config-parser

Generated by Doxygen 1.13.2

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 configEntry Struct Reference

```
#include <config.h>
```

Public Attributes

- char ∗ sectionName
- char ∗ keyName
- char ∗ keyValue

3.1.1 Detailed Description

Definition at line 30 of file config.h.

3.1.2 Member Data Documentation

3.1.2.1 keyName

```
char∗ configEntry::keyName
```

Definition at line 33 of file config.h.

3.1.2.2 keyValue

```
char∗ configEntry::keyValue
```

Definition at line 34 of file config.h.

3.1.2.3 sectionName

```
char∗ configEntry::sectionName
```

Definition at line 32 of file config.h.

The documentation for this struct was generated from the following file:

- config.h

Chapter 4

File Documentation

4.1 config.c File Reference

```
#include "config.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
```

Functions

- int checkSection (char ∗str, char delimiterLeft, char delimiterRight, char ∗∗sectionName)

  Here we check if the given string contains a section for example [SECTIONName] here delimiterLeft is [and delimiterRight is] if a section is found, the section name will be written to sectionName.
- int getStrAtPos (char ∗str, int fromPos, int toPos, char ∗∗name, int sizeName)

  Here we get / cut from fromPos to toPos and write it to an address.
- int getNameValuePair (char ∗str, char leftDelimiterPos, char rightDelimiterPos, char ∗∗name, char ∗∗value, int sizeName, int sizeValue)

  Input:
- int parseConfig (char ∗originalBuffer, struct configEntry ∗∗entry, int configSizeCount, int ∗returnedCount)

4.1.1 Function Documentation

4.1.1.1 checkSection()

```
int checkSection (
            char * str,
            char delimiterLeft,
            char delimiterRight,
            char ** sectionName)
```

Here we check if the given string contains a section for example [SECTIONName] here delimiterLeft is [and delimiterRight is] if a section is found, the section name will be written to sectionName.

Input:

Parameters

char	∗∗str: the string to check
char	delimiterLeft: the left delimiter to check for
char	delimiterRight: the right delimiter to check for

Output:

Parameters

char	∗∗sectionName: if found the section Name of the section

Return:

Parameters

int	status_code: one of the following values:

Definition at line 24 of file config.c.

4.1.1.2 getNameValuePair()

```
int getNameValuePair (
            char * str,
            char leftDelimiterPos,
            char rightDelimiterPos,
            char ** name,
            char ** value,
            int sizeName,
            int sizeValue)
```

Input:

Here we get a "pair" of data, parsed from ∗str, witch consists of a keyname and a keyvalue. These are written to the pointers at char ∗∗name and char ∗∗value. These pair can be in the form of:
ex1:
NAME=VALUE
____^_____^
____|_____Here we have no delimiter this means rightDelimiterPos must be NULL
____This is the left delimiter
ex2:
name(value)
____^_____^
____|_____This is the rightDelimiterPos and must be ')'
____This is leftDelimiterPos which must be '('

Parameters

char	∗str: the line in the form of a string where a key value pair is stored
char	leftDelimiterPos: the left delimiter for example '='
char	rightDelimiterPos: the right delimiter can be NULL in most cases, if NULL we assume that there is only one delimiter, which is in this case the leftDelimiterPos

Output:

Parameters

char	∗∗name: The address where we store the name of the Key
char	∗∗value: The address where we store the key value
int	sizeName: for size checking against memory allocated at ∗∗name
int	sizeValue: for size checking against memory allocated at ∗∗value

Return:

Returns

will return NO_ERROR (0) if successfull. If not it will return the error of the subroutine

Definition at line 151 of file config.c.

4.1.1.3 getStrAtPos()

```
int getStrAtPos (
            char * str,
            int fromPos,
            int toPos,
            char ** name,
            int sizeName)
```

Here we get / cut from fromPos to toPos and write it to an address.

