Documentation for parse.h and parse.c

Steven Andrews, © 2008-2023

Header

```
#ifndef __parse_h
#define __parse_h

typedef struct ParseFileStruct {
  char *froot;                  // root of file name
  char *fname;                  // complete file name, including root
  FILE *fptr;                   // pointer to open file
  int lctr;                       // line counter
  int maxlinechar;              // allocated size of line and linecopy
  char *line;                   // the line being parsed
  char *linecopy;               // copy of the line being parsed
  int incomment;                // flag for block comments
  struct ParseFileStruct *prevfile;// previous file in linked list
  int maxdef;                   // allocated size of define lists
  int ndef;                       // current length of define lists
  char **defkey;                // search keys for defines
  char **defreplace;            // replacement text for defines
  int *defgbl;                  // flag for if define is global
  int inifdef;                  // counter for depth in ifdefines
  } *ParseFilePtr;

ParseFilePtr Parse_AllocFilePtr(char *fileroot,char *filename);
void Parse_FreeFilePtr(ParseFilePtr pfp);
int Parse_ExpandDefine(ParseFilePtr pfp,int maxdef);
int Parse_AddDefine(ParseFilePtr pfp,const char *key,const char *replace,int
global);
int Parse_RemoveDefine(ParseFilePtr pfp,char *key);
void Parse_DisplayDefine(ParseFilePtr pfp);
int Parse_DoDefine(ParseFilePtr pfp);

int Parse_CmdLineArg(int *argcptr,char **argv,ParseFilePtr pfp) {
ParseFilePtr Parse_Start(char *fileroot,char *filename,char *erstr);
int Parse_ReadLine(ParseFilePtr *pfpptr,char *word,char **line2ptr,char
*erstr);
int Parse_ReadFailure(ParseFilePtr pfp,char *erstr);

#endif
```

Requires: `<stdio.h>`, `<stdio.h>`, `<stdlib.h>`, `"string2.h"`

Example program: `Smoldyn`

History:

6/3/08	Spun off from Smoldyn.
2/9/11	Added `Parse_CmdLineArg`.

3/5/11 Removed successive replacements from `Parse_DoDefine`, added `Parse_DisplayDefine`, and some other minor improvements.
2/29/12 Added recursion to `Parse_DoDefine`.
4/16/12 Changed declaration for Parse_AddDefine for C++ conformity.
5/17/15 Fixed a bug in global definitions in which they weren't being passed to upstream files.
10/5/15 Fixed a bug that I created in the 5/17/15 bug repair.
8/15/23 Tidied up code and documentation. Planned to enable arbitrarily long input lines but then reverted. Changed all strings to 4096 characters instead of 256.

Overview

This library provides a data structure and several functions for reading configuration files in a sequential fashion. These function take care of file opening, closing, files that read other files, comments within the file, error message generation, etc. Note that all strings here have `STRCHARLONG` (defined in string2.h to be 4096) characters.

Recognized text

Several terms are recognized by the parsing function

`#`	single-line comment
`/*`	start of a multi-line comment
`*/`	end of a multi-line comment
end of file	reading continues to prior file or terminates
`end_file`	reading continues to prior file or terminates
blank line	ignored
`read_file`	reading continues at listed file
`define`	define new macro substitution text
`define_global`	define new macro substitution text with global scope
`undefine`	remove macro subtitution text
`ifdefine`	continue reading if it is defined, otherwise skip to else or endif
`ifundefine`	continue reading if not defined, otherwise skip to else or endif
`else`	toggle reading on or off based on from prior ifdefine or ifundefine
`endif`	if not reading because of ifdefine or ifundefine, then start reading

These are described in more detail in the Smoldyn manual.

Data structure

If there is just one configuration file, then one ParseFileStruct (pfp) is used. On the other hand, this configuration file might call another, and it might call another, and so on. This list of dependencies is kept track of with a linked list of pfps. This list takes care of itself remarkably smoothly. While the configuration file collection might represent a tree, the pfp data structures are created and freed as needed, so they only trace the current track down the tree.

The data structure contains information about the file that is currently being read, including the root of its name (the path), the complete file name, and a pointer to the open

file. Also, there is some information about the reading status, including the line number that is being parsed, the actual line being parsed and a copy of it, and whether the current line is within a block comment or not. Because a file can instruct another file to be read, this leads to a stack of open files, which is stored with a linked list. The `prevfile` element links the record for the current file back to the previous file, which is used when the current one is closed.

It is possible to perform macro substitution in configuration files, which are called defines. For these, `maxdef` is the allocated size of the lists for the defines, `ndef` is the actual size of the these lists, `defkey` is the list of macro identification keys, `defreplace` is the list of replacement strings, and `defgbl` is a list of flags which are 1 if the define is global, meaning that it applies to this file and all files called by this file, or is 0 if the define is local to just this file. In the data structure, the defines are sorted by the lengths of the key strings, with the longest ones first. This is done to prevent replacement of short keys when they are part of longer keys. For example, if the key-replace pairs include "`KEY`", "`xx`" and "`KEY1`", "`yy`", then if the parser finds "`KEY1`" at some point in the text, it should replace it with "`yy`" and not with "`xx1`".

