

FD2Pragma

COLLABORATORS

	<i>TITLE :</i> FD2Pragma		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		August 7, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	FD2Pragma	1
1.1	FD2Pragma - the programmers file generator	1
1.2	about	2
1.3	options	3
1.4	examples	5
1.5	Option explanation of all user options	6
1.6	Options for detailed control	7
1.7	Options for direct pragma generation	9
1.8	includes	9
1.9	linker libraries	10
1.10	proto files	10
1.11	defines used in include files	11
1.12	local library base files	12
1.13	Which way the header scan works	12
1.14	Way of tag-function handling	13
1.15	FD2Pragma.types definition file	14
1.16	Functions using FPU registers	14
1.17	Files used and created for Phase5's PowerUP boards	14
1.18	Words and phrases	15
1.19	Known bugs and problems	17
1.20	How self made libraries should be designed	18
1.21	Design description of varargs pragmas	19
1.22	Design description of FD files	21
1.23	How registers of 680x0 processor are used	22
1.24	Scripts for automatic file creation	22
1.25	Greetings, last words and authors address	24
1.26	index	25

Chapter 1

FD2Pragma

1.1 FD2Pragma - the programmers file generator

FD2Pragma - the programmers file generator

About the program

What this is program able to do

Options

What options may be used to control it

Useful example calls

Most useful calls

The important options

Main options for everyone

The advanced options

Options for more detailed control

The pragma direct options

Options for direct pragma generation

About includes

What C includes are useful and required

Some words about

Linker libraries

Generated linker library files

Proto files

Generated proto files

Include definitions

The used defines

Local library base files

C includes for local library base support

Header scan	The way the HEADER option works
Tag-functions	The way tag-functions are found
FD2Pragma.types	The FD2Pragma.types definition file
FPU usage	Functions using FPU registers
PowerUP	Phase 5 PowerUP files
Important words	Explanation of used words and phrases
Bugs and Problems	Known bugs and problems
Library design	Short words how to design own libraries
Pragma design	Design description of pragmas
FD file design	Design description of FD files
Register usage	Usage of 680x0 registers
Scripts	Useful script to generate needed files
The End - Last words	Greetings, authors address ,...

Calling the program seems to be (is) very difficult, but it offers you a large set of functions. A lot of options need a lot of abilities to turn them on/off!

Read this documentation very carefully, because there are some notes you may not see on fast reading, but which will help you a lot. (for example HEADER option and "" filename)

1.2 about

This is a utility to create:

- following pragma statements for certain C compilers: amicall, libcall, flibcall, tagcall and syscall
- proto files for C compilers
- offset (LVO) files for assembler programs
- stub functions for either tag-functions or all library functions
- stub functions as assembler text
- stub functions as useable link library file
- FD files out of pragma files
- stubs for C++ compilers (SPECIAL 11, 12, 13 and CLIB)
- the files with your own headers inserted
- files for using local pointers for shared library bases in compilers which do not support this normally
- stub functions for Pascal compilers
- inline files for GCC
- inline, pragma and stub libraries using FPU registers
- files needed to develop for Phase5's PowerUP boards
- unit texts for FPC Pascal compiler
- BMAP files for AmigaBASIC and MaxonBASIC

Therefor only the FD file telling the library informations is needed. For some SPECIAL options you may additionally supply the CLIB keyword (or you need to supply it) giving FD2Pragma the prototypes file in clib directory.

Special option 80 does the reverse to normal: convert pragma to FD!

FD2Pragma does not produce special files for VBCC compiler, but this compiler seems to have included already all necessary conversion programs.

Commodore had a special program which created FD files, pragmas, stub libs, clib includes and all the missing stuff out of a special file format. Because this format was never released to the public we need to use a program like FD2Pragma which uses FD and clib files as input. It should be able to create all needed files as well. If some are missing, please tell me and I will add them. As FD2Pragma is a lot newer than Commodore tool, it should be able to do some more stuff (inline files, Storm files, C++ names), but I cannot guarantee that, as I newer saw Commodore tool.

I personally would prefer the one-file source file format and I will include support to that format when it will ever be released. I will not create such a format on my own (as there are already enough incompatible formats :-).

1.3 options

You get the command template with FD2Pragma ? .

```
FROM=FDFILE/A, SPECIAL/N, MODE/N, TO/K, CLIB/K, HEADER/K, AMICALL/K, LIBCALL/K,
AMITAGS/K, LIBTAGS/K, COMMENT/S, EXTERNC/S, NOFPU/S, PRIVATE/S, SMALLDATA/S,
SORTED/S, STORMFD/S, USESYS CALL/S:
```

In this position you may press <?> again and you get the following text!
Be carefull, because this text is longer than one normal high resolution screen, so it is usefull to press a key in the middle of the text to stop

the output.

FDFILE: the FD file which should be used
 SPECIAL: 1 - Aztec compiler (xxx_lib.h, MODE 2, AMICALL)
 2 - DICE compiler (xxx_pragmas.h, MODE 3, LIBCALL)
 3 - SAS compiler (xxx_pragmas.h, MODE 3, LIBCALL, LIBTAGS)
 4 - MAXON compiler (xxx_lib.h, MODE 1, AMICALL)
 5 - STORM compiler (xxx_lib.h, MODE 1, AMITAGS, AMICALL)
 6 - all compilers [default]
 7 - all compilers with pragma to inline redirect for GCC
 8 - pragma to inline redirect for GCC
 10 - stub-functions for C - C text
 11 - stub-functions for C - assembler text
 12 - stub-functions for C - assembler text (SECTION)
 13 - stub-functions for C - link library
 14 - defines and link library for local library base (register call)
 15 - defines and link library for local library base (stack call)
 16 - stub-functions for Pascal - assembler text
 17 - stub-functions for Pascal - link library
 18 - BMAP file for AmigaBASIC and MaxonBASIC
 20 - assembler lvo _lvo.i file
 21 - assembler lvo _lib.i file
 22 - assembler lvo _lvo.i file no XDEF
 23 - assembler lvo _lib.i file no XDEF
 30 - proto file with pragma/..._lib.h call
 31 - proto file with pragma/..._pragmas.h call
 32 - proto file with pragmas/..._lib.h call
 33 - proto file with pragmas/..._pragmas.h call
 34 - proto file with local/..._loc.h call
 35 - proto file for all compilers
 36 - proto file for GNU-C compiler only
 40 - GCC inline file (preprocessor based)
 41 - GCC inline file (old type - inline based)
 42 - GCC inline file (library stubs)
 50 - GCC inline files for PowerUP (preprocessor based)
 51 - GCC inline files for PowerUP (old type - inline based)
 52 - GCC inline files for PowerUP (library stubs)
 53 - SAS-C include file for PowerUP
 54 - Proto file for PowerUP
 60 - FPC pascal unit text
 80 - FD file (source is a pragma file!)
 MODE: SPECIAL 1-6, AMICALL, LIBCALL, AMITAGS, LIBTAGS:
 1 - _INCLUDE_PRAGMA..._LIB_H definition method [default]
 2 - _PRAGMAS..._LIB_H definition method
 3 - _PRAGMAS..._PRAGMAS_H definition method
 4 - no definition
 SPECIAL 11-15:
 1 - all functions, normal interface
 2 - only tag-functions, tagcall interface [default]
 3 - all functions, normal and tagcall interface
 TO: the destination directory (self creation of filename) or
 the destination file
 CLIB: name of the prototypes file in clib directory
 HEADER: inserts given file into header of created file (" is scan)
 The following four need a string as argument. This string is used to set
 a #if<given string> before the set method.
 AMICALL: creates amicall pragmas

