--

--

TDF and Portability

Issue 1.0 (June 1993)

-- --

Notice to readers

TDF is a portability technology and an architecture neutral format for expressing software applications which was developed by the United Kingdom's Defence Research Agency (DRA).

Requests for information about TDF should be directed to :

Dr N. E. Peeling
Defence Research Agency
St Andrews Road
Malvern
Worcestershire
United Kingdom WR14 3PS

Fax : +44 684 894303

Internet : peeling%hermes.mod.uk@relay.mod.uk

June 1993

Introduction

TDF is the name of the technology developed at DRA which has been adopted by the Open Software Foundation (OSF), Unix System Laboratories (USL), the European Community's Esprit Programme and others as their Architecture Neutral Distribution Format (ANDF). To date much of the discussion surrounding it has centred on the question, "How do you distribute portable software?". This paper concentrates on the more difficult question, "How do you write portable software in the first place?" and shows how TDF can be a valuable tool to aid the writing of portable software. Most of the discussion centres on programs written in C and is Unix specific. This is because most of the experience of TDF to date has been in connection with C in a Unix environment, and not because of any inbuilt bias in TDF.

It is assumed that the reader is familiar with the ANDF concept (although not necessarily with the details of TDF), and with the problems involved in writing portable C code.

The discussion is divided into two sections. Firstly some of the problems involved in writing portable programs are considered. The intention is not only to catalogue what these problems are, but to introduce ways of looking at them which will be important in the second section. This deals with the TDF approach to portability[1].

Part I : Portable Programs

We start by examining some of the problems involved in the writing of portable programs. Although the discussion is very general, and makes no mention of TDF, many of the ideas introduced are of importance in the second half of the paper, which deals with TDF.

1. Portable Programs

1.1. Definitions and Preliminary Discussion

Let us firstly say what we mean by a portable program. A program is *portable* to a number of machines if it can be compiled to give the same functionality on all those machines. Note that this does not mean that exactly the same source code is used on all the machines. One could envisage a program written in, say, 68020 assembly code for a certain machine which has been translated into 80386 assembly code for some other machine to give a program with exactly equivalent functionality. This would, under our definition, be a program which is portable to these two machines. At the other end of the scale, the C program :

```
#include <stdio.h>

int main ()
{
	fputs ( "Hello world\n", stdout ) ;
	return ( 0 ) ;
}
```

which prints the message, "Hello world", onto the standard output stream, will be portable to a vast range of machines without any need for rewriting. Most of the portable programs we shall be considering fall closer to the latter end of the spectrum - they will largely consist of target independent source with small sections of target dependent source for those constructs for which target independent expression is either impossible or of inadequate efficiency.

Note that we are defining portability in terms of a set of target machines and not as some universal property. The act of modifying an existing program to make it portable to a new target machine is called *porting*. Clearly in the examples above, porting the first program would be a highly complex task involving almost an entire rewrite, whereas in the second case it should be trivial.

[1] Note to the June 1993 revision : The description of the TDF system in this paper reflects a very slightly idealised version of the current technology; one in which all the features we intend to put in, but have not had time to, are included. I have tried to indicate the small number of features described which are not currently implemented, and those which are "future directions". However the documentation on the individual components of the TDF system should be consulted for a precise description of the current technology.

1.2. Separation and Combination of Code

So why is the second example above more portable (in the sense of more easily ported to a new machine) than the first? The first, obvious, point to be made is that it is written in a high-level language, C, rather than the low-level languages, 68020 and 80386 assembly codes, used in the first example. By using a high-level language we have abstracted out the details of the processor to be used and expressed the program in an architecture neutral form. It is one of the jobs of the compiler on the target machine to transform this high-level representation into the appropriate machine dependent low-level representation.

The second point is that the second example program is not in itself complete. The objects *fputs* and *stdout*, representing the procedure to output a string and the standard output stream respectively, are left undefined. Instead the header *stdio.h* is included on the understanding that it contains the specification of these objects.

A version of this file is to be found on each target machine. On a particular machine it might contain something like :

```
typedef struct {
        int __cnt ;
        unsigned char *__ptr ;
        unsigned char *__base ;
        short __flag ;
        char __file ;
} FILE ;

extern FILE __iob [60] ;
#define stdout ( &__iob [1] )

extern int fputs ( const char *, FILE * ) ;
```

meaning that the type *FILE* is defined by the given structure, *__iob* is an external array of 60 *FILE*'s, *stdout* is a pointer to the second element of this array, and that *fputs* is an external procedure which takes a *const char* * and a *FILE* * and returns an *int*. On a different machine, the details may be different (exactly what we can, or cannot, assume is the same on all target machines is discussed below).

These details are fed into the program by the pre-processing phase of the compiler. (The various compilation phases are discussed in more detail later - see Fig. 1.) This is a simple, preliminary textual substitution. It provides the definitions of the type *FILE* and the value *stdout* (in terms of *__iob*), but still leaves the precise definitions of *__iob* and *fputs* still unresolved (although we do know their types). The definitions of these values are not provided until the final phase of the compilation - linking - where they are linked in from the precompiled system libraries.

Note that, even after the pre-processing phase, our portable program has been transformed into a target dependent form, because of the substitution of the target dependent values from *stdio.h*. If we had also included the definitions of *__iob* and, more particularly, *fputs*, things would have been even worse - the procedure for outputting a string to the screen is likely to be highly target dependent.

To conclude, we have, by including *stdio.h*, been able to effectively separate the target independent part of our program (the main program) from the target dependent part (the details of *stdout* and *fputs*). It is one of the jobs of the compiler to recombine these parts to produce a complete program.

1.3. Application Programming Interfaces

As we have seen, the separation of the target dependent sections of a program into the system headers and system libraries greatly facilitates the construction of portable programs. What has been done is to define an interface between the main program and the existing operating system on the target machine in abstract terms. The program should then be portable to any machine which implements this interface correctly.

The interface for the "Hello world" program above might be described as follows : defined in the header *stdio.h* are a type *FILE* representing a file, an object *stdout* of type *FILE* * representing the standard output file, and a procedure *fputs* with prototype :

```
int fputs ( const char *s, FILE *f ) ;
```

which prints the string *s* to the file *f*. This is an example of an *Application Programming Interface (API)*. Note that it can be split into two aspects, the syntactic (what they are) and the semantic (what they mean). On any machine which implements this API our program is both syntactically correct and does what we expect it to.

The benefit of describing the API at this fairly high level is that it leaves scope for a range of implementation (and thus more machines which implement it) while still encapsulating the main program's requirements.

In the example implementation of *stdio.h* above we see that this machine implements this API correctly syntactically, but not necessarily semantically. One would have to read the documentation provided on the system to be sure of the semantics.

Another way of defining an API for this program would be to note that the given API is a subset of the ANSI C standard. Thus we could take ANSI C as an "off the shelf" API. It is then clear that our program should be portable to any ANSI-compliant machine.

It is worth emphasising that all programs have an API, even if it is implicit rather than explicit. However it is probably fair to say that programs without an explicit API are only portable by accident. We shall have more to say on this subject later.

1.4. Compilation Phases

The general plan for how to write the extreme example of a portable program, namely one which contains no target dependent code, is now clear. It is shown in the compilation diagram in Fig. 1[2] which represents the traditional compilation process. This diagram is divided into four sections. The left half of the diagram represents the actual program and the right half the associated API. The top half of the diagram represents target independent material - things which only need to be done once - and the bottom half target dependent material - things which need to be done on every target machine.

So, we write our target independent program (top left), conforming to the target independent API specification (top right). All the compilation actually takes place on the target machine. This machine must have the API correctly implemented (bottom right). This implementation will in general be in two parts - the system headers, providing type definitions, macros, procedure prototypes and so on, and the system libraries, providing the actual procedure definitions. Another way of characterising this division is between syntax (the system headers) and semantics (the system libraries).

The compilation is divided into three main phases. Firstly the system headers are inserted into the program by the pre-processor. This produces, in effect, a target dependent version of the original program. This is then compiled into a binary object file. During the compilation process the compiler inserts all the information it has about the machine - including the *Application Binary Interface (ABI)* - the sizes of the basic C types, how they are combined into compound types, the system procedure calling conventions and so on. This ensures that in the final linking phase the binary object file and the system libraries are obeying the same ABI, thereby producing a valid executable. (On a dynamically linked system this final linking phase takes place partially at run time rather than at compile time, but this does not really affect the general scheme.)

The compilation scheme just described consists of a series of phases of two types ; code combination (the pre-processing and system linking phases) and code transformation (the actual compilation phases). The existence of the combination phases allows for the effective separation of the target independent code (in this case, the whole program) from the target dependent code (in this case, the API implementation), thereby aiding the construction of portable programs. These ideas on the separation, combination and transformation of code underlie the TDF approach to portability.

[2] All the figures are at the end of the paper.

2. Portability Problems

We have set out a scheme whereby it should be possible to write portable programs with a minimum of difficulties. So why, in reality, does it cause so many problems? Recall that we are still primarily concerned with programs which contain no target dependent code, although most of the points raised apply by extension to all programs.

2.1. Programming Problems

A first, obvious class of problems concern the program itself. It is to be assumed that as many bugs as possible have been eliminated by testing and debugging on at least one platform before a program is considered as a candidate for being a portable program. But for even the most self-contained program, working on one platform is no guarantee of working on another. The program may use undefined behaviour - using uninitialised values or dereferencing null pointers, for example - or have built-in assumptions about the target machine - whether it is big-endian or little-endian, or what the sizes of the basic integer types are, for example. This latter point is going to become increasingly important over the next couple of years as 64-bit architectures begin to be introduced. How many existing programs implicitly assume a 32-bit architecture?

Many of thes built-in assumptions may arise because of the conventional porting process. A program is written on one machine, modified slightly to make it work on a second machine, and so on. This means that the program is "biased" towards the existing set of target machines, and most particularly to the original machine it was written on. This applies not only to assumptions about endianness, say, but also to the questions of API conformance which we will be discussing below.

Most compilers will pick up some of the grosser programming errors, particularly by type checking (including procedure arguments if prototypes are used). Some of the subtler errors can be detected using the *-Wall* option to the Free Software Foundation's GNU C Compiler (*gcc*) or separate program checking tools such as *lint*[3], for example, but this remains a very difficult area.

2.2. Code Transformation Problems

We now move on from programming problems to compilation problems. As we mentioned above, compilation may be regarded as a series of phases of two types : combination and transformation. Transformation of code - translating a program in one form into an equivalent program in another form - may lead to a variety of problems. The code may be transformed wrongly, so that the equivalence is broken (a compiler bug), or in an unexpected manner (differing compiler interpretations), or not at all, because it is not recognised as legitimate code (a compiler limitation). The latter two problems are most likely when the input is a high level language, with complex syntax and semantics.

Note that in Fig. 1 all the actual compilation takes place on the target machine. So, to port the program to *n* machines, we need to deal with the bugs and limitations of *n*, potentially different, compilers. For example, if you have written your program using prototypes, it is going to be a large and rather tedious job porting it to a compiler which does not have prototypes (this particular example can be automated; not all such jobs can). Other compiler limitations can be surprising - not understanding the *L* suffix for long numeric literals and not allowing members of enumeration types as array indexes are among the problems drawn from my personal experience.

The differing compiler interpretations may be more subtle. For example, there are differences between ANSI and "traditional" C which may trap the unwary. Examples are the promotion of integral types and the resolution of the linkage of static objects.

Many of these problems may be reduced by using the "same" compiler on all the target machines. For example, *gcc* has a single front end (C $\rightarrow$ RTL) which may be combined with an appropriate back end (RTL $\rightarrow$ target) to form a suitable compiler for a wide range of target machines. The existence of a single front end virtually eliminates the problems of differing interpretation of code and compiler quirks. It also reduces the exposure to bugs. Instead of being exposed to the bugs in *n* separate compilers, we are now

[3] Most versions of *lint* one comes across are not ANSI C compliant. Their support for library routines also tends to be erratic.

only exposed to bugs in one half-compiler (the front end) plus *n* half-compilers (the back ends) - a total of $(n + 1) / 2$. (This calculation is not meant totally seriously, but it is true in principle.) Front end bugs, when tracked down, also only require a single workaround.

2.3. Code Combination Problems

If code transformation problems may be regarded as a time consuming irritation, involving the rewriting of sections of code or using a different compiler, the second class of problems, those concerned with the combination of code, are far more serious.

The first code combination phase is the pre-processor pulling in the system headers. These can contain some nasty surprises. For example, consider a simple ANSI compliant program which contains a linked list of strings arranged in alphabetical order. This might also contain a routine :

```
void index ( char * ) ;
```

which adds a string to this list in the appropriate position, using *strcmp* from *string.h* to find it. This works fine on most machines, but on some it gives the error :

Only 1 argument to macro 'index'

The reason for this is that the system version of *string.h* contains the line :

```
#define index ( s, c )      strchr ( s, c )
```

But this is nothing to do with ANSI, this macro is defined for compatibility with BSD.

In reality the system headers on any given machine are a hodge podge of implementations of different APIs, and it is often virtually impossible to separate them (feature test macros such as *_POSIX_SOURCE* are of some use, but are not always implemented and do not always produce a complete separation; they are only provided for "standard" APIs anyway). The problem above arose because there is no transitivity rule of the form : if program *P* conforms to API *A*, and API *B* extends *A*, then *P* conforms to *B*. The only reason this is not true is these namespace problems.

