FPL

"

COLLABORATORS

	TITLE : FPL	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	"	August 9, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

FPL

1.1 FPL Library Documentation

For you who'd like to know about the FPL language, installation, warranty, bug report address and other things refer to FPL programming!

For all the rest, who you want to implement fpl.library support in your program...

```
   Survey               - short overview to get an idea of what FPL is about
   Multi file concepts  - how to enable cross file function calls with FPL
   Implement guide      - step by step guide how to implement FPL library
   Coding hints         - general information
   Error exceptions     - errors that you are responsible of
   Custom functions     - functions you supply FPL to take care of things
   Using in software    - rules about using FPL in your software

[Amiga only]
   Funclib overview     - use/create shareable third party FPL functions
```

1.2 Funclib overview

From fpl.library version 7, there is 'funclib' support added. Funclibs are simply programs that add functions to a running FPL session. The funclibs should be placed in FPLLIBS: and are opened by an FPL program or through the fpl.library function 'fplOpenLib()'.

Funclibs are nothing but common executable files that are run (with a specified parameter setup) by FPL. The program adds functions to the FPL session that opened the lib just like any other program. The functions are removed again when the funlib is closed. For details in how to program such a funclib, check out the files in the funclib/ directory of the FPL distribution package.

Funclibs work much like shared libraries (in the eyes of the FPL programmer) with an open counter that increases for multiple opens, and decreases on each close until it reaches zero and then is removed. Funclibs that are opened with the library function can be opened in such a way that it isn't possible to close it (decrease the counter to zero) from within an FPL program, but must be closed by the library function 'fplCloseLib()'.

All funclibs are automatically closed when the 'fplFree()' function is called.

1.3 Using FPL in software

I erge *all* implementors of FPL to write me a short note when you decide to include this library in your software. I really would like to keep track of the amount of programs using FPL. It also gives me a possibility to inform you about new releases and such things. Please include your opinions about FPL in your "notify mail". I use your criticism as guide in which directions I should develop FPL, which features I should prioritize and which I shouldn't!

1.4 Multi files concepts

What is the multi file system?

The multi file system allows FPL program writing divided into any number of source files and calling FPL functions cross-file. A function defined and exported in one file may be called from another file's FPL program.

How do you activate it?

Multi file systems means that FPL has to know in which files which functions are found. As you know, all functions declared as 'export' will be accessible from other files. To enable such functionality to the FPL programmer, you must either let FPL cache the files that export functions, or cache (keep track of) them by yourself. This is done by using the fplInit() tag FPLTAG_CACHEALLFILES (which sets the cache default on/off) or by specifying the fplExecuteXXXXX() tags FPLTAG_CACHEFILE (which enables/diables caching of this specific program)

and FPLTAG_FILENAME (which tells FPL the name of this program). FPL programs must have names to get cached.

File caching
~~~~~~~~~~~~

The term "file caching" means that FPL will, if any global symbols were declared, remember that program. If the program was started with fplExecuteFile(), FPL will keep the entire program in memory and if it was invoked using fplExecuteScript(), FPL expects you to have the program accessible just as it is.

File flush
~~~~~~~~~~

When using cached files, you may end up with a lot of FPL programs cached, occupying a lot of memory. You can tell FPL to flush a specific file or all files not currently in use by calling fplSend() with the tag FPLSEND_FLUSHFILE. All files that FPL is in control over can then be removed from memory, and for every file that you have control over, a FPL_FLUSH_FILE argument ID will be sent to the interface function with ->argv[0] set to the name of that file. If you decide to flush such a program from memory, and removing the accessibility for FPL, it must be confirmed to FPL by using a {FPLSEND_CONFIRM, TRUE} tag to fplSend(). No confirmation means you didn't flush the file.