LIBCALL: creates libcall pragmas
 AMITAGS: creates tagcall pragmas (amicall like method (StormC++))
 LIBTAGS: creates tagcall pragmas (libcall like method (SAS C))
 Switches:
 COMMENT: copy comments found in FD file
 EXTERNC: add a #ifdef __cplusplus ... statement to pragma file
 FPUONLY: work only with functions using FPU register arguments
 NOFPU: disable usage of FPU register arguments
 PRIVATE: includes private declared functions
 SMALLDATA: generate small data link libraries or assembler text
 SORTED: sort generated files by name and not by bias value
 STORMFD: converts FD files of strange
 StormC++ format
 USESYSCALL: uses syscall pragma instead of libcall SysBase

1.4 examples

Useful examples (with intuition.library):

1) FD2Pragma <FD file> TO <pragma dir>

```
FD2Pragma FD:intuition_lib.h TO INCLUDE:pragma/
```

Creates a pragma file for all C compilers and copies it to the given directory.

2) FD2Pragma <FD file> CLIB <clib file> SPECIAL 13 TO <lib dir>

```
FD2Pragma FD:intuition_lib.h CLIB INCLUDE:clib/intuition_protos.h
SPECIAL 13 TO LIB:
```

Creates a link library holding stub functions to call tag-functions from compilers which do not support them (MaxonC++).

3) FD2Pragma <FD file> CLIB <clib file> SPECIAL 14 MODE 3

```
FD2Pragma FD:intuition_lib.fd CLIB INCLUDE:clib/intuition_protos.h
SPECIAL 14 MODE 3
```

Creates a link library and an include file which allow you to call library functions with local base variables in compilers which do not support that (MaxonC++). See

below
, how to handle these files.

4) FD2Pragma <FD file> SPECIAL 34 TO <proto dir>

```
FD2Pragma FD:intuition_lib.h SPECIAL 34 TO INCLUDE:proto/
```

Creates a proto file for the local library base file include, which was created in example 3 and copies it to the given directory.

5) FD2Pragma <FD file> SPECIAL 35 TO <proto dir>

```
FD2Pragma FD:intuition_lib.h SPECIAL 35 TO INCLUDE:proto/
```

Creates a proto file for all C compilers and copies it to the given directory.

3) FD2Pragma <FD file> CLIB <clib file> SPECIAL 40

```
FD2Pragma FD:intuition_lib.fd CLIB INCLUDE:clib/intuition_protos.h
SPECIAL 40
```

Creates inline include for GCC. This is used by GCC instead of pragma files for other compilers.

1.5 Option explanation of all user options

FDFILE is the always needed source file, which describes the library.

SPECIAL option:

(create a pragma file)

- 1: Creates a pragma file for the Aztec compiler, what this means you see in the brackets above.
- 2: Same as 1 for DICE compiler.
- 3: Same as 1 for SAS compiler.
- 4: Same as 1 for MAXON compiler.
- 5: Same as 1 for STORM compiler.
- 6: This option creates a pragma file useable for nearly all compilers. This is default, when no other mode is given.

NOTE: Please do always use option 6.

- 7: same as 6, but redirects file to inline directory for GCC.
- 8: no pragmas, but only redirection to inline.

(link libraries and their assembler code)

- 10: Creates stub functions in correct C code which handle the varargs feature. CLIB parameter is useful with this to get correct functions. The only problem with these files is, that there is space wasted, when not all functions are used.
 - 11: Creates STUB functions for C compilers, which are not able to call a library directly (result is ASM source code), accepts option CLIB to create additional function names for C++ compilers like MaxonC++.
 - 12: Same as 11, but including SECTION keywords to divide the files into parts.
 - 13: Same as 11, but the result as a link library, which can be used by the C compiler directly.
 - 14: Creates two files (a link library and a C include) which allows you to use local library base variables also in compilers, which do normally not support them (MaxonC++). Most time it is useful to set option MODE to 3 or 1. This options needs CLIB keyword for correct results.
 - 15: Same as 14, but parameters are passed on stack.
 - 16: Creates STUB functions for PCQ Pascal compilers. The tagcall function
-

names are ignored, as they cannot be used with Pascal. The result is readable assembler text. The code equals the one for C compilers, but the args are taken from stack in reversed order.

17: same as 16, but produces link library.

18: Creates BMAP files used by certain BASIC variants (e.g. AmigaBASIC, MaxonBASIC).

(assembler LVO files)

20: Creates lvo file for an assembler.

21: Same as 20, but other name.

22: Same as 20, but there are no XDEF statements in the resulting file.

23: Same as 22, but other name.

(proto files - no prototypes)

30,31,32,33,34: Creates proto files for the C compiler (the difference is in the name of the called file).

35: Creates proto file with calls inline file for GNU-C and pragma/xxx_lib.h for all the others.