A second example demonstrates a slightly different point. The POSIX standard states that *sys/stat.h* contains the definition of the structure *struct stat*, which includes several members, amongst them :

```
time_t st_atime ;
```

representing the access time for the corresponding file. So the program :

```
#include <sys/types.h>
#include <sys/stat.h>

time_t st_atime ( struct stat *p )
{
        return ( p->st_atime ) ;
}
```

should be perfectly valid - the procedure name *st_atime* and the field selector *st_atime* occupy different namespaces[4]. However at least one popular operating system has the implementation :

```
struct stat {
        ....
        union {
                time_t st__sec ;
                timestruc_t st__tim ;
        } st_atim ;
        ....
} ;
#define st_atime    st_atim.st__sec
```

[4] See however the appendix on namespaces and APIs below.

This seems like a perfectly legitimate implementation. In the program above the field selector *st_atime* is replaced by *st_atim.st__sec* by the pre-processor, as intended, but unfortunately so is the procedure name *st_atime*, leading to a syntax error.

The problem here is not with the program or the implementation, but in the way they were combined. C does not allow individual field selectors to be defined. Instead the indiscriminate sledgehammer of macro substitution was used, leading to the problem described.

Problems can also occur in the other combination phase of the traditional compilation scheme, the system linking. Consider the ANSI compliant routine :

```
#include <stdio.h>

int open ( char *nm )
{
        int c, n = 0 ;
        FILE *f = fopen ( nm, "r" ) ;
        if ( f == NULL ) return ( -1 ) ;
        while ( c = getc ( f ), c != EOF ) n++ ;
        ( void ) fclose ( f ) ;
        return ( n ) ;
}
```

which opens the file *nm*, returning its size in bytes if it exists and −1 otherwise. As a quick porting exercise, I compiled it under six different operating systems. On three it worked correctly; on one it returned −1 even when the file existed; and on two it crashed with a segmentation error.

The reason for this lies in the system linking. On those machines which failed the library routine *fopen* calls (either directly or indirectly) the library routine *open* (which is in POSIX, but not ANSI). The system linker, however, linked my routine *open* instead of the system version, so the call to *fopen* did not work correctly.

So code combination problems are primarily namespace problems. The task of combining the program with the API implementation on a given platform is complicated by the fact that, because the system headers and system libraries contain things other than the API implementation, or even because of the particular implementation chosen, the various namespaces in which the program is expected to operate become "polluted".

2.4. API Problems

We have said that the API defines the interface between the program and the standard library provided with the operating system on the target machine. There are three main problems concerned with APIs. The first, how to choose the API in the first place, is discussed separately. Here we deal with the compilation aspects : how to check that the program conforms to its API, and what to do about incorrect API implementations on the target machine(s).

2.4.1. API Checking

The problem of whether or not a program conforms to its API - not using any objects from the operating system other than those specified in the API, and not making any unwarranted assumptions about these objects - is one which does not always receive sufficient attention, mostly because the necessary checking tools do not exist (or at least are not widely available). Compiling the program on a number of API compliant machines merely checks the program against the system headers for these machines. For a genuine portability check we need to check against the abstract API description, thereby in effect checking against all possible implementations.

Recall from above that the system headers on a given machine are an amalgam of all the APIs it implements. This can cause programs which should compile not to, because of namespace clashes; but it may also cause programs to compile which should not, because they have used objects which are not in their API, but which are in the system headers. For example, the supposedly ANSI compliant program :

```
#include <signal.h>
int sig = SIGKILL ;
```

will compile on most systems, despite the fact that *SIGKILL* is not an ANSI signal, because *SIGKILL* is in POSIX, which is also implemented in the system *signal.h*. Again, feature test macros are of some use in trying to isolate the implementation of a single API from the rest of the system headers. However they are highly unlikely to detect the error in the following supposedly POSIX compliant program which prints the entries of the directory *nm*, together with their inode numbers :

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

void listdir ( char *nm )
{
        struct dirent *entry ;
        DIR *dir = opendir ( nm ) ;
        if ( dir == NULL ) return ;
        while ( entry = readdir ( dir ), entry != NULL ) {
                printf ( "%s : %d\n", entry->d_name, ( int ) entry->d_ino ) ;
        }
        ( void ) closedir ( dir ) ;
        return ;
}
```

This is not POSIX compliant because, whereas the *d_name* field of *struct dirent* is in POSIX, the *d_ino* field is not. It is however in XPG3, so it is likely to be in many system implementations.

The previous examples have been concerned with simply telling whether or not a particular object is in an API. A more difficult, and in a way more important, problem is that of assuming too much about the objects which are in the API. For example, in the program :

```
#include <stdio.h>
#include <stdlib.h>

div_t d = { 3, 4 } ;

int main ()
{
        printf ( "%d,%d\n", d.quot, d.rem ) ;
        return ( 0 ) ;
}
```

the ANSI standard specifies that the type *div_t* is a structure containing two fields, *quot* and *rem*, of type *int*, but it does not specify which order these fields appear in, or indeed if there are other fields. Therefore the initialisation of *d* is not portable. Again, the type *time_t* is used to represent times in seconds since a certain fixed date. On most systems this is implemented as *long*, so it is tempting to use *(t & 1)* to determine for a *time_t t* whether this number of seconds is odd or even. But ANSI actually says that *time_t* is an arithmetic, not an integer, type, so it would be possible for it to be implemented as *double*. But in this case *(t & 1)* is not even type correct, so it is not a portable way of finding out whether *t* is odd or even.

2.4.2. API Implementation Errors

Undoubtedly the problem which causes the writer of portable programs the greatest headache (and heartache) is that of incorrect API implementations. However carefully you have chosen your API and checked that your program conforms to it, you are still reliant on someone (usually the system vendor) having implemented this API correctly on the target machine. Machines which do not implement the API at all do not enter the equation (they are not suitable target machines), what causes problems is incorrect implementations. As the implementation may be divided into two parts - system headers and system libraries - we shall similarly divide our discussion. Inevitably the choice of examples is personal; anyone who has ever attempted to port a program to a new machine is likely to have their own favourite examples.

2.4.2.1. System Header Problems

Some header problems are immediately apparent because they are syntactic and cause the program to fail to compile. For example, values may not be defined or be defined in the wrong place (not in the header prescribed by the API).

A common example (one which I have to include a workaround for in virtually every program I write) is that *EXIT_SUCCESS* and *EXIT_FAILURE* are not always defined (ANSI specifies that they should be in *stdlib.h*). It is tempting to change *exit (EXIT_FAILURE)* to *exit (1)* because "everyone knows" that *EXIT_FAILURE* is 1. But this is to decrease the portability of the program because it ties it to a particular class of implementations. A better workaround would be :

```
#include <stdlib.h>
#ifndef EXIT_FAILURE
#define EXIT_FAILURE 1
#endif
```

which assumes that anyone choosing a non-standard value for *EXIT_FAILURE* is more likely to put it in *stdlib.h*. Of course, if one subsequently came across a machine on which not only is *EXIT_FAILURE* not defined, but also the value it should have is not 1, then it would be necessary to resort to *#ifdef machine_name* statements. The same is true of all the API implementation problems we shall be discussing : non-conformant machines require workarounds involving conditional compilation. As more machines are considered, so these conditional compilations multiply.

As an example of things being defined in the wrong place, ANSI specifies that *SEEK_SET*, *SEEK_CUR* and *SEEK_END* should be defined in *stdio.h*, whereas POSIX specifies that they should also be defined in *unistd.h*. It is not uncommon to find machines on which they are defined in the latter but not in the former. A possible workaround in this case would be :

```
#include <stdio.h>
#ifndef SEEK_SET
#include <unistd.h>
#endif
```

Of course, by including "unnecessary" headers like *unistd.h* the risk of namespace clashes such as those discussed above is increased.

A final syntactic problem, which perhaps should belong with the system header problems above, concerns dependencies between the headers themselves. For example, the POSIX header *unistd.h* declares functions involving some of the types *pid_t*, *uid_t* etc, defined in *sys/types.h*. Is it necessary to include *sys/types.h* before including *unistd.h*, or does *unistd.h* automatically include *sys/types.h*? The approach of playing safe and including everything will normally work, but this can lead to multiple inclusions of a header. This will normally cause no problems because the system headers are protected against multiple inclusions by means of macros, but it is not unknown for certain headers to be left unprotected. Also not all header dependencies are as clear cut as the one given, so that what headers need to be included, and in what order, is in fact target dependent.

There can also be semantic errors in the system headers : namely wrongly defined values. The following two examples are taken from real operating systems. Firstly the definition :

```
#define DBL_MAX 1.797693134862316E+308
```

in *float.h* on an IEEE-compliant machine is subtly wrong - the given value does not fit into a *double* - the correct value is :

```
#define DBL_MAX 1.7976931348623157E+308
```

Again, the type definition :

```
typedef int size_t ; /* ??? */
```

(sic) is not compliant with ANSI, which says that *size_t* is an unsigned integer type[5]. These particular examples are irritating because it would have cost nothing to get things right, correcting the value of *DBL_MAX* and changing the definition of *size_t* to *unsigned int*. These corrections are so minor that the modified system headers would still be a valid interface for the existing system libraries (we shall have more to say about this later). However it is not possible to change the system headers, so it is necessary to build workarounds into the program. Whereas in the first case it is possible to devise such a workaround :

```
#include <float.h>
#ifdef machine_name
#undef DBL_MAX
#define DBL_MAX 1.7976931348623157E+308
#endif
```

for example, in the second, because *size_t* is defined by a *typedef* it is virtually impossible to correct in a simple fashion. Thus any program which relies on the fact that *size_t* is unsigned will require considerable rewriting before it can be ported to this machine.

2.4.2.2. System Library Problems

The system header problems just discussed are primarily syntactic problems. By contrast, system library problems are primarily semantic - the provided library routines do not behave in the way specified by the API. This makes them harder to detect. For example, consider the routine :

```
void *realloc ( void *p, size_t s ) ;
```

which reallocates the block of memory *p* to have size *s* bytes, returning the new block of memory. The ANSI standard says that if *p* is the null pointer, then the effect of *realloc (p, s)* is the same as *malloc (s)*, that is, to allocate a new block of memory of size *s*. This behaviour is exploited in the following program, in which the routine *add_char* adds a character to the expanding array, *buffer* :

```
#include <stdio.h>
#include <stdlib.h>

char *buffer = NULL ;
int buff_sz = 0, buff_posn = 0 ;

void add_char ( char c )
{
	if ( buff_posn >= buff_sz ) {
		buff_sz += 100 ;
		buffer = ( char * ) realloc ( ( void * ) buffer, buff_sz * sizeof ( char ) ) ;
		if ( buffer == NULL ) {
			fprintf ( stderr, "Memory allocation error\n" ) ;
			exit ( EXIT_FAILURE ) ;
		}
	}
	buffer [ buff_posn++ ] = c ;
	return ;
}
```

On the first call of *add_char*, *buffer* is set to a real block of memory (as opposed to *NULL*) by a call of the form *realloc (NULL, s)*. This is extremely convenient and efficient - if it was not for this behaviour we would have to have an explicit initialisation of *buffer*, either as a special case in *add_char* or in a separate initialisation routine.

---

[5] I'm not sure if this is better or worse than another system which defines *ptrdiff_t* to be *unsigned int* when it is meant to be signed. This would mean that the difference between any two pointers is always positive.

Of course this all depends on the behaviour of *realloc (NULL, s)* having been implemented precisely as described in the ANSI standard. The first indication that this is not so on a particular target machine might be when the program is compiled and run on that machine for the first time and does not perform as expected. To track the problem down will demand time debugging the program.

Once the problem has been identified as being with *realloc* a number of possible workarounds are possible. Perhaps the most interesting is to replace the inclusion of *stdlib.h* by the following :

```
#include <stdlib.h>
#ifdef machine_name
#define realloc ( p, s )    ( ( p ) ? ( realloc ) ( p, s ) : malloc ( s ) )
#endif
```

where *realloc (p, s)* is redefined as a macro which is the result of the procedure *realloc* if *p* is not null, and *malloc (s)* otherwise[6].

The only alternative to this trial and error approach to finding API implementation problems is the application of personal experience, either of the particular target machine or of things that are implemented wrongly by many machines and as such should be avoided. This sort of detailed knowledge is not easily acquired. Nor can it ever be complete : new operating system releases are becoming increasingly regular and are on occasions quite as likely to introduce new implementation errors as to solve existing ones. It is in short a "black art".

3. APIs and Portability

We now return to our discussion of the general issues involved in portability to more closely examine the role of the API.

3.1. Target Dependent Code

So far we have been considering programs which contain no conditional compilation, in which the API forms the basis of the separation of the target independent code (the whole program) and the target dependent code (the API implementation). But a glance at most large C programs will reveal that they do contain conditional compilation. The code is scattered with *#if*'s and *#ifdef*'s which, in effect, cause the preprocessor to construct slightly different programs on different target machines. So here we do not have a clean division between the target independent and the target dependent code - there are small sections of target dependent code spread throughout the program.

Let us briefly consider some of the reasons why it is necessary to introduce this conditional compilation. Some have already been mentioned - workarounds for compiler bugs, compiler limitations, and API implementation errors; others will be considered later. However the most interesting and important cases concern things which need to be done genuinely differently on different machines. This can be because they really cannot be expressed in a target independent manner, or because the target independent way of doing them is unacceptably inefficient.

Efficiency (either in terms of time or space) is a key issue in many programs. The argument is often advanced that writing a program portably means using the, often inefficient, lowest common denominator approach. But under our definition of portability it is the functionality that matters, not the actual source code. There is nothing to stop different code being used on different machines for reasons of efficiency.

To examine the relationship between target dependent code and APIs, consider the simple program :

```
#include <stdio.h>

int main ()
{
#ifdef mips
        fputs ( "This machine is a mips\n", stdout ) ;
#endif
```

[6] In fact this macro will not always have the desired effect (although it does in this case). Why (exercise)?