Input:

Parameters

char	∗str: the string to cut
int	fromPos: from which position we want to copy, the pos is included and zero indexed
int	toPos: to which position we want to copy, the pos is included and zero indexed

Output:

Parameters

char	∗∗name: here we will write the section name to

Input:

Parameters

int	sizeName: the size of the user allocated buffer at ∗∗name

Definition at line 98 of file config.c.

4.1.1.4 parseConfig()

```
int parseConfig (
            char * originalBuffer,
            struct configEntry ** entry,
            int configSizeCount,
            int * returnedCount)
```

Given a null terminated buffer of content(ex. from a file) an parse it with a default syntax of: Sections -> [SECTIONNAME] name / value -> name=value

Input: char ∗originalBuffer: the buffer which holds the string for parsing. struct configEntry ∗∗entry: a user allocated buffer to which we write the parsed config data. int configSizeCount: the user allocated size of ∗∗entry. Output: int ∗returnedCount: to this variable the function writes the count of struct, which will be required to store all alavaible data.

Note: The funtion returns the maximum allowed data structures, determined by configSizeCount. All data which exceeds will be dropped. To get the whole data we run the funtion a second time with the realloced configSizeCount in the size of [returnedCount] structures.

Definition at line 215 of file config.c.

4.2 config.c

Go to the documentation of this file.

```
#include"config.h"
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>

int checkSection(char *str,char delimiterLeft,char delimiterRight,char **sectionName)
{
    char section[MAX_LEN_SECTIONNAME];
    int i = 0;                  //most outer loop -> character of string
    if(str == NULL)
    {
        return ERROR_STR;
    }
    int len = strlen(str);
    int state = ST_INIT;
    int leftDelimiterPos=0;
    int rightDelimiterPos=0;
    char *sectionName2 = NULL;
    while(state != ST_FINISH)
    {
        switch(state)
        {
            case ST_INIT:
                for(i=0;i<=len;i++)         //find first (left) delimiter
                {
                    if(str[i] == delimiterLeft)
                    {
                        state = ST_FOUND_LEFT_DELIMITER;
                        leftDelimiterPos = i;
                        break;
                    }
                }
                if(state != ST_FOUND_LEFT_DELIMITER)
                {
                    return ST_ERROR_NOT_FOUND_LEFT_DELIMITER;
                }
                break;
            case ST_FOUND_LEFT_DELIMITER:
                for(i=0;i<=len;i++)         //find second (right) delimiter
                {
                    if(str[i] == delimiterRight)
                    {
                        state = ST_FOUND_RIGHT_DELIMITER;
                        rightDelimiterPos = i;
```

```c
                        break;
                    }
                }
                if(state != ST_FOUND_RIGHT_DELIMITER)
                {
                    return ST_ERROR_NOT_FOUND_RIGHT_DELIMITER;
                }
                break;
            case ST_FOUND_RIGHT_DELIMITER:
                int ret=0;
                if((ret=getStrAtPos(str,leftDelimiterPos,rightDelimiterPos,sectionName,MAX_LEN_SECTIONNAME)) == NO_ERROR)
                {
                    state = ST_FINISH;
                }else{
                    return ret;
                }
            case ST_FINISH:
                return FOUND_SECTION;
        }
    }
    return NO_SECTION;
}

int getStrAtPos(char *str,int fromPos,int toPos,char **name,int sizeName)
{
    if(*name == NULL)
    {
        int error = errno;
        printf("Pointer NULL:%d\n",error);
        return error;
    }
    int i=fromPos;      //character iterator, which starts from specified pos
    int j=0;            //the character iterator for the target string
    int diffLen=toPos-fromPos;
    char *ptr_name=*name;
    if(diffLen > MAX_LEN_SECTIONNAME || diffLen>sizeName)
    {
        return ERROR_MAX_LEN;
    }
    for(i=fromPos,j=0;i<=toPos;i++,j++)
    {
        ptr_name[j]  = str[i];
    }
    ptr_name[j+1]='\0';