Auto flush/get file
~~~~~~~~~~~~~~~~~~~

Using fplExecuteFile() will set all flags to make FPL automatically load the file from disk again when it wants access a flushed file. By setting the fplExecuteXXXXX() tags FPLTAG_PROGNAME and FPLTAG_FILENAMEGET, you tell FPL that your program can be loaded with the program name as file name whenever it wants to access this specific file after a flush. Those flags can be added run time by FPLSEND_SETPROGNAME and FPLSEND_GETFILENAMEGET.

1.5 Important note

Most important of everything: USE THE INCLUDES AND THE VALUES AND STRUCTURES DEFINED IN THERE, cause if there is one thing I've learned, it is that I'm terribly good at changing things making a re-compile necessary when updating library version!

Amiga programming details:

Indexing register
~~~~~~~~~~~~~~~~~

Compilers often have the cabability to produce output code that do address all data 16-bit indexed by an address register (standard SAS/C code uses A4). When fpl.library calls the interface or intercal function specified by you, that register will contain the same as it did when you called fplInit(). In all standard, normal cases this means the register is preserved just as you want it!

Note that this is not true when dealing with the FPLTAG_INTERNAL_ALLOC and FPLTAG_INTERNAL_DEALLOC patches. If you want those functions to be able to access global data, you must re-load the indexing register. Some ways to do it:

```
  LATTICE C version 5.xx:
    Use the -y flag (which forces A4 to be loaded in every function)
    when compiling the functions called from within the library or
    specify the keyword __saveds (in front of the function declaration)
    to perform the same task. Compiling with -b0 *SHOULD* remove the
    problem even though I have had problems trying this...

  SAS/C version 6.x
    Like the above text. The compiler keywords affected are DATA=NEAR
    instead of -b1 and SAVEDS instead of -y. The -b0 is equivalent to
    DATA=FAR. Starting the functions (called from the library) with
    "geta4();" will do the same too.

  Axtec C:
    Start the functions with "geta4();".

  DICE:
    Use the keyword __geta4 before the function name where you define/
  declare it.

    Stack checks/expandings
    ~~~~~~~~~~~~~~~~~~~~~~~
  Were before version 9.5 not allowed/recommeded/working, but are now if not
required at least recommended. Since 9.5, your program has to deal with its
own stack, FPL only uses its own for internal use.

    Stack Usage
    ~~~~~~~~~~~
 fpl.library allocates and uses very little of the host process' stack (approx
200-300 bytes on the fplInit() and 100-200 bytes in other function calls). It
uses mainly an own allocated stack with expanding possibilities!
```

1.6 Survey

```
  FPL interprets a common text file.  Expressions  are evaluated,  variables
declared,  keywords  executed. (Every time FPL finds a keyword or a closing
brace, it calls an  interval function .)

  Whenever FPL finds a  function  name (and it's not one of the internal ones)
that matches one of those you have told it to recognize, it parses the
parameters as you have declared.

  If all parameters were read without problems, FPL initializes another
structure and calls a function supplied by you, with a pointer to that
structure as argument.

  You send return code to FPL and return from the function with the proper
result code and FPL continues to interpret the text file.

  LOOP!
```

1.7 Step by step

Let's take a step by step look at what you have to do to implement a real and working FPL interface. Start your favourite editor and edit your source while reading this!

1. If your programming Amiga, open the library by simply doing:
   "struct Library *FPLBase = OpenLibrary(FPLNAME, version);"

2. Call the fplInit() function with proper arguments.

3. Inform FPL about all functions you want it to accept by calling the fplAddFunction() for every function FPL should approve. Since version 10, you can also add variables with fplAddVariable()!
   See adding functions to FPL !

4. Code the interface function . This function will get called whenever FPL finds one of your predeclared functions or wants something specific. There are a few reserved messages that this function can receive and that it should answer/react to.

5. Call fplExecuteScript() or fplExecuteFile() depending on how you have your data stored.

6. FPL takes control and performs:

 - FPL executes the program as fast as possible and whenever it finds a function or variable that's not an internal, compares it with the ones added by you.
   If you hadn't specified it, FPL returns an error code.

 - If it is a function, FPL will check the argument string you specified for that particular function and read the arguments according to that.

 - The interface function is called with the pointer to the fplArgument as argument.