```
        return ( 0 ) ;
}
```

which prints a message if the target machine is a mips. What is the API of this program? Basically it is the same as in the "Hello world" example discussed in sections I.1.1 and I.1.3, but if we wish the API to fully describe the interface between the program and the target machine, we must also say that whether or not the macro *mips* is defined is part of the API. Like the rest of the API, this has a semantic aspect as well as a syntactic - in this case that *mips* is only defined on mips machines. Where it differs is in its implementation. Whereas the main part of the API is implemented in the system headers and the system libraries, the implementation of either defining, or not defining, *mips* ultimately rests with the person performing the compilation. (In this particular example, the macro *mips* is normally built into the compiler on mips machines, but this is only a convention.)

So the API in this case has two components : a system-defined part which is implemented in the system headers and system libraries, and a user-defined part which ultimately relies on the person performing the compilation to provide an implementation. The main point to be made in this section is that introducing target dependent code is equivalent to introducing a user-defined component to the API. The actual compilation process in the case of programs containing target dependent code is basically the same as that shown in Fig. 1. But whereas previously the vertical division of the diagram also reflects a division of responsibility - the left hand side is the responsibility of the programmer (the person writing the program), and the right hand side of the API specifier (for example, a standards defining body) and the API implementor (the system vendor) - now the right hand side is partially the responsibility of the programmer and the person performing the compilation. The programmer specifies the user-defined component of the API, and the person compiling the program either implements this API (as in the mips example above) or chooses between a number of alternative implementations provided by the programmer (as in the example below).

Let us consider a more complex example. Consider the following program which assumes, for simplicity, that an *unsigned int* contains 32 bits :

```
#include <stdio.h>
#include "config.h"

#ifndef SLOW_SHIFT
#define MSB ( a )        ( ( unsigned char ) ( a >> 24 ) )
#else
#ifdef BIG_ENDIAN
#define MSB ( a )        *( ( unsigned char * ) &( a ) )
#else
#define MSB ( a )        *( ( unsigned char * ) &( a ) + 3 )
#endif
#endif

unsigned int x = 100000000 ;

int main ()
{
        printf ( "%u\n", MSB ( x ) ) ;
        return ( 0 ) ;
}
```

The intention is to print the most significant byte of *x*. Three alternative definitions of the macro *MSB* used to extract this value are provided. The first, if *SLOW_SHIFT* is not defined, is simply to shift the value right by 24 bits. This will work on all 32-bit machines, but may be inefficient (depending on the nature of the machine's shift instruction). So two alternatives are provided. An *unsigned int* is assumed to consist of four *unsigned char*'s. On a big-endian machine, the most significant byte is the first of these *unsigned char*'s; on a little-endian machine it is the fourth. The second definition of *MSB* is intended to reflect the former case, and the third the latter.

The person compiling the program has to choose between the three possible implementations of *MSB* provided by the programmer. This is done by either defining, or not defining, the macros *SLOW_SHIFT* and *BIG_ENDIAN*. This could be done as command line options, but we have chosen to reflect another commonly used device, the configuration file. For each target machine, the programmer provides a version of the file *config.h* which defines the appropriate combination of the macros *SLOW_SHIFT* and *BIG_ENDIAN*. The person performing the compilation simply chooses the appropriate *config.h* for the target machine.

There are two possible ways of looking at what the user-defined API of this program is. Possibly it is most natural to say that it is *MSB*, but it could also be argued that it is the macros *SLOW_SHIFT* and *BIG_ENDIAN*. The former more accurately describes the target dependent code, but is only implemented indirectly, via the latter.

3.2. Making APIs Explicit

As we have said, every program has an API even if it is implicit rather than explicit. Every system header included, every type or value used from it, and every library routine used, adds to the system-defined component of the API, and every conditional compilation adds to the user-defined component. What making the API explicit does is to encapsulate the set of requirements that the program has of the target machine (including requirements like, I need to know whether or not the target machine is big-endian, as well as, I need *fputs* to be implemented as in the ANSI standard). By making these requirements explicit it is made absolutely clear what is needed on a target machine if a program is to be ported to it. If the requirements are not explicit this can only be found by trial and error. This is what we meant earlier by saying that a program without an explicit API is only portable by accident.

Another advantage of specifying the requirements of a program is that it may increase their chances of being implemented. We have spoken as if porting is a one-way process; program writers porting their programs to new machines. But there is also traffic the other way. Machine vendors may wish certain programs to be ported to their machines. If these programs come with a list of requirements then the vendor knows precisely what to implement in order to make such a port possible.

3.3. Choosing an API

So how does one go about choosing an API? In a sense the user-defined component is easier to specify that the system-defined component because it is less tied to particular implementation models. What is required is to abstract out what exactly needs to be done in a target dependent manner and to decide how best to separate it out. The most difficult problem is how to make the implementation of this API as simple as possible for the person performing the compilation, if necessary providing a number of alternative implementations to choose between and a simple method of making this choice (for example, the *config.h* file above). With the system-defined component the question is more likely to be, how do the various target machines I have in mind implement what I want to do? The abstraction of this is usually to choose a standard and widely implemented API, such as POSIX, which provides all the necessary functionality.

The choice of "standard" API is of course influenced by the type of target machines one has in mind. Within the Unix world, the increasing adoption of Open Standards, such as POSIX, means that choosing a standard API which is implemented on a wide variety Unix boxes is becoming easier. Similarly, choosing an API which will work on most MSDOS machines should cause few problems. The difficulty is that these are disjoint worlds; it is very difficult to find a standard API which is implemented on both Unix and MSDOS machines. At present not much can be done about this, it reflects the disjoint nature of the computer market.

To develop a similar point : the drawback of choosing POSIX (for example) as an API is that it restricts the range of possible target machines to machines which implement POSIX. Other machines, for example, BSD compliant machines, might offer the same functionality (albeit using different methods), so they should be potential target machines, but they have been excluded by the choice of API. One approach to the problem is the "alternative API" approach. Both the POSIX and the BSD variants are built into the program, but only one is selected on any given target machine by means of conditional compilation. Under our "equivalent functionality" definition of portability, this is a program which is portable to both POSIX and BSD compliant machines. But viewed in the light of the discussion above, if we regard a program as a program-API pair, it could be regarded as two separate programs combined on a single source code tree. A more interesting approach would be to try to abstract out what exactly the functionality which both POSIX and BSD offer is and use that as the API. Then instead of two separate APIs we would have a single API with two broad classes of implementations. The advantage of this latter approach becomes clear if wished to port the program to a machine which implements neither POSIX nor BSD, but provides the equivalent functionality in a third way.

As a simple example, both POSIX and BSD provide very similar methods for scanning the entries of a directory. The main difference is that the POSIX version is defined in *dirent.h* and uses a structure called *struct dirent*, whereas the BSD version is defined in *sys/dir.h* and calls the corresponding structure *struct direct*. The actual routines for manipulating directories are the same in both cases. So the only abstraction required to unify these two APIs is to introduce an abstract type, *dir_entry* say, which can be defined by :

typedef struct dirent dir_entry ;

on POSIX machines, and :

typedef struct direct dir_entry ;

on BSD machines. Note how this portion of the API crosses the system-user boundary. The object *dir_entry* is defined in terms of the objects in the system headers, but the precise definition depends on a user-defined value (whether the target machine implements POSIX or BSD).

3.4. Alternative Program Versions

Another reason for introducing conditional compilation which relates to APIs is the desire to combine several programs, or versions of programs, on a single source tree. There are several cases to be distinguished between. The reuse of code between genuinely different programs does not really enter the argument : any given program will only use one route through the source tree, so there is no real conditional compilation per se in the program. What is more interesting is the use of conditional compilation to combine several versions of the same program on the same source tree to provide additional or alternative features.

It could be argued that the macros (or whatever) used to select between the various versions of the program are just part of the user-defined API as before. But consider a simple program which reads in some numerical input, say, processes it, and prints the results. This might, for example, have POSIX as its API. We may wish to optionally enhance this by displaying the results graphically rather than textually on machines which have X Windows, the compilation being conditional on some boolean value, *HAVE_X_WINDOWS*, say. What is the API of the resultant program? The answer from the point of view of the program is the union of POSIX, X Windows and the user-defined value *HAVE_X_WINDOWS*. But from the implementation point of view we can either implement POSIX and set *HAVE_X_WINDOWS* to false, or implement both POSIX and X Windows and set *HAVE_X_WINDOWS* to true. So what introducing *HAVE_X_WINDOWS* does is to allow flexibility in the API implementation.

This is very similar to the alternative APIs discussed above. However the approach outlined will really only work for optional API extensions. To work in the alternative API case, we would need to have the union of POSIX, BSD and a boolean value, say, as the API. Although this is possible in theory, it is likely to lead to namespace clashes between POSIX and BSD.

Part II : TDF

Having discussed many of the problems involved with writing portable programs, we now eventually turn to TDF. Firstly a brief technical overview is given, indicating those features of TDF which facilitate the separation of program. Secondly the TDF compilation scheme is described. It is shown how the features of TDF are exploited to aid in the separation of target independent and target dependent code which we have indicated as characterising portable programs. Finally, the various constituents of this scheme are considered individually, and their particular roles are described in more detail.

1. Features of TDF

It is not the purpose of this paper to explain the exact specification of TDF - this is described elsewhere[7] - but rather to show how its general design features make it suitable as an aid to writing portable programs.

TDF is an abstraction of high-level languages - it contains such things as *exps* (abstractions of expressions and statements), *shapes* (abstractions of types) and *tags* (abstractions of variable identifiers). In general form it is an abstract syntax tree which is flattened and encoded as a series of bits, called a *capsule*. This fairly high level of definition (for a compiler intermediate language) means that TDF is architecture neutral in the sense that it makes no assumptions about the underlying processor architecture.

The translation of a capsule to and from the corresponding syntax tree is totally unambiguous, also TDF has a "universal" semantic interpretation as defined in the TDF specification.

1.1. Capsule Structure

A TDF capsule consists of a number of units of various types. These are embedded in a general linkage scheme (see Fig. 3). Each unit contains a number of variable objects of various sorts (for example, tags and tokens) which are potentially visible to other units. Within the unit body each variable object is identified by a unique number. The linking is via a set of variable objects which are global to the entire capsule. These may in turn be associated with external names. For example, in Fig. 3, the fourth variable of the first unit is identified with the first variable of the third unit, and both are associated with the fourth external name.

This capsule structure means that the combination of a number of capsules to form a single capsule is a very natural operation. The actual units are copied unchanged into the resultant capsule - it is only the surrounding linking information that needs changing. Many criteria could be used to determine how this linking is to be organised, but the simplest is to link two objects if and only if they have the same external name. This is the scheme that the current TDF linker has implemented. Furthermore such operations as changing an external name or removing it altogether ("hiding") are very simple under this linking scheme.

1.2. Tokens

So, the combination of program at this high level is straightforward. But TDF also provides another mechanism which allows for the combination of program at the syntax tree level, namely *tokens*. Virtually any node of the TDF tree may be a token : a place holder which stands for a subtree. Before the TDF can be decoded fully the definition of this token must be provided. The token definition is then macro substituted for the token in the decoding process to form the complete tree (see Fig. 4). Tokens may also take arguments (see Fig. 5). The actual argument values (from the main tree) are substituted for the formal parameters in the token definition.

As mentioned above, tokens are one of the types of variable objects which are potentially visible to external units. This means that a token does not have to be defined in the same unit as it is used in. Nor do these units have originally to have come from the same capsule, provided they have been linked before they need to be fully decoded. Tokens therefore provide a mechanism for the low-level separation and combination of code.

2. TDF Compilation Phases

We have seen how one of the great strengths of TDF is the fact that it facilitates the separation and combination of program. We now demonstrate how this is applied in the TDF compilation strategy. This section is designed only to give an outline of this scheme. The various constituent phases are discussed in more detail later.

Again we start with the simplest case, where the program contains no target dependent code. The strategy is illustrated in Fig. 2, which should be compared with the traditional compilation strategy shown in Fig. 1. The general layout of the diagrams is the same. The left halves of the diagrams refers to the program itself, and the right halves to the corresponding API. The top halves refer to machine independent

[7] See [7] and [5].

material, and the bottom halves to what happens on each target machine. Thus, as before, the portable program appears in the top left of the diagram, and the corresponding API in the top right.

The first thing to note is that, whereas previously all the compilation took place on the target machines, here the compilation has been split into a target independent (C → TDF) part, called *production*, and a target dependent (TDF → target) part, called *installation*. One of the synonyms for TDF is ANDF, Architecture Neutral Distribution Format, and we require that the production is precisely that - architecture neutral - so that precisely the same TDF is installed on all the target machines.

This architecture neutrality necessitates a separation of code. For example, in the "Hello world" example discussed in sections I.1.1 and I.1.3, the API specifies that there shall be a type *FILE* and an object *stdout* of type *FILE* *, but the implementations of these may be different on all the target machines. Thus we need to be able to abstract out the code for *FILE* and *stdout* from the TDF output by the producer, and provide the appropriate (target dependent) definitions for these objects in the installation phase.

2.1. API Description (Top Right)

The method used for this separation is the token mechanism. Firstly the syntactic element of the API is described in the form of a set of target independent headers. Whereas the target dependent, system headers contain the actual implementation of the API on a particular machine, the target independent headers express to the producer what is actually in the API, and which may therefore be assumed to be common to all compliant target machines. For example, in the target independent headers for the ANSI standard, there will be a file *stdio.h* containing the lines :