    return NO_ERROR;
}

int getNameValuePair(char *str,char leftDelimiterPos,char rightDelimiterPos,char **name,char **value,int sizeName,int sizeValue)
{
    if(*name == NULL || *value == NULL)
    {
        int error = errno;
        printf("Pointer NULL:%d\n",error);
        return error;
    }
    int state=ST_INIT;
    char *ptr_name=*name;
    char *ptr_value=*value;
    int ret=0;

    char *ptrDelimiter=NULL;
    int posDelimiter=0;
    int posEnd=0;

    ptrDelimiter=strchr(str,leftDelimiterPos);
    if(ptrDelimiter==NULL)
    {
        return ERROR_DELIMITER_NOT_FOUND;
    }
    posDelimiter = (ptrDelimiter - str);
    printf("LenUntilDelimiter: %d\n",posDelimiter);
    if((ret=getStrAtPos(str,0,posDelimiter-1,&ptr_name,sizeName)) == NO_ERROR)
    {
       printf("ptr_name:%s\n",ptr_name);
    }else {
        printf("Error at getStrAtPos:%d\n",ret);
        return ret;
    }
    if(rightDelimiterPos == 0)
    {
        posEnd = strlen(str);
    }else{
        posEnd = rightDelimiterPos;
    }

    if((ret=getStrAtPos(str,posDelimiter+1,posEnd,&ptr_value,sizeValue)) == NO_ERROR)
    {
       printf("ptr_name:%s\n",ptr_value);
    }else {
        return ret;
    }

    return NO_ERROR;
}

int parseConfig(char *originalBuffer,struct configEntry **entry,int configSizeCount,int *returnedCount)
{
    int state=ST_INIT;
    int ret=0;
    int i=0;
    *returnedCount=0;
    char *sectionName=NULL;
    struct configEntry *ptr_entry = *entry;
    char *keyName=NULL;
    char *keyValue=NULL;
    char *buffer=NULL;

    buffer = malloc(strlen(originalBuffer)+1);
    if(buffer == NULL)
    {
        int error = errno;
        printf("ERROR MALLOC:%d",error);
        return ERR_PARSECONFIG_UNKNOWN;
    }
    memset(buffer,0,strlen(originalBuffer)+1);
    strcpy(buffer,originalBuffer);

    //this step is necessary, because the strok function modifys the originalBuffer
    //so we make a copy of it
    strcpy(buffer,originalBuffer);

    printf("buffer:%s\n--",buffer);

    sectionName = malloc(MAX_LEN_SECTIONNAME);
    memset(sectionName,0,MAX_LEN_SECTIONNAME);

    //read buffer line by line
    char *token;
    token = strtok(buffer,"\n");
    while(token != NULL && state != ST_FINISH)
    {

        switch(state)
        {
            case ST_INIT:
                if((ret=checkSection(token,'[',']',&sectionName))==FOUND_SECTION)
                {
                    state = ST_FOUND_SECTION;
                    printf("FOUND_SECTION:%s\n",sectionName);
                }
                break;
            case ST_FOUND_SECTION:
                keyName = malloc(MAX_LEN_SECTIONNAME);
                keyValue = malloc(MAX_LEN_SECTIONNAME);
                if(keyName == NULL || keyValue == NULL)
                {
                    int error = errno;
                    printf("MALLOC:%d\n",error);
                    return error;
                }
                memset(keyName,0,MAX_LEN_SECTIONNAME);
                memset(keyValue,0,MAX_LEN_SECTIONNAME);

                ret=getNameValuePair(token,'=',0,&keyName,&keyValue,MAX_LEN_SECTIONNAME,MAX_LEN_SECTIONNAME);
                if(ret==NO_ERROR)
                {
                    printf("configSizeCount: %d, i:%d\n",configSizeCount,i);
                    if(i<configSizeCount)
                    {
                        ptr_entry[i].sectionName = strdup(sectionName);
                        ptr_entry[i].keyValue= strdup(keyValue);
                        ptr_entry[i].keyName = strdup(keyName);
                    }
                    i++;
                    *returnedCount = i;
                    state = ST_FOUND_SECTION;
                }
                else {
                    state = ST_SKIP_READ;
                }
                free(keyName);
                free(keyValue);
                break;
        }

        if(state != ST_SKIP_READ)
        {
            token = strtok(NULL,"\n");
        }
        else {
            state = ST_INIT;
        }
    }
    printf("token: %s\n",token);
    printf("buffer:%s\n--",buffer);

    free(sectionName);
    free(buffer);
    printf("finish exitting parsing\nSTATE:%d\n",state);
    return NO_ERROR;
}
```

