OSF DCE SIG
Request For Comments: 34.2

H. Melman (OSF)
August 1993

DCE 1.1 CODING STYLE GUIDE

1. INTRODUCTION

This RFC presents an internal coding style guide to be used for the DCE 1.1 project. It includes style information on C source files, build environment issues (including makefile style and source tree organization), and user interface style. This document is intended to be used as a reference guide for internal DCE programmers when writing DCE code.

> *Note to Readers: This RFC contains contains information about cleanup issues relative to the build environment which are incomplete. All such items are clearly identified by ''Notes to Readers'' like this one. They will be completed in an updated version of this RFC as soon as the OSF RATs (Release and Tools) group has time to consider the issues. All other information described here should be considered complete.*

1.1. Changes Since Last Publication

A number of changes have been made since this document was previously published in [RFC 34.1]. The changes that were made, in no particular order, are:

(a) Section 3.1.2 item (a) formerly suggested that the value of *`table-index`* be of the form: *cmp*_s_*subcomp*, it now suggests that it be of the form: *cmp*_svc_*subcomp*, which matches the examples in the serviceability code.

(b) Changed the name of the header file `dce_assert.h` to `assert.h`. Since it it is found in the `/usr/include/dce` directory, the proper way to include this file is by using:

```
#include <dce/assert.h>
```

(c) The prototype of the routine `dce_priv_proc_is_approprpriate()` changed so that it now returns a `boolean` rather than an `int`.

(d) Updated the `dce.h` header file in Appendix B.

(e) Added recommendation on the use of `extern`'s in section 2.3 item (n).

(f) In Section 2.6.1.1 the variable `dsh__table` changed to `dsh_msg_table` matching serviceablity guidelines.

(g) Various editorial changes and improvements to the DCE Error Message Style Guidelines.

(h) Migrated to the new RFC template (`rfc0.1.roff`).

2. C SOURCE CODE

The following sections describe guidelines for writing C source code. In DCE 1.1 an [ANSI C] environment is assumed to exist. DCE 1.1 will *not* guarantee non-ANSI [K&R] compatibility.

2.1. Compiler Switches

OSF/1, running on a specified platform, will serve as one of the DCE 1.1 reference platforms. On that platform `gcc` version 2.*X*[1] will be the ANSI C compiler used. The switches listed below will be used to have the compiler

1. Currently version 2.3 is used at OSF.

generate additional warnings which *must* be fixed in all DCE code. The exception is any kernel code which will follow the conventions of the kernel build environment.[2] For a complete description of these switches (and others which are not required), see Appendix A.

(a) `-ansi`
(b) `-pedantic`
(c) `-Wchar-subscripts`
(d) `-Wcomment`
(e) `-Wformat`
(f) `-Wid-clash-32`
(g) `-Wimplicit`
(h) `-Wmissing-prototypes`
(i) `-Wreturn-type`
(j) `-Wstrict-prototypes`
(k) `-Wswitch`
(l) `-Wtraditional`[3]
(m) `-Wunused`

2.2. System Wide Definitions in dce.h

All DCE wide definitions will be in the machine dependent public header file `dce.h`. This file will include all typedefs, macro definitions, and common includes that span components. It is expected that `dce.h` will be included in most if not all DCE source files. See Appendix B for the current version of `dce.h` on the reference PMAX platform.

OSF will ''own'' `dce.h`; providers are responsible for including it in their code. Providers who have typedefs, macros, prototypes, etc., that need to be included in `dce.h`, must present them to OSF for inclusion.

It is required that each component use these definitions where appropriate for all existing and new code. No duplication of functions in `dce.h` will occur in component header files. (This is what will simplify the porting effort. It would make no sense to have a standard set of definitions and still use others.) The functions defined in `dce.h` are described below.

2.3. Common C Style

(a) C Preprocessor Macros.

All C preprocessor macros that look like functions must be in all uppercase. Macros that implement standard routines will follow whatever conventions the standard requires (e.g., the pthreads routines defined by DCE Threads must be in lower-case). Existing macros in *public* header files may remain in lowercase.

(b) Do *NOT* Use `__STDC__`.

Since some compilers only define this token in a strict ANSI mode, it cannot be used throughout DCE code. It can be used in limited configuration header files (such as `dce.h`) to set up global switches for certain compilers, but using it as a general prototype macro many times in a file is not allowed.

(c) Function Prototypes.

In addition to supporting ANSI C, DCE 1.1 will require prototypes for *all* functions.[4] Functions which are defined as `static` should have their prototypes at the top of the C source file, whether there are forward

---

2. Any C code generated by `idl` or `mavros` must also be [ANSI C] compliant and is subject to the same requirements as other DCE code.

3. There are some issues with including the `-Wtraditional` switch in the above list. This switch is included to help catch any unintentional programming errors. Several programmers have found that it generates a lot of useless warning messages that are not correctable. If you find that the erros are innocuous ignore then, or remove them from the compile line of files that generate many of these warnings. If this switch does not show any useful warnings, I will remove it. If you do find that it generates a warning that helps you find a bug, please let me know.

4. All kernel code will follow the conventions of the kernel source code.

references to these functions or not. This will prevent possible maintenance problems. No variable names are to be used in prototypes since they are subject to macro expansion. All function definitions will use the new style function definition format as defined in [ANSI C]. For example:

```
int foo (a, b, c)
int        a;
void       *b;
struct bar *c;
```

becomes:

```
int foo (int a, void *b, struct bar *c)
```

This is required as there are slightly different semantics associated with the two forms. If the old form is used, all arguments are subject to promotion, in the new form, the compiler is not required to perform the promotion for performance reasons. For this reason, using a prototype with an old-style function declaration can lead to problems with arguments that are subject to promotion.

DCE 1.1 code is *not* required to support compilers which do not handle prototypes. However, much DCE code currently does this by using a prototype macro. This support must either be removed completely (i.e., only use prototypes) or changed to use the following macro defined in `dce.h`:

```
/* Declare a prototype like this (don't use variables) */
int foo _DCE_PROTOTYPE_((int, void *, struct bar *))

/* Define a function like this */
int foo
#if defined(_DCE_PROTO_)
    (
    int a,
    void *b,
    struct bar *c
    )
#else
    (a, b, c)
    int a;
    void *b;
    struct bar *c;
#endif
```

(d) Generic Pointers.

DCE 1.1 assumes the presence of ANSI C, and therefore the generic pointer is to be of type `void *`. Those components wishing to support platforms where void pointers are not available, must use the following typedef for a generic pointer type defined in `dce.h`.

```
/*
 * Note that this definition is for the PMAX specific
 * version of dce.h.  In this version we check for
 * __STDC__ to see if to use void * or char *.  This does
 * NOT violate coding conventions since it is being
 * checked in an isolated instance in a machine
 * dependent file.
 */
```

```
#if defined(__STDC__)
  typedef void * pointer_t;
#else
  typedef char * pointer_t;
#endif
```

(e) Token Concatenation.

DCE 1.1 assumes the presence of ANSI C, and therefore the use of the `##` construct to perform token concatenations. Those components wishing to support platforms where `##` is not available, must use the following macro for a token concatenation defined in `dce.h`.

```
#if defined(_DCE_TOKENCONCAT_)
#  define DCE_TOKENCONCAT(a, b)    a ## b
#else
#  define DCE_TOKENCONCAT(a, b)    a/**/b
#endif
```

(f) Endian Checking.

If code is endian specific, the endianess of the machine should be obtained by including `dce.h` and using `#ifdef` to check for either `LITTLE_ENDIAN` or `BIG_ENDIAN`. One of these is guaranteed to be defined by `dce.h`.

If a port of DCE requires another form of endianess, it is the responsibility of the porter to handle the new form correctly in all DCE code. Each occurrence of endian-specific code must be documented in the *DCE Porting and Testing Guide* [PORTG].

(g) `TRUE` and `FALSE`.

If the macro definitions of `TRUE` and `FALSE` are required, they should be obtained by including `dce.h`. They should not be redefined, nor should variants of these be defined or used (e.g., `TP_TRUE` in `tp2.h`).

(h) `MIN` and `MAX`.

The classic definitions of `MIN` and `MAX` can be obtained by including `dce.h`. They should not be redefined in any form. The definitions are as follows. Note that the arguments are evaluated twice, and therefore arguments with side effects will probably not do what is desired. Also note that these routines return code set dependent values and therefore should not be called with `char`'s as parameters.

```
#if !defined(MIN)
#  define MIN(x, y)          ((x) < (y) ? (x) : (y))
#endif

#if !defined(MAX)
#  define MAX(x, y)          ((x) > (y) ? (x) : (y))
#endif
```

(i) Obtaining the Path of the DCE Installation.

DCE components which need to obtain the paths of the installation[5] must all use the mechanism described below.