```
#pragma token TYPE FILE # ansi.stdio.FILE
#pragma token EXP rvalue : FILE * : stdout # ansi.stdio.stdout
#pragma token FUNC int ( const char *, FILE * ) : fputs # ansi.stdio.fputs
```

These *#pragma token* directives are extensions to the C syntax which enable the expression of abstract syntax information to the producer. The directives above tell the producer that there exists a type called *FILE*, an expression *stdout* which is an rvalue (that is, a non-assignable value) of type *FILE* *, and a procedure *fputs* with prototype :

```
int fputs ( const char *, FILE * ) ;
```

and that it should leave their values unresolved by means of tokens[8]. Note how the information in the target independent header precisely reflects the syntactic information in the ANSI API.

The names *ansi.stdio.FILE* etc. give the external names for these tokens, those which will be visible at the outermost layer of the capsule; they are intended to be unique (this is discussed below). It is worth making the distinction between the internal names and these external token names. The former are the names used to represent the objects within C, and the latter the names used within TDF to represent the tokens corresponding to these objects.

2.2. Production (Top Left)

Now the producer can compile the program using these target independent headers. As will be seen from the "Hello world" example, these headers contain sufficient information to check that the program is syntactically correct. The produced, target independent, TDF will contain tokens corresponding to the various uses of *stdout*, *fputs* and so on, but these tokens will be left undefined. In fact there will be other undefined tokens in the TDF. The basic C types, *int* and *char* are used in the program, and their implementations may vary between target machines. Thus these types must also be represented by tokens. However these tokens are implicit in the producer rather than explicit in the target independent headers.

Note also that because the information in the target independent headers describes abstractly the contents of the API and not some particular implementation of it, the producer is in effect checking the program against the API itself.

---

[8] For more details on the *#pragma token* directive see [4].

2.3. API Implementation (Bottom Right)

Before the TDF output by the producer can be decoded fully it needs to have had the definitions of the tokens it has left undefined provided. These definitions will be potentially different on all target machines and reflect the implementation of the API on that machine.

The syntactic details of the implementation are to be found in the system headers. The process of defining the tokens describing the API (called TDF library building) consists of comparing the implementation of the API as given in the system headers with the abstract description of the tokens comprising the API given in the target independent headers. The token definitions thus produced are stored as TDF libraries, which are just archives of TDF capsules.

For example, in the example implementation of *stdio.h* given in section I.1.2, the token *ansi.stdio.FILE* will be defined as the TDF compound shape corresponding to the structure defining the type *FILE* (recall the distinction between internal and external names). *__iob* will be an undefined tag whose shape is an array of 60 copies of the shape given by the token *ansi.stdio.FILE*, and the token *ansi.stdio.stdout* will be defined to be the TDF expression corresponding to a pointer to the second element of this array. Finally the token *ansi.stdio.fputs* is defined to be the effect of applying the procedure given by the undefined tag *fputs*[9].

These token definitions are created using exactly the same C → TDF translation program as is used in the producer phase. This program knows nothing about the distinction between target independent and target dependent TDF, it merely translates the C it is given (whether from a program or a system header) into TDF. It is the compilation process itself which enables the separation of target independent and target dependent TDF.

In addition to the tokens made explicit in the API, the implicit tokens built into the producer must also have their definitions inserted into the TDF libraries. The method of definition of these tokens is slightly different. The definitions are automatically deduced by, for example, looking in the target machine's *limits.h* header to find the local values of *CHAR_MIN* and *CHAR_MAX*, and deducing the definition of the token corresponding to the C type *char* from this. It will be the *variety* (the TDF abstraction of integer types) consisting of all integers between these values.

Note that what we are doing in the main library build is checking the actual implementation of the API against the abstract syntactic description. Any variations of the syntactic aspects of the implementation from the API will therefore show up. Thus library building is an effective way of checking the syntactic conformance of a system to an API. Checking the semantic conformance is far more difficult - we shall return to this issue later.

2.4. Installation (Bottom Left)

The installation phase is now straightforward. The target independent TDF representing the program contains various undefined tokens (corresponding to objects in the API), and the definitions for these tokens on the particular target machine (reflecting the API implementation) are to be found in the local TDF libraries. It is a natural matter to link these to form a complete, target dependent, TDF capsule. The rest of the installation consists of a straightforward translation phase (TDF → target) to produce a binary object file, and linking with the system libraries to form a final executable. Linking with the system libraries will resolve any tags left undefined in the TDF.

2.5. Illustrated Example

In order to help clarify exactly what is happening where, Fig. 2a shows a simple example superimposed on the TDF compilation diagram. The program to be translated is simply :

FILE f ;

and the API is as above, so that *FILE* is an abstract type. This API is described as target independent headers containing the *#pragma token* statements given above. The producer combines the program with

[9] In fact, this picture has been slightly simplified for the sake of clarity. See the section on C → TDF mappings below (section II.3.2).

the target independent headers to produce a target independent capsule which declares a tag *f* whose shape is given by the token representing *FILE*, but leaves this token undefined. In the API implementation, the local definition of the type *FILE* from the system headers is translated into the definition of this token by the library building process. Finally in the installation, the target independent capsule is combined with the local token definition library to form a target dependent capsule in which all the tokens used are also defined. This is then installed further as described above.

3. Aspects of the TDF System

Let us now consider in more detail some of the components of the TDF system and how they fit into the compilation scheme.

3.1. The C → TDF Producer

Above it was emphasised how the design of the compilation strategy aids the representation of program in a target independent manner, but this is not enough in itself. The C → TDF producer must represent everything symbolically; it cannot make assumptions about the target machine. For example, the line of C containing the initialisation :

```
int a = 1 + 1 ;
```

is translated into TDF representing precisely that, 1 + 1, not 2, because it does not know the representation of *int* on the target machine. The installer does know this, and so is able to replace 1 + 1 by 2 (provided this is actually true).

As another example, in the structure :

```
struct tag {
        int a ;
        double b ;
} ;
```

the producer does not know the actual value in bits of the offset of the second field from the start of the structure - it depends on the sizes of *int* and *double* and the alignment rules on the target machine. Instead it represents it symbolically (it is the size of *int* rounded up to a multiple of the alignment of *double*). This level of abstraction makes the tokenisation required by the target independent API headers very natural. If we only knew that there existed a structure *struct tag* with a field *b* of type *double* then it is perfectly simple to use a token to represent the (unknown) offset of this field from the start of the structure rather than using the calculated (known) value. Similarly, when it comes to defining this token in the library building phase (recall that this is done by the same C → TDF translation program as the production) it is a simple matter to define the token to be the calculated value.

Furthermore, because all the producer's operations are performed at this very abstract level, it is a simple matter to put in extra portability checks. For example, it would be a relatively simple task to put most of the functionality of *lint* (excluding intermodular checking) or *gcc*'s *-Wall* option into the producer, and moreover have these checks applied to an abstract machine rather than a particular target machine. Indeed a number of these checks have already been implemented.

These extra checks are switched on and off by using *#pragma* statements[10]. For example, ANSI C states that any undeclared function is assumed to return *int*, whereas for strict portability checking it is more useful to have undeclared functions marked as an error (indeed for strict API checking this is essential). This is done by inserting the line :

```
#pragma no implicit definitions
```

either at the start of each file to be checked or, more simply, in a start-up file - a file which can be *#include*'d at the start of each source file by means of a command line option.

[10] For more details on the *#pragma* syntax and which portability checks are currently supported by the producer see [4].

Because these checks can be turned off as well as on it is possible to relax as well as strengthen portability checking. Thus if a program is only intended to work on 32-bit machines, it is possible to switch off certain portability checks. The whole ethos underlying the producer is that these portability assumptions should be made explicit, so that the appropriate level of checking can be done.

As has been previously mentioned, the use of a single front-end to any compiler not only virtually eliminates the problems of differing code interpretation and compiler quirks, but also reduces the exposure to compiler bugs. Of course, this also applies to the TDF compiler, which has a single front-end (the producer) and multiple back-ends (the installers). As regards the syntax and semantics of the C language, the producer is by default a strictly ANSI C compliant compiler[11]. However it is possible to change its behaviour (again by means of *#pragma* statements) to implement many of the features found in "traditional" or "K&R" C. Hence it is possible to precisely determine how the producer will interpret the C code it is given by explicitly describing the C dialect it is written in in terms of these *#pragma* statements.

3.2. C → TDF Mappings

The nature of the C → TDF transformation implemented by the producer[12] is worth considering. Although it is only indirectly related to questions of portability, it does illustrate some of the problems the producer has in trying to represent program in an architecture neutral manner.

Once the initial difficulty of overcoming the syntactic and semantic differences between the various C dialects is overcome, the C → TDF mapping is quite straightforward. In a hierarchy from high level to low level languages C and TDF are not that dissimilar - both come towards the bottom of what may legitimately be regarded as high level languages. Thus the constructs in C map easily onto the constructs of TDF (there are a few exceptions, for example coercing integers to pointers, which are discussed elsewhere[13]). Eccentricities of the C language specification such as doing all integer arithmetic in the promoted integer type are translated explicitly into TDF. So to add two *char*'s, they are promoted to *int*'s, added together as *int*'s, and the result is converted back to a *char*. These rules are not built directly into TDF because of the desire to support languages other than C (and even other C dialects).

A number of issues arise when tokens are introduced. Consider for example the type *size_t* from the ANSI standard. This is a target dependent integer type, so bearing in mind what was said above it is natural for the producer to use a tokenised variety (the TDF representation of integer types) to stand for *size_t*. This is done by a *#pragma token* statement of the form :

> *#pragma token VARIETY size_t # ansi.stddef.size_t*

But if we want to do arithmetic on *size_t*'s we need to know the integer type corresponding to the integral promotion of *size_t*. But this is again target dependent, so it makes sense to have another tokenised variety representing the integral promotion of *size_t*. Thus the simple token directive above maps to (at least) two TDF tokens, the type itself and its integral promotion.

As another example, suppose that we have a target dependent C type, *type* say, and we define a procedure which takes an argument of type *type*. In both the procedure body and at any call of the procedure the TDF we need to produce to describe how C passes this argument will depend on *type*. This is because C does not treat all procedure argument types uniformly. Most types are passed by value, but array types are passed by address. But whether or not *type* is an array type is target dependent, so we need to use tokens to abstract out the argument passing mechanism. For example, we could implement the mechanism using four tokens : one for the type *type* (which will be a tokenised shape), one for the type an argument of type *type* is passed as, *arg_type* say, (which will be another tokenised shape), and two for converting values of type *type* to and from the corresponding values of type *arg_type* (these will be tokens which take one exp argument and give an exp). For most types, *arg_type* will be the same as *type* and the conversion tokens

[11] Alas, this is no longer true; however strict ANSI can be specified by means of a simple command line option (see [1]). The decision whether to make the default strict and allow people to relax it, or to make the default lenient and allow people to strengthen it, is essentially a political one. It does not really matter in technical terms provided the user is made aware of exactly what each compilation mode means in terms of syntax, semantics and portability checking.

[12] Not all the feature described in this section are fully implemented in the current (June 1993) producer.

[13] See [4].

will be identities, but for array types, *arg_type* will be a pointer to *type* and the conversion tokens will be "address of" and "contents of".

So there is not the simple one to one correspondence between *#pragma token* directives and TDF tokens one might expect. Each such directive maps onto a family of TDF tokens, and this mapping in a sense encapsulates the C language specification. Of course in the TDF library building process the definitions of all these tokens are deduced automatically from the local values.

3.3. TDF Linking

We now move from considering the components of the producer to those of the installer. The first phase of the installation - linking in the TDF libraries containing the token definitions describing the local implementation of the API - is performed by a general utility program, the TDF linker (or builder). This is a very simple program which is used to combine a number of TDF capsules and libraries into a single capsule. As has been emphasised previously, the capsule structure means that this is a very natural operation, but, as will be seen from the previous discussion (particularly section I.2.3), such combinatorial phases are very prone to namespace problems.

In TDF tags, tokens and other externally named objects occupy separate namespaces, and there are no constructs which can cut across these namespaces in the way that the C macros do. There still remains the problem that the only way to know that two tokens, say, in different capsules are actually the same is if they have the same name. This, as we have already seen in the case of system linking, can cause objects to be identified wrongly.

In the main TDF linking phase - linking in the token definitions at the start of the installation - we are primarily linking on token names, these tokens being those arising from the use of the target independent headers. Potential namespace problems are virtually eliminated by the use of unique external names for the tokens in these headers (such as *ansi.stdio.FILE* in the example above). This means that there is a genuine one to one correspondence between tokens and token names. Of course this relies on the external token names given in the headers being genuinely unique. In fact, as is explained below, these names are normally automatically generated, and uniqueness of names within a given API is checked. Also incorporating the API name into the token name helps to ensure uniqueness across APIs. However the token namespace does require careful management. (Note that the user does not normally have access to the token namespace; all variable and procedure names map into the tag namespace.)

We can illustrate the "clean" nature of TDF linking by considering the *st_atime* example given in section I.2.3. Recall that in the traditional compilation scheme the problem arose, not because of the program or the API implementation, but because of the way they were combined by the pre-processor. In the TDF scheme the target independent version of *sys/stat.h* will be included. Thus the procedure name *st_atime* and the field selector *st_atime* will be seen to belong to genuinely different namespaces - there are no macros to disrupt this. The former will be translated into a TDF tag with external name *st_atime*, whereas the latter is translated into a token with external name *posix.stat.struct_stat.st_atime*, say. In the TDF library reflecting the API implementation, the token *posix.stat.struct_stat.st_atime* will be defined precisely as the system header intended, as the offset corresponding to the C field selector *st_atim.st__sec*. The fact that this token is defined using a macro rather than a conventional direct field selector is not important to the library building process. Now the combination of the program with the API implementation in this case is straightforward - not only are the procedure name and the field selector name in the TDF now different, but they also lie in distinct namespaces. This shows how the separation of the API implementation from the main program is cleaner in the TDF compilation scheme than in the traditional scheme.

TDF linking also opens up new ways of combining code which may solve some other namespace problems. For example, in the *open* example in section I.2.3, the name *open* is meant to be internal to the program. It is the fact that it is not treated as such which leads to the problem. If the program consisted of a single source file then we could make *open* a *static* procedure, so that its name does not appear in the external namespace. But if the program consists of several source files the external name is necessary for intra-program linking. The TDF linker allows this intra-program linking to be separated from the main system linking. In the TDF compilation scheme described above each source file is translated into a separate TDF capsule, which is installed separately to a binary object file. It is only the system linking which finally combines the various components into a single program. An alternative scheme would be to use the TDF linker to combine all the TDF capsules into a single capsule in the production phase and install that. Because all the intra-program linking has already taken place, the external names required for it can be "hidden" - that is to say, removed from the tag namespace. Only tag names which are used but not defined (and so are not internal to the program) and *main* should not be hidden. In effect this linking phase has made all the internal names in the program (except *main*) *static*.

In fact this type of complete program linking is not always feasible. For very large programs the resulting TDF capsule can to be too large for the installer to cope with (it is the system assembler which tends to cause the most problems). Instead it may be better to use a more judiciously chosen partial linking and hiding scheme.

3.4. The TDF Installers

The TDF installer on a given machine typically consists of four phases : TDF linking, which has already been discussed, translating TDF to assembly source code, translating assembly source code to a binary object file, and linking binary object files with the system libraries to form the final executable. The latter two phases are currently implemented by the system assembler and linker, and so are identical to the traditional compilation scheme.

It is the TDF to assembly code translator which is the main part of the installer. Although not strictly related to the question of portability, the nature of the translator is worth considering. Like the producer (and the assembler), it is a transformational, as opposed to a combinatorial, compilation phase. But whereas the transformation from C to TDF is "difficult" because of the syntax and semantics of C and the need to represent everything in an architecture neutral manner, the transformation from TDF to assembly code is much easier because of the unambiguous syntax and uniform semantics of TDF, and because now we know the details of the target machine, it is no longer necessary to work at such an abstract level.

The whole construction of the current generation of TDF translators is based on the concept of compilation as transformation. They represent the TDF they read in as a syntax tree, virtually identical to the syntax tree comprising the TDF. The translation process then consists of continually applying transformations to this tree - in effect TDF → TDF transformations - gradually optimising it and changing it to a form where the translation into assembly source code is a simple transcription process[14].

Even such operations as constant evaluation - replacing 1 + 1 by 2 in the example above - may be regarded as TDF → TDF transformations. But so may more complex optimisations such as taking constants out of a loop, common sub-expression elimination, strength reduction and so on. Some of these transformations are universally applicable, others can only be applied on certain classes of machines. This transformational approach results in high quality code generation[15] while minimising the risk of transformational errors. Moreover the sharing of so much code - up to 70% - between all the TDF translators, like the introduction of a common front-end, further reduces the exposure to compiler bugs.

Much of the machine ABI information is built into the translator in a very simple way. For example, to evaluate the offset of the field *b* in the structure *struct tag* above, the producer has already done all the hard work, providing a formula for the offset in terms of the sizes and alignments of the basic C types. The translator merely provides these values and the offset is automatically evaluated by the constant evaluation transformations. Other aspects of the ABI, for example the procedure argument and result passing conventions, require more detailed attention.

One interesting range of optimisations implemented by many of the current translators consists of the inlining of certain standard procedure calls. For example, *strlen ("hello")* is replaced by 5. As it stands this optimisation appears to run the risk of corrupting the programmer's namespace - what if *strlen* was a user-defined procedure rather than the standard library routine (cf. the *open* example in section I.2.3)? This risk only materialises however if we actually use the procedure name to spot this optimisation. In code compiled from the target independent headers all calls to the library routine *strlen* will be implemented by means of a uniquely named token, *ansi.string.strlen* say. It is by recognising this token name as the token is expanded that the translators are able to ensure that this is really the library routine *strlen*.

---

[14] See [7].
[15] See [6].

Another example of an inlined procedure of this type is *alloca*. Many other compilers inline *alloca*, or rather they inline *__builtin_alloca* and rely on the programmer to identify *alloca* with *__builtin_alloca*. This gets round the potential namespace problems by getting the programmer to confirm that *alloca* in the program really is the library routine *alloca*. By the use of tokens this information is automatically provided to the TDF translators.

4. TDF and APIs

What the discussion above has emphasised is that the ability to describe APIs abstractly as target independent headers underpins the entire TDF approach to portability. We now consider this in more detail.

4.1. API Description

The process of transforming an API specification into its description in terms of *#pragma token* directives is a time-consuming but often fascinating task. In this section we discuss some of the issues arising from the process of describing an API in this way.

4.1.1. The Description Process

As may be observed from the example given in section II.2.1, the *#pragma token* syntax is not necessarily intuitively obvious. It is designed to be a low-level description of tokens which is capable of expressing many complex token specifications. Most APIs are however specified in C-like terms, so an alternative syntax, closer to C, has been developed in order to facilitate their description. This is then transformed into the corresponding *#pragma token* directives by a specification tool called *tspec*[16], which also applies a number of checks to the input and generates the unique token names. For example, the description leading to the example above was :