4.3 config.h File Reference

Classes

- struct configEntry

Macros

- #define NO_SECTION 0
- #define FOUND_SECTION 10
- #define ERROR_STR 1001
- #define ERROR_MAX_LEN 2
- #define NO_ERROR 0
- #define ERROR_DELIMITER_NOT_FOUND 40
- #define ST_INIT 0
- #define ST_FOUND_LEFT_DELIMITER 1
- #define ST_FOUND_RIGHT_DELIMITER 2
- #define ST_ERROR_NOT_FOUND_RIGHT_DELIMITER 3
- #define ST_ERROR_NOT_FOUND_LEFT_DELIMITER 4
- #define ST_FOUND_SECTION 5
- #define ST_SKIP_READ 6
- #define ST_FINISH 20
- #define ST_ERROR_GETSTRATPOS 30
- #define MAX_LEN_SECTIONNAME 128
- #define ERR_PARSECONFIG_UNKNOWN 201

Functions

- int loadConfig (char ∗file, char ∗∗str_entry, char ∗∗host, int ∗intervall, int size)
- int checkSection (char ∗str, char delimiterLeft, char delimiterRight, char ∗∗sectionName)

  Here we check if the given string contains a section for example [SECTIONName] here delimiterLeft is [and delimiterRight is] if a section is found, the section name will be written to sectionName.
- int getStrAtPos (char ∗str, int fromPos, int toPos, char ∗∗name, int sizeName)

  Here we get / cut from fromPos to toPos and write it to an address.
- int getNameValuePair (char ∗str, char leftDelimiterPos, char rightDelimiterPos, char ∗∗name, char ∗∗value, int sizeName, int sizeValue)

  Input:
- int parseConfig (char ∗buffer, struct configEntry ∗∗entry, int configSize, int ∗returnedCount)

4.3.1 Macro Definition Documentation

4.3.1.1 ERR_PARSECONFIG_UNKNOWN

```
#define ERR_PARSECONFIG_UNKNOWN 201
```

Definition at line 28 of file config.h.

4.3.1.2 ERROR_DELIMITER_NOT_FOUND

```
#define ERROR_DELIMITER_NOT_FOUND 40
```

Definition at line 7 of file config.h.

4.3.1.3 ERROR_MAX_LEN

```
#define ERROR_MAX_LEN 2
```

Definition at line 4 of file config.h.

4.3.1.4 ERROR_STR

```
#define ERROR_STR 1001
```

Definition at line 3 of file config.h.

4.3.1.5 FOUND_SECTION

```
#define FOUND_SECTION 10
```

Definition at line 2 of file config.h.

4.3.1.6 MAX_LEN_SECTIONNAME

```
#define MAX_LEN_SECTIONNAME 128
```

Definition at line 25 of file config.h.

4.3.1.7 NO_ERROR

```
#define NO_ERROR 0
```

Definition at line 5 of file config.h.

4.3.1.8 NO_SECTION

```
#define NO_SECTION 0
```

Definition at line 1 of file config.h.

4.3.1.9 ST_ERROR_GETSTRATPOS

```
#define ST_ERROR_GETSTRATPOS 30
```

Definition at line 22 of file config.h.

4.3.1.10 ST_ERROR_NOT_FOUND_LEFT_DELIMITER

```
#define ST_ERROR_NOT_FOUND_LEFT_DELIMITER 4
```

Definition at line 15 of file config.h.

