A Tool for RTF Processing
Release 1.10

Paul DuBois
dubois@primate.wisc.edu

Wisconsin Regional Primate Research Center
Revision date: 5 April 1994

Introduction

This document describes a general purpose tool for processing RTF files—an RTF reader which may be configured in a well-defined manner to allow it to be used with a variety of writers generating different output formats. This provides a method for generating RTF-to-*XXX* translators.

I assume that you have some familiarity with RTF syntax and semantics, and that you're willing to study the source code of the RTF distribution described here. If you don't have the RTF specification, you can get it from the FTP site listed under "Distribution Availability" at the end of this document. References to "the specification" refer to the RTF specification document.

If you use this tool and find that you have an RTF file that won't pass through the sample translator *rtf2null*, or for which *rtf2null* announces unknown symbols, please contact me so the tool can be improved. It is best if you can supply the RTF file for which this behavior is observed.

Theory of Operation

Translator Architecture

This is a brief description of how translators are designed. For more details, see the document *RTF Tools Translator Architecture*.

There are three components to an RTF translator: reader code, writer code, and driver code. These break down as follows.

reader
: Responsible for peeling tokens out of the input stream, classifying them, and causing the writer to process them.

writer
: Responsible for translating tokens from the input stream into the required output format.

driver
: Responsible for making sure the reader and writer are initialized, and for calling the reader, to cause translation to occur.

This architecture allows the reader to remain constant, so that different translators can be built by supplying different writer and driver code. Also, for a given translator, the reader and writer remain constant and the translator can be ported to different types of systems by supplying system-specific driver code.

In practice, to build a new translator, you supply a *main()* function and the writer code, and link in the RTF reader. *main()* includes the driver code and is responsible to see that the following are done:

- Determine which files are to be translated
- Configure the reader, which may involve:
  - — Reset the input stream if necessary
  - — Configure other reader behavior, such as whether or not to process the font and color tables internally
  - — Install writer callbacks into the reader so it knows what functions to call when various kinds of tokens occur
- Initialize the writer
- Call the reader to process the input stream
- Terminate the writer

A minimal translator (for a UNIX system) looks something like this:

```
# include   <stdio.h>
# include   "rtf.h"

int
main ()
{
    RTFSetOpenLibFileProc (UnixOpenLibFile);
    RTFInit ();
    RTFRead ();
    exit (0);
}
```

This installs a function that's suitable for opening RTF library files on a UNIX system, initializes the reader, and calls it to read *stdin* (the default input stream). The writer portion is null (i.e., there is no writer), so all that happens is that the reader tokenizes the input and discards it. That isn't very interesting; most of the sample translators are examples of more elaborate translators.

Reader Operation

Each time a token is read, several global variables are set:

rtfClass	token class
rtfMajor	token major number
rtfMinor	token minor number
rtfParam	token parameter value
rtfTextBuf	token text
rtfTextLen	length of token (including parameter text)

Tokens are classified using up to three numbers: token class, and major and minor numbers. The major and minor numbers may be meaningless depending on the kind of token.

The class number can be:

rtfUnknown	unrecognized token
rtfGroup	"{" or "}"
rtfText	plain text character
rtfControl	token beginning with "\"
rtfEOF	fake class number; indicates end of input stream

There are some exceptions. A few tokens beginning with \ actually belong to other classes, a tab character is treated like `\tab`, and unrecognized tokens are put in class *rtfUnknown* no matter what they look like.

Within a class, tokens are assigned a major number, and perhaps a minor number. For the *rtfText* class, the major number is the value of the input character (0..255), and the minor number is assigned a standard character code. Text characters have different mappings in different RTF character sets, so to avoid the problems associated with this, the reader maps the character onto a standard character code using a charset-dependent translation table. Translators should generally use the standard character code in *rtfMinor* rather than the raw character code in *rtfMajor*. Character mapping issues are described further in the document *RTF Tools Character Mapping*.

A "plain text" character can be a literal character, a character specified in hex notation (\´*xx*) or one of the special escaped characters (`\{`, `\}`, `\\`). The sequence `\:` is treated as a plain text colon. This is arguably wrong; the rationale is given later under the description of the *RTFGetToken()* function.