 - You return the function's result by using fplSend().

 - Every now and then (with very irregular intervals - after about every statement in the program) the interval function will be called, if specified!

 - If the keyword exit() isn't found and there is more program to execute, FPL continues.

7. When completed, FPL returns a zero (0) if everything went ok, or an error code. If FPL discovered an error, the interface function will be called just before returning control to you, with the ID set to FPL_GENERAL_ERROR .

1.8 Adding functions to FPL

The very soul of FPL is to add functions that has no visual difference from

the built-in functions. When such a function is used in an executed program, the ' interface function ' is called.

Adding functions is done with the fplAddFunction() function with which you specify function name, return data type, number of parameters and their data types and a few other things. See fpl.doc for closer details.

Since version 10, FPL also supports variable addings by calling fplAddVariable().

1.9 fplInit() calling

GENERAL

The very first action to do (Amiga: after you've opened "fpl.library"), or at least before you can or should use any other FPL library function, fplInit() must be called with the proper arguments.

fplInit() initializes the FPL session and tells FPL how you would like a few things to be in the soon coming FPL program executings.

FUNCTION

The major things that you control with this function is of course the ones that can't be changed later on. Among those things are: (for more details, refer to the fpl.doc file)

* The function that is to be called on each discovered function and every time FPL wants to tell you something. The function is called the ' interface function '.

* Allocation and deallocation functions patches. You can change the function FPL is using to get and leave memory to the system and the smoothest way to use that is to tell fplInit() using certain tags.

* Hash table size. The size of the symbol hash table must be static during an FPL session.

RESULT

The return code of this function is the 'handle' which you must use when you call any of the other FPL library functions!

1.10 Errors

Whenever an error occurs, the execution is halted and an error code returned. Error message is available by using the proper tag to fplInit() or fplExecuteXXXX(). The old and still working way is to use the fplGetErrorMsg() function.
Just before the FPL functions returns, the interface function is called with the FPL_GENERAL_ERROR ID.

Most of the returned error codes is returned due to programming errors in

```
the FPL code. Some is caused by your incompetent coding!

INTERNAL ERROR
  Returned if you try to flush a non existing file or if FPL requested a
  program from you, using the  FPL_REQUEST_FILE  ID
  and no proper message were returned.

ILLEGAL ANCHOR
  You called an FPL function with an illegal parameter!

FILE ERROR
  Might be caused by you. This error code is returned when FPL has
  trouble with any of the file related function on the specified file.
  Only happens when using fplExecuteFile().
  and most often if the specified file doesn't exist.
```

1.11 Your functions

```
    Interface function
    Interval function
    Memory functions
```

1.12 Memory functions

```
 To keep FPL as flexible and powerful as possible, you can supply a function
pointer to a function that takes care of allocating/freeing memory instead of
the internal FPL default functions.

 (Amiga) The free function will get the memory pointer in A1 and the size in
D0, and the malloc function will get the size in D0. Both functions will get
the userdata in A1.

 This sounds more complicated than it really is. Let's take a look on a small
example:

 You program a software product that allocates a large memory area at startup
and then you handle all allocate/deallocate within your program inside that
memory area. You open fpl.library and it would be nice if you could make that
use the same memory area and allocating functions.

 Just use the tag  FPLTAG_INTERNAL_DEALLOC  and specify the free function and
 FPLTAG_INTERNAL_ALLOC  to specify the allocate function. The functions could
look something like this:

 (NOTE: the "__asm" and "register __xx" is Amiga and SAS/C specific keywords
for receiving parameters into specified registers.)

  void __asm FPLfree(register __a1 void *pointer,
                     register __d0 long size,
                     register __a0 void *userdata)
  {
     extern int memory; /* external variable counting malloced memory */
```

```
    MyFree(pointer);   /* free the memory using our own free() */
    memory-=size;      /* decrease the memory counter */
  }

  void __asm *FPLmalloc(register __d0 long size,
                        register __a0 void *userdata)
  {
    extern int memory:  /* external variable counting malloced memory */
    MyAlloc(size);      /* allocate with your own function */
    memory+=size;       /* increase memory counter! */
  }
```

and the fplInit() call to apply this could look like:

```
  void main(void)
  {
    unsigned long tags[]={
      FPLTAG_INTERNAL_DEALLOC, (unsigned long)FPLfree,
      FPLTAG_INTERNAL_ALLOC,   (unsigned long)FPLmalloc,
      FPLTAG_DONE
    };
    void *anchor =  fplInit ( interfaceFunction , tags);

    /* and more should be added here */
  }
```