36: Creates proto file for GNU-C. This differs from SPECIAL 35 only by define `__CONSTLIBBASEDECL__` and removed pragma call.

FD2Pragma knows the correct library base structures for some libraries. All the other libraries get 'struct Library' as default.

(inline files)

40: Creates new style GCC inline files.

41: Creates old style GCC inline files.

42: Same as 41, but no extern keyword before functions.

(Phase5 PowerUP files)

50: Creates files like 40, but for PowerUP.

51: Creates files like 41, but for PowerUP.

52: Creates files like 42, but for PowerUP.

53: Creates SAS-C include file for PowerUP (named as pragma files in PowerUP files for no logical reason).

54: Creates a proto file for PowerUP (usable for GNU-C and SAS).

(Pascal stuff)

60: This creates a unit text file for FPC Pascal compiler.

(FD file)

80: This creates a FD file! The option FDFILE has to be a pragma file here!

1.6 Options for detailed control

MODE:

- 1) given with SPECIAL 1 to 6, AMITAGS, AMICALL, LIBTAGS, LIBCALL or CSTUBS:
 - Defines, which `#ifdef ...\n#define ...` statement is used in the pragma file. Option 1 is default.
- 2) given with SPECIAL 11 to 15:
 - Defines, which functions should be created. Option 2 is default.
 - 1 - all functions are taken in normal way with normal name

- 2 - only tag-functions are taken with tagcall method and tag name
- 3 - means 1 and 2 together

TO: Here you specify either the destination directory or the destination file!

- If this argument is a directory, the internal names are used, but the file will be in the given directory.
- If this argument is a filename (not existing or already existing), then the resulting file has the here given name!

CLIB: Supply name of the prototypes file in clib directory. If this option is given together with SPECIAL 11, 12 and 13, additional functions names with C++ names are created. FD2Pragma knows all standard parameter types defined in exec/types.h and dos/dos.h, all structures and some more types. All other #typedef's bring a warning. Do not use them in prototypes files! This parameter is needed by option SPECIAL 10, 13, 14 and 40 to 42. You may define unknown types using the

```
    FD2Pragma.types
    file.
```

HEADER: This option gives you the ability to specify a file, which should be inserted after the normal headers and before the clib call of standard headers (in LVO and ASM files too). If you give "" as filename (empty string), the destination file (if already exists) will be scanned for an existing header. This is useful for updating files. See

```
    HeaderScan
    to find
```

out how a header must be built to be recognized.

COMMENT: Comments which are in the FD file are copied to the pragma or LVO file, when this option is given! When SORTED is given, this options is disabled.

EXTERNC: This options adds an #ifdef __cplusplus ... statement to the pragma file. This options is useful for C compilers runing in C++ mode, but it seems, that they do not really need this statement. Only useful with SPECIAL option 1-6, 14 and 15.

FPUONLY: This options is the opposite to NOFPU. It forces FD2Pragma to ignore all functions not using FPU registers for argument passing.

NOFPU: This disables usage of FPU arguments. Functions using FPU arguments are not converted with this option (as e.g. Maxon does not support amicall with FPU args). You get a warning for every line containing FPU arguments.

It is really useless to both specify NOFPU and FPUONLY.

PRIVATE: Also gives you the pragmas or LVO's of private functions. Normally these functions should never be used!

SMALLDATA: Normal large data model references the library base as global variable. In small data model the reference is relativ to register A4 instead. This option is useful for SPECIAL 11, 12 and 13 only.

SORTED: This option sorts generated files by name and not by bias value. This is only for visibility and does not change the use of the files.

STORMFD: This option allows to convert FD files in strange StormC++ format. It's a FD file format defining the C tag-function name directly in FD file. These files cannot be used with other FD scanners without changes.

USESYSYSCALL: Instructs FD2Pragma to use the syscall pragma instead of a libcall SysBase. This is useful only, when using a SPECIAL option with LIBCALL or by giving LIBCALL directly and converting exec_lib.fd. I think only SAS compiler supports this statement.

1.7 Options for direct pragma generation

The now following options are for not recommended to be used. They are designed to be used without any SPECIAL option, but you also can give SPECIAL and any of the following options! In this case the corresponding settings of SPECIAL are overwritten, when they are in conflict.

AMICALL: creates amicall pragmas

```
--> #pragma amicall(IntuitionBase,0x294,SetGadgetAttrsA(a0,a1,a2,a3))
```

LIBCALL: creates libcall pragmas

```
--> #pragma libcall IntuitionBase SetGadgetAttrsA 294 BA9804
```

AMITAGS: creates tagcall pragmas (amicall like method (StormC++))

```
--> #pragma tagcall(IntuitionBase,0x294,SetGadgetAttrs(a0,a1,a2,a3))
```

LIBTAGS: creates tagcall pragmas (libcall like method (SAS C))

```
--> #pragma tagcall IntuitionBase SetGadgetAttrs 294 BA9804
```

These four functions need a string as argument. It is used to set a

```
#if<given string>
```

before the data of that option. So it is possible to create a file like this:

```
--> FD2Pragma FDFILE xxx AMICALL " defined(__MAXON__) || defined(AZTEC_C) "
LIBCALL "def __SASC"
```

```
#if defined(__MAXON__) || defined(AZTEC_C)
/* do amicalls */
#endif
#ifdef __SASC
/* do libcalls */
#endif
```

If you give "" as string, then no '#if<text>' statement will be added.

As you see, the text is added without space after '#if'. This gives you the ability to use also other '#if' clauses, than '#ifdef' (e.g. #ifndef). If needed, you have to add the space in the parameter text (" defined...").

1.8 includes

Useful include system for C compilers:

After programming a long time I arranged my includes in a way, that all my C compilers are able to use the system includes in one directory.

I copied all Amiga system includes to one directory and added some files, which were created with FD2Pragma. The system includes you get for example on Amiga Developer CD.

- New directory 'pragma' contains xxx_lib.h pragma files for every library. These files were created with SPECIAL option 6.
- New directory 'proto' contains xxx.h proto files which were created with SPECIAL option 35.
- New directory 'inline' contains xxx.h inline files for GCC. These files were created with SPECIAL option 40.

Directories like 'pragmas' were deleted, when existing. Remain should only 'clib', 'pragma', 'proto' and library specific directories (like 'dos', 'exec', 'libraries' and 'utility').