4.3.1.11 ST_ERROR_NOT_FOUND_RIGHT_DELIMITER

```
#define ST_ERROR_NOT_FOUND_RIGHT_DELIMITER 3
```

Definition at line 14 of file config.h.

4.3.1.12 ST_FINISH

```
#define ST_FINISH 20
```

Definition at line 19 of file config.h.

4.3.1.13 ST_FOUND_LEFT_DELIMITER

```
#define ST_FOUND_LEFT_DELIMITER 1
```

Definition at line 12 of file config.h.

4.3.1.14 ST_FOUND_RIGHT_DELIMITER

```
#define ST_FOUND_RIGHT_DELIMITER 2
```

Definition at line 13 of file config.h.

4.3.1.15 ST_FOUND_SECTION

```
#define ST_FOUND_SECTION 5
```

Definition at line 16 of file config.h.

4.3.1.16 ST_INIT

```
#define ST_INIT 0
```

Definition at line 11 of file config.h.

4.3.1.17 ST_SKIP_READ

```
#define ST_SKIP_READ 6
```

Definition at line 17 of file config.h.

4.3.2 Function Documentation

4.3.2.1 checkSection()

```
int checkSection (
             char * str,
             char delimiterLeft,
             char delimiterRight,
             char ** sectionName)
```

Here we check if the given string contains a section for example [SECTIONName] here delimiterLeft is [and delimiterRight is] if a section is found, the section name will be written to sectionName.

Input:

Parameters

char	**str: the string to check
char	delimiterLeft: the left delimiter to check for
char	delimiterRight: the right delimiter to check for

Output:

Parameters

char	**sectionName: if found the section Name of the section

Return:

Parameters

int	status_code: one of the following values:

Definition at line 24 of file config.c.

4.3.2.2 getNameValuePair()

```
int getNameValuePair (
             char * str,
             char leftDelimiterPos,
             char rightDelimiterPos,
             char ** name,
             char ** value,
             int sizeName,
             int sizeValue)
```

Input:

Here we get a "pair" of data, parsed from ∗str, witch consists of a keyname and a keyvalue. These are written to the pointers at char ∗∗name and char ∗∗value. These pair can be in the form of:
ex1:
NAME=VALUE
____^_____^
____|_____Here we have no delimiter this means rightDelimiterPos must be NULL
____This is the left delimiter
ex2:
name(value)
____^_____^
____|_____This is the rightDelimiterPos and must be ')'
____This is leftDelimiterPos which must be '('

Parameters

char	∗str: the line in the form of a string where a key value pair is stored
char	leftDelimiterPos: the left delimiter for example '='
char	rightDelimiterPos: the right delimiter can be NULL in most cases, if NULL we assume that there is only one delimiter, which is in this case the leftDelimiterPos

Output:

Parameters

char	∗∗name: The address where we store the name of the Key
char	∗∗value: The address where we store the key value
int	sizeName: for size checking against memory allocated at ∗∗name
int	sizeValue: for size checking against memory allocated at ∗∗value

Return:

Returns

will return NO_ERROR (0) if successfull. If not it will return the error of the subroutine

Definition at line 151 of file config.c.

4.3.2.3 getStrAtPos()

```
int getStrAtPos (
             char * str,
             int fromPos,
             int toPos,
             char ** name,
             int sizeName)
```

Here we get / cut from fromPos to toPos and write it to an address.

Input:

Parameters

char	∗str: the string to cut
int	fromPos: from which position we want to copy, the pos is included and zero indexed
int	toPos: to which position we want to copy, the pos is included and zero indexed

Output:

Parameters

char	∗∗name: here we will write the section name to

Input:

Parameters

int	sizeName: the size of the user allocated buffer at ∗∗name

Definition at line 98 of file config.c.