For the *rtfControl* class, most tokens have both a major and minor number. For instance, all paragraph attribute control symbols have major number *rtfParAttr* and a minor number indicating a paragraph formatting property, such as *rtfLeftIndent* or *rtfSpaceBefore*. A few oddball control tokens have no minor number.

Control symbols may have a parameter value, e.g., `\margr720` specifies a right margin (in units of 720 twentieths of a point).

If no parameter value is given, *rtfParam* is *rtfNoParam*.

Ideally, there should never be any tokens in the *rtfUnknown* class, but as the RTF standard continues to develop, unknown tokens are inevitable.

To write a translator, you'll need to familiarize yourself with the token classification scheme by reading *rtf.h*. A skeleton translator *rtfskel* is included with the distribution and may be used as a basis for new translators.

As of release 1.10, the reader allows an 8-bit character set since the current RTF specification (version 1.2) now allows 8-bit characters. Formerly, if the reader saw an 8-bit character, it converted the character to the equivalent \´*xx* hex notation sequence and returned that as the token.

Generally, a translator will configure the RTF reader to call particular writer functions when certain kinds of tokens are encountered in the input stream. These functions are known as *class callbacks*. Writer callbacks can be registered with the reader using *RTFSetClassCallback()* for each token class.

The reader reads each token, classifies it, and sends it to a token routing function *RTFRouteToken()*, which tries to find a writer callback function to process the token. Tokens in a given class are ignored if no callback is registered for the class.

Class callbacks make it quite easy to receive notification when certain types of tokens occur in the input. For instance, a crude RTF text extractor could be written by installing a callback function for the *rtfText* class.[1] Whenever the function is invoked, *rtfMajor* will contain a value in the range 0..255 representing the character value.

```
# include   <stdio.h>
# include   "rtf.h"

void
TextCallback ()
{
    putchar (rtfMajor);
}

int
main ()
```

[1] Reasons this is a crude translator are that: (i) some text characters occur in contexts where the characters are not intended to be output, e.g., font tables, stylesheets; (ii) some control symbols like `\tab` represent output text characters; (iii) it writes output based on the raw input character value in *rtfMajor* rather than mapping the standard character code in *rtfMinor*. The sample translator *rtf2text* addresses these problems in a (slightly) more sophisticated manner.

```
{
    RTFSetOpenLibFileProc (UnixOpenLibFile);
    RTFInit ();
    RTFSetClassCallback (rtfText, TextCallback);
    RTFRead ();
    exit (0);
}
```

Callbacks for the *rtfControl* and *rtfGroup* classes typically operate by selecting on the token major number to determine the action to take. A callback for the *rtfGroup* class usually will do something like this:

```
void
BraceCallback ()
{
    switch (rtfMajor)
    {
    case rtfBeginGroup:
        ...push state...
        break;
    case rtfEndGroup:
        ...pop state...
        break;
    }
}
```

Destination Readers

Grouping in RTF documents occurs within braces "{" and "}". One kind of group is the *destination*. The token immediately following the opening brace is a destination control symbol. These indicate such things as headers, footers, footnotes, etc.

Three destinations which specify information for internal use (i.e., information which affects output but isn't itself written) are the font table, color table and stylesheet. Since these three destinations occur so commonly and have a special syntax, the RTF reader by default gobbles them up itself when it recognizes them. The functions which do this are called *destination readers* and are probably the nearest thing in the reader to what might be called parsers. They are installed by default so that translators can be written without the burden of understanding the syntax or digesting the contents of these destinations. Each of them constructs a list of the entries specified in the destination and the reader includes functions providing access to these lists.

Translators can turn off or override these defaults with *RTFSetDestinationCallback()* if necessary. To override one, pass the address of a different destination reader function. To turn one off, pass `NULL`.

Destination callbacks may be called for any destination, not just *rtfFontTbl*, *rtfColorTbl* and *rtfStyleSheet*. Destinations for which no callback is registered are not treated specially.

Other destinations for which there is a default reader are the information (`\info`), picture (`\pict`), and object (`\object`) destinations; all they do is skip to the end of the group.

Using the Built-in Destination Readers

The font table, color table and stylesheet information is maintained internally, and the reader either acts on that information itself, or allows itself to be queried by the writer about it, as described below. These descriptions do not apply if the translator shuts off or overrides the default destination readers, of course.