```
+TYPE FILE ;
+EXP FILE *stdout ;
+FUNC int fputs ( const char *, FILE * ) ;
```

Note how close this is to the English language specification of the API given previously.

tspec is not capable of expressing the full power of the *#pragma token* syntax. Whereas this makes it easier to use in most cases, for describing the normal C-like objects such as types, expressions and procedures, it cannot express complex token descriptions. Instead it is necessary to express these directly in the *#pragma token* syntax. However this is only rarely required : the constructs *offsetof*, *va_start* and *va_arg* from ANSI are the only examples so far encountered during the API description programme at DRA. For example, *va_arg* takes an assignable expression of type *va_list* and a type *t* and returns an expression of type *t*. Clearly, this cannot be expressed abstractly in C-like terms; so the *#pragma token* description :

```
#pragma token PROC ( EXP lvalue : va_list : e, TYPE t ) EXP rvalue : t : \
        va_arg # ansi.stdarg.va_arg
```

must be used instead.

Most of the process of describing an API consists of going through its English language specification transcribing the object specifications it gives into the *tspec* syntax[17] (if the specification is given in a machine readable form this process can be partially automated). The interesting part consists of trying to interpret what is written and reading between the lines as to what is meant. It is important to try to represent exactly what is in the specification rather than being influenced by one's knowledge of a particular

[16] See [2]. There are a number of open issues relating to *tspec* and the *#pragma token* syntax, mainly concerned with determining the type of syntactic statements that it is desired to make about the APIs being described. The current scheme is adequate for those APIs so far considered, but it may need to be extended in future.

[17] There is syntactic information in the paper API specifications which *tspec* (and the *#pragma token* syntax) is not yet capable of expressing. In particular, some APIs go into very careful management of namespaces within the API, explicitly spelling out exactly what should, and should not, appear in the namespaces as each header is included (see the appendix on namespaces and APIs below). What is actually being done here is to regard each header as an independent sub-API. There is not however a sufficiently developed "API calculus" to allow such relationships to be easily expressed.

implementation, otherwise the API checking phase of the compilation will not be checking against what is actually in the API but against a particular way of implementing it[18].

4.1.2. Resolving Conflicts

Another consideration during the description process is to try to integrate the various API descriptions. For example, POSIX extends ANSI, so it makes sense to have the target independent POSIX headers include the corresponding ANSI headers and just add the new objects introduced by POSIX. This does present problems with APIs which are basically compatible but have a small number of incompatibilities, whether deliberate or accidental. As an example of an "accidental" incompatibility, XPG3 is an extension of POSIX, but whereas POSIX declares *malloc* by means of the prototype :

*void *malloc (size_t) ;*

XPG3 declares it by means of the traditional procedure declaration :

*void *malloc (s)*
size_t s ;

These are surely intended to express the same thing, but in the first case the argument is passed as a *size_t* and in the second it is firstly promoted to the integer promotion of *size_t*. On most machines these are compatible, either because of the particular implementation of *size_t*, or because the procedure calling conventions make them compatible. However in general they are incompatible, so the target independent headers either have to reflect this or have to read between the lines and assume that the incompatibility was accidental and ignore it.

As an example of a deliberate incompatibility, both XPG3 and SVID3 declare a structure *struct msqid_ds* in *sys/msg.h* which has fields *msg_qnum* and *msg_qbytes*. The difference is that whereas XPG3 declares these fields to have type *unsigned short*, SVID3 declares them to have type *unsigned long*. However for most purposes the precise types of these fields is not important, so the APIs can be unified by making the types of these fields target dependent. That is to say, tokenised integer types *__msg_q_t* and *__msg_l_t* are introduced[19]. On XPG3-compliant machines these will both be defined to be *unsigned short*, and on SVID3-compliant machines they will both be *unsigned long*. So, although strict XPG3 and strict SVID3 are incompatible, the two extension APIs created by adding these types are compatible. In the rare case when the precise type of these fields is important, the strict APIs can be recovered by defining the field types to be *unsigned short* or *unsigned long* at produce-time rather than at install-time.

This example shows how introducing extra abstractions can resolve potential conflicts between APIs. But it may also be used to resolve conflicts between the API specification and the API implementations. For example, POSIX specifies that the structure *struct flock* defined in *fcntl.h* shall have a field *l_pid* of type *pid_t*. However on at least two of the POSIX implementations examined at DRA, *pid_t* was implemented as an *int*, but the *l_pid* field of *struct flock* was implemented as a *short* (this showed up in the TDF library building process). The immediate reaction might be that these system have not implemented POSIX correctly, so they should be cast into the outer darkness. However for the vast majority of applications, even those which use the *l_pid* field, its precise type is not important. So the decision was taken to introduce a tokenised integer type, *__flock_pid_t*, to stand for the type of the *l_pid* field. So although the implementations do not conform to strict POSIX, they do to this slightly more relaxed extension. Of course, one could enforce strict POSIX by defining *__flock_pid_t* to be *pid_t* at produce-time, but the given implementations would not conform to this stricter API.

Both the previous two examples are really concerned with the question of determining the correct level of abstraction in API specification. Abstraction is inclusive and allows for API evolution, whereas specialisation is exclusive and may lead to dead-end APIs. The SVID3 method of allowing for longer messages

---

[18] There is a continuing API description programme at DRA. The current status (June 1993) is that ANSI (X3.159), POSIX (1003.1), XPG3 (X/Open Portability Guide 3) and SVID (System V Interface Definition, 3rd Edition) have been described and extensively tested. POSIX2 (1003.2), XPG4, AES (Revision A), X11 (Release 5) and Motif (Version 1.1) have been described, but not yet extensively tested.

[19] XPG4 uses a similar technique to resolve this incompatibility. But whereas the XPG4 types need to be defined explicitly, the tokenised types are defined implicitly according to whatever the field types are on a particular machine.

than XPG3 - changing the *msg_qnum* and *msg_qbytes* fields of *struct msqid_ds* from *unsigned short* to *unsigned long* - is an over-specialisation which leads to an unnecessary conflict with XPG3. The XPG4 method of achieving exactly the same end - abstracting the types of these fields - is, by contrast, a smooth evolutionary path.

4.1.3. The Benefits of API Description

The description process is potentially of great benefit to bodies involved in API specification. While the specification itself stays on paper the only real existence of the API is through its implementations. Giving the specification a concrete form means not only does it start to be seen as an object in its own right, rather than some fuzzy document underlying the real implementations, but also any omissions, insufficient specifications (where what is written down does not reflect what the writer actually meant) or built-in assumptions are more apparent. It may also be able to help show up the kind of over-specialisation discussed above. The concrete representation also becomes an object which both applications and implementations can be automatically checked against. As has been mentioned previously, the production phase of the compilation involves checking the program against the abstract API description, and the library building phase checks the syntactic aspect of the implementation against it.

The implementation checking aspect is considered below. Let us here consider the program checking aspect by re-examining the examples given in section I.2.4.1. The *SIGKILL* example is straightforward; *SIGKILL* will appear in the POSIX version of *signal.h* but not the ANSI version, so if the program is compiled with the target independent ANSI headers it will be reported as being undefined. In a sense this is nothing to do with the *#pragma token* syntax, but with the organisation of the target independent headers. The other examples do however rely on the fact that the *#pragma token* syntax can express syntactic information in a way which is not possible directly from C. Thus the target independent headers express exactly the fact that *time_t* is an arithmetic type, about which nothing else is known. Thus *(t & 1)* is not type correct for a *time_t t* because the binary *&* operator does not apply to all arithmetic types. Similarly, for the type *div_t* the target independent headers express the information that there exists a structure type *div_t* and field selectors *quot* and *rem* of *div_t* of type *int*, but nothing about the order of these fields or the existence of other fields. Thus any attempt to initialise a *div_t* will fail because the correspondence between the values in the initialisation and the fields of the structure is unknown. The *struct dirent* example is entirely analogous, except that here the declarations of the structure type *struct dirent* and the field selector *d_name* appear in both the POSIX and XPG3 versions of *dirent.h*, whereas the field selector *d_ino* appears only in the XPG3 version.

4.2. TDF Library Building

As we have said, two of the primary problems with writing portable programs are dealing with API implementation errors on the target machines - objects not being defined, or being defined in the wrong place, or being implemented incorrectly - and namespace problems - particularly those introduced by the system headers. The most interesting contrast between the traditional compilation scheme (Fig. 1) and the TDF scheme (Fig. 2) is that in the former the program comes directly into contact with the "real world" of messy system headers and incorrectly implemented APIs, whereas in the latter there is an "ideal world" layer interposed. This consists of the target independent headers, which describe all the syntactic features of the API where they are meant to be, and with no extraneous material to clutter up the namespaces (like *index* and the macro *st_atime* in the examples given in section I.2.3), and the TDF libraries, which can be combined "cleanly" with the program without any namespace problems. All the unpleasantness has been shifted to the interface between this "ideal world" and the "real world"; that is to say, the TDF library building.

The importance of this change may be summarised by observing that previously all the unpleasantnesses happened in the left hand side of the diagram (the program half), whereas in the TDF scheme they are in the right hand side (the API half). So API implementation problems are seen to be a genuinely separate issue from the main business of writing programs; the ball is firmly in the API implementor's court rather than the programmer's. Also the problems need to be solved once per API rather than once per program.

It might be said that this has not advanced us very far towards actually dealing with the implementation errors. The API implementation still contains errors whoever's responsibility it is. But the TDF library building process gives the API implementor a second chance. Many of the syntactic implementation problems will be shown up as the library builder compares the implementation against the abstract API description, and it may be possible to build corrections into the TDF libraries so that the libraries reflect, not the actual implementation, but some improved version of it.

To show how this might be done, we reconsider the examples of API implementation errors given in section I.2.4.2. As before we may divide our discussion between system header problems and system library problems. Recall however the important distinction, that whereas previously the programmer was trying to deal with these problems in a way which would work on all machines (top left of the compilation diagrams), now the person building the TDF libraries is trying to deal with implementation problems for a particular API on a particular machine (bottom right).

4.2.1. System Header Problems

Values which are defined in the wrong place, such as *SEEK_SET* in the example given, present no difficulties. The library builder will look where it expects to find them and report that they are undefined. To define these values it is merely a matter of telling the library builder where they are actually defined (in *unistd.h* rather than *stdio.h*).

Similarly, values which are undefined are also reported. If these values can be deduced from other information, then it is a simple matter to tell the library builder to use these deduced values. For example, if *EXIT_SUCCESS* and *EXIT_FAILURE* are undefined, it is probably possible to deduce their values from experimentation or experience (or guesswork).

Wrongly defined values are more difficult. Firstly they are not necessarily detected by the library builder because they are semantic rather than syntactic errors. Secondly, whereas it is easy to tell the library builder to use a corrected value rather than the value given in the implementation, this mechanism needs to be used with circumspection. The system libraries are provided pre-compiled, and they have been compiled using the system headers. If we define these values differently in the TDF libraries we are effectively changing the system headers, and there is a risk of destroying the interface with the system libraries. For example, changing a structure is not a good idea, because different parts of the program - the main body and the parts linked in from the system libraries - will have different ideas of the size and layout of this structure. (See the *struct flock* example in section II.4.1.2 for a potential method of resolving such implementation problems.)

In the two cases given above - *DBL_MAX* and *size_t* - the necessary changes are probably "safe". *DBL_MAX* is not a special value in any library routines, and changing *size_t* from *int* to *unsigned int* does not affect its size, alignment or procedure passing rules (at least not on the target machines we have in mind) and so should not disrupt the interface with the system library.

4.2.2. System Library Problems

Errors in the system libraries will not be detected by the TDF library builder because they are semantic errors, whereas the library building process is only checking syntax. The only realistic ways of detecting semantic problems is by means of test suites, such as the Plum-Hall or CVSA library tests for ANSI and VSX for XPG3, or by detailed knowledge of particular API implementations born of personal experience. However it may be possible to build workarounds for problems identified in these tests into the TDF libraries.

For example, the problem with *realloc* discussed in section I.2.4.2.2 could be worked around by defining the token representing *realloc* to be the equivalent of :