In order to facilitate a change in these paths without requiring the recompilation of many DCE source code files, a single file will define variables used to store the paths. This file will be built into `libdceutil.a` which is part of `libdce.a`. These variables can be accessed by DCE components by including `dce.h` to obtain the following definitions:

---

5. This refers to the *`dceshared`* and *`dcelocal`* paths whose uses are specified in [RFC 35.0].

```
extern const char *dceshared_path;
extern const char *dcelocal_path;
```

The use of these variables in DCE code will allow for a change in these paths by merely recompiling one file and relinking programs (this step might be unnecessary if shared libraries are used). The results of changing this path at runtime (that is, at any time after starting any DCE processes on a host) are undefined.

(j) Datatypes.

The following machine dependent datatypes are to be used by all DCE code needing such datatypes. They are defined in `dce.h`.[6]

(i) `byte` — unsigned 8-bit integer (`idl_byte`)
(ii) `boolean` — unsigned 8-bit integer (`idl_boolean`)
(iii) `unsigned8` — unsigned 8-bit integer
(iv) `unsigned16` — unsigned 16-bit integer
(v) `unsigned32` — unsigned 32-bit integer
(vi) `unsigned64` — unsigned 64-bit integer (a `struct`)
(vii) `signed8` — signed 8-bit integer
(viii) `signed16` — signed 16-bit integer
(ix) `signed32` — signed 32-bit integer
(x) `signed64` — signed 64-bit integer

(k) Printing `unsigned32`'s.

In the face of 16, 32, and 64 bit machines, printing something like an `unsigned32` can be a problem. Use the method described below. Other solutions can cause difficulties during ports, as it is necessary to track down format strings, which can be a real problem when using format strings from message catalogs.

```
unsigned32 x = some_value();
/* ... */
printf("0x%lx\n", (long) x);
```

(l) Null Pointers and the use of `NULL`.

If a null pointer is required use `NULL`. Do *not* define a variable or macro to be a type or component specific null pointer (e.g., `cma_c_null_ptr`). The proper definition of `NULL` is as follows:

```
#define NULL 0
```

There are no casts required in either the definition or in any use of `NULL`. If the definition of `NULL` is incorrect on a platform, then it should be redefined in the `dce.h` file for that platform. The macro `NULL` is meant to be used only as a null pointer, do *not* use it to mean the integer zero. The following is incorrect although a compiler will allow it:

```
int x = NULL;
```

(m) Opaque Datatypes.

The correct way to define a new opaque datatype in a public header file is as follows. In the public header file define something like:

```
typedef struct private_foo_s_t *public_foo_t;
```

The definition of `struct private_foo` is not included in the public header file, C allows this. In order for the private code to use this `typedef`, it needs a definiton of the private structure in a private header file as follows:

6. Some are defined in `dce.h` by including `idlbase.h` via `nbase.h`

```
struct private_foo_s_t {
    int   a;
    char *b;
    /* ... */
}
```

Any code needed to access the internals of a `public_foo_t` should include first the private header file, then the public one, and then can access the contents of the pointer. No casting is required. Also this provides some form of type checking, and is therefore much better than using a void (generic) pointer. Finally, this practice, allows the opaque datatypes to be understood by a debugger.

(n) Extern Declarations

This item is a recommendation for all code, it is *not* required for existing or new code. Declarations for `extern`'s should only be in header files. Any `.c` files that need an `extern` should `#include` the appropriate header file. This header file must also be included in the `.c` file that contains the definition of the function, this way the compiler can check if the declaration and the definition match.

2.4. Standard Library Routines

The following hierarchy must be used when finding library routines to call in new DCE code. A routine found at some point in the list performing some function must be used instead of any other routine further down list (or not on the list, such as routines from [BSD]) performing a similar function. Any deviations from this hierarchy *must* be documented in the *DCE Porting and Testing Guide*.

(a) ANSI C.

Routines in the standard C library defined in [ANSI C] (e.g., `memcpy()` instead of `bcopy()`, and `stdarg` instead of `vararg`). See Appendix C for a complete listing of all functions defined in the Standard C Library.

(b) POSIX.1.

Routines defined in [POSIX.1] (e.g., `sigaction()`). See Appendix D for a complete listing of all functions defined by [POSIX.1].

(c) XPG4.

Routines defined by X/Open in [XPG4] (e.g., message and wide character routines `catopen()`, `fgetws()`, `iswalpha()`). See Appendix E for a complete listing of all functions defined by [XPG4].

(d) AES.

Routines defined in the Operating System Programming Interfaces Volume of the Application Environment Specification [AES] from OSF. Its organization is similar to the other items on this list.

(e) Other.

Finally, if none of the above standards are applicable, the routines need to be localized (isolated in as few places as is possible) and/or `#ifdef`'ed. This applies as well to alternative approaches that are used along side of the ''standard'' approach (e.g., sockets implementation as well as XTI in GDS).

2.4.1. Threads wrappers

All called non-thread-safe library routines must be protected by DCE Threads wrappers. For a list of all routines for which wrappers currently exist, see Appendix F. If you are calling a routine that is not wrapped and needs to be, you must have a wrapper created for it.

2.4.2. Standard defined constants

The standards mentioned above define many constants available for the application programmer to determine things like the size of an `unsigned char`, and the maximum number of characters in a filename. These constants must be used where applicable. See Appendix G for a list of the relevant constants.

2.4.3. Exceptions to standards hierarchy

There are some cases where this hierarchy is to be violated, such as certain functions whose use is expanded by some other implementation. For example, the internationalized version of `printf`, which is defined in [XPG4], is to be used instead of the one specified in ANSI C. All such instances are defined below and will be documented in the *DCE Porting and Testing Guide*.

(a) Signal Handling.

[ANSI C] defines the `signal()` routine, however it is not to be used. Instead use the [POSIX.1] defined `sigaction()` routine.

(b) Internationalized Routines.

[ANSI C] defines many routines that are limited in the code sets they can support. Where such differences are limiting factors in the code, expanded versions of those same (or similar) routines found in [XPG4] should be used.

(c) `isascii()`.

Some occurrences of `isascii()` must be replaced with `isdcepcs()` to support internationalization.

(d) Assertion Checking.

The ANSI C `assert()` macro should *not* be used. Instead, use the `dce_assert()` macro (See below).

(e) Message Catalog Routines.

The message catalog routines defined in [XPG4], `catopen()`, `catclose()`, and `catgets()`, should *not* be used directly. Instead use the DCE serviceability routines described below in ‘‘Messaging interface’’ that ‘‘wrap’’ these routines.[7]

2.5. DCE Utilities Library

OSF will provide a library of utility routines to be used by DCE code known as `libdceutils.a`. These routines must be used in place of any others. Any deviations from this *must* be documented in the *DCE Porting and Testing Guide*. Providers wishing to add routines to `libdceutils.a` should make a request to OSF.

This library will be part of `libdce.a` so that DCE library routines can call these routines without forcing a DCE application programmer to add another library to the link line.[8] Users of this library will include the file `dceutils.h` in their code to obtain prototypes of the routines.

2.5.1. Internationalization routines

The following routines should be used to write correctly internationalized code.

(a) `int dce_rpmatch(const char *response);`

This routine is a DCE version of the OSF/1 routine `rpmatch`. It determines whether the string value of the parameter matches (using extended regular expressions) the affirmative or negative response expression as specified by the `LC_MESSAGES` category in the current locale. This routine will return one of three values:

(i) 1 — Returned if the string value of the parameter matches the affirmative expression.

(ii) 0 — Returned if the string value of the parameter matches the negative expression.

---

7. Certain uses of message catalogs in GDS can call the message catalog routines directly.

8. The DCE utilities library will *not* be public, i.e, the routines will not be exposed to DCE application programmers.

(iii) −1 — Returned if the string value of the parameter matches the neither the affirmative or negative expression.

(b) `int isdcepcs(int c);`

This routine returns TRUE or FALSE based on whether or not the character argument is in the DCE Portable Character Set. It is basically a replacement for the XPG4 routine `isascii()`.

2.5.2. Command line input

The following two functions provide a line editing interface with text recall which should be used for all command line input.[9]

(a) `char * readline(char *prompt);`

This function displays the prompt to the user and returns an input line entered by the user. An emacs-like editing interface can be used while entering the line.

(b) `void add_history(char *line);`

This function makes a copy of the specified line and adds it to the internal history list. The user has access to this history when input is read with the `readline` function. It is a separate function so that the programmer can determine what lines are entered in the history. For example, some interfaces might not want continued lines entered in the history.