All the others (ANSI-C stuff, compiler specials) were copied to another directory. In S:User-StartUp I use 'Assign ADD' to join the two directories, so that the compiler may access both.

1.9 linker libraries

About created link libraries (SPECIAL Option 13-15):

The created link libraries are relatively big compared to other link libraries. The size of the link library has nothing to do with the size of the resulting program you create. The code part of my link libraries is relatively short, but I define a lot of texts (which are NOT copied to the created executable program). These texts are for easier identification and every function also gets different names:

- 1) the normal asm name: <name> (e.g. CopyMem)
- 2) the normal C name: _<name> (e.g. _CopyMem)
- 3) the normal C++ name: <name>_<params> (e.g. CopyMem_PvPvUj)
- 4) when a function parameter is STRPTR, a second C++ name is created

Forms 3 and 4 occur only, when you use CLIB keyword. With SPECIAL options 14 and 15 the number of strings is twice as much. The different names give a lot more flexibility and only make the link library bigger. These names are only visible to the linker program. The resulting executable most time is a lot smaller than the link library!

I think the code part of the link libraries is optimized totally. I do not know any possible improvement to make it shorter.

1.10 proto files

FD2Pragma is able to generate different proto files, but I suggest using

only the file generated with SPECIAL option 35.

For system libraries and some others the correct base structure is used. Other unknown basenames get "struct Library *" as default. You may change that in the created proto files, when another structure is correct.

The proto files support following define:

```
__NOLIBBASE__
```

When this is set before calling the proto file, the declaration of the global library base is skipped, so that can be done in source-code. This define is also used for GCC.

The file created with SPECIAL 36 supports an additionally define:

```
__CONSTLIBBASEDECL__
```

When this is set to "const" the library base is handled as const. This may shorten the produced code, but you need a sourcefile without this define to initialize the base variable.

1.11 defines used in include files

The first #ifdef/#define statements of created C includes:

FD2Pragma has a set of different define names for different include files. These names are internally to allow double-inclusion of one include files without getting errors. Standard system includes use the same system.

The normal names are: (example intuition.library)

```
proto files:      _PROTO_INTUITION_H
local library base files:  _INCLUDE_PROTO_INTUITION_LOC_H
standard pragma files:  _INCLUDE_PRAGMA_INTUITION_LIB_H
C stubs files:    _INCLUDE_INTUITION_CSTUB_H
inline files for GCC  _INLINE_INTUITION_H
```

Non-FD2Pragma names are:

```
clib files      CLIB_INTUITION_PROTOS_H
other includes (path_name_extension) INTUITION_INTUITION_H
```

These names never should be used in other files or sources! This rule is broken for some standard system includes, but is generally true. Compiling may be some seconds faster, when you check these names before #include line, but in this case the names must be standard and they are not!

Some defines allow the user to change the behaviour of the includes:

```
__NOLIBBASE__
```

This is used in proto files. See
Proto files
for more information.

```
NO_INLINE_STDARG
```

For GCC inline files a lot of defines exists, but this seems to be the most

important. It disables the creation of varargs/tagcall functions. For other inline defines check the created inline files.

<library>_BASE_NAME

Function definitions in the inline files refer to the library base variable through the <library>_BASE_NAME symbol (e.g. INTUITION_BASE_NAME). At the top of the inline file, this symbol is redefined to the appropriate library base variable name (e.g., IntuitionBase), unless it has been already defined. This way, you can make the inlines use a field of a structure as a library base, for example.

1.12 local library base files

When using SPECIAL options 14 and 15 you get two files called libname_loc.h and libname_loc.lib. The second one is a link library and should be passed to the compiler with program settings or in makefile. The first one is a C header file and should be used as a replacement for files in clib, pragma, proto and pragmas directories. Use always the libname_loc.h file instead of these files and not together with them! Do not mix them. I suggest copying the header file into a directory called "local".

This file holds prototypes equal to the prototypes in directory clib, but with struct Library * as first parameter and the name prefix LOC_. Together with the prototypes there are some defines redefining the function name to the old one and passing the library base as first parameter. These defines allow you to use the local library bases as normal as global bases. For tag-functions and some exceptions these defines do not work and you have to call the LOC_ function directly and pass the library base as first parameter.

Use need the CLIB keyword together with SPECIAL option 14 and 15.

1.13 Which way the header scan works

Giving the HEADER option lets FD2Pragma insert the file (you have to give filename with HEADER option) at start of LVO/Pragma/Proto/stub file. When you pass "" as filename, FD2Pragma scans the destination file (if already existing) for a header and copies this header to the new file.

How is scanned:

FD2Pragma scans for a block of comment lines. So when a line starting with '/*', ';' or '//' is found, this line is the first header line. The header ends before the first line not starting this way. Additionally, when FD2Pragma finds first a line starting with '/*' it scans until a line holds '*/'. This then is the last line of header. Same is done for Pascal comments using '(*' and '*)' or '{' and '}'.

C and ASM files are scanned same way, so sometimes FD2Pragma may get a wrong header.

1.14 Way of tag-function handling

The tag-functions are supported by certain comments. Note, that the official includes from the Native Developer Update Kit do not have these comments included. Lets look at an excerpt from the FD file `muimaster_lib.fd`:

```
MUI_NewObjectA(class, tags) (a0, a1)
*tagcall
MUI_DisposeObject (obj) (a0)
MUI_RequestA (app, win, flags, title, gadgets, format, params) (d0, d1, d2, a0, a1, a2, a3)
*tagcall
MUI_AllocAslRequest (type, tags) (d0, a0)
*tagcalltags
```

The comments tell us, that `MUI_NewObjectA`, `MUI_RequestA` and `MUI_AllocAslRequest` should have stub routines. The respective names are `MUI_NewObject`, `MUI_Request` (as the comment has just the word `tagcall`) and `MUI_AllocAslRequestTags` (as the comment has the word `tags` included).

Another possibility would be to write something like

```
SystemTagList (command, tags) (d1/d2)
*tagcall-TagList+Tags
```

This would create a stub routine or `tagcall` pragma `SystemTags` (dropping the word `TagList`, adding the word `Tags`).

FD2Pragma is also able to create the names automatically. Most times this should be enough, so you do not have to use the above mentioned method. In case you really use the above method, I suggest using always the one with '+' and '-' signs!