```
#define realloc ( p, s )    ( void *q = ( p ) ? ( realloc ) ( q, s ) : malloc ( s ) )
```

(where the C syntax has been extended to allow variables to be introduced inside expressions) or :

```
static void *__realloc ( void *p, size_t s )
{
        if ( p == NULL ) return ( malloc ( s ) ) ;
        return ( ( realloc ) ( p, s ) ) ;
}
```

```
#define realloc ( p, s )    __realloc ( p, s )
```

Alternatively, the token definition could be encoded directly into TDF (not via C), using the TDF notation compiler[20].

4.2.3. TDF Library Builders

The discussion above shows how the TDF libraries are an extra layer which lies on top of the existing system API implementation, and how this extra layer can be exploited to provide corrections and workarounds to various implementation problems. The expertise of particular API implementation problems on particular machines can be captured once and for all in the TDF libraries, rather than being spread piecemeal over all the programs which use that API implementation. But being able to encapsulate this expertise in this way makes it a marketable quantity. One could envisage a market in TDF libraries : ranging from libraries closely reflecting the actual API implementation to top of the range libraries with many corrections and workarounds built in.

All of this has tended to paint the system vendors as the villains of the piece for not providing correct API implementations, but this is not entirely fair. The reason why API implementation errors may persist over many operating system releases is that system vendors have as many porting problems as anyone else - preparing a new operating system release is in effect a huge porting exercise - and are understandably reluctant to change anything which basically works. The use of TDF libraries could be a low-risk strategy for system vendors to allow users the benefits of API conformance without changing the underlying operating system.

Of course, if the system vendor's porting problems could be reduced, they would have more confidence to make their underlying systems more API conformant, and thereby help reduce the normal programmer's porting problems. So whereas using the TDF libraries might be a short-term workaround for API implementation problems, the rest of the TDF porting system might help towards a long-term solution.

Another interesting possibility arises. As we said above, many APIs, for example POSIX and BSD, offer equivalent functionality by different methods. It may be possible to use the TDF library building process to express one in terms of the other. For example, in the *struct dirent* example in section I.3.3, the only differences between POSIX and BSD were that the BSD version was defined in a different header and that the structure was called *struct direct*. But this presents no problems to the TDF library builder : it is perfectly simple to tell it to look in *sys/dir.h* instead of *dirent.h*, and to identify *struct direct* with *struct dirent*. So it may be possible to build a partial POSIX lookalike on BSD systems by using the TDF library mechanism.

5. TDF and Conditional Compilation

So far our discussion of the TDF approach to portability has been confined to the simplest case, where the program itself contains no target dependent code. We now turn to programs which contain conditional compilation. As we have seen, many of the reasons why it is necessary to introduce conditional compilation into the traditional compilation process either do not arise or are seen to be distinct phases in the TDF compilation process. The use of a single front-end (the producer) virtually eliminates problems of compiler limitations and differing interpretations and reduces compiler bug problems, so it is not necessary to introduce conditionally compiled workarounds for these. Also API implementation problems, another prime reason for introducing conditional compilation in the traditional scheme, are seen to be isolated in the TDF library building process, thereby allowing the programmer to work in an idealised world one step removed from the real API implementations. However the most important reason for introducing conditional compilation is where things, for reasons of efficiency or whatever, are genuinely different on different machines. It is this we now consider.

[20] See [3].

5.1. User-Defined APIs

The things which are done genuinely differently on different machines have previously been characterised as comprising the user-defined component of the API. So the real issue in this case is how to use the TDF API description and representation methods within one's own programs. A very simple worked example[21] is given below (in section II.5.2).

For the *MSB* example given in section I.3.1 we firstly have to decide what the user-defined API is. To fully reflect exactly what the target dependent code is, we could define the API, in *tspec* terms, to be :

```
+MACRO unsigned char MSB ( unsigned int a ) ;
```

where the macro *MSB* gives the most significant byte of its argument, *a*. Let us say that the corresponding *#pragma token* statement is put into the header *msb.h*. Then the program can be recast into the form :

```
#include <stdio.h>
#include "msb.h"

unsigned int x = 100000000 ;

int main ()
{
	printf ( "%u\n", MSB ( x ) ) ;
	return ( 0 ) ;
}
```

The producer will compile this into a target independent TDF capsule which uses a token to represent the use of *MSB*, but leaves this token undefined. The only question that remains is how this token is defined on the target machine; that is, how the user-defined API is implemented. On each target machine a TDF library containing the local definition of the token representing *MSB* needs to be built. There are two basic possibilities. Firstly the person performing the installation could build the library directly, by compiling a program of the form :

```
#pragma implement interface "msb.h"
#include "config.h"

#ifndef SLOW_SHIFT
#define MSB ( a )		( ( unsigned char ) ( a >> 24 ) )
#else
#ifdef BIG_ENDIAN
#define MSB ( a )		*( ( unsigned char * ) &( a ) )
#else
#define MSB ( a )		*( ( unsigned char * ) &( a ) + 3 )
#endif
#endif
```

with the appropriate *config.h* to choose the correct local implementation of the interface described in *msb.h*. Alternatively the programmer could provide three alternative TDF libraries corresponding to the three implementations, and let the person installing the program choose between these. The two approaches are essentially equivalent, they just provide for making the choice of the implementation of the user-defined component of the API in different ways[22].

---

[21] For more detailed examples see [9].

[22] An interesting alternative approach would be to provide a short program which does the selection between the provided API implementations automatically. This approach might be particularly effective in deciding which implementation offers the best performance on a particular target machine.

5.2. User Defined Tokens - Example

As an example of how to define a simple token consider the following example. We have a simple program which prints "hello" in some language, the language being target dependent. Our first task is choose an API. We choose ANSI C extended by a tokenised object *hello* of type *char ** which gives the message to be printed. This object will be an rvalue (i.e. it cannot be assigned to). For convenience this token is declared in a header file, *tokens.h* say. This particular case is simple enough to encode by hand; it takes the form :

```
#pragma token EXP rvalue : char * : hello #
#pragma interface hello
```

consisting of a *#pragma token* directive describing the object to be tokenised, and a *#pragma interface* directive to show that this is the only object in the API. An alternative would be to generate *tokens.h* from a *tspec* specification of the form :

```
+EXP char *hello ;
```

The next task is to write the program conforming to this API. This may take the form of a single source file, *hello.c*, containing the lines :

```
#include <stdio.h>
#include "tokens.h"

int main ()
{
        printf ( "%s\n", hello ) ;
        return ( 0 ) ;
}
```

The production process may be specified by means of a *Makefile*. This uses the TDF C compiler, *tcc*, which is an interface to the TDF system which is designed to be like *cc*, but with extra options to handle the extra functionality offered by the TDF system[23].

```
produce : hello.j
        @echo "PRODUCTION COMPLETE"

hello.j : hello.c tokens.h
        @echo "PRODUCTION : C->TDF"
        tcc -Fj hello.c
```

The production is run by typing *make produce*. The ANSI API is the default, and so does not need to be specified to *tcc*. The program *hello.c* is compiled to a target independent capsule, *hello.j*. This will use a token to represent *hello*, but it will be left undefined.

On each target machine we need to create a token library giving the local definitions of the objects in the API. We shall assume that the library corresponding to the ANSI C API has already been constructed, so that we only need to define the token representing *hello*. This is done by means of a short C program, *tokens.c*, which implements the tokens declared in *tokens.h*. This might take the form :

```
#pragma implement interface "tokens.h"
#define hello "bonjour"
```

to define *hello* to be "bonjour". On a different machine, the definition of *hello* could be given as "hello", "guten Tag", "zdrastvetye" (excuse my transliteration) or whatever (including complex expressions as well as simple strings). Note the use of *#pragma implement interface* to indicate that we are now implementing the API described in *tokens.h*, as opposed to the use of *#include* earlier when we were just using the API.

The installation process may be specified by adding the following lines to the *Makefile* :

---

[23] See [1].

```
install : hello
        @echo "INSTALLATION COMPLETE"

hello : hello.j tokens.tl
        @echo "INSTALLATION : TDF->TARGET"
        tcc -o hello -J. -jtokens hello.j

tokens.tl : tokens.j
        @echo "LIBRARY BUILDING : LINKING LIBRARY"
        tcc -Ymakelib -o tokens.tl tokens.j

tokens.j : tokens.c tokens.h
        @echo "LIBRARY BUILDING : DEFINING TOKENS"
        tcc -Fj -not_ansi tokens.c
```

The complete installation process is run by typing *make install*. Firstly the file *tokens.c* is compiled to give the TDF capsule *tokens.j* containing the definition of *hello*. The *-not_ansi* flag is needed because *tokens.c* does not contain any real C (declarations or definitions), which is not allowed in ANSI C. The next step is to turn the capsule *tokens.j* into a TDF library, *tokens.tl*, using the *-Ymakelib* option to *tcc*[24]. This completes the API implementation.

The final step is installation. The target independent TDF, *hello.j*, is linked with the TDF libraries *tokens.tl* and *ansi.tl* (which is built into *tcc* as default) to form a target dependent TDF capsule with all the necessary token definitions, which is then translated to a binary object file and linked with the system libraries. All of this is under the control of *tcc*.

Note the four stages of the compilation : API specification, production, API implementation and installation, corresponding to the four regions of the compilation diagram (Fig. 2).

5.3. Conditional Compilation within TDF

Although tokens are the main method used to deal with target dependencies, TDF does have built-in conditional compilation constructs. For most TDF sorts *X* (for example, exp, shape or variety) there is a construct *X_cond* which takes an exp and two *X*'s and gives an *X*. The exp argument will evaluate to an integer constant at install time. If this is true (nonzero), the result of the construct is the first *X* argument and the second is ignored; otherwise the result is the second *X* argument and the first is ignored. By ignored we mean completely ignored - the argument is stepped over and not decoded. In particular any tokens in the definition of this argument are not expanded, so it does not matter if they are undefined.

These conditional compilation constructs are used by the C $\rightarrow$ TDF producer to translate certain statements containing :

#if condition

where *condition* is a target dependent value. Thus, because it is not known which branch will be taken at produce time, the decision is postponed to install time. If *condition* is a target independent value then the branch to be taken is known at produce time, so the producer only translates this branch. Thus, for example, code surrounded by *#if 0 ... #endif* will be ignored by the producer.

Not all such *#if* statements can be translated into TDF *X_cond* constructs. The two branches of the *#if* statement are translated into the two *X* arguments of the *X_cond* construct; that is, into sub-trees of the TDF syntax tree. This can only be done if each of the two branches is syntactically complete.

The producer interprets *#ifdef* (and *#ifndef*) constructs to mean, is this macro is defined (or undefined) at produce time? Given the nature of pre-processing in C this is in fact the only sensible interpretation. But if such constructs are being used to control conditional compilation, what is actually intended is, is this macro defined at install time? This distinction is necessitated by the splitting of the TDF compilation into production and installation - it does not exist in the traditional compilation scheme. For example, in the mips example in section I.3.1, whether or not *mips* is defined is intended to be an installer property, rather

[24] With older versions of *tcc* it may be necessary to change this option to *-Ymakelib -M -Fj*.

than what it is interpreted as, a producer property. The choice of the conditional compilation path may be put off to install time by, for example, changing *#ifdef mips* to *#if is_mips* where *is_mips* is a tokenised integer which is either 1 (on those machines on which *mips* would be defined) or 0 (otherwise). In fact in view of what was said above about syntactic completeness, it might be better to recast the program as :

