Interface Translation for Reuse of Assembly-Language Modules in a Two-Language Environment

A mixture of low-level and high-level implementation languages is likely when old modules are reused. In a two-language system, some interfaces must be expressed in both languages. This paper describes the design and implementation of a production-quality software tool that solves this problem for the C programming language.

by James R. Buffenbarger

The advantages of developing a software system in a high-level programming language rather than a low-level assembly language are well-known. Nevertheless, many embedded systems have been and continue to be developed in a low-level language. This implementation language paradox is typically justified in terms of the required execution efficiency or the lack of high-level software development tools.

Typically, a new system is built from a mixture of old and new modules. The old modules are often from a previous system. Depending on their age, they may be implemented in a low-level language. The new modules might be implemented in a low-level or high-level language.

This paper is concerned with the development and reuse of intermodule interfaces in a system requiring a mixture of low-level and high-level implementation languages. Such a mixture is likely when old modules are reused. New modules might also be implemented in a low-level language, but they are more likely to be implemented in a high-level language, since processors are getting faster, memories are getting larger, and high-level software-development tools are becoming available.

In this paper, a problem is identified and defined, several possible approaches are proposed, and the most promising approach is selected and pursued. The design and implementation of a production-quality software tool that solves the problem for the C programming language are described in detail and short-term industrial experiences with this tool are briefly described.

The Problem and Possible Solutions

Suppose a two-language system contains a module named f. Clearly, f should have only one implementation. However, f may need two equivalent interfaces: one for each language. There are several possible approaches:

- Manually develop and maintain a low-level interface only. This approach requires f and every user of its interface to be implemented in the low-level language. However, f or one of its users may be a new module, which suggests a high-level implementation.
- Manually develop and maintain a high-level interface only. This approach requires f and every user of its interface to be implemented in the high-level language. However, f or one of its users may be an old low-level module, which should be reused rather than reimplemented.
- Manually develop and maintain both interfaces. This approach allows f to be implemented in either language. However, the probability of inconsistency is very high, as it is for any instance of double maintenance.
- Develop and maintain the high-level interface manually and derive the low-level interface automatically, according to a well-defined transformation. This approach also allows f to be implemented in either language, but avoids double maintenance. Note that the reverse transformation is not possible.

Interface Translation

The last of these approaches seems to be the most reasonable, and is pursued here. Interface translation is flexible, accurate, and fast.

In many languages, an interface is contained in what is called an *include* file. Thus, deriving a low-level interface from a high-level interface means translating an include file written in a high-level language into an include file written in a low-level language. An include-file translator can be used in at least two ways.

Suppose a programmer is writing a module named g and needs to use a module named f. If the programmer is an assembly-language programmer, module f needs to consist of an assembly-language include file f.inc, which specifies the interface of f, and a binary object file f.obj for linking. On the other hand, if the g programmer is a C-language programmer, module f needs to consist of a C include file f.h, which specifies the interface of f, and a binary object file f.obj for linking. Therefore, the "user view" of module f is the three files: f.inc, f.h, and f.obj. These three files must be available so that both assembly-language and C-language programmers can use module f.

Module f may be written either in assembly language or in C. If it is written in assembly-language, there will be an assembly-language file f.a96 which implements the interface of f and includes f.inc. If f is written in C, there will be a C file f.c which implements the interface of f and includes f.h.

Therefore, if f is written in assembly language, the following files are necessary so that both assembly-language and C programmers can use module f: f.h, f.a96, f.inc, and f.obj. With an include-file translator, only f.h and f.a96 need to be maintained manually; f.inc can be translated from f.h, and f.obj is assembled from f.a96. The processes are shown in Fig. 1.

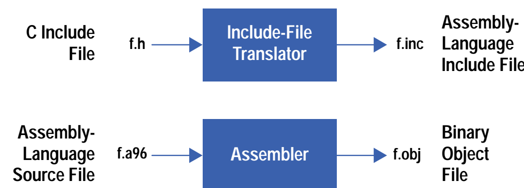

Fig. 1. *For a module f, written in assembly language, to be usable by both assembly-language and C programmers, the four files shown here are needed. With an include-file translator, only the files on the left need to be maintained manually.*

If f is written in C, the following files are necessary: f.h, f.c, f.inc, and f.obj. Again, with an include-file translator, only f.h and f.c need to be maintained manually; f.inc can be translated from f.h, and f.obj is compiled from f.c. The processes are illustrated in Fig. 2.