Stylesheet—The reader acts on this itself. When the stylesheet destination is encountered, the style contents are remembered. Thereafter, whenever the writer receives notification that a style number control symbol (`\s`*nnn*) has occurred, it can call *RTFExpandStyle(rtfParam)* to cause the style to be expanded. The reader consults contents of the stylesheet and each token in the style definition is routed in turn back to the writer. This effects a sort of macro expansion.

If the writer doesn't care about style expansion, it simply refrains from calling *RTFExpandStyle()*.

If the writer wants information about a style, it can call *RTFGetStyle()*.

Font table—For each entry in the font table, the font number, type and name are maintained by the reader. The writer finds out that a font number has been specified in the input when its control class callback is invoked and *rtfMajor* = *rtfCharAttr* and *rtfMinor* = *rtfFontNum*. To obtain a pointer to the appropriate *RTFFont* structure, the reader function *RTFGetFont(rtfParam)* may be called.

Color table—For each entry in the color table, the color number is maintained along with the red, green and blue values. The writer finds out that a color number has been specified in the input when its control class callback is invoked and *rtfMajor* = *rtfCharAttr* and *rtfMinor* = *rtfColorNum*. To obtain a pointer to the appropriate *RTFColor* structure, the reader function *RTFGetColor(rtfParam)* may be called.

One subtle point about the built-in destination readers: destinations cannot be recognized until *after* the occurrence of the "{" symbol that begins the destination. This means the writer, if it maintains a state stack, will already have pushed a state. In order to allow the writer to properly pop that state in response to the "}", these destination readers feed the "}" back into the token router after they pull it from the input stream. What the writer actually sees is a "{" followed immediately by a "}".

Applications that maintain a state stack may find it necessary to do something similar if they supply their own destination readers.

Programming Interface

Source files using the RTF reader should #include *rtf.h*. The library files common to all translators are used to build a library *librtf.a* in the distribution's *lib* directory. This library should be part of the final application link.

The best way to learn how these source files work is to study the sample translators, which vary in complexity from very simple (e.g., *rtf2text*, *rtfwc*), to wretchedly messy (e.g., *rtf2troff*). You should be aware that one implication of the way the translators are built (callbacks and switch statements) is that it's quite easy to build them incrementally. You can start with a very bare-bones model, and start plugging in callbacks as you progress. Within the callbacks, your switch statements can progressively handle more cases.

An alternative approach is to start with a copy of *rtfskel*, which includes a full set of class callbacks and complete switch statements for all tokens. Each case is empty; you simply add code for those cases you want to handle. You can also rip out the code for the cases you don't care about.

Types

Most types are pretty standard. The one of note is *RTFFuncPtr*, a generic function pointer which is defined like so:

```
typedef void (*RTFFuncPtr) ();
```

That is, it's a pointer to a function that takes no arguments and returns no value.

Global variables

The global RTF reader variables are:

int	rtfClass;	token class
int	rtfMajor;	token major number
int	rtfMinor;	token minor number
int	rtfParam;	parameter value for control symbols
char	*rtfTextBuf;	token text
int	rtfTextLen;	length of token text

These variables always apply to the token with which the writer should be concerned. This may be either the last token read or the current token within a style which is being reprocessed.

Warning: *rtfTextBuf* is `NULL` until *RTFInit()* has been called.

Two other global variables which may be of interest provide the current input line number and position within the line:

long	rtfLineNum;	current input line
int	rtfLinePos;	position within current line

These variables can be used to provide feedback to the user when a problem is found in an input file as to the location of the problem. They indicate the position immediately after the last token read.

Functions

```
void
RTFInit ()
```

Initialize the RTF reader. This should be called once for each input file to be processed. It performs some initialization such as computing hash values for the token lookup table and installation of some built-in destination and token class readers.

RTFInit() may be called multiple times. Each invocation resets the reader's state completely, except that the input stream is not disturbed.

```
void
RTFRead ()
```

RTFRead() calls *RTFGetToken()* to tokenize the input stream and *RTFRouteToken()* to process each token, until input is exhausted. When *RTFRead()* returns, input has been completely read and the writer can perform any cleanup or termination needed.

If you want to read multiple files per invocation of your translator, you should do the following for each file: call *RTFInit()*, install callbacks, etc., then call *RTFRead()*.

```
void
RTFRouteToken ()
```

This routine decides what to do with the current token and routes it to the correct place for processing. Usually this is directly to the writer via a class callback. The token is *not* passed to the writer (i.e., the class callback is bypassed) when it is a destination token for which a reader callback is installed.