Tag-functions in standard Amiga includes differ a bit in there naming conventions, so it is easy to find them:

normal function name	tag-function name
<code>xxxA</code>	<code>xxx</code>
<code>xxxTagList</code>	<code>xxxTags</code>
<code>xxxArgs</code>	<code>xxx</code>

Also the arguments given in the FD file may define a function as tag-function. If the last argument equals one of the words `"tags"`, `"taglist"` or `"args"`, then the function has a tag-function named `xxxTags` or `xxxArgs`.

There are some exceptions for this rules (some `dos.library` and `utility.library` functions) which are handled automatically.

Sometimes FD2Pragma detects functions as tag-functions, which are not. This is a problem of the detection, which is based on function name. Not all programmers follow the unofficial guidelines and thus FD2Pragma produces wrong stuff. If you detect such functions inform me and I will remove that behaviour. In any case you may add the line

```
*notagcall
```

as next line in FD file and the detection is turned of for that function.

1.15 FD2Pragma.types definition file

This file allows you to define unknown types, so that correct C++ names can be built and also created inline files are valid. The file must be in current directory or in program directory.

When FD2Pragma does not recognize a type you get a warning, telling you line number and argument number. Argument number 0 means return value of the function. Check the type and add it to the list and FD2Pragma will know it afterwards.

Lines starting with * are seen as comment and are ignored.
The description format is one 'unknown type : known type' in every line.

Unknown type: The type FD2Pragma does not know of. It consists of only one word. All other stuff like C keywords and * is handled internally and cannot be supplied at these place.

Known type: The type which is hidden behind the unknown one. It consists of struct + name, enum + name, signed, unsigned, const, long (32 bit), short (16 bit), char (8 bit), double and float.
For nameless structs or enums set ? as name.

Structs without a name, but used by a typedef (like: typedef struct {...} name) get the typedef name (struct name).

See already added example types in supplied file. Send me new types whenever you get one. Best is to send definition file also, so I can check the definitions myself.

1.16 Functions using FPU registers

There may be FD files containing functions with FPU register arguments.

You may use NOFPU keyword to disable processing of these functions.

For link library stub creation of such functions the clib file is required to determine if argument is double (64 bit) or float/single (32 bit). An warning message appears, when it was not possible to get the correct type. In this case float is used always.
Extend format is currently not supported.

FD2Pragma creates amicall and flibcall pragmas for these functions, but some compilers cannot use them (e.g. MaxonC).

1.17 Files used and created for Phase5's PowerUP boards

It is very hard to tell something about that, as I myself do not know very much about it. Maybe that is caused by missing documentation.

FD2Pragma creates needed files with SPECIAL options starting at number 50. FD2Pragma produces nearly the same files, which are made by special fd2inline version 1.12 (by Ralph Schmidt). I fixed the protos a bit and arranged files to follow my general style, but changed nothing in the way they work.

The files need following directory structure

```
PPCINCLUDE:
  powerup
    gcclib      support files
    ppcinline   GCC inline files (SPECIAL 50)
    ppplib      support files
    ppcpragmas  SAS-C function definitions (SPECIAL 53)
    proto       proto files for GCC/SAS-C (SPECIAL 54)
```

Using this SMakeFile with SAS-C I had not yet larger problems:

```
SCOPTS = RESOPT PARAMETERS=REGISTERS NOSTACKCHECK STRINGMERGE OPTIMIZE \
  OPTIMIZERINLINELOCAL MEMORYSIZE=HUGE OPTIMIZERTIME
SLOPTS = SMALLCODE STRIPDEBUG NOICONS
```

```
PROGRAM = # enter name here
```

```
ALL: ELF PPC
ELF: $(PROGRAM).elf
PPC: $(PROGRAM).ppc
```

```
$(PROGRAM).ppc: $(PROGRAM).ppc.o
  slinkppc $(SLOPTS) PPC FROM LIB:c.o $? LIB LIB:scppc.lib TO $@
  StripHunks $@ # *** Aminet/util/misc/StripHunks.lha ***
```

```
$(PROGRAM).ppc.o: $(PROGRAM).c
  scppc $(SCOPTS) $? HUNKOBJ OBJNAME=$@ DEFINE=AMIGA
```

```
# ***** GCC stuff needs ADE 2 CD
```

```
PPCLD      = ADE-2:AmigaOS/ready-to-run/bin/ppc-amigaos-ld
PPCSTRIP   = ADE-2:AmigaOS/ready-to-run/bin/ppc-amigaos-strip
```

```
$(PROGRAM).elf: $(PROGRAM).elf.o
  $(PPCLD) -r -s -o $@ lib:c_ppc.o $? lib:scppc.a lib:end.o
  $(PPCSTRIP) --strip-unneeded $@
  Protect $@ rwed
```

```
$(PROGRAM).elf.o: $(PROGRAM).c
  scppc $(SCOPTS) $? OBJNAME=$@ DEFINE=AMIGA
```

1.18 Words and phrases

clib-files, prototypes:

For Amiga C functions the prototypes needed in C compilers are stored in

a directory called `clib`. The files are named `libname_protos.h`. The `CLIB` option needs the name of such a file as parameter. These files are needed by `FD2Pragma` to create correct data with some options.

data models:

Most C compilers offer 2 different data models called large and small data (or far and near).

Large data means all data is stored in a `HUNK_DATA` as a normal variable, which is accessed by its address. One access normally needs 4 byte space in program code and 4 byte as relocation entry.

The small data model needs less space. Here the data is stored as one structure. At the program start the compiler adds an instruction, which loads the structure address into register `A4`. In the following program the data is always accessed related to the register `A4`. One access now only needs 2 byte in program code and no relocation entry (saving 6 bytes each access).

But this model has some problems:

- Data size may not exceed 65KB, as relative information uses 2 bytes only.
- When program code is called from outside the program (e.g. hook code), it is not guaranteed that `A4` register still holds the base reference. So these functions need to be `__saveds` (SAS-C) or call functions like `GetBaseReg()` (Maxon-C++).

inline system calls:

GNU-C (GCC) uses a different system to call Amiga system functions. The needed files are stored in a directory called `inline`. Starting with version 2.45 this program is able to produce inline files as well. Before you needed to use `fd2inline` program. I suggest using the proto file you can create with `SPECIAL 35` instead calling inline files directly.