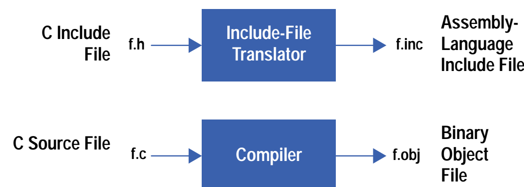

Fig. 2. *For a module f, written in C, to be usable by both assembly-language and C programmers, the four files shown here are needed. With an include-file translator, only the files on the left need to be maintained manually.*

Include-file translation is similar to compilation. In fact, one way of interpreting these ideas is as additional requirements for a compiler. Perhaps compiler/assembler vendors will eventually incorporate these features into their products.

The ideas presented here are programming language independent. However, for demonstration purposes, a particular assembly language and a particular high-level language (C) are discussed. Furthermore, just one implementation of what should be considered a class of translation tools is described.

Pretranslator Environment

We begin with a description of an environment that needs an include-file translator.

The software is developed in an evolutionary way, in a heterogeneous environment. The software is large and mature, residing in an centralized version-controlled repository. The *development platforms* are workstations based on the UNIX®, MS-DOS®, and OS2 operating systems. The *target platform* is an embedded system based on an Intel microcontroller. The software is developed in assembly language because good cross-compilers for the target platform do not exist, compiler-generated code is too slow, and compiler-generated code is too big.

Eventually, the software developers and their managers are ready for a change. New microcontrollers are much faster, and they can address much more memory. Management recognizes the potential benefits of assembler and processor independence, not the least of which is the freedom to buy development tools (e.g., emulators) from more than one vendor. Good cross-compilers are finally available. New developers would become productive more quickly if they did not have to learn an esoteric assembly language. All developers would become more productive if they could program in a high-level language when appropriate.

In this environment, the high-level language of choice is C, but its complete and immediate adoption has well-founded resistance. For speed and space efficiency, some parts of the software should never be rewritten in C. Furthermore, those parts of the software that should be rewritten in C represent an enormous corporate investment. An effort to rewrite all of them at once would have severe short-term productivity and quality costs. Instead, the unanimous understanding is that only some of the assembly-language modules should be replaced by C modules, and then only in a prioritized piecemeal fashion. Thus, the environment must support two-language development.

A Tool for Two-Language Development

Consider the consequences of rewriting an assembly-language module named f in C. The original module has two parts: an assembly-language file named f.inc, specifying the module's interface, and an assembly-language file named f.a96, implementing the module's interface.

Likewise, the new module has two parts: a C file named f.h, specifying the module's interface, and a C file named f.c, implementing the module's interface

However, other assembly language modules still need the assembly-language interface to f. That is, they need to include f.inc in one way or another. One solution is to maintain both f.h and f.inc manually. This solution is tedious and error-prone. A better solution is to maintain f.h manually and automatically translate it into f.inc with a tool. Such a tool needs to be able to translate the subset of C that occurs in a .h file into assembly language.

Two brief definitions are relevant. A ***target assembler*** is a cross-assembler from a development platform to the target platform. Analogously, a ***target compiler*** is a C cross-compiler from a development platform to the target platform. Note that the word size of the compiler with which the translator is developed typically differs from that of a target assembler/compiler pair.

The characteristics of C are fairly standardized, stable, and well-known; those of assembly language are not. Thus, the translator needs to be retargetable, primarily for different target assembler/compiler pairs. Several aspects of a particular target assembler/compiler pair are described below, not because they are interesting in themselves, but because they provide insight into the translation task and help explain the example presented later.

The target assembler has several relevant characteristics. It uses a preprocessor, similar to a C preprocessor, to effect file inclusion, macro definition, and conditional inclusion. It processes two kinds of equate directive, EQU and SET, whose difference is irrelevant here. It processes EXTRN directives, which are analogous to C extern declarations. Symbols and values have types such as: NULL, BYTE, WORD, POINTER, and ENTRY. During assembly, the output for each statement goes to one of several segments that are available (e.g., code, data, stack, or register).

The target compiler has only one relevant characteristic. Its algorithm for determining the distance from the beginning of a struct to a member requires alignment analysis of the types of all members of the struct.

Include-File Translation versus Compilation

Although the translator is much like a C-subset compiler, there are significant differences. For example, the translator generates code from a #define directive, but a compiler does not. On the other hand, a compiler generates code from statements included by a #include directive, but the translator does not.

