fd2inline

COLLABORATORS			
	TITLE : fd2inline		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 5, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

fd2inline

1.1 fd2inline.guide

```
                    This is a user's guide to FD2Inline 1.11, a parser that  ↩
                       converts the
AmigaOS shared library FD files to format accepted by the GNU CC.

   See file COPYING for the GNU General Public License.

   Last updated January 31st, 1997.



                Introduction
                 What is this program for?

                Installation
                 How to install it?

                Usage
                 How to use inlines and FD2InLine?

                Rebuilding
                 How to recompile it?

                Internals
                 How do inlines work?

                History
                 What has changed?

                Authors
                 Who wrote it?


                Index
                 Concept index.
```

1.2 fd2inline.guide/Introduction

```
                Introduction
************
```

FD2InLine is useful if you want to use GCC for AmigaOS-specific development and would like to call the functions in the AmigaOS shared libraries efficiently.

The format of calls to the AmigaOS shared library functions differs substantially from the default function call format of C compilers (see

Background

). Therefore, some tricks are necessary if you want to use these functions.

FD2InLine is a parser that converts fd files and clib files to GCC inlines.

fd and clib files contain information about functions in shared libraries (see

Background

).

FD2InLine reads these two files and merges the information contained therein, producing an output file suitable for use with the GCC compiler.

This output file contains so-called inlines -- one for each function entry. Using them, GCC can produce very efficient code for making function calls to the AmigaOS shared libraries.

> Note: the term inlines is misleading -- FD2InLine no longer uses the __inline feature of GCC (see
>
> New format
>
>).

1.3 fd2inline.guide/Installation

```
                Installation
************
```

The following assumes you have the fd2inline-1.11-bin.lha archive.

If you use a recent release of GCC, you might not need to install anything. Starting with GCC 2.7.2, the new format (see

New format

) of inlines should be available with the compiler. However, the separate fd2inline-1.11-bin.lha archive will always contain the latest version of FD2InLine and the inlines, which might not be the true for the ADE or Aminet distributions. ADE distribution might not contain

some of the supported 3rd party libraries' inlines.

The installation is very easy, so there is no Installer script :-).

If you have an older version of the inlines installed, please remove it now, or you might encounter problems later. Typically, you will have to remove the following subdirectories of the os-include directory: inline, pragmas and proto.

Next, please change your current directory to ADE: (or GNU:, if you use old Aminet release) and simply unpack the fd2inline-1.11-bin.lha archive. This should install everything in the right place. More precisely, the headers will go to the include directory, the libraries to lib, fd2inline executable to bin directory and the AmigaGuide documentation to the guide directory.

1.4 fd2inline.guide/Usage

```
                Usage
*****
```

This chapter describes two aspects of using FD2InLine:

1.5 fd2inline.guide/Using inlines

```
                Using inlines
=============
```

Using inlines is very simple. If you want to use a library called foo.library (or a device called bar.device), simply include file <proto/foo.h> (<proto/bar.h>) and that's it. For example:

```
#include <proto/dos.h>

int main(void)
{
   Delay(100); /* Wait for 2 seconds */
}
```

Please always include proto files, not inline files - proto files

often fix some incompatibilities between system headers and GCC.
Besides, this technique makes your code more portable across various
AmigaOS C compilers.

   There are a few preprocessor symbols which alter the behaviour of the
proto and inline files:

__NOLIBBASE__
     By default, the proto files make external declarations of the
     library base pointers. You can disable this behaviour by defining
     __NOLIBBASE__ before including a proto file.

__CONSTLIBBASEDECL__
     The external declarations described above declare plain pointer
     variables. The disadvantage of this is that the library base
     variable has to be reloaded every time some function is called. If
     you define __CONSTLIBBASEDECL__ to const, less reloading will be
     necessary, and better code will be produced. However, declaring a
     variable as const makes altering it impossible, so some dirty
     hacks are necessary (like defining the variable as plain in one
     file and altering it only there. However, this will not work with
     base relative code).