4.3.2.4 loadConfig()

```
int loadConfig (
             char * file,
             char ** str_entry,
             char ** host,
             int * intervall,
             int size)
```

4.3.2.5 parseConfig()

```
int parseConfig (
              char * originalBuffer,
              struct configEntry ** entry,
              int configSizeCount,
              int * returnedCount)
```

Given a null terminated buffer of content(ex. from a file) an parse it with a default syntax of: Sections -> [SECTIONNAME] name / value -> name=value

Input: char ∗originalBuffer: the buffer which holds the string for parsing. struct configEntry ∗∗entry: a user allocated buffer to which we write the parsed config data. int configSizeCount: the user allocated size of ∗∗entry. Output: int ∗returnedCount: to this variable the function writes the count of struct, which will be required to store all alavaible data.

Note: The funtion returns the maximum allowed data structures, determined by configSizeCount. All data which exceeds will be dropped. To get the whole data we run the funtion a second time with the realloced configSizeCount in the size of [returnedCount] structures.

Definition at line 215 of file config.c.

4.4 config.h

Go to the documentation of this file.

```
#define NO_SECTION 0
#define FOUND_SECTION 10
#define ERROR_STR 1001
#define ERROR_MAX_LEN 2
#define NO_ERROR 0
//ERROR DELIMITER
#define ERROR_DELIMITER_NOT_FOUND 40

//State Machine

#define ST_INIT 0
#define ST_FOUND_LEFT_DELIMITER 1
#define ST_FOUND_RIGHT_DELIMITER 2
#define ST_ERROR_NOT_FOUND_RIGHT_DELIMITER 3
#define ST_ERROR_NOT_FOUND_LEFT_DELIMITER 4
#define ST_FOUND_SECTION 5
#define ST_SKIP_READ 6

#define ST_FINISH 20

//state machine ERROR
#define ST_ERROR_GETSTRATPOS 30

//LIMITS
#define MAX_LEN_SECTIONNAME 128

//error parseConfig
#define ERR_PARSECONFIG_UNKNOWN 201

struct configEntry
{
    char *sectionName;
    char *keyName;
    char *keyValue;
};


int loadConfig(char *file, char **str_entry,char **host,int *intervall,int size);
int checkSection(char *str,char delimiterLeft,char delimiterRight,char **sectionName);

int getStrAtPos(char *str,int fromPos,int toPos, char **name,int sizeName);
int getNameValuePair(char *str,char leftDelimiterPos,char rightDelimiterPos,char **name,char
      **value,int sizeName,int sizeValue);

int parseConfig(char *buffer,struct configEntry **entry,int configSize,int *returnedCount);
```

4.5 file.c File Reference

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

Functions

- int getFile (char ∗fname, char ∗∗strContent, int cbSize, long ∗neededSize)

4.5.1 Function Documentation

4.5.1.1 getFile()

```
int getFile (
            char * fname,
            char ** strContent,
            int cbSize,
            long * neededSize)
```

Definition at line 17 of file file.c.