A more interesting difference concerns error handling. A compiler can stop producing output (i.e., object code) as soon as it detects any nonwarning error. In contrast, the translator must recover from most errors, including syntax errors, and produce as much correct output (i.e., assembly code) as it can. In many cases, translation errors are quite acceptable, representing C declarations that are simply inaccessible to assembly-language programmers.

Translation Details

The include-file translator accepts as input a subset of the C programming language. This subset consists of C modules (also known as translation units[1]) that do not define functions or variables. Such modules are conventionally stored in include files, with a .h file name extension. Accordingly, the translator need not recognize the following reserved words: auto do return break else static case for switch continue goto while default if. However, the translator does understand the reserved words near and far as qualifiers for pointer types.

Like a conventional compiler,[2] the translator's operation can be understood as a sequence of phases (see Fig. 3):

1. Prior-inclusion preprocessing
2. Define directive preprocessing
3. C preprocessing
4. Scanning and parsing
5. Code generation.

As usual, phases 4 and 5 are interleaved. The phases are described below.

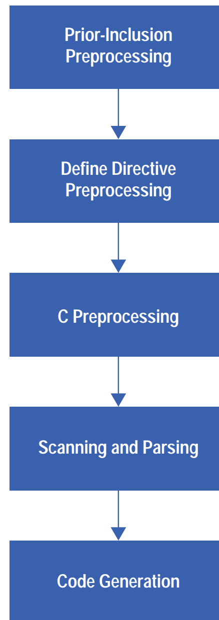

Fig. 3. *Include-file translator operation consists of a series of phases.*

Prior-Inclusion Preprocessor. An include file can reference an identifier declared in another include file, but the former does not necessarily include the latter. The prior-inclusion preprocessor solves this problem.

For example, suppose f.c contains the following:

```
#include "a.h"
#include "f.h"
```

and f.h references an identifier from a.h. To translate f.h, a.h must be processed first. However, declarations from a.h must not produce output.

The prior-inclusion preprocessor solves the problem by constructing a file containing #include directives, #line directives, and the content of the original input file (i.e., f.h). Later phases only produce output for declarations from the original input file.

The prior-inclusion preprocessor is implemented with Awk.[3]

Define Directive Preprocessor. In general, a #define directive in the input produces output. However, C preprocessors delete #define directives. The define directive preprocessor solves this problem.

For example, suppose the following line is line five in file f.h.

```
#define SIZE 100 /* size of something */
```

The define directive preprocessor translates this into the following lines.

```
#define SIZE 100
#line 5 "f.h"
%%% "SIZE" %%% SIZE %%%
```

Since normal C preprocessing has not yet been performed, the original #define directive must be retained.

The #line directive allows error messages from subsequent phases to accurately specify the location of an error. Here, subsequent phases must recognize that a line has been inserted during preprocessing.

The three-character token %%% is effectively a new reserved word. C preprocessing ignores all but the fourth token on this line, which it replaces with the definition of SIZE in the usual way. Phases after C preprocessing recognize this statement and produce output from it corresponding to the original #define directive.

Initially, an @ was used rather than %%% but some C preprocessors complained about it. The probability of a programmer accidently using three adjacent modulus operators seems low. The identifier is used as the fourth token, rather than its definition, to ensure correct operator precedence during evaluation in later phases. If the identifier has no definition, a 1 is used as the fourth token. The line ends with %%% to prevent the parser from skipping too many tokens during error recovery.

The define directive preprocessor is also implemented with Awk.

C Preprocessor. This preprocessor is an ordinary C preprocessor (e.g., cpp). The translator expects the value of an environment variable to specify a particular preprocessor.

Scanner/Parser. This part of the translator is essentially a front end for a normal C compiler. Several differences are:

- Only the subset of C described previously needs to be recognized.
- The new %%% statements are recognized.
- The definition part of a %%% statement is evaluated. Since it can be any C constant integer expression, its evaluation is not trivial. Furthermore, evaluation is simulated to occur according to the word size of the target compiler, which may be different from the word size of the translator, and simulated arithmetic overflow is detected and reported. The alternative to evaluating the expression during translation is to translate it into an equivalent assembly-language expression.
- Most errors, including syntax errors, do not preclude code generation.

The scanner/parser is implemented with Yacc,[4] Lex,[5] and GPerf.[6]

Code Generator. Since the translator only translates declarations, this part is simpler than the back end of a normal C compiler. Its primary task is to output assembly-language directives and declarations, based on the content of a typical symbol table, which is built by the scanner and parser.