```
#include <stdio.h>
#include "user_api.h"     /* For the specification of is_mips */

int main ()
{
        if ( is_mips ) {
                fputs ( "This machine is a mips\n", stdout ) ;
        }
        return ( 0 ) ;
}
```

because the branches of an *if* statement, unlike those of an *#if* statement, have to be syntactically complete is any case. The installer will optimise out the unnecessary test and any unreached code, so the use of *if (condition)* is guaranteed to produce as efficient code as *#if condition*.

In order to help detect such "installer macro" problems the producer has a mode for detecting them. All *#ifdef* and *#ifndef* constructs in which the compilation path to be taken is potentially target dependent[25] are reported.

The existence of conditional compilation within TDF also gives flexibility in how to approach expressing target dependent code. Instead of a "full" abstraction of the user-defined API as target dependent types, values and functions, it can be abstracted as a set of binary tokens (like *is_mips* in the example above) which are used to control conditional compilation. This latter approach can be used to quickly adapt existing programs to a TDF-portable form since it is closer to the "traditional" approach of scattering the program with *#ifdef*'s and *#ifndef*'s to implement target dependent code. However the definition of a user-defined API gives a better separation of target independent and target dependent code, and the effort to define such as API may often be justified. When writing a new program from scratch the API rather than the conditional compilation approach is recommended.

The latter approach of a fully abstracted user-defined API may be more time consuming in the short run, but this may well be offset by the increased ease of porting. Also there is no reason why a user-defined API, once specified, should not serve more than one program. Similar programs are likely to require the same abstractions of target dependent constructs. Because the API is a concrete object, it can be reused in this way in a very simple fashion. One could envisage libraries of private APIs being built up in this way.

5.4. Alternative Program Versions

Consider again the program described in section I.3.4 which has optional features for displaying its output graphically depending on the boolean value *HAVE_X_WINDOWS*. By making *HAVE_X_WINDOWS* part of the user-defined API as a tokenised integer and using :