Below is an example of the use of these functions:

```
while ((line = readline("> ") != NULL)
{
    add_history(line);    /* add all lines to history */
    /* ... other work here ... */
}
```

2.5.3. Getting unique machine identifiers

Some DCE code needs to get the hosts IEEE 802 address. Unfortunately there is no standard way of obtaining this from a system. Both the RPC and CDS components currently have different machine specific code to do this.[10] Both must be changed to use the following interface, as well as any other code needing the 802 address of the host.

```
char *dce_get_802_address (void);
```

2.5.4. Determining appropriate privileges

Presently, DCE relies upon a `uid` of `0` to determine appropriate (root) privilege (e.g., `src/security/server/rs/rs_main.c`). The [POSIX.1] standard does not prescribe that `uid 0` is the root or superuser. In fact, `uid 0` need not exist for POSIX compliance. Hence, the mechanism DCE employs to check for appropriate privilege, namely `setuid(0)` and subsequent checks for `geteuid() == 0`, may not suffice for some compliant POSIX systems. In fact, these will fail where a `uid` of `0` is not allowed/defined.

The following new functions will be defined, in a target machine specific file, to isolate these issues. All existing and new code must call these routines where appropriate.

(a) `int dce_priv_proc_enable(void);`

(b) `int dce_priv_proc_disable(void);`

---

9. While the interface is similar to GNU code, this implementation is not under the conditions of a copyleft.

10. RPC in `src/rpc/runtime/PMAX/uuidsys.c`, and CDS in `src/directory/cds/library/PMAX/unix_getaddr.c` for the PMAX cases.

(c) `boolean dce_priv_proc_is_appropriate(void);`

2.5.5. Common 64 and 128 bit arithmetic utilities

The following macros for 64 bit math are to be used.[11] Any operations required that are not defined below will be added by OSF upon request. Note that DTS may continue to use the five operations defined in `src/time/common/arith.c`, namely `emul()`, `uemul()`, `ueemul()`, `ediv()`, and `uediv()`, since no other component has need of these operations.

Note that these routines sometimes evaluate their parameters more than once. Also venders are free to modify these to suit their platform.

```
/*
 * In these macro definitions, the arguments are named as follows:
 *    d  destination        (only used when a value is changed)
 *    s  source             (s1 and s2 for two sources)
 *    h  the high value     (either source or destination)
 *    l  the low value      (either source or destination)
 *    i  some integer amount
 */

/*
 * U64CMP returns:  1 if s1 is greater;
 *                 -1 if s2 is greater,
 *                  0 if equal
 */
#define U64CMP(s1,s2)    ((s1).hi < (s2).hi? -1 :                       \
                          ((s1).hi > (s2).hi? 1 :                       \
                           ((s1).lo < (s2).lo? -1 :                     \
                            ((s1).lo > (s2).lo? 1 : 0))))
#define U64EQUAL(s1,s2)  ((s1).lo == (s2).lo && (s1).hi == (s2).hi)
#define U64ISZERO(s)     ((s).lo == 0 && (s).hi == 0)
#define U64FITSIN32(s)   ((s).hi == 0)

#define U64SET(d,s)      ((d)=(s))
#define U64SET32(d,l)    ((d).hi = 0, (d).lo = (l))
#define U64SET64(d,h,l)  ((d).hi = (h), (d).lo = (l))
#define U64GET32(l,s)    ((l) = (s).lo)
#define U64GET64(h,l,s)  ((l) = (s).lo, (h) = (s).hi)
#define U64ZERO(d)       U64SET64(d,0,0)
#define U64ONES(d)       U64SET64(d,~0x0,~0x0)


#define U64LSHIFT(d,i)   ((d).hi <<= (i),                               \
                          (d).hi |= (((d).lo) >>                        \
                          ((sizeof(unsigned32)*8)-(i))),                \
                          (d).lo <<= (i))
#define U64RSHIFT(d,i)   ((d).lo >>= (i),                               \
                          (d).lo |= (((d).hi) <<                        \
                          ((sizeof(unsigned32)*8)-(i))),                \
                          (d).hi >>= (i))
```

---

11. These macros are based on macros in `src/file/config/stds.h` for performing math on the `hyper` datatype. The only modifications have been to change the names of the macros (since they needed to be changed to uppercase anyway.)

```
/*
 * The algorithm here is to check for two cases that cause a carry.
 * If the top two bits are different then if the sum has the top
 * bit off then there must have been a carry.  If the top bits are
 * both one then there is always a carry.  We assume 32 bit longs
 * and twos complement arithmetic.
 */
#define SIGN          0x80000000
#define U64ADD32(a,i) (((((a).lo ^ (long)(i)) & SIGN) ?          \
                        (((((a).lo + (long)(i)) & SIGN) == 0) &&    \
                         (a).hi++) :                                 \
                        (((a).lo & (long)(i) & SIGN) && (a).hi++)), \
                       (a).lo += (long)(i))

#define U64ADD(a,b)   (U64ADD32((a),(b).lo), (a).hi += (b).hi)
#define U64SUB(a,b)   (U64ADD32((a),~((b).lo)),                \
                       (a).hi += ~((b).hi),                    \
                       U64ADD32((a),1))
```

2.6. Serviceability

Many of the internationalization and serviceability requirements require code clean-up-like activities (e.g., modifying all `printf` calls in the code). Guidelines for these requirements are presented here so that this one document can serve as a single reference source. Details on these areas of work can be found in [RFC 23.0] and [RFC 24.0] respectively.

Much of the serviceability requirements are to be met by using the `sams` processor to create source files for messages. Coding guidelines for these source files are presented in this RFC after all guidelines for C sources files.

Use of the `sams` processor will assign each message text a unique 32 bit message id. All messages for a given component are stored in a single message catalog. They are also compiled into the program in the form of a global table that contains id-text pairs generated by the `sams` processor. There are three sets of serviceability interfaces that deal with these message id's to process messages, and support for `dce_error_inq_text`:

(a) A set of routines that log serviceability messages as routed by the serviceability component. These are meant to be used for error and warning messages (mostly in servers, but perhaps clients as well). All of these routines begin with `dce_svc_` and are documented in `svc(3)`.

(b) A set of routines that handle strings to be shown to the end-user (mostly in clients, but perhaps servers as well). These are not error or warning messages, but rather questions or prompts. They are not meant to appear in a log file. These routines are similar to the `printf` family of the ANSI C library, and are documented in `dceprintf(3)`.

(c) A set of routines that provide low level access to message catalogs. These are for the few times when complete control of the information in the catalogs is required. They are similar to the X/Open message catalog routines, but incorporate knowledge of the DCE status code space to find the proper catalogs. They all are prefixed with `dce_msg_` and are documented in `msg(3)`.

(d) The existing `dce_error_inq_text()` routine has been rewritten to use the messaging facilities. Any existing code that calls this routine may continue to do so. Most new code should use other messaging routines. The `dce_error_inq_text()` routine can be used to append the component name to the output string.

2.6.1. Initialization

There are two considerations for initialization:

(a) If there is to be an in-core message table of default messages (versus only obtaining message from a catalog and depending on the ‘‘Message number 0x12345678’’ as a default message), then the in-core table must be registered with the messaging subsystem.

(b) Each serviceability component in a DCE executable must have its own serviceability sub-component table. The table must be registered with the serviceability library and turned into a handle (`dce_svc_handle_t`) at initialization time.

2.6.1.1. Message initialization

Initialization code should contain a call to:

```
void dce_msg_define_msg_table(
    dce_msg_table_t *table,
    unsigned32       size,
    unsigned32      *status);
```

It adds a message table to the in-core table. The `table` parameter should be a message table as generated by the `sams` processor; `size` specifies the number of elements in the table. In case of an error, `status` will be filled in with an error code. An example of its use (taken from DCE Shell code) is as follows:

```
error_status_t  st = msg_s_ok;

dce_msg_define_msg_table(dsh_msg_table,
                sizeof dsh_msg_table / sizeof dsh_msg_table[0],
                &st);
/*
 * i18n not done here, since this reports an error
 * during i18n bootstraping
 */
if (st != msg_s_ok) {
    (void)
    fprintf(stderr,
        "Cannot register dsh message table, error 0x%lx\n",
        st);
    exit(1);
}
```

2.6.1.2. Serviceability initialization

There are two ways to register a table described below. When to use which method is also described; it is based on how and when initialization can happen for a component.

```
#define DCE_SVC_DEFINE_HANDLE(handle, table, comp)

dce_svc_handle_t dce_svc_register(
    dce_svc_subcomp_t *table,
    const idl_char    *comp_name,
    error_status_t    *status);

void dce_svc_unregister(
    dce_svc_subcomp_t *table,
    error_status_t    *status);
```

The first method to register a table is to create a global variable using the `DCE_SVC_DEFINE_HANDLE` macro. The first parameter is the name of the handle; the second is a pointer to the table, and the third is the name of the serviceability component. This macro creates a skeleton variable that will be completed the first time the handle is used. It should be used in cases where there is no convenient initialization code (e.g., the security library can be entered from any one of a number of API calls).

When there is a convenient location for initialization code (e.g., in an executable near the top of `main()`, or if all parts of a library call a common initialization routine), the second method is preferred since a status is returned that can be checked. The second method is to call the `dce_svc_register` routine. The first parameter is a pointer to the sub-component table and the second is the name of the serviceability component.