.lib file, link library:

A link library is a file holding functions, which are added to the final executable at linking time. The other method are runtime or shared libraries (`#!.library`) which are called in runtime and thus take no space in the executable program.

pragma:

C allows non standard (compiler private) definitions called pragmas. Most Amiga compilers use them to define system library calls. There exists 5 different `#pragma` statements, which are used by different compilers. I suggest using the proto file you can create with `SPECIAL 35` instead calling pragma files directly.

proto file:

C compilers like SAS have a special directory called `proto` with files in it calling the pragma and prototypes files. This is useful, because different compilers store their pragma files in different directories (or use other methods to define system calls), but all use one proto file. I did not use them till now and called the pragma files directly, but this makes it harder to switch to another compiler. So I now use proto always.

stub, stub function:

A stub functions is a function, which converts between different interfaces. For example C supplies function parameters on stack, but Amiga libraries get them in registers. A stub function for that gets the arguments on stack, copies them into the registers and call's the Amiga function. Newer C compilers have `#pragmas` to do that internally, but some

calling mechanisms are not supported by all compilers. MaxonC++ for example does not support the tagcall.

tag-functions:

C allows to have functions getting a variable number of parameters, everytime they are called. These varargs functions have in there prototypes "... " at the end (e.g. printf). Amiga system libraries use this mechanism for supplying so called tags. (See Amiga programmers documentation for that.)

The name tag-functions is not the best, because there are also some functions getting variable args, which are no tags (e.g. Printf), but it expresses good, what is meant.

1.19 Known bugs and problems

- Pragma creation with cia_lib.fd fails with no ##basename error. This is wanted by Amiga OS programmers to allow passing the library base as first argument. In this case the C compiler function call does not work. You may add a ##basename statement to the FD file and get a working pragma file, but this file will not work together with clib/cia_protos.h file. Using option 14 or 15 instead generates a valid link file to use with clib file. The created text file is of no use and can be deleted. When using created proto file (e.g. SPECIAL 35), you may remove the lines calling pragma file.
- mathieeedoubtrans_lib.fd and mathieeedoubbas_lib.fd both use 2 registers for one double value. FD2Pragma creates only link libraries and definition files for these. Pragma and inline files are not created.
- The redefines of SPECIAL 14 and 15 are illegal, when a function has same name as a structure. (e.g. DateStamp of dos.library). You have to remove the #define line for that function.
- When clib file contains more than one function with same name, FD2Pragma always takes the first one which may not always be the right one. For good include files this should never happen.
- Return register is D0 for all functions. Also flibcall uses D0 as return register always. All the pragma types normally support other return registers in their design. But I never saw any pragmas really using that and I think some compilers (f.e. MaxonC++) do not even accept such pragmas.

Include file errors:

- Datatypes function RefreshDTObject is called RefreshDTObjects in prototypes file. For creating inline files (or other types, which need CLIB argument) this should be corrected.
- Using created graphics pragma brings an error on GetOutlinePen. This is not my fault, but an include error. Remove the line


```
#define GetOutlinePen(rp) GetOPen(rp)
```

 in graphics/gfxmacros.h or turn it around to


```
#define GetOPen(rp) GetOutlinePen(rp)
```

 as this works ok.

- OpenAmigaGuideA of amigaguide.library has a '*' as argument name in FD file. This causes an error for inline files.
- ActivateCxObj of commodities.library has 'true' as argument name in clib and FD file. This may cause errors.
- * Automatic created files may not always be fully correct (may happen * seldom, but sometimes). When you find such a condition (not mentioned * above), please contact me and when useful and possible I will include a * fix in the program.
- * There are so many exceptions in the official include files. How many are * in non default system include files?

1.20 How self made libraries should be designed

As the main author of xpkmaster.library packer interface system ←
and include
creator for other libraries (not to forget: author of FD2Pragma) I got
some experience how library functions should be designed.

Expanding the possibilities of functions most time brings certain problems.
Some ways your reduce these problems:

- Design your functions as tag-functions. For these it is really easy to implement new functionality. Believe me, it was a hard lesson to learn for me. :-)
- It is always a good idea to add a "AllocStructures" function, which allocates all structures, which are needed by your library system. Force the user to ALWAYS use this function. Future additions to the structures are really easy then. A "FreeStructures" function frees the stuff later. The "AllocStructures" may get an ULONG type and tags as argument. The tags allow to change initialisation behaviour.
- Structures should use pointers instead of byte-arrays (for texts) or directly included other structures. This makes expanding possible, but is a little more complicated.
- The function name should reflect your library name, for example xpkmaster functions all start with "Xpk" and xfdmaster functions start with "xfd".

A bit how you should design names to make the work of utilities like FD2Pragma possible:

- Tag-functions should always receive the tagitem array (struct TagItem *) as last element.
- The normal function should end in a big 'A', the tag-function has same name without 'A'. Other methods are described in that document, but this seems to be the best.
- Pointers should be in A-registers, data should be in D-registers. The tagitem array pointer should always be in A-register.
- Try to sort your registers in a order, so that MOVEM.L can be used to get them from stack into registers: D0,...,D7,A0,...A3(,A4,A5)
This affects autodocs, prototype files and FD files. Assembler programmers have no problems with argument order, but it is useful for C.
- Do not use A6 and A7 registers for arguments (nearly impossible).
- Try not to use A4 and A5 register. Some C compilers store data in these registers. There may be problems with functions using these (e.g. inline creation has some restrictions).

- Return value should always be in D0 (and only there). It is really complicated to access multi-return functions from C programs.

How to design

FD files

:

- For tag-functions the last element in FD file should be named 'tags'.
- The argument names should not be equal for different arguments, as these names are used for certain files by FD2Pragma and other utilities.
- If you do not follow above mentioned name convention for tag-functions, use the

tagcall comments

to define tag-function names.

- For separating registers, you may use a "," or a "/". In standard system FD files, the "/" is used to separate these arguments that are in correct order to be used with MOVEM.L call (see above). It is not important which separator you use, as FD2Pragma and nearly all other tools accept both, but I suggest either using "," only or the standard style.
- The file should be named <library>_lib.fd, as FD2Pragma interprets the file name and uses the first part (before _lib.fd) to generate the destination filename. If the file does not end in _lib.fd the defined basename is used.