<library>_BASE_NAME
     Function definitions in the inline files refer to the library base
     variable through the <library>_BASE_NAME symbol (e.g.,
     AMIGAGUIDE_BASE_NAME for amigaguide.library). At the top of
     the inline file, this symbol is redefined to the appropriate
     library base variable name (e.g., AmigaGuideBase), 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.

NO_INLINE_STDARG
     This symbol prevents the definition of inline macros for varargs
     functions (see
                Old format
).

USEOLDEXEC
     This symbol is used only in proto/exec.h. Unlike SAS/C,
     proto/exec.h uses the SysBase variable as the Exec library
     base by default. This is usually faster than direct dereferencing
     of 0x00000004 (see
                Background
                ), since it does not require reading
     from CHIP memory (things might be even worse if you use Enforcer
     or CyberGuard, which protect the low memory region). However, in
     some low-level cases (like startup code) you might prefer
     dereferencing 0x00000004. To do this, define _USEOLDEXEC_ before
     including proto/exec.h.

1.6 fd2inline.guide/Using fd2inline

Using fd2inline

You invoke FD2InLine by writing:

```
fd2inline [options] FD-FILE CLIB-FILE [[-o] OUTPUT-FILE]
```

The command line arguments have the following meaning:

FD-FILE
: The name of the input fd file.

CLIB-FILE
: The name of the input clib file.

OUTPUT-FILE
: The name of the output inline file. If it is not specified (or if - is specified), standard output will be used instead. The file name can be preceded by a -o, for compatibility with most UN*X software.

The following options can be specified (anywhere on the command line):

--new
: Produce new format inlines.

--old
: Produce old format inlines.

--stubs
: Produce library stubs.

--proto
: Produce proto file. If this option is specified, providing clib file is not necessary. fd2inline will only read fd file and will generate a proto file, to be put in include/proto directory.

--version
: Print version information and exit.

See Internals, for more information.

Example:

```
fd2inline ADE:os-lib/fd/exec_lib.fd ADE:os-include/clib/exec_protos.h -o ADE: ↩
    include/inline/exec.h
```

This will build file exec.h containing new format inlines of exec.library in directory ADE:include/inline.

If you want to add support for GCC to a library, there are a few things you should remember about.

Sometimes, FD2InLine might not know how to handle a function found in a clib file, if this function doesn't have a corresponding entry in the fd file. This is most often a case of varargs functions (see

Background

), if they use nonstandard naming convention. FD2InLine will warn you if it finds such a function. There is an array of such exceptions in FD2InLine source code. You should add the name of this function there and send a patch to FD2InLine maintainer (see

Authors

),
for inclussion in the next release of FD2InLine.

FD2InLine assumes that the type of the base variable is struct Library *. If it is something different in your case, you should extend an array of exceptions in FD2InLine source code (see above).

FD2InLine handles void functions in a special way. It recognizes them by the return value -- it has to be void (the case is not significant). If a clib file uses a different convention, it has to be modified before running FD2InLine.

In addition to creating inlines you must also create a proto file.

The easiest way to do it is to call FD2InLine with --proto option. Most often, the generated file will be ready to use. Unfortunately, some libraries (like, for example, dos.library) have broken header files and GCC generates warning messages if you try to use them. To avoid these warnings, you have to include various headers in the proto file before including the clib file.

You might also want to create a pragmas file, which might be necessary for badly written SAS/C sources. pragmas are generated automatically during the building of FD2InLine by an AWK script, so you might either have a look at the fd2inline-1.11-src.lha archive, or simply create pragmas file by hand.

Creating a linker library with stubs might also be useful, in case somebody doesn't want to, or can't, use inline headers.

fd2inline-1.11-src.lha contains necessary support for this. For example, to generate a library libexec.a with exec.library stubs, you should type:

```
make alllib INCBASE=exec LIBBASE=exec
```

This will create three libexec.a libraries in lib subdirectory: plain, base relative and 32-bit base relative one. Of course, this particular example doesn't make much sense since libamiga.a already contains these stubs.