To destroy a handle, call `dce_svc_unregister`. This closes any open message routes and frees allocated resources. It is normally not necessary to call this routine, as enough cleanup will be done as part of the normal process exit.

2.6.2. Logging API

To generate all serviceability messages that can be written to a log file (e.g., errors, warnings, etc), the following macro and routine should be used:

```
#define DCE_SVC(handle, argtypes)

void dce_svc_printf(
    DCE_SVC(handle, argtypes) ,
    const unsigned32 table_index,
    const unsigned32 attributes,
    const unsigned32 msgid,
    ...);
```

The `DCE_SVC` macro is used to generate the initial arguments for `dce_svc_printf`. The first argument is the handle. The second is a string that describes the variable arguments to the `dce_svc_printf` call. The format of the string is the same as the format string of a standard `printf(3)`, except that only the conversion specifiers separated by a space are needed (e.g., `"%d %u %s"`), any addition characters are ignored. If the message will be routed to a binary file then the format is extended to add a `%b` specifier which takes two arguments: an integer size, and a buffer pointer. Using `%b` in a different routing will give unpredictable results.

The `dce_svc_printf` routine is used to generate all messages. The first argument must be a call to the `DCE_SVC` macro. The `table_index` is an index into the serviceability sub-component table associated with the handle. It indicates which sub-component within the component is responsible for the message. The `attributes` categorizes the message. It must be chosen from the following set (defined in the header file):

(a) `svc_c_sev_fatal`
(b) `svc_c_sev_error`
(c) `svc_c_sev_warning`
(d) `svc_c_sev_notice`
(e) `svc_c_sev_notice_verbose`

In addition, any of the following flags may be or'd in to cause additional action to happen:

(a) `svc_c_route_stderr`
(b) `svc_c_route_nolog`
(c) `svc_c_action_abort`
(d) `svc_c_action_exit_bad`
(e) `svc_c_action_exit_ok`
(f) `svc_c_action_brief`

The last item is used to suppress the standard prolog. Note that using `svc_c_route_nolog` without using `svc_c_route_stderr` will result in no message being generated.

2.6.3. Serviceability printf API

For those times when a string needs to be obtained from a message catalog but no logging event is to be generated (e.g., obtaining a question to display on the command line, or checking a locale-sensitive response) do not use the logging API, instead use the following serviceability versions of the standard `printf`-familiy routines.

(a) `int dce_printf(const unsigned32 msgid, ...);`

    The `dce_printf` routine retrieves the message text that has the specified `msgid` and prints the message and its arguments on the standard output.

(b) `int dce_fprintf(FILE *stream, const unsigned32 msgid, ...);`

The dce_fprintf routine performs the same action as dce_printf, but on the specified stream.

(c) `unsigned char *dce_sprintf(const unsigned32 msgid, ...);`

The dce_sprintf routine is a convenience function that wraps around the DCE messaging calls and the standard sprintf(3) library routine. The first argument is the id of the message containing the format string. The remaining arguments are the arguments needed by the format. The routine returns a pointer to allocated space containing the formatted output, or NULL on error. The final formatted string generated by dce_sprintf must not exceed 1000 bytes.

2.6.4. Messaging API

For those times when tighter control is needed in obtaining messages, use the following routines. An example, might be GDS's use of message catalogs to store information about curses screens.

(a) unsigned char *dce_msg_get_msg(
 unsigned32 msgid,
 unsigned32 *status);

This routine is intended to be used to get a user-visibile message string. See also the dce_printf-family of routines described above. It retrieves the text for the specified message by first searching the appropriate message catalog, and then the in-core message table. It returns a pointer to allocated space that must be unallocated by using free(). If the message cannot be found it returns a default string and fills in status with an error code. If memory cannot be allocated it returns NULL and fills in status with the msg_s_no_memory error code.

An example of its use is as follows:

```
unsigned char *mp;
unsigned32 st;

if (argc != 2) {
    mp = dce_msg_get_msg(dsh_s_wrong_num_args, &st);
    printf((char *)mp);
    free((char *)mp);
}
```

Note that calling dce_msg_get_msg inside the call to printf will make it impossible to free the string returned.

(b) unsigned char *dce_msg_get_default_msg(
 unsigned32 msgid,
 unsigned32 *status);

unsigned char *dce_msg_get_cat_msg(
 unsigned32 msgid,
 unsigned32 *status);

These routines are used when more control over where messages are obtained from is required. They obtain the message from either the in-core table of default message strings or from the message catalog respectively.

(c) dce_msg_cat_handle_t dce_msg_cat_open(
 unsigned32 msgid,
 unsigned32 *status);

unsigned char *dce_msg_cat_get_msg(
 dce_msg_cat_handle_t handle,
 unsigned32 msgid,
 unsigned32 *status);

```
void dce_msg_cat_close(
   dce_msg_cat_handle_t  handle,
   unsigned32          *status);
```

These routines provide a DCE abstraction over the XPG4 message catalog routines. The difference between these and their XPG4 counter-parts is that these take DCE message id's, and map them into message catalog names.

2.6.5. Routing API

The two routing routines may be used to specify the disposition of production and debug messages (described below). If called before the component is registered, then the disposition is stored until it is needed. In case of error, both of these routines fill in `status` with an error code. These routines a probably only needed when processing user input to decide message disposition (in DCE Shell and when processing a command-line switch to a server).

(a) void dce_svc_routing(unsigned char *where,
            error_status_t *status);

The `dce_svc_routing` routine specifies how `dce_svc_printf` messages are routed. The `where` parameter is a three-field routing parameter as described in `svcroute(5)`. This routine is normally called with the value of a command-line flag. For convenience, the first field of the routing specifier may be omitted to indicate that all levels should be routed as specified.

(b) void dce_svc_debug_routing(
   unsigned char *where,
   error_status_t *status);

The `dce_svc_debug_routing` routine performs the same action for `DCE_SVC_DEBUG` messages; `where` is four-field routing parameter as described in `svcroute(5)`. No shorthand is allowed; all four fields are required.

(c) void dce_svc_debug_set_levels(
   unsigned char *where,
   error_status_t *status);

To set the debugging level for a component, use the `dce_svc_debug_set_levels` routine. The `where` parameter should consist of a component name and a comma-separated list of sub-component/level pairs, as described in `svcroute(5)`.

2.6.6. Code for debugging

Debugging instructions will be activated if the macro `DCE_DEBUG` is defined. Defining this one macro via ODE for all of DCE will turn on all debugging statements. Further, individual components should define this macro in an internal, component-wide header file, if a component debug macro is defined. This must be done in an internal header file that other components will not be including, or else setting debugging on for one component could have an effect on another. This component debug macro is DCE_*CMP*_DEBUG, where *CMP* is the three letter acronym for the component in uppercase.[12] For example, the following should appear in a `cds.h` file:

```
#if defined(DCE_CDS_DEBUG) && !defined(DCE_DEBUG)
#  define DCE_DEBUG
#endif                       /* defined(DCE_CDS_DEBUG) */
```

2.6.6.1. DCE_SVC_DEBUG

If (and only if) `DCE_DEBUG` is defined, then `DCE_SVC_DEBUG()` is defined to be a call to `dce_svc__debug()` (otherwise it is defined to be null). It must be called with four or more arguments: a

12. One of: `THR`, `RPC`, `CDS`, `SEC`, `DTS`, `GDS`, `DFS`.

handle, a table index, a debug level, a format string, and then any parameters to the format string. Since C does not allow a variable number of arguments to a macro, the double parentheses trick[13]

is used. The `debug_level` indicates the level of detail associated with this message and must be of the form `svc_c_debugN`, where *N* is an integer from 1 to 9. Larger numbers imply more detail, but the actual meaning of the debug level is left to the individual component. The effects of each debug level must be docoumented in the [PORTG]. An example of its use follows:

```
void *allocate(int bytes)
{
    char          *new;

    DCE_SVC_DEBUG((mem__svc_handle,
                   mem_svc_alloc,
                   svc_c_debug2,
                   "Allocating %d bytes\n",
                   bytes));

    if ((new = malloc(bytes)) == NULL) {
        dce_svc_printf(DCE_SVC(mem__svc_handle, "d"),
                       mem_svc_alloc,
                       svc_c_sev_fatal |
                           svc_c_action_exit_bad,
                       mem_s_noalloc,
                       bytes);
    }

    DCE_SVC_DEBUG((mem__svc_handle,
                   mem_svc_alloc,
                   svc_c_debug1,
                   "Allocated %d bytes at 0x%x\n",
                  bytes,
                   new));
    return new;
}
```

Notes for the above code:

(a) `mem__svc_handle` is a global variable that was assigned the result of a call to `dce_svc_register()`.

(b) `mem_svc_alloc` is meant to be the index into the service table associated with the handle for the `alloc` facility of the `mem` component.