4.6 file.c

Go to the documentation of this file.

```
00001 #include "file.h"
00002 #include <stdio.h>
00003 /*
00004  * Here we load a file and write the file as string to a memory address:
00005  * Input:
00006  *      char* fname: the name of the file to open
00007  *      int cbSize: the size of the strContent
00008  * Output:
00009  *      char **strContent: a pointer to a address where you have to allocate enougth memory to store
      the whole file
00010  *      int neededSize: this returns the needed size of the buffer strContent
00011  *
00012  *  Note:
00013  *      to get the size define strContent as NULL, then the needed size will return the needed
      buffersize
00014  *
00015  *
00016  */
00017 int getFile(char *fname, char **strContent,int cbSize,long *neededSize)
00018 {
00019     FILE *hfile = fopen(fname,"r");
00020     long file_size=0;
00021     int ret=0;
00022     if(hfile == NULL)
00023     {
00024         return FILE_ERROR_OPEN;
00025     }
00026 
00027     printf("file openend:%s",fname);
00028    fseek(hfile,0,SEEK_END);
00029    file_size = ftell(hfile);
00030     if(strContent == NULL) //we must determine the size and return to neededSize
00031     {
00032         *neededSize = file_size+1;
00033         fclose(hfile);
00034         return NO_ERROR;
00035     }
00036 
00037     if(cbSize < file_size)
```

```
    {
        *neededSize = file_size+1; //null terminator
        fclose(hfile);
        return FILE_ERROR_STRCONTENT_TO_SMALL;
    }

    fseek(hfile,0,SEEK_SET);
    ret=fread(*strContent,file_size,1,hfile);
    if(ret != 1)
    {
        printf("Reading error read and should read missmatch\nnumber written elements:%d\n",
                ret);
        fclose(hfile);
        return FILE_ERROR_READ_MISMATCH;
    }
    //buffer end-1 = \0
    //buffer end-2 = \n
    //buffer end-3 = last character
    printf("strcontent: %ld, [%c]\n",*neededSize,*(*strContent+ *neededSize-3));
    *(*strContent + *neededSize -1) ='\0';
    printf("after zero assign: %ld, %s\n",*neededSize,(*strContent+ *neededSize-1));
    //printf("content:%s",*strContent);
    fclose(hfile);
    return NO_ERROR;
}


```

4.7 file.h File Reference

Macros

- #define NO_ERROR 0
- #define FILE_ERROR_OPEN 11
- #define FILE_ERROR_STRCONTENT_TO_SMALL 12
- #define FILE_ERROR_READ_MISMATCH 13

Functions

- int getFile (char *fname, char **strContent, int cbSize, long *neededSize)

4.7.1 Macro Definition Documentation

4.7.1.1 FILE_ERROR_OPEN

```
#define FILE_ERROR_OPEN 11
```

Definition at line 4 of file file.h.

4.7.1.2 FILE_ERROR_READ_MISMATCH

```
#define FILE_ERROR_READ_MISMATCH 13
```

Definition at line 6 of file file.h.

4.7.1.3 FILE_ERROR_STRCONTENT_TO_SMALL

```
#define FILE_ERROR_STRCONTENT_TO_SMALL 12
```

Definition at line 5 of file file.h.

4.7.1.4 NO_ERROR

```
#define NO_ERROR 0
```

Definition at line 1 of file file.h.

4.7.2 Function Documentation

4.7.2.1 getFile()

```
int getFile (
            char * fname,
            char ** strContent,
            int cbSize,
            long * neededSize)
```

Definition at line 17 of file file.c.

4.8 file.h

Go to the documentation of this file.

```
#define NO_ERROR 0

//ERRORS
#define FILE_ERROR_OPEN 11
#define FILE_ERROR_STRCONTENT_TO_SMALL 12
#define FILE_ERROR_READ_MISMATCH 13

int getFile(char *fname, char **strContent,int cbSize,long *neededSize);
```

4.9 test.c File Reference

```
#include "config.h"
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "file.h"
```

Functions

- int main (void)

4.9.1 Function Documentation

4.9.1.1 main()

```
int main (
            void )
```

Definition at line 8 of file test.c.

