convenient to use the primary output file name as a basis for deriving related file names.

The writer is free to abort if an output file name is required.

- *WriterBeginFile()* installs token class callbacks into the reader for writer functions that should be invoked when instances of particular token classes occur. *RTFInit()* resets all the callbacks, so they must be installed for each file processed. This is one reason *WriterBeginFile()* must be called after *RTFInit()* and not before. Another reason is that the writer might also read a few tokens from the input stream to make sure the input file looks like a valid RTF file (e.g., beginning with "{" and "\rtf"). This can only be done after *RTFInit()* has been called.
- *WriterBeginFile()* determines the status value to return to the driver. The value should be non-zero if *WriterBeginFile()* is able to set up for translating a new file, or zero if some error occurs that prevents translation of the input file.

  If *WriterBeginFile()* fails, it should generate its own error message, not leave that for the driver. If a failure occurs, the driver decides whether to abort or to continue with any further files that may be queued for translation.

Interface Guidelines Summary

- Install OS-specific callbacks when the translator starts up.
- To generate diagnostic message that aren't written to the primary output file, call *RTFMsg()*. Don't write to *stderr*, since there might not be any open file associated with it.
- To generate an error message and terminate the process in error, call *RTFPanic()*. Don't write to *stderr*, and don't call *exit()*.
- To open a library file, pass the file's basename and open mode to *RTFOpenLibFile()*.
- The driver is responsible for opening the input file on *stdin*.
- Either the driver or the writer may open the output file. If the driver opens it, it should do so on *stdout*.
- If the writer needs to know the input or output file names, the driver is responsible to set them with *RTFSetInputName()* and This must be done after calling *RTFInit()* and before calling the writer's begin-file function. The writer may then obtain the names with *RTFGetInputName()* and If the writer wants to modify the strings returned by these functions, it should make copies of them.

Handling User Preferences

This section is, uh, not very good yet.