(c) `mem_s_noalloc` is the symbolic name of a message in the `alloc` facility.

```
#define DCE_SVC_LOG((handle, table_index, debug_level, \
messageid, argtypes, ...))
```

The `DCE_SVC_LOG` macro is a similar to `DCE_SVC_DEBUG` except that, like `dce_svc_printf` it takes a message identifier and a format specifier.

---

13. This is where the arguments are grouped as one argument by placing them in parentheses, so the call must have two sets of parentheses, one for the macro itself, and one to group the arguments

2.6.6.2. Other DCE_SVC_DEBUG macros

```
#define DCE_SVC_DEBUG_ATLEAST(handle, table_index, debug_level)
#define DCE_SVC_DEBUG_IS(handle, table_index, debug_level)
```

If debugging code is enabled, these macros may be used in an expression (e.g., in an if statement) to test the debug level of a subcomponent (as indexed in the serviceability table by `table_index`) for the specified `handle`. The first macro tests if the `debug_level` is not less then the specified level, while the second macro tests for an exact match. In either case, `debug_level` should be a number from 1 to 9.

2.6.6.3. dce_assert

```
void dce_assert(dce_svc_handle_t h, int expression)
```

This routine is a replacement for the [ANSI C] `assert()` macro which uses DCE style error messages as defined by the serviceability component. Its arguments are a registered DCE serviceability handle (if it's `NULL` then an internal serviceability handle is used) and an expression.

On execution, if the expression evaluates to zero, then `dce_svc_printf()` is called with parameters to identify the expression, and source file and line number. The message is generated as a `svc_c_fatal` message, with the `svc_c_action_abort` flag.

Assertion-checking can be enabled or disabled at compile-time. If the macro `DCE_ASSERT` is defined, then assertion checking is enabled. If it is not defined, it is compiled out of the program.

To use this routine include the header file `dce/assert.h`, which is shown in Appendix H.

2.7. Internationalization

In addition to following the messaging guidelines given above, all code should follow the internationalization guidelines described below.

(a) Use `setlocale()`.

All DCE programs that do locale-sensitive processing (i.e., display a date or time, generate messages, or compare lexicographic strings) must use `setlocale()` as described in [I18N]. Library routines should not call `setlocale()`.

```
(void) setlocale(LC_ALL, "");
```

(b) Localized Date and Time Format.

All non-DTS-style time which is presented to the user, must be formatted with `strftime()` as shown in [I18N] section 5.5. To convert from a formatted time string to a `tm` structure, use the routine `strptime()`. Converting the code to use DTS style time is recommended.

```
#include <time.h>
#include <stdio.h>
#include <locale.h>
#include <langinfo.h>

main()
{
    char        buff[BUFSIZ];
    time_t      ct;
    char        *dt_fmt;            /* date/time format */
    struct tm   *local;             /* local time */
```

```
        (void) setlocale(LC_ALL, "");

        /* get the time in secs and convert to tm format */
        ct = time(NULL);
        local = localtime(&ct);

        /* get format from locale */
        dt_fmt = nl_langinfo(D_T_FMT);

        /* Now format the string and print it */
        if (strftime(buff, BUFSIZ, dt_fmt, local)) {
            printf("%s\n", buff);
        }
    }
```

(c) Yes and No User Responses.

Internationalized user questions that require an affirmative or negative (i.e., yes/no) answer must use the `dce_rpmatch()` routine to parse the response.. If `dce_rpmatch()` is not used, the prompt must indicate appropriate responses (e.g., y/n).

(d) Character Manipulation.

Remember that DCE code needs to support multiple code sets. Even those areas that are restricted to portable characters must support both ASCII and EBCDIC; other areas must handle a larger range of encodings. Keep in mind the following:

(i) Only a few places in DCE will need to worry about wide characters or multi-byte data. This is due the use of the Portable Character Set (PCS) for some data. DCE code will *only* need to worry about either of these, if the data can come from a user and is not limited to a particular code set by some DCE code.

For example, all code that accepts and processes a cellname can be single-byte since cell names are limited to the PCS. However, if extended registry attributes are implemented then all code that accepts or processes them, must be wide-char or multi-byte capable. See [I18N] section 5 for details.

(ii) Hard-coded character constants must be removed from all non-protocol related code. Instead of `\141` use `'a'`.

(iii) Use of `isascii()` must change to `isdcepcs()`.

(iv) Dependencies on code set constructions, such as whether or not numbers are contiguous in the code set, must be removed from all code.

(v) Arithmetic on character data must not be performed. To change the case of a character do *not* add or subtract a value to it, call `toupper()` or `tolower()` (for single-byte cases).

(vi) Range checking on character data should only be performed using one of the library routines such as `isprint()` or `islower()`. In particular do *not* do something like the following when doing an `isalpha()`-like check.

```
if ((code >= 'a') && (code <= 'z'))
```

since, in EBCDIC the values for 'a' to 'z' are not contiguous.

(vii) For collation of single-byte or multiple-byte data, use one of the following:

[a] `strcoll()` by itself

[b] `strxfrm()` followed by one or more calls to `strcmp()`.

When collating wide character data use the similar functions `wcscoll()` or `wcsxfrm()` and `wcscmp()`.

(e) Guidelines For New Protocols.

If a new protocol is being added to DCE, refer to [RFC 23.0] for mechanisms to use to allow for internationalized data to be properly handled.

2.8. Include Files

All header files in DCE 1.1 must be protected from duplicate inclusion in the following manner:

```
#ifndef _HEADER_H
#define _HEADER_H
    /* ... body of header.h ... */
#endif /* _HEADER_H */
```

where `HEADER` is the base name of the header file in uppercase.

3. SAMS SOURCE FILES AND MESSAGES

In DCE 1.1 all user-visible messages are to be isolated into message catalogs.[14] These message catalogs must be maintained using the `sams` processor and should follow internationalization guidelines.

3.1. sams

Files which will be input to `sams` should end with the extension `.sams`. The `sams` processor will generate several files as described below. In all of the following filenames, *cmp* is the serviceability component name. Also the `dce` is generated based on the technology name, for all core components this will be `dce`, for DFS it will be `dfs`.

(a) `dce`*cmp*`msg.h` — Header file defining constants for message numbers.

(b) `dce`*cmp*`msg.c` — In-core table mapping message numbers to default text.

(c) `dce`*cmp*`.msg` — Message catalog file for input to the `gencat` program.

(d) `dce`*cmp*`.cat` — Message catalog generated by `gencat`.

(e) `dce`*cmp*`msg.man` — A subset of a man page containing error codes and their descriptions.

(f) `dce`*cmp*`msg.sgm` — Markup file for Problem Determination Guide.

(g) `dce`*cmp*`svc.h` — Serviceability header file.

(h) `dce`*cmp*`svc.c` — Serviceability sub-component table.

(i) `dce`*cmp*`mac.h` — Convenience macros to be used in calls to serviceability routines.

The input file is divided into three parts. A header containing information used in processing the whole file, such as the component name. A second part used to define information about the serviceability table[15] which can be generated by `sams`, and final part which describes each message.

14. The isolation of debug messages is not required, although debug messages are to use the provided serviceability interfaces.

15. This is also known as the sub-component table as it has one entry for each sub-component.

3.1.1. The header

The header of the `.sams` file syntacticly must contain the `component` field with the serviceability component name.[16] This style guide also requires the use of the `technology` field. Most components will set this to the default of `dce` but DFS will have to set it to `dfs`. All other fields are optional and can (and in most cases should) be left out.

3.1.2. The serviceability table information

The second part of the file is used to specify the DCE serviceability table and handle. All `.sams` files that define serviceability messages should contain such a specification. It should appear in the following format:

```
serviceability table table_name handle handle_name start
    sub-component table-index subcomp full_descrip_id
    sub-component table-index subcomp full_descrip_id
    ...
end
```

Where:

(a) *table-index*

The name of a macro whose value will be generated by `sams`. It is used as an index into the serviceability table. It is also used as a value of a `sub-component` field as presented below. It is recommended that the name be *cmp*`_svc_`*subcomp*.

(b) *subcomp*

A one word name that will only be used to identify a sub-component to route debug messages.

(c) *full_descrip_id*

The code of a message listed somewhere in the file that provides the full description of the sub-component. It is recommended that the name be *cmp*`_i_svc_`*subcomp*.

An example follows. Note that portions of the third part of the input file are listed here. The syntax of that portion of the input file is described in the next section. This style guide recommends listing all the messages that are used to describe a sub-component together before any other messages in the input file.

```
serviceability table dts_svc_table handle dts_svc_handle
start
    sub-component dts_s_general "general" dts_i_svc_general
    sub-component dts_s_provider "provider" dts_i_svc_provider
end

## Messages for serviceability table
start       !intable undocumented
code        dts_i_svc_general
text        "General administrative actions"
explanation ""
action      ""
end
```

---

16. This component name must be a three-letter word chosen from the set of 38 characters, `a-z_0-9-` (i.e., the RAD40 alphabet).

```
start       !intable undocumented
code        dts_i_svc_provider
text        "Interactions with time-providers"
explanation ""
action      ""
end
```

3.1.3. The messages

The final part of the file defines the individual messages. It consists of a series of records where each record specifies one message. Each record is of the form:

```
start optional_flags
field_list
end
```

The *field_list* is a set of key-value pairs from the following lists. All messages must have the following fields:

(a) `code` *name*[=*value*]

This is the symbolic name of the message. It is used to create a header file that has `#define` statements that map all message ID's to their numeric value. It also appears in the generated `.man` file. If the text of this field is of the form `name=value`, then that the numeric value is explicitly assigned as opposed to being generated with a simple counter. This should be used only when needed to ensure compatibility with older message versions.