INCBASE and LIBBASE specify the base names of the (input) proto and fd files and the (output) library. This will often be the same, but not always. For example, in the case of MUI, INCBASE has to be set to muimaster, but LIBBASE should be set to mui.

1.7 fd2inline.guide/Rebuilding

```
Rebuilding
**********

   First, you have to get the fd2inline-1.11-src.lha archive.

   Unarchive it. You might either build FD2InLine in source directory
or in a separate, build directory. The latter is recommended. Type:

     lha -mraxe x fd2inline-1.11-src.lha
     mkdir fd2inline-bin
     cd fd2inline-bin
     sh ../fd2inline-1.11/configure --prefix=/ade
     make

   This should build the FD2InLine executable, headers, libraries and
so on.

   Please note that the fd files should be available in the directory
ADE:os-lib/fd. If you store them in some other place, you will have
to edit the Makefile and modify variable FD_DIR before invoking make.

   You can then type:

     make install

   This will install fd2inline, the inlines and the documentation in
the appropriate subdirectories of ADE:.

   The fd2inline-1.11-src.lha archive contains four patches in unified
diff format, in directory patches. They fix bugs in OS 3.1 headers and
fd files. Without applying amigaguide_lib.fd.diff to
amigaguide.library fd file, the produced inlines will be broken.
Applying timer.h.diff to devices/timer.h will prevent collision with
IXEmul's sys/time.h. Two other patches rename an argument name from
true to tf, since true is a reserved word in C++. Use patch to
apply these patches, for example:

     cd ADE:os-lib/fd
     patch -p0 <amigaguide_lib.fd.diff

   ADE and Aminet distributions contain more complete sets of patches.

   A few words about the source code:

   I know, it's not state-of-the-art C programming example.  However,
believe me, it was in much worse condition when I took it over. In its
current state it is at least readable (if you use tab size 3, as I do
:-). I think that rewriting it in C++ would clean it up considerably
(it's already written in OO fashion, so this should be quite easy).
Using flex and bison to create the parser would also be a nice thing, I
guess. However, I don't think it's worth the effort. But, if somebody
```

wants to do it: feel free, this is GNU software, so everybody can
modify it.

1.8 fd2inline.guide/Internals

```
                Internals
*********

   This chapter describes the implementation details of inlines.



                Background
                 Function calls in shared libraries.

                Old format
                 Inlines that use __inline.

                New format
                 Inlines that use the preprocessor.

                Stubs format
                 Not really inlines, but...
```

1.9 fd2inline.guide/Background

```
Background
==========

   This section describes the calling conventions used in the AmigaOS
shared libraries.

   User-callable functions in the AmigaOS are organized in libraries.

   From our point of view, the most important part of a library is the
library base. It always resides in RAM and contains library
variables and a jump table. The location of the library base varies.
You can obtain the library base location of the main system library --
exec.library -- by dereferencing 0x00000004. Locations of other
library bases can be obtained using the OpenLibrary function of
exec.library.

   Without providing unnecessary details, every function in a library
has a fixed location in the library's jump table. To call a function,
one has to jump to this location.

   Most functions require some arguments. In C, these are usually
passed on the CPU stack. However, for some obscure reason, AmigaOS
system designers decided that arguments to shared libraries should be
```

passed in CPU registers.

All the information required to make library function calls is provided in fd files. Every shared library should have such a file. It provides the name a library base variable should have, the offset in the jump table where each library function resides, and information about which arguments should be passed in which registers.

In order to check if arguments passed to a function have the correct type, the C compiler requires function prototypes. These are provided in clib files -- every library should have such a file.

Starting with the AmigaOS release 2.0, certain functions have been provided which accept a variable number of arguments (so-called varargs functions). Actually, these are only C language stubs. Internally, all optional arguments have to be put into an array of long ints and the address of this array must be passed to a fixed args library function.

