fd2inline

COLLABORATORS			
	TITLE : fd2inline		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 7, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

fd2inline

1.1 fd2inline.guide

This is a user's guide to 'FD2Inline' 1.0, a parser that converts Amiga shared libraries' FD files to format accepted by GNU CC.

Last updated 14 July 1996.

1.2 fd2inline.guide/Introduction

Introduction
****

'FD2InLine' is useful if you want to use 'GCC' for Amiga-specific development and would like to make very efficient calls to Amiga shared libraries' functions.

Format of calls to Amiga shared libraries' functions differs substantially from the default 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 outputs a file containing information from input files merged in a special format, suitable to use with 'GCC' compiler.

This output file contains so-called "inlines" -- one for each function entry. Thanks to them 'GCC' can produce very efficient code for Amiga shared libraries' functions calls.

Note: the term 'inlines' is misleading -- 'FD2InLine' does not use '__inline' feature of 'GCC' any longer (see New format).

1.3 fd2inline.guide/Installation

Installation
****

I assume you have 'fd2inline-1.0-bin.lha' archive.

If you use recent release of 'GCC', you might not need to install anything. Starting with 'GCC' 2.7.2, new format (see New format) of 'inlines' should be available with the compiler. However, the separate 'fd2inline-1.0-bin.lha' archive will always contain the latest version of 'FD2InLine' and 'inlines', which might not be the truth in case of 'ADE' or 'Aminet' distributions.

Installation is very easy, so I didn't bother writing an Installer script :-).

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

Next, please change current directory to 'ADE:' and simply extract 'fd2inline-1.0-bin.lha' archive. This should install everything in right places. More precisely, 'inlines' of standard system libraries and devices will go to 'include/inline' directory and 'inlines' of 3rd party libraries will go to 'local/include/inline' directory. 'fd2inline' executable will go to 'bin' directory and AmigaGuide documentation to 'guide'.

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'), just 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 Amiga compilers.

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

'__NOLIBBASE__'
: By default, 'proto' files make an external declaration of library base pointer. 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 library base variable has to be reloaded every time some function is called. If you redefine '__CONSTLIBBASEDECL__' to 'const', less reloading will be necessary, so better code will be produced. However, declaring a variable as 'const' makes altering it impossible, so some dirty hacks are necessary (like defining variable as plain in one file and altering it only there. This *won't* work with base relative code, though).

'<library>_BASE_NAME'
: Function definitions in 'inline' files refer to library base variable through '<library>_BASE_NAME' symbol ('AMIGAGUIDE_BASE_NAME' for 'amigaguide.library', for example). On top of 'inlines' file, this symbol is redefined to appropriate library base variable name ('AmigaGuideBase', for example), *unless* it's already defined. This way, you can make inlines use a field of a structure as a library base, for example.

'NO_INLINE_STDARG'

```
     This symbol prevents defining of inline macros for varargs
     functions (see Old format).

'_USEOLDEXEC_'
     This one is used in 'proto/exec.h' only. Unlike in 'SAS/C',
     'proto/exec.h' uses 'SysBase' variable as 'Exec' library base by
     default. This is usually faster than direct dereferencing of
     '0x00000004' (see Background), since it doesn't require reading
     from 'CHIP' memory (things might be even worse if you use
     'Enforcer' or 'CyberGuard', which protect 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]

   Variables have the following meaning:

FD-FILE
     A file name of an input 'fd' file.

CLIB-FILE
     A file name of an input 'clib' file.

OUTPUT-FILE
     A file name of an output 'inlines' file. If it is not specified
     (or if '-' is specified), standard output will be used instead.
     The file name can be preceded with 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.

   See Internals, for more information.

   Example:
```

```
    fd2inline 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 ‘ADE:include/inline’ directory.

   If you want to add support for ‘GCC’ to a library, creating ‘inlines’ is not enough -- you should also create ‘proto’ file.

   Use one of existing files as a template. Some libraries (like for example ‘dos.library’) have broken header files and ‘GCC’ generates warning messages. In order to avoid them, you have to include various headers in ‘proto’ file before including ‘clib’ file.

   You might also want to create ‘pragmas’ file, which might be necessary for badly written ‘SAS/C’ sources. ‘pragmas’ are generated automatically during building of ‘FD2InLine’ by an ‘AWK’ script, so you might either have a look at ‘fd2inline-1.0-src.lha’ archive or simply create ‘pragmas’ by hand.

   Creating a linker library with stubs might also be useful, in case somebody didn’t want to, or couldn’t, use inline headers.

   ‘fd2inline-1.0-src.lha’ contains necessary support for this. For example, to generate ‘libexec.a’ library with ‘exec.library’ stubs, you’d have to type:

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

   This would create two ‘libexec.a’ libraries: plain and base relative one. Of course, this particular example doesn’t make much sense since ‘libamiga.a’ already contains this stubs.

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

1.7 fd2inline.guide/Rebuilding

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

   First, you have to download ‘fd2inline-1.0-src.lha’ archive.

   Unarchive it. You might either build ‘FD2InLine’ in source archive or in a separate, build archive. I suggest the latter. Type:

```
    lha -mraxe x fd2inline-1.0-src.lha
    mkdir fd2inline-bin
    cd fd2inline-bin
    sh ../fd2inline-1.0/configure --prefix=/ade
    make
```

```
   This should build 'FD2InLine' executable and 'inlines'.