(b) `text` *text*

This is a short description of the error. It is stored in the in-core table of default messages and in the message catalog. It is intended to be displayed by messaging routines. The text must be shorter than the size of the `dce_error_string_t` typedef defined in `dce_error.h`.

(c) `explanation` *text*

This is a verbose description of the message; it can be blank if the message is not for an error condition. It is used in the `.man` and `.sgm` files and should provide additional information that can be used for fault isolation.

(d) `action` *text*

The text describes the action that should be taken when this error occurs. The text appears in the generated `.sgm` file.

All messages used in serviceability calls must have the following two fields. They are both only used to generate convenience macros in the `dce`*cmp*`mac.h` file.

(a) `attributes` *text*

Indicates the attributes assigned to the message. Multiple attributes can be separated by |'s. Valid attributes are defined in `dce_svc.h`; do *not* use the ones reserved for vender use. Valid ones are listed below.

(i) `svc_c_action_abort`
(ii) `svc_c_action_brief`
(iii) `svc_c_action_exit_bad`
(iv) `svc_c_action_exit_ok`
(v) `svc_c_action_none`
(vi) `svc_c_route_nolog`
(vii) `svc_c_route_stderr`
(viii) `svc_c_sev_alarm`
(ix) `svc_c_sev_error`
(x) `svc_c_sev_fatal`

(xi) `svc_c_sev_notice`
(xii) `svc_c_sev_notice_verbose`
(xiii) `svc_c_sev_warning`

(b) `sub-component` *`table-index`*

Indicates the sub-component that the message belongs to. The value must be one of the *`table-index`*'s listed in the serviceability table description.

The following fields are optional:

(a) `notes` *`text`*

Optional notes for translators. This text, if it exists, appears as comments in the message catalog. Note that the internationalization guidelines specify that translator notes must be used copiously. See section 2.6.1.3 item (f) for details.

(b) `tables (`*`keywords`*`)`

If a single component is used for several executables, the message table can get unreasonably large, containing texts that will never be used. Using the `table` command, a space-separated list of *`keywords`* can be assigned to each message. If the `sams` processor is invoked with a `-t` switch specifying a keyword, then only those messages having that *`keyword`* in the table field will be included in the generated in-core default message table.

(c) `engineer` *`text`*

This is used to specify the software engineer responsible for the code where this message could occur. It may be omitted (and will be in most cases).

3.2. How to Write Proper Messages

In addition to using the `sams` processor described above, all messages must meet internationalization and guidelines described in [I18N]. They are summarized below:

(a) Messages must be clear and easy to understand. Remember that translators will be working from the message catalog only, and may not understand the what the code does. Do *not* use slang.

(b) Messages must *not* use abbreviations. Examples of bad abbreviations are ''pkt'' and ''ack''. Some common acronyms are permissible such as DCE, official component names (e.g., thr not cma), and command names (e.g., cdsd instead of the CDS server). If it is not clear if an acronym is acceptable, do not use it. If it must be used, then include a `notes` field in the message description to define it.

(c) Messages must *not* use fragmentation. Do not use common C tricks to make words in strings plural or to negate them. The following code should be changed to use two separate messages:

```
printf("Server directory access is %sokay\n",
        (result ? "" : "not "));
```

Messages that have fragments can be difficult or impossible to translate. This does not mean that variable substitution cannot be used in messages, however, it must be used for complete units such as a filename, a numerical amount, a process name, etc.

(d) Initial message strings (i.e., the default messages) do not need to use the XPG4 variable parameter syntax; it is unnecessary since the argument order is correct for the default string, and it slows down the code.

(e) Default text must be included in the `catgets` call and must be in English encoded in ASCII. Proper use of the messaging API and the `sams` processor will ensure that this requirement is met with no explicit action required by the programmer.

(f) Notes must be included in the message catalog (i.e., the `.sams` file) for any item needing further explanation. The example in the above item referring to variable parameter substitution used a string to represent the data. If the catalog contained only the string, then a translator would have no way to know what it is in order to translate it. An explanation of this type message is required in the `notes` field of the `.sams` file.

In this case it should say something like: ‘‘This is a data format string. The first field is the month, the second is the day, and the third is the year.’’

Notes should also be used to explain any potentially ambiguous parts of the message and to specify words or phrases that should not be translated. For example, in the message string `"The read system call failed. Cannot read %s"`, the notes section should say something like: ‘‘Do not translate "read" in the phrase "read system call". %s is a filename.’’

The following guidelines were developed by the DCE Technical Writers for writing message text in `sams` files. Follow these guidelines in addition to the internationalization guidelines stated previously. All new messages must, and all existing messages should meet these guidelines.

(a) The most important thing is to be clear and concise. When in doubt refer to this guideline.

(b) The `text` field should be descriptive and not prescriptive. In other words, describe the problem, not what to do about it. The `action` field should be used to describe corrective actions to be taken.

(c) Use complete short sentences in the `explanation` and `action` fields. Sentence fragments may be used in `text` fields. Sentence fragments do not need to end with a period (or other punctuation), but complete sentences must. This means a message like ‘‘file not found’’ is acceptable, although it would be better if it gave the name of the file.

(d) Use active voice and change passive voice to active voice wherever possible. For example, the following sentences are in passive voice:

```
The job was cancelled.
Month and year must be entered as numbers.
Data was updated by administrator.
```

The following sentences are in active voice:

```
Job cancelled by operator.
Enter month and year as numbers.
The administrator updated data.
```

(e) Use correct case for macro names, commands, keyword parameters, module names, and acronyms. Capitalize the first word of a sentence, but not a sentence fragment. Do not begin a sentence with a command name that must be in lowercase. Use ‘‘Use cdscp to...’’, instead of ‘‘cdscp is used...’’.

(f) In `text` fields, do not use any mechanism (e.g., double or single quotes, asterisks, uppercase, SML markups) to show emphasis for words. However, if the text contains a literal (e.g., a filename, command name, or any name) do use the appropriate fonts for literals as described below.

In `explanation` and `action` fields use the standard SML markup for bold, italic, and constant width fonts. To begin the font change use the appropriate markup from the following table. To end it, use `\*O` (that’s a capital letter ‘‘O’’) to return to the ordinary text font.

(i)	Commands	`\*L`
(ii)	Keywords	`\*L`
(iii)	Functions	`\*L`
(iv)	Data types	`\*L`
(v)	Command Options	`\*L`
(vi)	Arguments	`\*V`
(vii)	Pathnames	`\*L`
(viii)	Return Values	Uppercase
(ix)	Sample Commands:	
	[a] user input	`\*L`
	[b] system response	`\*C`

(g) Follow internationalization practices for expressing plurality. Do not use ‘‘(s)’’ to show possible plurality (e.g., ‘‘error(s) cannot be translated’’). Two separate messages should be written and output appropriately.

(h) Indicate ranges using the word ‘‘to’’ (do not use a dash, or the word ‘‘through’’) and by being inclusive. Include the first and last elements in the range. If the legal values for an entry are 0, 1, 2, and 3, use ‘‘The legal values are 0 to 3.’’

(i) Do not use the construction ‘‘and/or’’; it does not exist in other languages. Usually it is better to use ‘‘or’’ to indicate that it is not necessary to do both.

(j) Do not use the first or second person (i.e., ‘‘I’’, ‘‘we’’, or ‘‘you’’) in `text` fields; however, they are acceptable in `explanation` and `action` fields.

(k) You can use contractions, as long as they’re grammatical contractions (e.g., ‘‘don’t’’, ‘‘you’ve’’, etc.). Do not use non-standard contractions or abbreviations for words; for example, ‘‘req’d’’ for ‘‘required’’ or ‘‘ack’’ for ‘‘acknowledged’’.

(l) Only use acronyms that are better known to your audience than their spelled-out version (e.g., ‘‘DCE’’ instead of ‘‘Distributed Computing Environment’’). If some acronym is border-line, use the acronym in the `text` field and the spelled-out version in the `explanation` or `action` field.

To make a plural of an acronym, add a lowercase ‘‘s’’, without an apostrophe. A list of acronyms is provided in the next section.

(m) Use the present tense. For example, use ‘‘The foo command displays a calendar.’’ instead of ‘‘The foo command will display a calendar.’’ In general, don’t mix tenses. Of course there are some cases where it is necessary to refer to something that happened in the past, and what the program is now doing in response.