To implement calls to shared library functions, compiler vendors have to either use some compiler-dependent tricks to make these calls directly (so-called in line), or provide linker libraries with function stubs, usually written in assembler. In the latter case, a function call from the user's code is compiled as usual -- arguments are passed on the stack. Then, in the linking stage, a library stub gets linked in. When this stub is called during program execution, it moves the arguments from the stack to the appropriate registers and jumps to the library jump table. Needless to say, this is slower than making a call in line.

1.10 fd2inline.guide/Old format

```
Old format
==========

     extern __inline APTR
     OpenAmigaGuideA(BASE_PAR_DECL struct NewAmigaGuide *nag, struct TagItem * ↩
        attrs)
     {
        BASE_EXT_DECL
        register APTR res __asm("d0");
        register struct Library *a6 __asm("a6") = BASE_NAME;
        register struct NewAmigaGuide *a0 __asm("a0") = nag;
        register struct TagItem *a1 __asm("a1") = attrs;
        __asm volatile ("jsr a6@(-0x36:W)"
        : "=r" (res)
        : "r" (a6), "r" (a0), "r" (a1)
        : "d0", "d1", "a0", "a1", "cc", "memory");
        return res;
     }
```

In this implementation, the AmigaOS shared library function stubs are external functions. They are defined as __inline, making GCC insert them at every place of call. The mysterious BASE_PAR_DECL and BASE_EXT_DECL defines are hacks necessary for local library base

support (which is quite hard to achieve, so it will not be described here). The biggest disadvantage of these inlines is that compilation becomes very slow, requiring huge amounts of memory. Besides, inlining only works with optimization enabled.

```
#ifndef NO_INLINE_STDARG
#define OpenAmigaGuide(a0, tags...) \
   ({ULONG _tags[] = { tags }; OpenAmigaGuideA((a0), (struct TagItem *)_tags) ↩
      ;})
#endif /* !NO_INLINE_STDARG */
```

The source above shows how varargs functions are implemented. Handling them cannot be made using __inline functions, since __inline functions require a fixed number of arguments. Therefore, the unique features of the GCC preprocessor (such as varargs macros) have to be used, instead. This has some drawbacks, unfortunately. Since these are actually preprocessor macros and not function calls, you cannot make tricky things involving the preprocessor inside them. For example:

```
#include <proto/amigaguide.h>

#define OPENAG_BEG OpenAmigaGuide(
#define OPENAG_END , TAG_DONE)

void f(void)
{
   OPENAG_BEG "a_file.guide" OPENAG_END;
   OpenAmigaGuide(
#ifdef ABC
      "abc.guide",
#else
      "def.guide",
#endif
      TAG_DONE);
}
```

Neither of the above OpenAmigaGuide() calls is handled correctly.

In the case of the first call, you get an error:

```
unterminated macro call
```

By the time the preprocessor attempts to expand the OpenAmigaGuide macro, OPENAG_END is not yet expanded, so the preprocessor cannot find the closing bracket. This code might look contrived, but MUI, for example, defines such macros to make code look more pretty.

In the case of the second call, you'll see:

```
warning: preprocessing directive not recognized within macro arg
```

A workaround would be to either surround entire function calls with conditions, or to conditionally define a preprocessor symbol GUIDE somewhere above and simply put GUIDE as a function argument:

```
#ifdef ABC
#define GUIDE "abc.guide"
```

```
#else
#define GUIDE "def.guide"
#endif

void f(void)
{
#ifdef ABC
   OpenAmigaGuide("abc.guide", TAG_DONE);
#else
   OpenAmigaGuide("def.guide", TAG_DONE);
#endif
   OpenAmigaGuide(GUIDE, TAG_DONE);
}
```

Another problem is that when you pass a pointer as an argument, you get a warning:

```
warning: initialization makes integer from pointer without a cast
```

This is because all optional arguments are put as initializers to an array of ULONG. And, if you attempt to initialize an ULONG with a pointer without a cast, you get a warning. You can avoid it by explicit casting of all pointer arguments to ULONG.