By default, built-in readers are installed for font table, color table, stylesheet and information and picture group destinations. The built-in readers can be disabled if the writer wants to see all tokens directly.

```
int
RTFGetToken ()
```

Reads one token from the input stream, classifies it, sets the global variables, and returns the class number. If the class is *rtfEOF* the end of the input stream has been reached. Newlines (`\n`), carriage returns (`\r`), and nulls are silently discarded by *RTFGetToken()*, as they have no meaning. All are passed to the token hook if one is installed, however.

The sequence `\:` is treated as a plain text character, with *rtfClass* set to *rtfText* and *rtfMajor* set to the colon ASCII code. Strictly speaking, `\:` is the control word for an index subentry, but some versions of Microsoft Word write out plain text colons with a preceding backslash, while others don't. This unfortunate ambiguity results in an ugly dilemma. It seems the lesser burden to require translators to recognize that plain text colons should "really" be treated as index subentry indicators while inside of an index entry destination, than to recognize that an index subentry control word should "really" be treated as a plain text colon everywhere else.

Writer code usually does not call *RTFGetToken()* directly except within specialized destination readers. Driver code usually does not call *RTFGetToken()* if it calls *RTFRead()*. However, the following loop is an alternative to *RTFRead()*:

```
while (RTFGetToken () != rtfEOF)
{
    RTFRouteToken ();
}
```

If a driver wants to regain control after reading each token, this loop may be preferable to *RTFRead()*.

```
int
RTFUngetToken ()
```

Pushes the last token back on the input stream so that *RTFGetToken()* returns it again. You can't put back the same token twice unless you read it again in the interim.

```
int
RTFPeekToken ()
```

Reads a token from the input stream and sets the global token variables, but does not remove the token from the input stream.

```
void
RTFSetToken (class, major, minor, param, text)
int       class, major, minor, param;
char      *text;
```

It is sometimes useful to construct a fake token and run it through the token router to cause the effects of the token to be applied. *RTFSetToken()* allows you to do this, by setting the reader's global variables to the values supplied. If *param* is *rtfNoParam*, the token text *rtfTextBuf* is constructed from *text* and *param*, otherwise *rtfTextBuf* is just copied from *text*.

```
void
RTFSetReadHook (f)
RTFFuncPtrf;
```

Install a function to be called by *RTFGetToken()* after each token is read from the input stream. The function takes no arguments and returns no value. Within the function, information about the current token can be obtained from the global variables. This function is for token examination purposes only, and should not modify those variables.

```
RTFFuncPtr
RTFGetReadHook ()
```

Returns a pointer to the current read hook, or `NULL` if there isn't one.

```
void
RTFSkipGroup ()
```

This function can be called to skip to the end of the current group (including any subgroups). It's useful for explicitly ignoring `\*\`*dest* groups, where *dest* is an unrecognized destination, or for causing groups that you don't want to deal with to effectively "disappear" from the input stream.

Calling this function in the middle of expanding a style may cause problems. However, it is typically called when you have just seen a destination symbol, which won't happen during a style expansion—I think.

Be careful with this function if your writer maintains a state stack, because you will already have pushed a state when the opening group brace was seen. After *RTFSkipGroup()* returns, the group closing brace has been read, and you'll need to pop a state. All global token variables will still be set to the closing brace, so you may only need to call *RTFRouteToken()* to cause the state to be unstacked.

```
void
RTFExpandStyle (num)
int       num;
```

Performs style expansion of the given style number, or does nothing if there is no such style. The writer should call this when it notices that the current token is a style number indicator.

```
void
RTFSetStream (stream)
FILE      *stream;
```

Redirects the RTF reader to the given stream. This should be called before any reading is done. The default input stream is *stdin*. An alternative to *RTFSetStream()* is to simply *freopen()* the input file on *stdin* (that's what all the sample translators do).

The input stream is *not* modified by *RTFInit()*.

```
void
RTFSetClassCallback (class, callback)
int       class;
RTFFuncPtrcallback;
```

Installs a writer callback function for the given token class. The first argument is a class number, the second is the function to call when tokens from that class are encountered in the input stream. This will cause *RTFRouteToken()* to invoke the callback when it encounters a token in the class. If *callback* is `NULL` (which is the default for all classes), tokens in the class are ignored, i.e., discarded.