(n) Use positive language. For example, ‘‘Do not use the f option more than once.’’ uses negative language, while ‘‘Use the -f flag only once.’’, uses positive language. There are some exceptions; for example, ‘‘file does not exist’’ is clearer and more easily understandable (and translatable) than the message ‘‘file nonexistent’’. In `explanation` and `action` fields, make positive statements when you can. Explanations may or may not be positive, but actions should always be positive. Avoid using double negatives.

(o) Use the imperative mood (command phrase) and active verbs; for example, ‘‘specify’’, ‘‘use’’, ‘‘check’’, ‘‘choose’’, and ‘‘wait’’. This applies for all fields, but will be relevant mostly to `explanation` and `action` fields.

(p) Do not use humor, slang, or your company’s internal jargon. Industry-standard terminology such as ‘‘queue’’, ‘‘toggle’’ or ‘‘click’’ is acceptable.

(q) Be as precise as possible when refering to objects such as a file, cell, principal, etc. Use the specific name. If it is appropriate, use an article (i.e., ‘‘a’’, ‘‘an’’, or ‘‘the’’). The following three examples are bad, better, and best:

  (i) Fix client configuration.

  (ii) Fix the client configuration.

  (iii) Fix configuration of client /.../my_cell.osf.org/hosts/gumby

(r) Once you use a name for an object, continue to refer to it the same way. For example, if you call something ‘‘the cdsd.log file’’ in one place within the message explanation, do not refer to it as the ‘‘log output’’ later.

(s) When in doubt refer to a dictionary. Do this for spelling and grammar. Don’t use nouns as verbs (e.g., use ‘‘enter your response’’ rather than ‘‘key in your response’’,). If you’re still in doubt, ask a writer.

(t) While it is not concise to repeat things, go back and reread guideline (a).

3.2.1. Approved acronyms in messages

Here is a list of DCE acronyms that can be used in message texts:

(a) ACF Attribute Configuration File
(b) ACL Access Control List
(c) ACSE Association Control Service Element

(d)	API	Application Programming Interface
(e)	ASN.1	Abstract Syntax Notation One
(f)	AVA	Attribute Value Assertion
(g)	C-ISAM	C-language Indexed Sequential Access Method
(h)	CDS	Cell Directory Service
(i)	CPU	Central Processing Unit
(j)	DAP	Directory Access Protocol
(k)	DCE	Distributed Computing Environment
(l)	DFS	Distributed File Service
(m)	DIB	Directory Information Base
(n)	DIT	Directory Information Tree
(o)	DLC	Diskless Configuration Service
(p)	DNS	Domain Name Service
(q)	DSA	Directory System Agent
(r)	DSP	Directory System Protocol
(s)	DTS	Distributed Time Service
(t)	DUA	Directory User Agent
(u)	GDA	Global Directory Agent
(v)	GDS	Global Directory Service
(w)	IDL	Interface Definition Language
(x)	ISO	International Organization for Standardization
(y)	LAN	Local Area Network
(z)	LFS	Local File System
(aa)	NSAP	Network Service Access Point (OSI)
(ab)	NTP	Network Time Protocol
(ac)	OSF	Open Software Foundation
(ad)	OSI	Open Systems Interconnection
(ae)	OSS	OSI Session Service
(af)	RDN	Relative Distinguished Name
(ag)	ROS	Remote Operation Service
(ah)	ROSE	Remote Operation Service Elements
(ai)	RPC	Remote Procedure Call
(aj)	TCP/IP	Transmission Control Protocol/Internet Protocol
(ak)	TDF	Time Differential Factor
(al)	UDP/IP	User Datagram Protocol/Internet Protocol
(am)	UFS	UNIX File System
(an)	UTC	Coordinated Universal Time
(ao)	UUID	Universal Unique Identifier
(ap)	WAN	Wide Area Network
(aq)	XDS	X/Open Directory Service
(ar)	XOM	X/Open OSI-Abstract-Data Manipulation

4. FILENAMES AND DIRECTORY NAMES

The names of files and directories (in the filesystem) used by DCE services must comply with the following:

(a) There must be at most one period in filenames and it should be immediately before the extension.

(b) There must be no periods in directory names. The only exceptions to this rule are directories under `/opt` such as `dce1.0`

(c) [RFC 35.0] must be followed for choosing where files will reside.

4.1. Location of core Files

All processes implementing DCE core services should, if at all possible, change directory into a *running* directory early on so that any core dumps they generate will be easy to locate, and not show up in the root directory of the system. The path will be:

`/opt/dcelocal/var/`*`component-name`*`/adm`

This path is what is documented in [RFC 35.0] as where servers should place log files, caches, and core dumps. These directories should be owned by root and mode 700 by default, to protect the core dumps (which probablay contain encryption keys) from prying eyes.

5. BUILD ENVIRONMENT

While DCE is generally thought of as C source code, those who port DCE learn very quickly that the DCE distribution comes with some very complex `Makefile`'s as well. In fact, many of the complaints about DCE's portability are related to the `Makefile`'s. In order to make porting DCE as easy as possible, the following guidelines must be followed when writing `Makefile`'s and when addressing other build environment issues.

5.1. Makefiles

The following rules must be followed in all `Makefile`'s in DCE. See Appendix I for overview information about ODE and a description of the variables that ODE defines that can be used in `Makefile`'s.

(a) One Target per Makefile or Directory.

All new build targets (programs, libraries, etc.) must ''live'' in their own directory, with a `Makefile` that builds only that target. This is desirable for existing targets as well, but is not required. However, existing directories that have two targets where one is a library and one is a program that depends on that library, should be separated wherever possible. This particular situation exacerbates a bug in `make`.

(b) Avoid Target-Specific Variables.

Almost all variables can be set in a target-specific manner, by setting *`target_variable`*. For example, compiler flags for only the `cdsd` target could be set in the `Makefile` by setting `cdsd_CFLAGS`. This should be *avoided* wherever possible by having only one target in a directory. However, where this is not avoidable, care should be taken so that the defaults for these variables (those variables which are not target-specific) are set in a general manner, and only a few exceptions needed are set in a target-specific manner.

(c) Use the `install` Tree Properly.

The goal of the build is to populate the `install` tree. Both the `obj` and `export` trees are intermediaries in achieving this goal. All programs, libraries, header files, and datafiles used by an end-user; and all tests, scripts and datafiles used by a DCE developer must be in the `install` tree after the build completes successfully. ODE will look first in the `obj` tree and then in the `src` tree to find targets.

(d) Use ODE Passes Properly.

ODE implements the build process as a number of passes. Each pass performs different actions. Directories in the `src` tree should either contain source code, or contain subdirectories, not both.[17] For those that contain subdirectories, they must have a `Makefile` that defines the appropriate variables for subdirectory traversal by `make` and nothing else, except for some logic if necessary, such as not visiting the `src/file` directory if `NO_DFS` is defined. The variables must be listed in the order the passes are performed.[18]

(e) Describe All Dependencies in `Makefile`'s.

All dependencies must be listed in `Makefile`'s. This includes library dependences for executables and any cross-directory dependencies. Do *not* depend on another pass being run the another directory to create dependencies (e.g., idl being run on a `.idl` file in a headers directory creating stub files). It is permissible to depend on another pass being run in the same directory, or on an export pass being run in another

17. The exception are subdirectories that handle machine dependencies are permitted in directories with source code.

18. This is not an ODE requirement, but rather a stylistic requirement.

directory.

(f) Use Appropriate `Makefile`'s.

Makefile information must be properly placed in either a `Makefile`, `machdep.mk` file, or the `src/Makeconf` file. Note that *`component`*`.mk` files are being removed in DCE 1.1. The information currently in these files must be moved to the appropriate `Makefile`.

(g) Use of `VPATH` is discouraged.

(h) Use Libraries Where Needed.

If an object is to be included in more than one executable it must be in a library and that library should be linked into each executable.

(i) Build DCE in NFS Mounted Sandbox.

DCE should be able to be built in a sandbox which is mounted via NFS. Currently this is not possible due to the way a database is created for GDS. This must be investigated and corrected if possible. This work is to be done by provider companies with support from the OSF Release and Tools Group.

(j) Structure of New Source Subtrees.

New subtrees must be created in the following style. There should be one target per subdirectory. Each target should reside in a directory which is named the same as the target. Component wide header files should reside in a single subdirectory as should `.idl` and `.sams` files.

(i)	`comp`	
(ii)	`comp/h`	all headers that are component wide
(iii)	`comp/idl`	IDL files
(iv)	`comp/sams`	SAMS files
(v)	`comp/lib`	directory for libraries
(vi)	`comp/lib/lib`*A*	contains source for library *A*
(vii)	`comp/lib/lib`*B*	contains source for library *B*
(viii)	`comp/`*foo*`d`	all sources for the *foo* daemon
(ix)	`comp/`*foo*`cp`	all sources for the *foo* control program
(x)	`comp/`*foo*`clerk`	all sources for the *foo* clerk

(k) IDL Files.