Because of these drawbacks, varargs inlines can be disabled by defining NO_INLINE_STDARG before including a proto file. In such a case, you will need a library with function stubs.

1.11 fd2inline.guide/New format

```
                New format
==========

#define OpenAmigaGuideA(nag, attrs) \
   LP2(0x36, APTR, OpenAmigaGuideA, struct NewAmigaGuide *, nag, a0, struct  ↩
       TagItem *, attrs, a1, \
   , AMIGAGUIDE_BASE_NAME)
```

As you can see, this implementation is much more compact. The LP2 macro (and others) are defined in inline/macros.h, which is included at the beginning of every inline file.

```
#define LP2(offs, rt, name, t1, v1, r1, t2, v2, r2, bt, bn) \
({             \
   t1 _##name##_v1 = (v1);       \
   t2 _##name##_v2 = (v2);       \
   {             \
      register rt _##name##_re __asm("d0");    \
      register struct Library *const _##name##_bn __asm("a6") = (struct  ↩
         Library*)(bn);\
      register t1 _n1 __asm(#r1) = _##name##_v1; \
      register t2 _n2 __asm(#r2) = _##name##_v2; \
      __asm volatile ("jsr a6@(-"#offs":W)"    \
      : "=r" (_##name##_re)        \
```

```
           : "r" (_##name##_bn), "r"(_n1), "r"(_n2)    \
           : "d0", "d1", "a0", "a1", "cc", "memory"); \
           _##name##_re;          \
       }             \
    })
```

   If you compare this with the old inlines (see
               Old format
) you will
notice many similarities. Indeed, both implementations use the same tricks. This means that there should be small, if any, difference in code quality between old and new inlines.

   With the new inlines, however, inlining is performed very early, at the preprocessing stage. This makes compilation much faster, less memory hungry, and independent of the optimization options used. This also makes it very easy to use local library bases -- all that is needed is to define a local variable with the same name as library base.

   Unfortunately, using the preprocessor instead of the compiler for making function calls has its drawbacks, as described earlier (see

               Old format
). There is not much you can do about it apart from modifying your code.

   Depending on the type of a function, FD2InLine generates calls to different LP macros.

   Macros are distinguished by one or more of the qualifiers described below:

digit
     As you may have already guessed, digit indicates the number of arguments a function accepts. Therefore, it is mandatory.

NR
     This indicates a "no return" (void) function.

A4, A5
     These two are used when one of the arguments has to be in either the a4 or a5 register. In certain situations, these registers have special meaning and have to be handled more carefully.

UB
     This indicates "user base" -- the library base pointer has to be specified explicitly by the user. Currently, this is used for cia.resource only. Since there are two CIA chips, the programmer has to specify which one [s]he wants to use.

FP
     This means that one of the arguments is of type "pointer to a function". To overcome strange C syntax rules in this case, inside FP macros a typedef to __fpt is performed. The inline file passes __fpt as the argument type to the LP macro. The actual type of the argument, in a form suitable for a typedef, is passed as an additional, last argument.

As you can see, there could be more than a hundred different variations of the LP macros. inline/macros.h contains only 34, which are used in the current OS version and supported 3rd party libraries. More macros will be added in the future, if needed.

If you look carefully at the definition of OpenAmigaGuideA at the beginning of this section, you might notice that the next to last argument to the LP macro is not used. New inlines were not implemented in one evening, and they went through many modifications. This unused argument (which was once a type of library base pointer) is provided for backwards compatibility. Actually, there are more unnecessary arguments, like function and argument names, but it was decided to leave them in peace.

1.12 fd2inline.guide/Stubs format

```
                Stubs format
============
```

Stubs format is very similar to old format (see
Old format
). The
functions are not defined as extern, however.