To simplify retargeting, target compiler type sizes and type alignment rules are encoded in a tabular fashion. Likewise, target assembler mnemonics and type names are easily modifiable. Code is generated only for declarations from the original input file. Declarations from files it includes produce no output.

Translation Algorithm

The translation from C to assembly language is predominantly syntax-directed. In other words, each syntactic construct in the C subset is translated to a line of assembly language. However, as in a normal compiler, the symbol table provides context. The translated constructs are described in the following paragraphs.

Define Directives. A #define directive, without arguments and whose replacement text is a constant integer expression, is translated to a SET directive. The value is that of the expression, which is evaluated according to the word size of the target compiler. Simulated arithmetic overflow is detected and reported. For example,

```
#define TWO 1+1
```

is translated to:

```
TWO  SET  2:NULL
```

Other kinds of #define directives are not translated.

Enumeration Members. An enum declaration declares at least one member. Each such member is translated to an EQU directive. The value is that which would be assigned to the member by the target compiler. For example,

```
 enum  numbers {
     zero,
     one,
     two,
     ten=5*2,
     eleven,
     twelve
 };
```

is translated to:

```
zero   EQU  0:NULL
one    EQU  1:NULL
two    EQU  2:NULL
ten    EQU  10:NULL
eleven EQU  11:NULL
twelve EQU  12:NULL
```

Structure and Union Members. A struct or union declaration also declares at least one member. Again, each such member is translated to an EQU directive. The value is the offset, in bytes, from the beginning of the nearest enclosing structure or union, according to the target compiler. For example,

```
struct STR {
    double d;
    double *dp;
    int i;
```

```
        char c,*cp,b;
        float f;
    };
```

is translated to:

```
    d    EQU    0:NULL
    dp   EQU    4:NULL
    i    EQU    8:NULL
    c    EQU    10:NULL
    cp   EQU    12:NULL
    b    EQU    16:NULL
    f    EQU    18:NULL
```

Notice that alignment occurs. A command line option causes structure and union member names to be qualified by the nearest enclosing structure or union tag (e.g., the translation of the second member would equate the identifier STR_dp to four).

Nested structure and union declarations force an interesting decision. A member's offset can be computed as the distance from the beginning of its (1) nearest enclosing structure or union or (2) outermost structure or union. Both offsets are valuable to an assembly-language programmer. Both offsets require name qualification to differentiate identical member names in distinct structures or unions. However, offset 2 requires a qualification for each level of nesting, which can quickly exhaust the 31-character length limit for C identifiers. In addition, offset 1 allows a programmer to compute offset 2, as necessary. Therefore, offset 1 is considered the best choice.

Structure and Union Tags. In addition to its members, a struct or union declaration optionally declares a tag. Such a tag is translated to an EQU. The value is the size in bytes of the structure or union, according to the target compiler. For example, for the previous structure, the translation is:

```
    STR    EQU    22:NULL
```

Variable Declarations. A variable declaration (i.e., a variable definition prefixed by extern) is translated to an EXTRN directive. For example,

```
    extern int a[10];
    extern int e;
```

is translated to:

```
    EXTRN    a:POINTER
    EXTRN    e:WORD
```

These directives are output in data segment space. If the C reserved word register is present, they are output in register segment space.

Function Declarations. A function declaration (i.e., a prototype) is also translated to an EXTRN directive. For example,

```
    extern int f(int x, int y);
```

is translated to:

```
    EXTRN    f:ENTRY
```

These directives are output in code segment space.

An Example

This section provides a concrete demonstration of the include-file translator described in earlier sections. Relevant aspects of the translator's interface are presented and an example translation is shown.

The translator recognizes a variety of command-line arguments and environment variables, but most of them are not very interesting. The important command-line arguments are:

- –i specifies the input file
- –o specifies the output file
- –q simulates prior inclusion of a #include "..." file
- –a simulates prior inclusion of a #include <...> file

The only important environment variable is cppcmd, which allows any C preprocessor to be used, rather than the default. Thus, the translator can be executed as follows:

```
    c2as -i cars.h -o cars.inc
```

Suppose cars.h contains the following C code:

```
#define MAKELEN 9
#define CARS 3
typedef enum {black=10,red,blue} Color;
typedef char Make[MAKELEN];
typedef double Price;
typedef struct Car {
    Color color;
    Make make;
    Price price;
    struct Car *oldcars[CARS-1];
} Car;
extern Car *car;
extern Car FixCar(Car car);
```