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

   You can then type:

     make install

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

   'fd2inline-1.0-src.lha' archive contains three patches in context
diff format. They fix bugs in OS 3.1 headers and 'fd' files. Without
applying 'amigaguide_lib.fd.diff' you won't be able to build 'inlines'
for 'amigaguide.library'.  Two other patches rename an argument 'true'
to 'tf', since 'true' is a reserved word in 'C++'. Use 'patch' to apply
the patches, for example:

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

   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 wanted to do it: feel free, I'll be more than happy.
```

1.8 fd2inline.guide/Internals

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

   This chapter describes the implementation details of 'inlines'.


  Background                  Function calls in shared libraries.
  Old format                  Inlines that used '__inline'
  New format                  Inlines that use preprocessor.
  Stubs format                Not really inlines, but...
```

1.9 fd2inline.guide/Background

Background

This section describes calling conventions used in Amiga shared libraries.

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

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

Without providing unnecessary details, every function has a fixed place in library's jump table. In order to call a function, one has to jump into this place.

Most functions require some arguments. In 'C', these are usually passed on 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 function calls are provided in "fd" files. Every shared library should have such a file. It provides information about the name a library base variable should have and, for every function separately, about offset in jump table where the function resides and about registers in which arguments should be passed.

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

Starting with AmigaOS release 2.0, some functions have been provided which accept 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 address of this array is passed to fixed args library function.

To implement calls to shared libraries' functions, compiler vendors have to either use some compiler-dependent tricks to make this calls directly (so-called "in line"), or provide linker libraries with functions stubs, usually written in assembler. In the latter case, function call from user's code is compiled as usual -- arguments are passed on stack. Then, in linking stage, a library stub gets linked in, and when it's called, it moves arguments from stack to appropriate registers and jumps to 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, Amiga shared libraries' function stubs are external functions. They are defined as '__inline', what makes 'GCC' insert them at every place of call. Mysterious 'BASE_PAR_DECL' and 'BASE_EXT_DECL' defines are hacks necessary for local library base support (which is quite hard to achieve, so I won't describe it here). The biggest disadvantage of this 'inlines' is the fact that 'GCC' becomes very slow and requires huge amount of memory when compiling them. Besides, inlining works with optimization enabled only.

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

This is how 'varargs' functions are implemented. Handling them cannot be made using '__inline' functions, since '__inline' functions require fixed number of arguments. Therefore, unique features of GCC preprocessor (such as 'varargs macros') have to be used, instead. This has some drawbacks, unfortunately. Since this are actually preprocessor macros and not function calls, you can't make tricky things that involve 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 'OpenAmigaGuide()' calls above is handled correctly.

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

```
  unterminated macro call
```

By the time preprocessor attempts to unwind 'OpenAmigaGuide' macro, 'OPENAG_END' is not yet unwound, so preprocessor can't find closing bracket. This code might look artificial, but 'MUI' for example defines such macros to make code look more pretty.

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

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

A workaround would be to either surround whole 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);
    }
```

Because of this drawbacks, 'varargs inlines' can be disabled by defining 'NO_INLINE_STDARG' before including 'proto' file. You'll need a library with function stubs in such case.

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. ‘LP2’ macro (and others) are defined in ‘inline/macros.h’, which is being 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 old inlines (see Old format) you’ll notice many similarities. Indeed, both implementations use the same tricks.

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

Unfortunately, using preprocessor instead of compiler for making function calls has its drawback, as described earlier (see Old format). There is nothing you can do about it but modify your code, I’m afraid.

Depending on the type of function, ‘fd2inline’ generates calls to different ‘LP’ macros.

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

‘digit’
: As you might have already guessed, digit indicates the number of arguments a function accepts. Therefore, it’s mandatory.

‘NR’
: This indicates "no return" (‘void’) function.

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

‘UB’
: This indicates "user base" -- library base pointer has to be specified explicitly by user. Currently, this is used for ‘cia.resource’ only. Since there are two ‘CIA’ chips, programmer

```
     has to specify which one [s]he wants to use.

'FP'
     This means that one of the arguments has a "pointer to function"
     type. To overcome strange 'C' syntax rules in this case, inside
     'FP' macros a 'typedef' to '__fpt' is performed.  'inline' file
     passes '__fpt' as argument's type to '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 'LP' macros. 'inline/macros.h' contains only 34, which are used in current OS version and supported 3rd party libraries. More macros will be added in future, if needed.

If you had a careful look at the definition of 'OpenAmigaGuideA' on the beginning of this section, you might have noticed that next to last argument to 'LP' macro is not used. New inlines were not implemented in one evening and they came 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 we decided to left 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 format of 'varargs' functions -- they are actual 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 building of linker libraries with stubs (see Using fd2inline).

1.13 fd2inline.guide/Authors

```
Authors
*******
```

First parser for 'GCC inlines' was written in 'Perl' by Markus Wild.

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

However, for some reason he has never finished it. In early 1995 Rainer F. Trunz took over its development and "improved, updated, simply made it workable" (it's a quote from changes log). It still contained quite a lot of bugs, though.

In more-or-less the same time I started a discussion on 'amiga-gcc-port' mailing list about improving quality of 'inlines'. The most important idea came from Matthias Fleischer -- he introduced new format of 'inlines' (see New format). Since I started the discussion, I volunteered to make improvements to '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 'FD2InLine' archives were created by me or 'FD2InLine'. Most of the files in 'include/proto', as well as a few files in 'include/inline' ('alib.h', 'strsub.h' and 'stubs.h') were written by Gunther Nikl (with some modifications by Joerg Hoehle).

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

ade-gcc@ninemoons.com

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

However, if you for some reason wanted to contact me personally, do it 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

1.14 fd2inline.guide/Index

Index
*****