4.10 test.c

Go to the documentation of this file.

```
00001 #include "config.h"
00002 #include <errno.h>
00003 #include <stdio.h>
00004 #include <stdlib.h>
00005 #include <string.h>
00006 #include "file.h"
00007
00008 int main(void)
00009 {
00010    char teststr[] =
      "sdafsdfsdf[12345asdfdddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
00011    char *name = NULL;
00012    char *sectionName=NULL;
00013    int ret=0;
00014    name = malloc(MAX_LEN_SECTIONNAME*sizeof(char));
00015    getStrAtPos(teststr, 11, 15, &name, MAX_LEN_SECTIONNAME);
00016
00017    sectionName = malloc(MAX_LEN_SECTIONNAME);
00018    memset(sectionName,0,MAX_LEN_SECTIONNAME);
00019    ret=checkSection(teststr,'[',']', &sectionName);
00020    if(ret == FOUND_SECTION)
00021    {
00022        printf("checkSection sucessfull\nsectionName=%s\n",sectionName);
00023    }
00024    else
00025    {
00026        free(sectionName);
00027        printf("an error occured:%d\n",ret);
00028        return 1;
00029    }
00030
00031    char testpair[] = "asifdsfo=s1254124";
00032    char *keyName=NULL;
00033    char *keyValue=NULL;
00034    char *content=NULL;
00035
00036    keyName = malloc(MAX_LEN_SECTIONNAME);
00037    keyValue = malloc(MAX_LEN_SECTIONNAME);
00038    memset(keyName,0,MAX_LEN_SECTIONNAME);
00039    memset(keyValue,0,MAX_LEN_SECTIONNAME);
00040
00041    if((ret=getNameValuePair(testpair,
00042                    '=',0,
00043                    &keyName,&keyValue,
00044                    MAX_LEN_SECTIONNAME,MAX_LEN_SECTIONNAME))==NO_ERROR)
00045    {
00046         printf("keyname:%s Value: %s\n",keyName,keyValue);
00047    }else{
00048        printf("error getNameValuePair: %d\n",ret);
00049    }
00050
00051     long neededSize=0;
00052     if((ret=getFile("config-segfault.cfg",NULL,0,&neededSize))==NO_ERROR)
00053     {
00054         printf("sucessfull retrieved size:%ld\n",neededSize);
00055     }
00056     else {
00057         printf("Error on getFile:%d\n",ret);
00058         return 1;
00059     }
00060     content = malloc(neededSize);
00061     printf("allocate buffer for filecontent size:%ld\n",neededSize);
00062     if(content == NULL)
00063     {
        int error = errno;
        printf("MALLOC: %d\n",error);
        return error;
    }
    memset(content,0,neededSize);
    if((ret=getFile("config-segfault.cfg",&content,neededSize,&neededSize))==NO_ERROR)
    {
        //printf("Sucessfull read file into buffer:%s|\n---\n",content);
    }
    else {
        printf("Error on getFile:%d\n",ret);
        free(content);
        return 1;
    }
    struct configEntry *entry=NULL;
    int returnedCount=0;
    int i=0;
    int count=10;

    entry = malloc(count*sizeof(struct configEntry));
    if(entry == NULL)
    {
        int error = errno;
        printf("MALLOC: %d\n",error);
        return 0;
    }
    ret = parseConfig(content,&entry,count,&returnedCount);
    if(ret!=NO_ERROR)
    {
        printf("Error on parseConfig:%d\nReallocate from size %d, to %d\n",ret,count,returnedCount);
    }
    if(returnedCount > count)
    {
        for(i=0;i<count;i++)
        {
            free(entry[i].keyName);
            free(entry[i].keyValue);
            free(entry[i].sectionName);
        }

        printf("Error on parseConfig:%d\nReallocate from size %d, to %d\n",ret,count,returnedCount);
        if((entry=realloc(entry,returnedCount*sizeof(struct configEntry)))==NULL)
        {
            printf("error could not reallocate from size %d to %d \n",count,returnedCount);
            return 1;
        }
        ret = parseConfig(content,&entry,returnedCount,&returnedCount);
        if(ret!=NO_ERROR)
        {
            printf("Error on parseConfig:%d\n",ret);
            return ret;
        }
        printf("returnedCount:%d\n",returnedCount);
    }



    free(content);

    for(i=0;i<returnedCount;i++)
    {
        printf("i:%d/%d, struct section: %s, keyname: %s, keyvalue:
%s\n",i,returnedCount,entry[i].sectionName,entry[i].keyName,entry[i].keyValue);
        free(entry[i].keyName);
        free(entry[i].keyValue);
        free(entry[i].sectionName);
    }


   free(keyValue);
   free(keyName);
   free(name);
   free(entry);
   free(sectionName);
    return 0;
}

```

Index