For plain assembler programmers: Contact a experienced C programmer to get a fine interface (or learn C :-).

1.21 Design description of varargs pragmas

Pragmas are a method to implement compiler specific behaviour in C language programs. Refer to ANSI-C standard to find out what this means.

Most Amiga compilers use the pragma system to implement calls to the Amiga shared libraries. Since there exist different compilers, there are different pragmas also, so I describe the design of these here:

The used format descriptors:

```
base      Name of the library base (e.g. IntuitionBase)
function  Name of the function (e.g. OpenWindowTagList)
hexoffset Bias value of the library (e.g. 25E)
offset    Bias value of the library (e.g. 0x25E or 606)
retret    return register (e.g. d0)
reglist   used registers (e.g. a0,a1), maybe empty
magic     magic value, see below how this is build
fmagic    magic value for floating point
```

pragma libcall (SAS-C, DICE):

Normal call method for shared libraries.

```
#pragma libcall base function hexoffset magic
#pragma libcall IntuitionBase OpenWindowTagList 25E 9802
```

pragma flibcall (SAS-C):

Call method, when arguments are passed in FPU registers (maybe mixed with CPU arguments).

```
#pragma flibcall base function hexoffset fmagic
#pragma flibcall mesamainBase glAlphaFunc 3C 10000002
```

pragma syscall (SAS-C):
Special method to call exec.library function without a Sysbase, but directly with address 4.W.

```
#pragma syscall function hexoffset magic
#pragma syscall AllocMem C6 1002
```

pragma tagcall (SAS-C):
This type allows to call shared library functions with variable argument lists (useful for tag-functions or e.g. Printf).

```
#pragma tagcall base function hexoffset magic
#pragma tagcall IntuitionBase OpenWindowTags 25E 9802
```

pragma amicall (Aztec-C, Maxon++, Storm, [SAS-C supports this with errors]):
Same as libcall for other compilers. FD2Pragma can produce amicall pragmas containing FPU arguments (FP0..FP7), but Storm seems to be the only compiler accepting these.

```
#pragma amicall(base, offset,[retreg=]function(reglist))
#pragma amicall(IntuitionBase,0x25E,d0=OpenWindowTagList(a0,a1))
```

Most compilers do not support return register place (and FD2Pragma too does not), so this really is:

```
#pragma amicall(base, offset,function(reglist))
#pragma amicall(IntuitionBase,0x25E,OpenWindowTagList(a0,a1))
```

pragma tagcall (Storm):
This type allows to call shared library functions with variable argument lists (useful for tag-functions or e.g. Printf).

```
#pragma tagcall(base, offset,function(reglist))
#pragma tagcall(IntuitionBase,0x25E,OpenWindowTags(a0,a1))
```

The magic value:

First thing of every register as one hexadecimal number:
d0..d7,a0..a7,fp0..fp7 --> 0..7,8..F,10..17

Now the magic value is build in following form: <reglist><retreg><numregs>
The reglist is build in reverse order, so the first argument is last entry:

```
OpenWindowTags(a0, a1): returns in d0, has 2 arguments
arguments in reverse order: 98
return register:      0
number of arguments:  2
result:               9802
```

A void function thus gets: 00

The magic value for flibcall is like that, but with 2 spaces for EVERY entry:

```
glAlphaFunc(d0,fp0): returns in d0, has 2 arguments
```

```
arguments in reverse order: 1000
return register:      00
number of arguments:   02
result:               10000002
```

1.22 Design description of FD files

FD files are a standard way to describe the interface for Amiga \leftrightarrow shared libraries. This format consists of 3 main line types:

- 1) Commands - always start with ##
- 2) Comments - always start with *
 - 2a) Plain comments - describe the contents for the user
 - 2b) Command comments - add some functionality (like *tagcall or *notagcall)
- 3) Function descriptions - provide the main information

1)

Know commands are:

```
##base Define the library base name, always starts with a "_"
##bias Define a bias value, minimum value is 30, always a sum of 6
##public Declare following functions as public
##private Declare following functions as private
##end Always in last line, finishes file
```

Normally a FD file is build this way:

```
##base _IntuitionBase
##bias 30
##public
....
##end
```

2)

Comments always start with a * and are normally ignored (except when they are "Command comments". Check the description of these to know what this means.

3)

A function description is build always like that:
name(argnames) (registers)

e.g. OpenWindowTagList(newWindow,tagList) (a0/a1)

For void functions argnames and registers are both empty. Registers are seperated by ",", or "/". See library design to find out the difference.

FD2Pragma accepts registers d0..d7,a0..a7,fp0..fp7.

If the last argument is "tags", "taglist" or "args", FD2Pragma creates additionally a varargs or tag-function. See tag-function

section to learn about that.

In the file you may add `##bias` lines, when you want to skip some internal functions, or `##private` lines, when you want some functions not to be used in normal case. The `##public` keyword can be used to enable the normal system again.

There exists an incompatible FD file format addition created by Haage&Partner for their Storm compiler. It supports an additional command `##shadow`.

```
The format is:
function description
##shadow
tag-function description
```

The first line is like above a normal function description. The `##shadow` command sets back the bias to same value as before and accepts the next line as a tag-function!

```
example:
OpenWindowTagList(newWindow,tagList) (a0/a1)
##shadow
OpenWindowTags(newWindow,tagList) (a0/a1)
```

Do not use that format! FD2Pragma has a STORMFD option to support this.

1.23 How registers of 680x0 processor are used

On Amiga computers exist some conventions how the registers of 680x0 processor are used:

- 1) Register A7 either hold "user stack pointer" USP or "supervisor stack pointer" SSP. Second should not be important for normal Amiga user. The stack is used to store variables, return addresses and for some compiler languages (e.g. C) to store function arguments.
- 2) Register A6 holds a pointer to the library base, when you call a library function. Before the library base is a jump table, which has an entry for every library function. The function itself is reached by jumping into the corresponding entry, which is specified by the base value in FD file (always negativ).
- 3) Registers D0,D1,A0,A1 are scratch registers. This means after calling a library function the contents of these registers is no longer valid! The contents of all the other register are preserved.
- 4) Register D0 normally contains the return value of a called function.
- 5) Register A4 holds data base pointer when you use C compiler in small data model. You should not use this register for argument passing.
- 6) Register A5 is used by C compilers to store a pointer to a field (part of the stack) for local variables.