The main difference is the format of the varargs functions -- they are plain functions, not preprocessor macros.

```
APTR OpenAmigaGuide(struct NewAmigaGuide *nag, int tag, ...)
{
   return OpenAmigaGuideA(nag, (struct TagItem *)&tag);
}
```

This format is not suitable for inlining, and it is not provided for this purpose. It is provided for the building of linker libraries with stubs (see
Using fd2inline
).

1.13 fd2inline.guide/History

```
History
*******
```

Version 1.0, July 14th, 1996, Kamil Iskra
* First officially available version.

Version 1.1, October 24th, 1996, Kamil Iskra

```
   * Removed a lot of language mistakes from the documentation (Kriton
     Kyrimis).

   * Inlines of dospath.library, screennotify.library, ums.library and
     wbstart.library integrated (Martin Steigerwald).

   * Inlines of muimaster.library integrated (Kamil Iskra).

   * Floating point registers fp0 and fp1 are now marked as clobbered
     (Kamil Iskra, reported by Kriton Kyrimis).

   * Improved handling of clib files, particularly recognition of
     function prototypes and varargs functions (Kamil Iskra).

   * Added support for --proto and --version options. Minor changes in
     output file generators, most notably making proto files work with
     other compilers than GCC, too. Finalized support for building
     linker libraries (Kamil Iskra, change in proto files suggested by
     Joop van de Wege).

   Version 1.11,  January 31st, 1997,  Kamil Iskra
   * Minor fixes in Makefile.in for ADE tree (Fred Fish).

   * Fixed handling of prototypes in which the argument name was the
     same as the type name (Kamil Iskra, reported by Martin
     Recktenwald).

   * Added support for building 32-bit base relative linker libraries
     (Kamil Iskra).

   * Added a paragraph about void functions (Kamil Iskra, suggested by
     Martin Recktenwald).
```

1.14 fd2inline.guide/Authors

```
                Authors
*******

   The first parser for GCC inlines was written in Perl by Markus Wild.

   It had several limitations, which were apparently hard to fix in
Perl. This is why Wolfgang Baron decided to write a new parser
in C.

   For some reason, however, he never finished it. In early 1995. Rainer
F. Trunz took over its development and "improved, updated, simply made
it workable" (quotation from the change log). It still contained quite
a few bugs, though.

   In more-or-less the same time, I started a discussion on the
amiga-gcc-port mailing list about improving the quality of inlines.
The most important idea came from Matthias Fleischer, who introduced
the new format of inlines (see
                New format
```

```
               ).  Since I started the
discussion, I volunteered to make improvements to the inlines parser.
Having no idea about programming in Perl, I decided to modify the
parser written in C. I fixed all the bugs known to me, added some new
features, and wrote this terribly long documentation :-).

   Not all of the files distributed in the FD2InLine archives were
created by me or FD2InLine. Most of the files in include/proto-src and
include/inline-src (alib.h, strsub.h and stubs.h) were written by
Gunther Nikl (with some modifications by Joerg Hoehle and me).

   If you have any comments concerning this work, please write to:

   ade-gcc@ninemoons.com

   This is a list to which most of the ADE GCC developers and activists
subscribe, so you are practically guaranteed to get a reply.

   However, if, for some reason, you want to contact me personally, you
can do so in one of the following ways:

   * E-mail (preferred :-):

     iskra@student.uci.agh.edu.pl

     Should be valid until October 1999 (at least I hope so :-).

   * Snail-mail (expect to wait long for a reply :-):

     Kamil Iskra
     Luzycka 51/258
     30-658 Krakow
     Poland

   Latest version of this package should always be available on my WWW
page:

http://student.uci.agh.edu.pl/~iskra
```

1.15 fd2inline.guide/Index

```
               Index
*****


                <library>_BASE_NAME
                  Using inlines

                __CONSTLIBBASEDECL__
                  Using inlines

                __NOLIBBASE__
                  Using inlines
```