1.13 Argument

The argument points to a fplArgument structure which is telling you about the function. That structure and all the data of it, is strictly READ ONLY. Do not make any stupid moves. These are the members of the fplArgument structure:

name The name of the function. Zero terminated.

ID The number you associate with the function/variable. This has no meaning to the library. This is the number that is sent in the ID parameter in the fplAddFunction/fplAddVariable call.

ALL negative ID numbers are reserved for FPL internal function calls, queries and handlers. See the reserved exception IDs .

argv This is a pointer to a void pointer array containing all arguments specified in the FPL program to this function. The arguments were read according to the "format" member of this structure informs us.

Different argument types creates different types of data:

format letter	real data
FPL_STRARG	char pointer.
FPL_INTARG	integer.
FPL_OPTARG	char pointer OR integer

```
        FPL_ARGLIST    Like the previous one. This tells the library to
                       accept any number of that argument type. Compare to
                       the '...' of the C programming language.

      When variables are read, this is set to zero (0).

  argc  Number of members in the argv array described above.

  key   This is the same pointer you received when you called fplInit().
        This can be used to get information with eg. fplSend() and even to
        add/delete functions and variables using future functions.

  format Pointer to a string holding the format string for this function.
        Note that FPL_ARGLIST is expanded so that every letter match one of
        the FPL_xxxARG defines. When using FPL_OPTARG, use this information
        to see which kind of argument you have recieved.

  funcdata This is the same pointer as specified in fplAddFunction()'s tag
           FPLTAG_FUNCDATA or NULL if the tag wasn't used.

  ret   The kind of the expected return code. Only really interesting for
        functions declared with optional return types. If this is different
        than FPL_OPTARG, you do better to return that type, or if that type
        isn't what you wanted, return a FPL syntax error code!!!

  variable If this is a variable-read, this structure field will hold the
        current [default]~value of this variable. For strings, its a regular
        FPL-string (with the length readable through the common macro
        'FPL_STRLEN()') and for integers a regular long.
```

1.14 Interface function

Whenever this function is called, you know that the library has found one of your functions or variables in the FPL program, discovered an error or wants an answer to a question. You should respond to such a call as fast as possible.

When this function is called, all registers (d2-d7, a2-a6) will be set to the same values as when you called fplInit(). Therefore, "__saveds", "__geta4" or "geta4();" won't be necessary in all standard cases. Note that the parameter is sent in register A0.

The interface function is specified as the first parameter in the fplInit() call. The function is declared as:

```
    int __asm InterfaceFunction ( register __a0  struct fplArgument *);

   Returning data to FPL   strings, ints, chars and such
   Errors to interpreter   error code from you
   Argument structure      what can I read from where
   Example functions       how can this be done
   Reserved exception IDs  questions from the interpreter
```

1.15 Reserved exception IDs

The interface function is used not only to activate user declared functions, but to answer and react to the special reservd messages that the FPL interpreter can send it in certain conditions. FPL reqruies a two way communication and this is the way it speaks to you. You answer to FPL by calling fplSend() with proper paramters.

These messages are using negative IDs which are read in the argument structure.

- FPL_GENERAL_ERROR:
  FPL has found an error in the executed FPL program. The error code is to be found in the ->argv[0] member! Do not send back any return value, they are just ignored!