The callback should take no arguments and return no value. Within the callback, information about the current token can be obtained from the global variables.

Installing a callback for the *rtfEOF* "class" is silly and has no effect.

```
RTFFuncPtr
RTFGetClassCallback (class)
int       class;
```

Returns a pointer to the callback function for the given token class, or `NULL` if there isn't one.

```
void
RTFSetDestinationCallback (dest, callback)
int       dest;
RTFFuncPtrcallback;
```

Installs a callback function for the given destination (*dest* is a token minor number). When *RTFRouteToken()* sees a token with class *rtfControl* and major number *rtfDestination*, it checks whether there is a callback for the destination indicated by the minor number. If so, it invokes it. If *callback* is `NULL`, the given destination is not treated specially (the control class callback is invoked as usual). By default, destination callbacks are installed for the font table, color table, stylesheet, and information and picture group.

The callback should take no arguments and return no value. When the functon is invoked, the current token will be the destination token following the destination's initial opening brace `{`. (For optional destinations, the destination token follows the `\*` symbol.)

```
RTFFuncPtr
RTFGetDestinationCallback (dest)
int       dest;
```

Returns a pointer to the callback function for the given token class, or `NULL` if there isn't one.

```
RTFStyle *
RTFGetStyle (num)
int       num;
```

Returns a pointer to the *RTFStyle* structure for the given style number. The "Normal" style number is 0. Pass −1 to get a pointer to the first style in the list. Styles are not stored in any particular order.

Be sure to check the result; it might be `NULL`.

This function is meaningless if the default stylesheet destination reader is overridden.

```
RTFFont *
RTFGetFont (num)
int       num;
```

Returns a pointer to the *RTFFont* structure for the given font number. Pass −1 to get a pointer to the first font in the list. Fonts are not stored in any particular order.

Be sure to check the result; it might be `NULL`. In particular, you might think that passing the number specified with the `\deff` (default font) control symbol would always yield a valid font structure, but that's not true. The default font might not be listed in the font table.

This function is meaningless if the default font table destination reader is overridden.

```
RTFColor *
RTFGetColor (num)
int       num;
```

Returns a pointer to the *RTFColor* structure for the given color number. Pass −1 to get a pointer to the first color in the list. Colors are not stored in any particular order. If the color values in the entry are −1, the default color should be used. The default color is translator-dependent.

Be sure to check the result; it might be `NULL`. I think this means you should use the default color.

This function is meaningless if the default color table destination reader is overridden.

```
int
RTFCheckCM (class, major)
int       class, major;
```

Returns non-zero if *rtfClass* and *rtfMajor* are equal to *class* and *major*, respectively, zero otherwise.

```
int
RTFCheckCMM (class, major, minor)
int       class, major, minor;
```

Returns non-zero if *rtfClass*, *rtfMajor* and *rtfMinor* are equal to *class*, *major* and *minor*, respectively, zero otherwise.

```
int
RTFCheckMM (major, minor)
int       major, minor;
```

Returns non-zero if *rtfMajor* and *rtfMinor* are equal to *major* and *minor*, respectively, zero otherwise.

```
char *
RTFAlloc (size)
int       size;
```

Returns a pointer to a block of memory *size* bytes long, or `NULL` if insufficient memory was available.

```
char *
RTFStrSave (s)
char      *s;
```

Allocates a block of memory big enough for a copy of the given string (including terminating null byte), copies the string into it, and returns a pointer to the copy. Returns `NULL` if insufficient memory was available.

```
void
RTFFree (p)
char      *p;
```

Frees the block of memory pointed to by *p*, which should have been allocated by *RTFAlloc()* or *RTFStrSave()*. It is safe to pass `NULL` to this routine.

```
void
RTFCharToHex (c)
char      c;
```

Returns 0..15 for the characters ‘0‘..‘9‘,‘a‘..‘f‘.

```
void
RTFHexToChar (i)
int       i;
```

Returns the characters ‘0‘..‘9‘,‘a‘..‘f‘ for 0..15.

```
int
RTFReadCharSetMap (file, csId)
char      *file;
int       csId;
```