```
#if HAVE_X_WINDOWS
```

to conditionally compile the X Windows code, the choice of whether or not to use this version of the program is postponed to install time. If both POSIX and X Windows are implemented on the target machine the installation is straightforward. *HAVE_X_WINDOWS* is defined to be true, and the installation proceeds as normal. The case where only POSIX is implemented appears to present problems. The TDF representing the program will contain undefined tokens representing objects from both the POSIX and X Windows APIs. Surely it is necessary to define these tokens (i.e. implement both APIs) in order to install the TDF. But because of the use of conditional compilation, all the applications of X Windows tokens will be inside *X_cond* constructs on the branch corresponding to *HAVE_X_WINDOWS* being true. If it is actually false then these branches are stepped over and completely ignored. Thus it does not matter that these tokens are

[25] See [4] and [9].

undefined. Hence the conditional compilation constructs within TDF give the same flexibility in the API implementation is this case as do those in C.

Conclusions

The philosophy underlying the whole TDF approach to portability is that of separation or isolation. This separation of the various components of the compilation system means that to a large extent they can be considered independently. The separation is only possible because the definition of TDF has mechanisms which facilitate it - primarily the token mechanism, but also the capsule linkage scheme.

The most important separation is that of the abstract description of the syntactic aspects of the API, in the form of the target independent headers, from the API implementation. It is this which enables the separation of target independent from target dependent code which is necessary for any Architecture Neutral Distribution Format. It also means that programs can be checked against the abstract API description, instead of against a particular implementation, allowing for effective API conformance testing of applications. Furthermore, it isolates the actual program from the API implementation, thereby allowing the programmer to work in the idealised world envisaged by the API description, rather than the real world of API implementations and all their faults.

This isolation also means that these API implementation problems are seen to be genuinely separate from the main program development. They are isolated into a single process, TDF library building, which needs to be done only once per API implementation. Because of the separation of the API description from the implementation, this library building process also serves as a conformance check for the syntactic aspects of the API implementation. However the approach is evolutionary in that it can handle the current situation while pointing the way forward. Absolute API conformance is not necessary; the TDF libraries can be used as a medium for workarounds for minor implementation errors.

The same mechanism which is used to separate the API description and implementation can also be used within an application to separate the target dependent code from the main body of target independent code. This use of user-defined APIs also enables a separation of the portability requirements of the program from the particular ways these requirements are implemented on the various target machines. Again, the approach is evolutionary, and not prescriptive. Programs can be made more portable in incremental steps, with the degree of portability to be used being made a conscious decision.

In a sense the most important contribution TDF has to portability is in enabling the various tasks of API description, API implementation and program writing to be considered independently, while showing up the relationships between them. It is often said that well specified APIs are the solution to the world's portability and interoperability problems; but by themselves they can never be. Without methods of checking the conformance of programs which use the API and of API implementations, the APIs themselves will remain toothless. TDF, by providing syntactic API checking for both programs and implementations, is a significant first step towards solving this problem.

References

[1] *tcc User's Guide*, DRA, 1993.

[2] *tspec - An API Specification Tool*, DRA, 1993.

[3] *The TDF Notation Compiler*, DRA, 1993.

[4] *The C to TDF Producer*, DRA, 1993.

[5] *A Guide to the TDF Specification*, DRA, 1993.

[6] *TDF Facts and Figures*, DRA, 1993.

[7] *TDF Specification*, DRA, 1993.

[8] *The 80386/80486 TDF Installer*, DRA, 1992.

[9] *A Guide to Porting using TDF*, DRA, 1993.

Appendix : Namespaces in Standard APIs

Namespace problems are amongst the most difficult faced by standard defining bodies (for example, the ANSI and POSIX committees) and they often go to great lengths to specify which names should, and should not, appear when certain headers are included[26].

For example, the intention, certainly in ANSI, is that each header should operate as an independent sub-API. Thus *va_list* is prohibited from appearing in the namespace when *stdio.h* is included (it is defined only in *stdarg.h*) despite the fact that it appears in the prototype :

*int vprintf (char *, va_list) ;*

This seeming contradiction is worked round on most implementations by defining a type *__va_list* in *stdio.h* which has exactly the same definition as *va_list*, and declaring *vprintf* as :

*int vprintf (char *, __va_list) ;*

This is only legal because *__va_list* is deemed not to corrupt the namespace because of the convention that names beginning with __ are reserved for implementation use.

This particular namespace convention is well-known, but there are others defined in these standards which are not generally known (and since no compiler I know tests them, not widely adhered to). For example, the ANSI header *errno.h* reserves all names given by the regular expression :

E[0-9A-Z][0-9a-z_A-Z]+

against macros (i.e. in all namespaces). By prohibiting the user from using names of this form, the intention is to protect against namespace clashes with extensions of the ANSI API which introduce new error numbers. It also protects against a particular implementation of these extensions - namely that new error numbers will be defined as macros.

A better example of protecting against particular implementations comes from POSIX. If *sys/stat.h* is included names of the form :

st_[0-9a-z_A-Z]+

are reserved against macros (as member names). The intention here is not only to reserve field selector names for future extensions to *struct stat* (which would only affect API implementors, not ordinary users), but also to reserve against the possibility that these field selectors might be implemented by macros. So our *st_atime* example in section I.2.3 is strictly illegal because the procedure name *st_atime* lies in a restricted namespace. Indeed the namespace is restricted precisely to disallow this program.

As an exercise to the reader, how many of your programs use names from the following restricted namespaces (all drawn from ANSI, all applying to all namespaces)?

is[a-z][0-9a-z_A-Z]+ (ctype.h)
to[a-z][0-9a-z_A-Z]+ (ctype.h)
str[a-z][0-9a-z_A-Z]+ (stdlib.h)

With the TDF approach of describing APIs in abstract terms using the *#pragma token* syntax most of these namespace restrictions are seen to be superfluous. When a target independent header is included precisely the objects defined in that header in that version of the API appear in the namespace. There are no worries about what else might happen to be in the header, because there is nothing else. Also implementation details are separated off to the TDF library building, so possible namespace pollution through particular implementations does not arise.

Currently TDF does not have a neat way of solving the *va_list* problem. The present target independent headers use a similar workaround to that described above (exploiting a reserved namespace). (See the footnote in section II.4.1.1.)

---

[26] The position is set out in D. J. Prosser, *Header and name space rules for UNIX systems* (private communication), USL, 1993.

None of this is intended as criticism of the ANSI or POSIX standards. It merely shows some of the problems that can arise from the insufficient separation of code.

Fig. 1. Traditional Compilation Phases

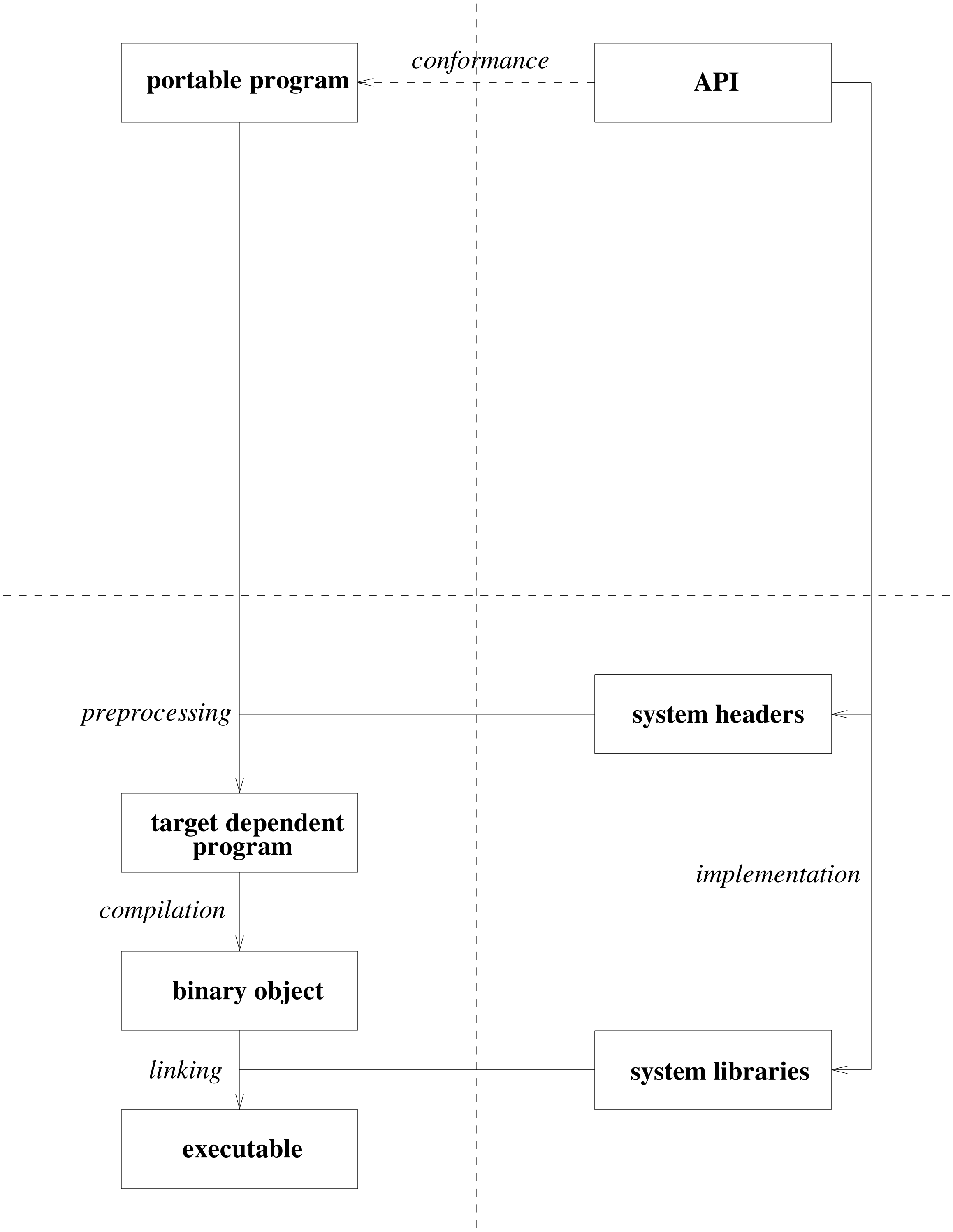

Fig. 2. TDF Compilation Phases

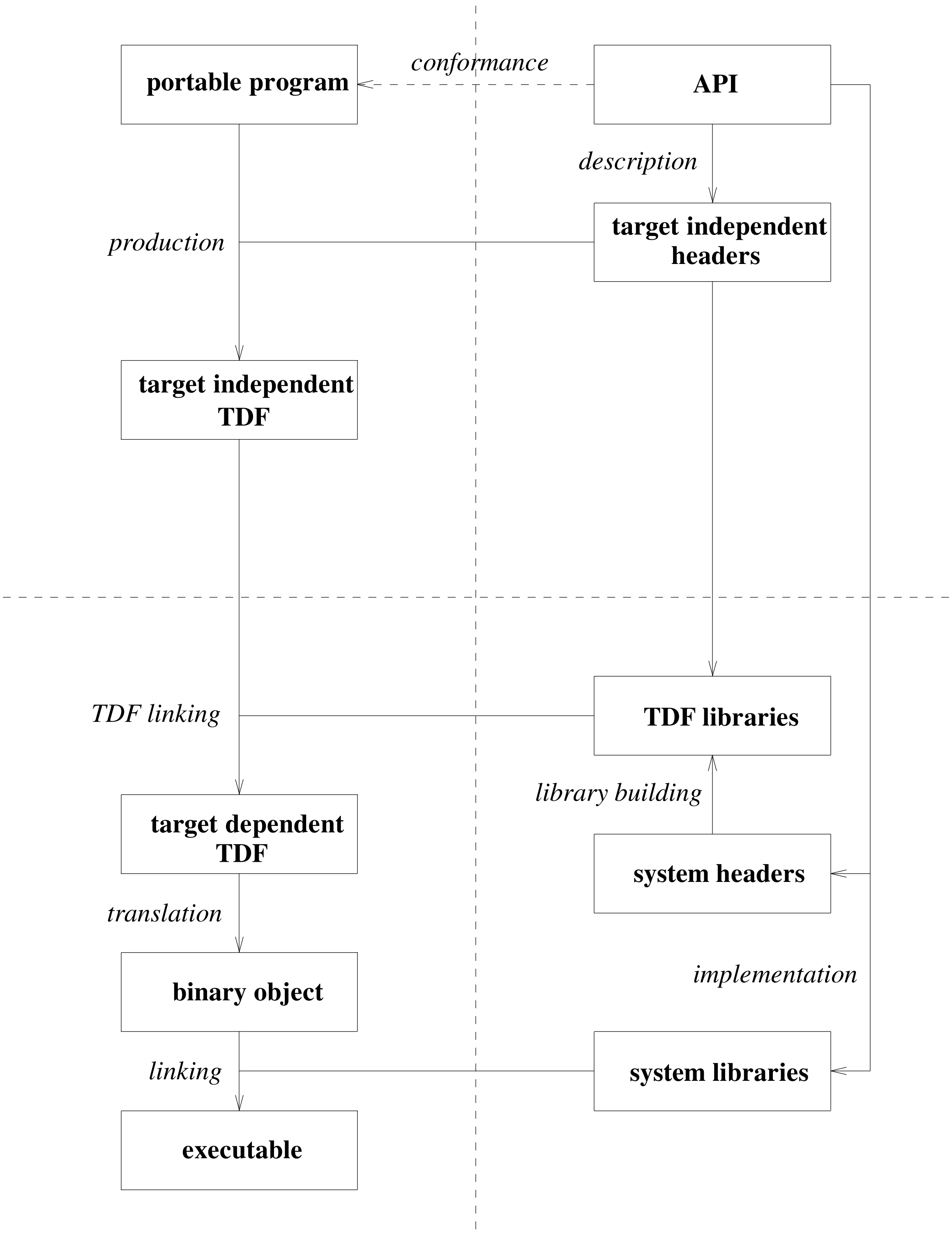

Fig. 2a. TDF Compilation Phases : Example

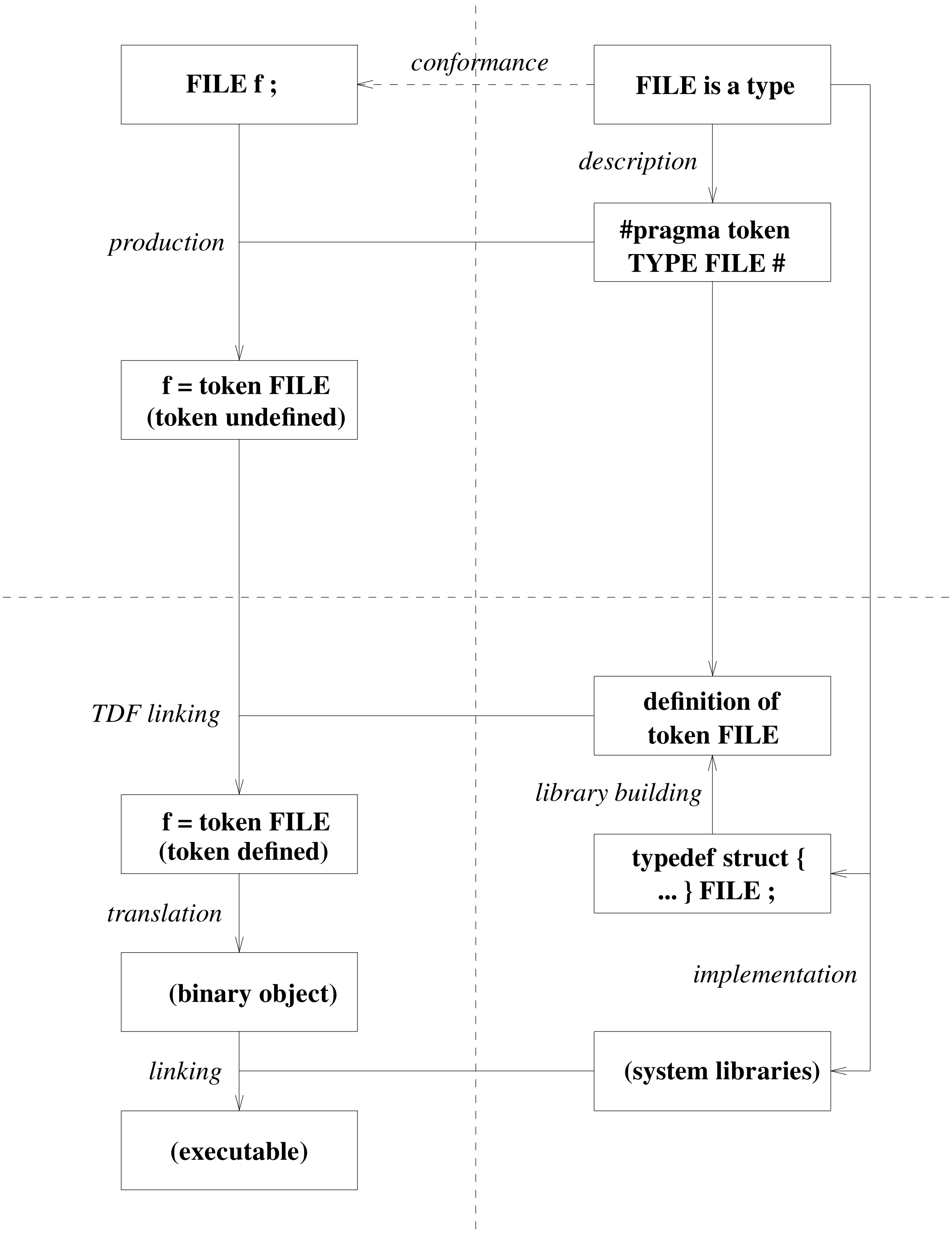

Fig. 3. TDF Capsule Structure

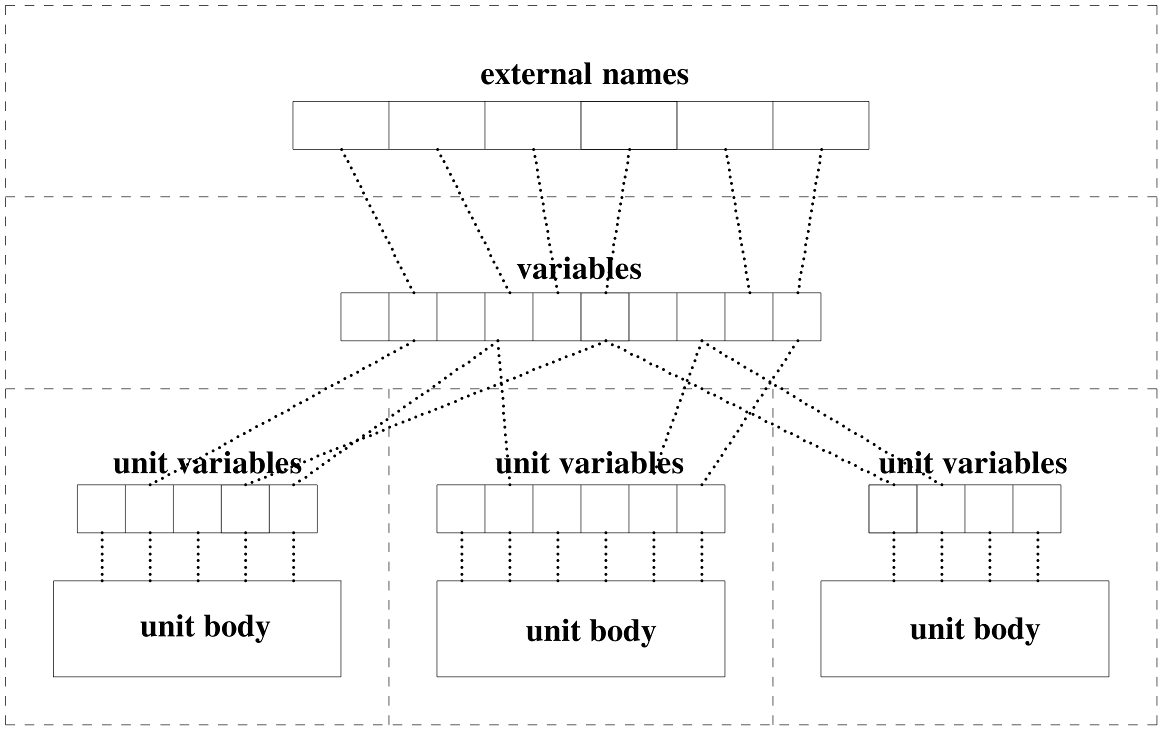

Fig. 4. TDF Token Mechanism I

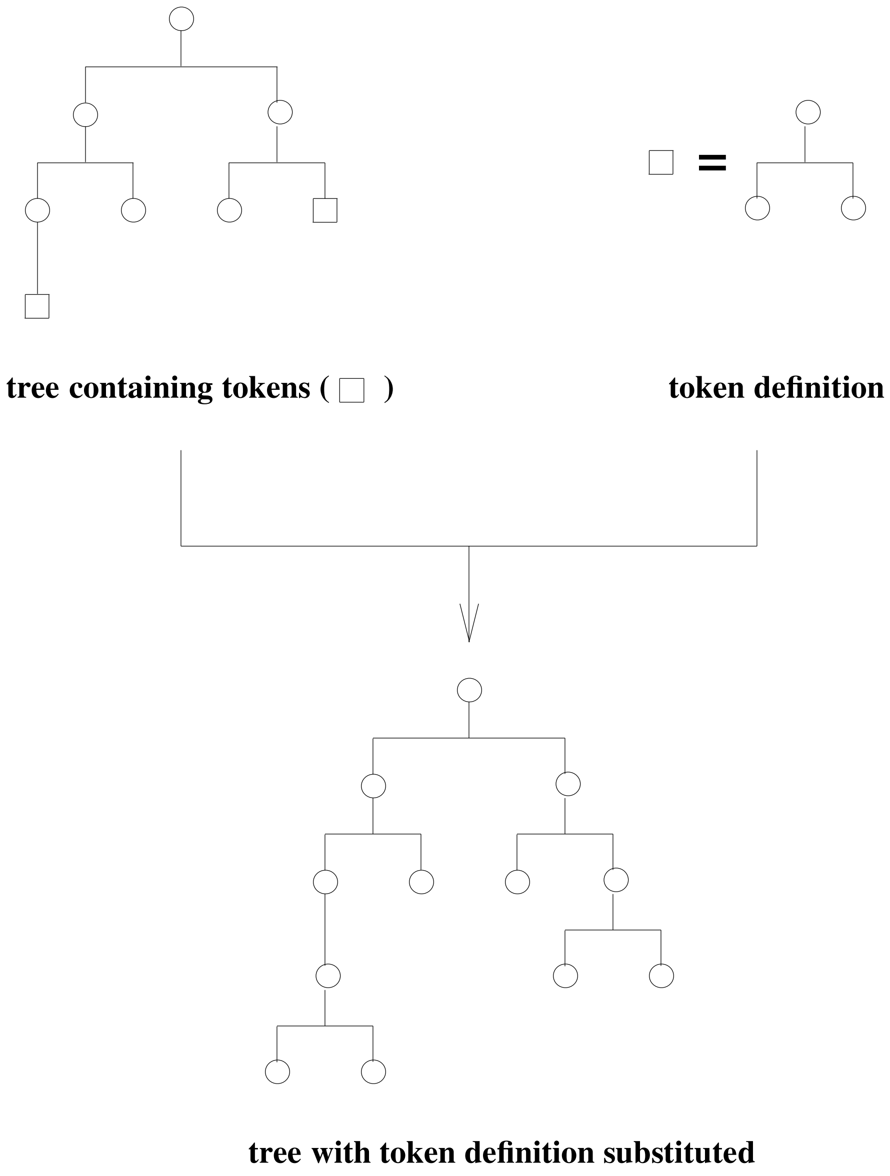

Fig. 5. TDF Token Mechanism II