- FPL_FILE_REQUEST:
  A file containing an exported function has been flushed from memory and now FPL wants it back (someone called it). The ->argv[0] contains the filename of the program. FPL wants the answer in the FPLSEND_PROGRAM or FPLSEND_PROGFILE tag of a fplSend() call. If none of these messages is sent, the program will fail with "internal error".

- FPL_FLUSH_FILE:
  When using non-cached files which is started with fplExecuteScript() and declares global symbols, FPL will tell you by sending this that it is ok for you to flush this file from memory now if you want. If you decide to remove it, tell FPL by the {FPLSEND_CONFIRM, true} tag of fplSend(). ->argv[0] contains the filename of the program. If you don't reply with that tag, you must not do anything to the function's program area!
  This might be called after a FPLSEND_FLUSHFILE call.

- FPL_WARNING:
  Whenever the FPL interpreter finds an error that it might be able to pass, it calls the interface function with this ID set. If a fplSend(FPLSEND_CONFIRM, TRUE) is sent, the error will be ignored and the interpreter keeps on working!

  Any warning reported in this way is a proof of incorrect coding in the FPL program. The errors should be instantly fixed as soon as possible. The interpreter can not be expected to always continue as the program should have done if it was written correct. The interpreter guesses and tries one or a few ways to continue. Those ways may _not_ be the ones that the FPL programmer desired when he wrote the failing program!

  Warnings caused by an error such as "a[2[=3;" will fail in a FPL_MISSING_BRACKET at the position right after the number "2". If you ignore such a failure, FPL will try to continue running on that position which only will cause a dead end FPL_SYNTAX_ERROR instead. The FPLSEND_STEP tag is done to be used in such occasions.

  A very important detail to remember when you decide to correct the FPL program, is that any global symbols that might be cached from that file must

be cleared and re-initialized!

The return code of this function is ignored!

- FPL_UNKNOWN_FUNCTION:

This can only happen if the FPLTAG_ALLFUNCTIONS has been specified. When this ID is received, it means that FPL has interpreted a function that it didn't recognize, it is unknown. FPL have parsed the arguments as usual and they can all be read as usual. A good habit in this case (if not any other) is to check the ->format member of the argument structure, to see which kinds of arguments that are received.

->funcdata is reserved for future use in this case, and should not be used as if it contained anything.

To return a return code to FPL, use the fplSend() as usual. Notice that it can be really tricky to know which kind you should return! Is it a string or
is it an int?

If you don't send anything, FPL will try to guess which kind of value that should be returned. Default is 'int'.

The return code of this function is returned to FPL as the standard FPL error code. Return zero for success!

1.16 Returning error codes to the FPL interpreter

Each call to the interface function will have a final "return" call. The return code will be the source of progress information to the interpreter. If zero is returned, everything went ok, but if anything else than that was returned from you FPL will fail and return that error on the current interpret position! You will find proper return codes to return in the <libraries/FPL.h> file. Of course you can return any value you please, but FPL will only understand the codes from that file.

Do not mix this return code with the return procedure when returning things to FPL.

For assembler programmers: the return code is of course always return in D0.

1.17 Sending data to FPL

The function that FPL found - that you have defined as a function (by calling fplAddFunction()) - requires a return value.

When you've done with doing what you should do, you should send to FPL what you specified that this function should return.

Sending an int
~~~~~~~~~~~~~~
~~~~~~~~~~~~~~

```
        unsigned long tags[]={FPLSEND_INT, code, FPLSEND_DONE};
        fplSend(arg->key, tags);

    Sending a string
    ~~~~~~~~~~~~~~~~
        unsigned long tags[]={FPLSEND_STRING, string,
                              FPLSEND_STRLEN, length_of_string,
                              FPLSEND_DONE};
        fplSend(arg->key, tags);
```

FPL will copy the string at the function call, which makes it perfectly legal to free or re-use the string immediately after the function call. Sending a NULL pointer is legal and will result in an zero length string.

If you set FPLSEND_STRLEN to -1 (or skip the tag), FPL will make a strlen() on the string to get the length of it.

If the length of the string you want to send back to FPL is zero, it's OK to simply ignore the sending, send a NULL pointer or send a zero length string. All ways will cause the same result.