Reads a charset map file into the charset map indicated by *csId*, which should be either *rtfCSGeneral* or *rtfCSSymbol*. Returns non-zero for success, zero otherwise.

```
void
RTFSetCharSetMap (file, csId)
char      *file;
int       csId;
```

Specify the name of the file to be read for the charset map indicated by *csId* (which should be either *rtfCSGeneral* or *rtfCSSymbol*) when auto-charset-file reading is done. This can be used to override the default charset map names. *RTFSetCharSetMap()* should be called after *RTFInit()* but before you begin reading any input.

```
void
RTFSetCharSet (csId)
int       csId;
```

Switches to the charset map given by *csId*, which should be either *rtfCSGeneral* or *rtfCSSymbol*.

```
int
RTFGetCharSet ()
```

Returns the id of the current charset map, either *rtfCSGeneral* or *rtfCSSymbol*.

```
int
RTFMapChar (c)
int       c;
```

Maps in input character onto a standard character code.

```
int
RTFStdCharCode (name)
char      *name;
```

Given a standard character name, returns the standard code corresponding to the name, or −1 if the name is unknown.

```
char *
RTFStdCharName (code)
int       code;
```

Given a standard character code, returns a string pointing to the standard character name, or `NULL` if the code is unknown.

```
int
RTFReadOutputMap (file, outMap, reinit)
char      *file;
char      *outMap[];
int       reinit;
```

Reads an output map from the named file into *outMap*. If *reinit* is non-zero, the map is cleared first. See the document *RTF Tools Character Mapping* for further details.

Generally, the output map needs to be read only once.

```
void
RTFSetInputName (name)
char      *name;

void
RTFSetOutputName (name)
char      *name;
```

These functions tell the RTF library the input or output file names. They're called by driver code so that writer code can determine the names by calling *RTFGetInputName()* and *RTFGetOutputName()*. Since *RTFInit()* sets the names to `NULL`, the driver should set the names after calling *RTFInit()* but before calling the writer to tell it to set up for a new file.

```
char *
RTFGetInputName ()

char *
RTFGetOutputName ()
```

These functions return pointers to the current input and output file names, assuming the driver has set them up. The caller should make a copy of the strings returned if it wants to modify them.

```
void
RTFMsg (args ...)
```

This function generates a diagnostic message. It takes *printf()*-like arguments.

See the description of *RTFSetMsgProc()*.

```
void
RTFPanic (args ...)
```

This function generates an error message and terminates the process. It takes *printf()*-like arguments.

See the description of *RTFSetPanicProc()*.

```
FILE *
RTFOpenLibFile (name, mode)
char      *name;
char      *mode;
```

This function opens a library file and returns a `FILE` pointer to it, or `NULL` if the file could not be opened.

See the description of *RTFSetOpenLibFileProc()*.

```
void
RTFSetMsgProc (proc)
void      (*proc) ();
```

This function installs a function for use by *RTFMsg()*; see *RTF Tools Translator Architecture* for details.

```
void
RTFSetPanicProc (proc)
void      (*proc) ();
```

This function installs a function for use by *RTFPanic()*; see *RTF Tools Translator Architecture* for details.

```
void
RTFSetOpenLibFileProc (proc)
FILE      *(*proc) ();
```

This function installs a function that the library will use to open library files. The driver must call this when it starts up or *RTFOpenLibFile()* will always return `NULL`. The function should take a library file basename and open mode, open the file, and return the `FILE` pointer, or `NULL` if the file could not be found and opened.

Distribution Availability

This software may be redistributed without restriction and used for any purpose whatsoever.

The RTF Tools distribution is available for anonymous *ftp* access on *ftp.primate.wisc.edu*. Look in the */pub/RTF* directory. Updates appear there as they become available.

A version of the RTF specification is available in this directory, as a binhex'ed Word for Macintosh document and in RTF and PostScript formats.

The software and documentation may also be accessed using gopher by connecting to *gopher.primate.wisc.edu* or using World Wide Web by connecting to *www.primate.wisc.edu* using the URL *http://www.primate.wisc.edu/*. In both cases, look under "Primate Center Software Archives".

If you do not have Internet access, send requests to *software@primate.wisc.edu*. Bug reports and questions should be sent to this address as well.

If you use this software as the basis for a translater not included in the current collection, please send me a description that indicates how it may be obtained and I'll add the description to the archive site.