1.24 Scripts for automatic file creation

In directory Scripts there are some scripts, which allow you to generate necessary files automatic.

MakeInline - Generates inline files for standard system libraries, which can be used with GCC compiler:

Options:

- FDPATH This is the path, where your system FD files are stored. The path must end in ':' or '/'.
- CLIBPATH This is the path, where your system prototype files are stored. The path must end in ':' or '/'.
- INLINEPATH This is the path, where created files should be stored. It must already exist and should be empty.

MakePragma - Generates pragma files for standard system libraries, which can be used with all compilers:

Options:

- FDPATH This is the path, where your system FD files are stored. The path must end in ':' or '/'.
- PRAGMAPATH This is the path, where created files should be stored. It must already exist and should be empty.

MakeStubLib - Generates a stub link library for all standard system libraries like amiga.lib (but with C++ name support). This library only holds stubs and nothing of the additionally amiga.lib stuff. The result is a file called stubs.lib in current directory. When you join this with small.lib distributed on NDUK, you get a complete replacement for amiga.lib and you need not to modify clib files for C++.

You need a Join command, which supports patterns matching, like "Aminet/util/sys/JoinReplace.lha" to get this script to work.

Options:

- FDPATH This is the path, where your system FD files are stored. The path must end in ':' or '/'.
- CLIBPATH This is the path, where your system prototype files are stored. The path must end in ':' or '/'.

MakeProto - Generates proto files for standard system libraries, which can be used with all compilers:

Options:

- FDPATH This is the path, where your system FD files are stored. The path must end in ':' or '/'.
- PROTOPATH This is the path, where created files should be stored. It must already exist and should be empty.

MakeUnit - Generates FPC Pascal compiler unit text files for standard system libraries:

Options:

- FDPATH This is the path, where your system FD files are stored. The path must end in ':' or '/'.
- CLIBPATH This is the path, where your system prototype files are stored. The path must end in ':' or '/'.
- UNITPATH This is the path, where created files should be stored. It must already exist and should be empty.

I added another Shell script called 'MakeStuff', which allows you to generate all the needed files for only library. This is mostly useful for library programmers, who need to release include files for these

libraries.

It gets 3 arguments:

FDFILE The FD file, which normally is named 'xxx_lib.fd'.
 CLIBFILE The prototypes file, which normally is called 'xxx_protos.h'.
 DEST The destination directory, which must already exist and should be empty.

In the destination path all needed directories are created, the FD and prototypes file are copied and inline, lvo, pragma and proto files are created. You only need to copy your library include files and the include stuff is complete.

MakePPCStuff has same arguments, creates needed directories, but does not copy the clib and FD file. It produces the files needed to support PPC PowerUp system.

1.25 Greetings, last words and authors address

Everyone using this program should tell me in a little mail which options he uses and if there are some problems. It took me lots of hours to make FD2Pragma as powerful as it is now. So please tell me if you are using it!

This program is in the public domain. Use it as you want, but WITHOUT ANY WARRANTY!

I want to send greetings also to the author making version 2.0 of this utility. Although this version of FD2Pragma has not much same with version 2.0 it was a big help. The authors address is stated in the source file of the current version!

Thanks Jochen for your great work!

The program is compiled using SAS-C 6.58 and has additionally been tested with MaxonC++ and GCC. Any other ANSI-C compiler should be enough as well.

Because FD2Pragma is very complex, it may be that there are some errors (maybe also serious ones) in the code. So if you find one, please tell me! I will also be glad if someone tells me what can be improved in the program! I will add new options, but a GUI will never come, because this utility is for experts, and they do not need a GUI for creating the needed files. :-) Beside this a GUI would make really much work and increase the file size a lot.

Please contact me:

```
*****
* snail-mail:          * e-mail:          *
* Dirk Stoecker       * stoecker@amigaworld.com      *
* Geschwister-Scholl-Str. 10 * stoecker@rcs.urz.tu-dresden.de      *
* 01877 Bischofswerda      * world wide web:          *
* GERMANY               * http://home.pages.de/~Gremlin/      *
* phone:                * pgp key:                *
* GERMANY +49 (0)3594/706666 * get with finger or from WWW pages *
*****
```

1.26 index

The here given entries link to the correct node entry, but not ↔
 always to
 correct line. This is because I often change the texts and the line
 numbers would need to be updated everytime.

```

##base

##bias

##end

##private

##public

##bias

#pragma amicall

#pragma flibcall

#pragma libcall

#pragma syscall

#pragma tagcall

.lib files

<xxx>_<xxx>_H define

<xxx>_BASE_NAME define

__CONSTLIBBASEDECL__

__INCLUDE_<xxx>_CSTUB_H define

__INCLUDE_PRAGMA_<xxx>_LIB_H define

__INCLUDE_PROTO_<xxx>_LOC_H define

__INLINE_<xxx>_H define

__PROTO_<xxx>_H define

__NOLIBBASE__ define
A
About

AMICALL option

AMITAGS option

```

Author
B
BMAP file

bugs and problems
C
clib files

CLIB option

CLIB_<xxx>_PROTOS_H define

COMMENT option

Copyright
D
data models
E
Examples

EXTERNC option
F
far data

FD file design

FD2Pragma.types

FDFILE option

FPU comments

FPUONLY option
H
HEADER option

Headerscan
I Include directory system

Include file definitions

inline files
L
large data

LIBCALL option

library design

LIBTAGS option

link libraries

Linker library comments

Local library base files
M

MakeInline

MakePPCStuff

MakePragma

MakeProto

MakeStubLib

MakeStuff

MakeUnit

MODE option

 N

near data

NOFPU option

NO_INLINE_STDARG define

 O

Options

 P

Pascal unit

PowerUP

pragma

pragma file design

PRIVATE option

proto file

Proto files

prototypes

 R

registers

 S

Scripts

small data

SMALLDATA option

SORTED option

SPECIAL option

STORMFD option

stub function

 T

tag function

tag-functions definition
U
USESYSALL option