Then, after translation, cars.inc would contain the following Intel i960 assembly-language code.

```
MAKELEN     SET    9:NULL
CARS        SET    3:NULL
black       EQU    10:NULL
red         EQU    11:NULL
blue        EQU    12:NULL
Car_color   EQU    0:NULL
Car_make    EQU    2:NULL
Car_price   EQU    12:NULL
Car_oldcarsEQU     16:NULL
Car         EQU    20:NULL
            DSEG
            EXTRN  car:POINTER
            CSEG
            EXTRN  FixCar:ENTRY
```

Integration with Build Process

From a build-process perspective, an include-file translator is no different than a compiler: it translates a source file into an intermediate form. Like a compiler, the translator's invocation should be automated by a build process. There are several details, which are described below. A build process based on Make[7] is assumed.

In general, only some of a system's C include files need to be translated. These .h files need to be identified. Typically, a Make variable is used for this purpose.

A single rule is sufficient to tell Make how to translate any of the previously identified .h files into a corresponding .inc assembly-language file. Some versions of Make can use ***static pattern rules*** for this, while others can use ***dynamic prerequisite macros*** to accomplish the same thing. If these Make features are unavailable, multiple rules are required.

Any additional dependencies of each resulting .inc file must be specified explicitly, in two ways. First, Make must be told about the dependencies, but this can be done with ordinary rules. Second, such a dependency may require a translator option specifying prior-inclusion preprocessing, as described above. This second kind of dependency cannot be computed automatically. Fortunately, an esoteric Make feature called ***computed variable names*** can customize translator options for different .h files. Again, if these Make features are unavailable, multiple rules are required.

Conclusion

Include-file translation supports the reuse of modules whose interfaces must be expressed in both a high-level language and an assembly language. These two-view interfaces allow a module to be implemented in either language and referenced by other modules implemented in either language. The basic approach is to maintain the high-level interface manually and automatically derive the low-level interface.

The C include-file translator described above has recently been implemented and incorporated into the software build process used by a group of about 20 firmware engineers at Hewlett-Packard's Disk Memory Division. The translator was implemented for the UNIX and MS-DOS operating systems by one person working half time for about six months. The firmware upon which the translator operates consists of about 275 files occupying over 13 megabytes. About one quarter of these files contain C source code, while the rest contain assembly-language source code. The translator is used on about 30 include files.

For the most part, the experiences and comments of those involved have been positive. Surprisingly, arithmetic-overflow checking caused problems, because the target assembler's word size depends on the complexity of an expression (operator-free expressions can be large). In addition, several users requested an output-file preamble of comments. These comments contain file names, version numbers, and time stamps.

The vast majority of difficulties have been with build process integration rather than translation. Users have had trouble understanding and correctly using the prior-inclusion preprocessor. Typically, a file containing a typedef is omitted, which causes the parser to produce a parse error message. In addition, a .inc file's dependence on its corresponding .h file is not properly recognized by the mechanism responsible for automatically checking out the .h file from the version control system.

Acknowledgments

I wish to acknowledge the contributions made by other members of HP's Disk Memory Division to this project: Michael Banther, especially for his work on the requirements document, Art Beale, Chris Grund, and Steve Folster, for their efforts in the inspections, Ken Hibbs, especially for his thoroughness and patience in the system-test phase, Don Peterson, Dan Martin, Steve King, and Jeff Aguilera, especially for their help in the build process integration phase, Roy Foote and Scott Dehart, my project managers, and of course, the firmware developers who use the tool.

References

1. B. Kernighan and D. Ritchie, ***The C Programming Language***, Prentice Hall, 1988.
2. A. Aho, R. Sethi, and J. Ullman, ***Compilers: Principles, Techniques, and Tools***, Addison-Wesley, 1986.
3. D. Close, A. Robbins, P. Rubin, and R. Stallman, ***The Gawk Manual***, anonymous ftp distribution, 1993.
4. C. Donnelly and R. Stallman, ***Bison***, anonymous ftp distribution, 1992.
5. V. Paxson, ***Using Flex***, anonymous ftp distribution, 1990.
6. D. Schmidt, ***GPerf Manual***, anonymous ftp distribution, 1989.
7. R. Stallman and R. McGrath, ***GNU Make: A program for directing recompilation***, anonymous ftp distribution, 1991.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open is a registered trademark and the X device is a trademark of X/Open Company Limited in the UK and other countries.

MS-DOS is a U.S. registered trademark of Microsoft Corporation.

▶ Go Table of Contents

▶ Go To HP Journal Home Page