Functions that should never need to be called from externally

```
ParseFilePtr Parse_AllocFilePtr(char *fileroot,char *filename);
```

This allocates a new parse file structure and initializes it. `froot` is initialized to `fileroot`, `fname` to a concatanation of `fileroot` and `filename`, and other values to 0 or `NULL` as appropriate. `fileroot` and/or `filename` can be entered as `NULL`, in which case those components of the internal strings are left as empty strings. A pointer to the structure is returned unless memory could not be allocated, in which case `NULL` is returned.

```
void Parse_FreeFilePtr(ParseFilePtr pfp);
```

Frees a parse file structure, including its members. The previous structure, pointed to by `prevfile`, is not freed.

```
int Parse_ExpandDefine(ParseFilePtr pfp,int maxdef);
```

Expands (or shrinks) the size of the define lists in the listed `pfp` to size `maxdef`. Any previous list contents are copied over into the new lists and previous lists are freed and then replaced with new ones. Returns 0 for success, 1 for inability to allocate the required memory, and 2 for illegal inputs; `maxdef` needs to be at least 1.

```
int Parse_AddDefine(ParseFilePtr pfp,char *key,char *replace,int global);
```

Adds a define to the listed `pfp`. The key text is `key`, the replacement text is `replace`, and it is made global if `global` is 1 and not if global is 0. `replace` may be entered as `NULL`, in which the replacement text is left as an empty string. Returns 0 for normal operation, 1 if required memory could not be allocated, or 2 if the new definition would overwrite an old definition (but this is not done; the old definition is kept).

```
int Parse_RemoveDefine(ParseFilePtr pfp,char *key);
```

Removes the define from the listed `pfp` that has key equal to `key`. Returns 1 if a matching key could be found and 0 if so. If `key` is set to `NULL`, this removes all defines, whether local or global.

```
void Parse_DisplayDefine(ParseFilePtr pfp);
```
Simply prints out the current file path and name, followed by all current definitions.

```
int Parse_DoDefine(ParseFilePtr pfp);
```
Performs all required define substitutions on the current line being parsed, which is in the line element, using the local and global defines. No substitutions are performed if the line starts with one of the following strings: “define”, “undefine”, “ifdefine”, or “ifundefine”. Returns 2 if replacement caused the line to overflow. Otherwise, returns 0 to indicate success.

If this function makes replacements, then it calls itself recursively to see if there are more replacements to be made. It does not recurse more than 10 times.

```
char* Parse_fgets(char *str, int num,FILE *stream);
```
This is essentially identical to the standard library fgets function. It reads a line from the specified `stream` and stores it in `str`. This string should be allocated to `num` characters. Reading terminates when `num-1` characters are read, the newline character is read or the end of file is reached, whichever comes first. This also stops when it finds a ‘\r’ character, which is the return character. This is the line termination character for Macs and is also in Windows.

Functions for normal library use

```
int Parse_CmdLineArg(int *argcptr,char **argv,ParseFilePtr pfp);
```
This function processes arguments that the user enters on the command line. It can run in either of two modes. In the first, `argcptr` and `argv` contain command line arguments and `pfp` is `NULL`; in this case, the arguments are copied and stored internally. In the second mode, `argcptr` and `argv` are `NULL` while `pfp` is defined; in this case, the function copies any stored arguments into the `pfp` list of definitions. This function can also work in a combination of the two modes where none of the inputs are `NULL`. In this case, arguments are registered directly in the `pfp` without using internal storage. Finally, if all inputs are `NULL`, this function simply returns 0.

For the first mode, the number of arguments should be pointed to by `argcptr` and the arguments should be in `argv`. At present, this function only recognizes an argument that equals “--define” and that is then followed by an arguments that has the format “*key=replacement*”, where key is the define key and replacement is the replacement text. If needed, the replacement text can be put in double quotes to include spaces within it. This function removes parsed arguments from the input and leaves others in `argv` (and it updates the contents of `argcptr`) for processing elsewhere.

The function returns 0 for success, 1 for inability to allocate memory (either for internal use, or for the definitions within the `pfp`), or 2 for inability to parse the argument list.

```
ParseFilePtr Parse_Start(char *fileroot,char *filename,char *erstr);
```
Call this to start reading the top level configuration file with the file path in `fileroot` and the file name in `filename`. This takes care of structure allocation, setting up, and error checking. Returns the `pfp` for success and `NULL` for failure. If there is a failure, an error message is copied into `erstr`. Possible failures are memory allocation failure or file not found.

```
int Parse_ReadLine(ParseFilePtr *pfpptr,char *word,char **line2ptr,char
*erstr);
```
Reads one line of the current configuration file, preprocesses it slightly, and takes care of intial parsing things. These include skipping blank lines and dealing with comments, define statements, and "read_file" and "end_file" statements. Send in `pfpptr` pointing to the current pfp, `word` as an allocated string, `line2ptr` as a pointer to a pointer to a character, and `erstr` as an allocated string. Only the contents of the first parameter are used; the other parameters are for returned information. Returns 0 if there is nothing more to do with the line, 1 if the word needs processing, 2 if the configuration file ended, 3 for illegal inputs or a problem with the entry. If 1 is returned, which is the most common situation, then the first word of the line is copied into `word`, while `*line2ptr` is set to point to the remainder of the line, after any whitespace. If 3 is returned, the error string describes the problem.

```
int Parse_ReadFailure(ParseFilePtr pfp,char *erstr);
```
Call this if an error arises during file parsing, along with an optional error message in `erstr`. `erstr` needs to be allocated to `STRCHAR`. This closes all open configuration files, frees the list of pfps, and concatenates the provided error message with some more details about the erroneous line and the file. If `pfp` is entered as `NULL`, this returns 0; otherwise, this returns the line number where the error arose.