See also return codes to FPL interpreter .

1.18 Example

Let's look at a small example. We have specified a few functions.

output() function requires a string as argument (S), outputs the string and returns zero.

double() requires an integer as argument (I) and return twice the input number.

get() takes no argument (NULL) and returns a string.

printf() takes a string and an list of optional parameters (So>) and performs a regular C language printf() operation!! Returns the number of printed characters.

```
   #include <proto/FPL.h>
   long fplSendTags(void *, unsigned long, ...);

   long __asm InterfaceFunction( register __a0  struct fplArgument
*argument)
   {
     APTR string;
     switch(argument->ID) {
     case OUTPUT:                        /* handle output() */
        fplSendTags(argument->key,
                    FPLSEND_INT, printf("%s", argument->argv[0],)
                    FPLSEND_DONE);
        break;

     case PRINTF:
        fplSendTags(argument->key,
```

```
                FPLSEND_INT, vprintf(arg->argv[0], (char *)&arg->argv[1])
                FPLSEND_DONE);
      break;

   case DOUBLE:                         /* handle double() */
      fplSendTags(argument->key,
                FPLSEND_INT, (int)argument->argv[0]*2,
                FPLSEND_DONE);
      break;

   case GET:                            /* handle get() */
      string=fplAllocString(argument->key, 100); /* Allocate memory */
      strcpy(string, GetThatString());/* Get the string */

      fplSend(argument->key,
              FPLSEND_STRING, string,
              FPLSEND_STRLEN, -1, /* let FPL make a strlen() */
              FPLSEND_DONTCOPY_STRING, TRUE, /* go ahead use my string! */
              FPLSEND_DONE);
      break;

   case FPL_GENERAL_ERROR:              /* error handling */
      {
        char buffer[FPL_ERRORMSG_LENGTH];
        long col;
        char *name;
        fplSendTags(arg->key,
                FPLSEND_GETCOLUMN, &col,
                FPLSEND_GETPROGNAME, &name,
                FPLSEND_DONE);
        printf("\n>>> %s\n",
                   fplGetErrorMsg(arg->key, (long)arg->funcdata, buffer));
        printf(">>> Byte position %d in file \"%s\".\n", col, name);
      }
     break;
   }

   return 0;
}

long fplSendTags(void *anchor, unsigned long tags, ...)
{
  return(fplSend(anchor, &tags));
}
```

1.19 Interval function

This function will be called a lot of times. About once for every statement FPL interprets. It _must_ be coded in consideration of execution speed. Returning anything but zero breaks the program execution. The returned value is later readable by calling fplSend() with the FPLSEND_GETRESULT tag.

The argument to this function is received in register A0 and points to the data you could specify in the FPLTAG_USERDATA tag of fplInit(), or NULL. All registers (d2-d7, a2-a6) will have the same contents as when you called

fplInit(). Therefore, "__saveds" is hardly needed.

In this example we have a function called any_keypress() which checks if any breaking key has been pressed since last check:

```
long __asm IntervalFunction(register __a0 void *userdata)
{
  return(any_keypress()); /* break if a key was pressed! */
}
```

Breaking the execution from the interval function will cause the library to return the error code FPL_PROGRAM_STOPPED.

1.20 Index

Index of database FPL

Documents

Buttons

~FPL_GENERAL_ERROR~
~FPL_REQUEST_FILE~
~FrexxWare~
~Funclib~overview~~~~
~function~
~functions~
~Implement~guide~~~~~
~interface~function~
~interfaceFunction~
~interval~function~
~Interval~function~~
~keyword~
~keywords~
~Memory~functions~~~
~Multi~file~concepts~
~reserved~exception~IDs~
~reserved~messages~
~return~codes~to~FPL~interpreter~
~Returning~data~to~FPL~~
~struct~fplArgument~
~Survey~~~~~~~~~~~~~~
~Using~in~software~~~
~variables~
~without~warranty~
Call~the~fplInit()~function~with~proper~arguments.~