Note to Readers: This needs to be defined by the RATs. Included here are the requirements from [RFC 17.0]. The right thing is to define a variable such as `IDLFILES` *to set all* `.idl` *files to, and let* `make` *do the rest. This might involve a separate pass, so that other* `Makefile`*'s will have to change to follow this path. It also might involve an enhancement to* `idl` *to add a switch that searches for the* `.acf` *over a path, or doesn't produce a* `.h` *file.*

There shall be a rule for creating all stubs and headers from `.idl` files so that they are not built more than once as they currently are. This is one of the few things preventing DCE from using the parallel `make`. The rule will create these files once, probably in their own pass after the headers are exported and before the libraries are exported (pass=SECOND?). It might involve a new switch to `idl` being created so that it can be run without generating a header file. Also, there are some places in DCE (e.g., Security) where the same `.idl` file is processed more than once with different `.acf` files so that the generated stubs are different. This must be done in an orderly way. The OSF Release and Tools Group will be responsible for determining the solution to this problem. Provider companies will implement the solution. Since the number of `.idl` files is small, this is not expected to be a large work item.

(l) SAMS Files.

Note to Readers: This needs to be defined by the RATs. I expect these files to be handled similarly as IDL files. The design work should be simpler since it only takes one input file. A suggestion for the variable name is `SAMSFILES`.

5.2. Machine Dependency Handling

The best way to handle machine dependencies is not to have them in the first place. To this end, machine dependencies should be removed and replaced with feature dependencies. For example, instead of coding endianess specific code in `#ifdef`'s for `mips` or `AIX`, they should be coded in an `#ifdef` that checks for `LITTLE_ENDIAN` or `BIG_ENDIAN` which will be defined in `dce.h`. Feature macros should be defined in the header file that most fits their scope. The endian macros mentioned above will be in `dce.h` as they could be used for many components, however, the `MISPACKED_HEADER` macro used only by RPC will be defined in an RPC header file. All feature macros must be documented in the *DCE Porting and Testing Guide*.

There will be cases where machine dependent code is necessary. Machine dependencies must be divided into those dependencies on machines (i.e., hardware) or on operating systems (i.e., system software). The term ''machine dependency'' will be used to refer to both. Wherever possible, machine dependencies must be separated out into functions that have interfaces documented in the *DCE Porting and Testing Guide* and that are implemented in machine dependent files. Where the code involved is small and isolated[19] then an `#ifdef` for a machine defined macro is appropriate. Another qualifier to use is how many machine dependencies appear in a file. If there are many sections of code that are small and isolated, then the whole file should be considered as a machine dependent file.

5.2.1. Machine dependent files

> *Note to Readers: This needs to be defined by the RATs. Included here are the requirements from [RFC 17.0]. My guess is that the resolution will involve some approved form of conventions for organizing machine and operating system dependent files under subdirectories, and the use of* `VPATH` *with* `TARGET_OS` *and* `TARGET_MACHINE`*, or their equivalent, in the* `Makefile`*'s.*

The methods of handling machine dependencies in DCE are both varied and unwieldy. First of all a solution to the PMAX versus MIPS debate must be finalized. It will include how to differentiate between `TARGET_MACHINE` and `TARGET_OS` and even `TARGET_OS_VERSION` (how will differences between OSF/1 1.0 and OSF/1 1.1 be handled?). The OSF Release and Tools Group will be responsible for determining the solution to this problem. Provider companies will be responsible for the implementation.

5.2.2. Machine defined macros

In those cases where machine dependent conditionalized code (i.e., `#ifdef`'s) is appropriate, the following macros are to be use in the check. They are the macros that the compilers used on these platforms define. In the table, platform names are the informally defined ones used up through DCE 1.0.2 development.

	Platform	*Machine*	*Operating System*
(i)	PMAX	`__mips__`	`__OSF__`
(ii)	AT386	`__i386__`	`__OSF__`
(iii)	AIX	`_IBMR2`	`_AIX`
(iv)	SVR4	`i386`	`unix`
(v)	HPUX	`__hppa`[20]	`__hpux`
(vi)	RISC Ultrix	`__MIPSEL`	`ultrix`
(vii)	VAX Ultrix	`vax`	`ultrix`
(viii)	VAX VMS	`vax`	`VMS`
(ix)	AXP OSF/1	`__alpha`	`__osf__`
(x)	AXP VMS	`__alpha`	`__VMS`

---

19. As a guideline, code that is under 10 continuous lines, closely tied to surrounding code, and not quite worth the overhead of a function call can be considered ''small and isolated''.

20. Having `__hppa` defined means that the machine supports the HP Precision Architecture rev 1.0 or greater, (i.e., all PA machines). Having `_PA_RISC1_1` defined means that the machine supports the HP Precision Architecture rev 1.1 or greater, (i.e., all s700 and all new s800 machines).

5.2.3. Machine dependencies in Makefile's

Note to Readers: This needs to be defined by the RATs. Included here are the requirements from [RFC 17.0]. The complaints are that there are too many places to look to find how `make` *variables are set, and some say that* `component.mk` *files are slow since they do tests for each directory. There are several choices:*

- *Leave things the way they are. Perhaps a cleanup of the* `component.mk` *files by the RATs could make things a little better than they are now.*
- *Remove* `component.mk` *files completely and put the information either in the* `Makefile`*'s (some say this will confuse the makefiles), or* `machdep.mk` *files (some don't like lots of little* `machdep.mk` *files, or in* `Makeconf` *or common makefiles as appropriate.*
- *Keep the* `component.mk` *files for only component wide things (not DCE wide things, they should be in* `Makeconf` *or the common makefiles if they are general to ODE and not DCE) and place all the directory specific stuff in either the* `Makefile`*'s or* `machdep.mk`*'s.*

The `component.mk` files will be removed and replaced with `machdep.mk` files. Currently all the components have a top level file which sets machine specific flags for each directory in the component, that is named with the name of the component followed by `.mk`, for example, `src/rpc/rpc.mk`. The result is that many (not all) of the machine specific flags are in one place. However, the build is longer since this lengthy file must be read every time that `make` is invoked, porting is less clear since machine dependencies occur in `component.mk` files, in the `osf.dce.mk` file, in the `Makeconf` file, in `machdep.mk` files, in the `Makefile`, etc. These will be isolated so that there is one file with project wide machine specific settings (either `Makeconf` or `osf.dce.mk`) and then separate files with machine specific settings in each directory where needed. These `component.mk` files were created to simplify porting, but they seem to have just made it more complicated. The OSF RATs will be responsible for this work.

6. RECOMMENDED CODING CONVENTIONS

This section describes recommended coding conventions. These conventions are not required for either old or new code. If the new code fits into another component with rigorous coding conventions, follow them. However, these are probably suitable for a new component. They are based on the coding conventions used in the RPC source.

This section uses the traditional DCE definition of component. A component is defined to be made up of facilities. For example, the rpc component has the following facilities: com, dg, cn, ip, ns, mgt, etc.

6.1. File Organization

Source files should be kept to a reasonable size. Reasonable must be determined by the developer, but a guideline is that a source file should implement some functionality that can be stated in a single concise statement, such as ''This file implements the messaging facility.'' For those that need a more precise guideline, 2000 lines is about as big as it should get.

The organization of `.c` files should be as follows:

(a) File header comment.

(b) `#include` preprocessor directives.

(c) `#define` and `typedef` declarations that are internal to this file.

(d) Internal variable definitions.

(e) Function declarations (i.e., prototypes) for all the internal functions .

(f) Function definitions.

A limitation of ODE is that all header files should be included by using angle brackets and not double quotes. In order to simplify the include file structure and make things as explicit as possible, header files should not be nested. The organization of `.h` files should be as follows:

(a) File header comment.

(b) Protect the header file from duplicate inclusion.

(c) `#define` and `typedef` declarations.

(d) Function declarations (i.e., prototypes).

6.2. Naming Conventions

All names are to be formed from the set of lowercase portable letters (a-z), digits (0-9), and the underscore character (not a dollar sign). Macros are to be in uppercase portable letters (A-Z) only. All names must be unique to 31 characters in length. All names must begin with a prefix that identifies the subcomponent to which it belongs. Names should be long enough to be descriptive. Avoid the need to notice the difference between 1 (a digit), l (a lowercase letter), and I (an uppercase letter).

File names should begin with a facility prefix which should be no longer than three characters. C source files, should be named as the facility prefix followed by a name for the file. If sub-facilities are required, they should be included after the facility prefix. C header files, should be named as a facility prefix followed by a name for the header file. If sub-facilities are required, they should be included after the facility prefix. If the header file is private, there should be a `p` before the `.h` extension.

Name functions, variables, and constants, etc., as follows:

(a) *comp_fac_name* public function
(b) *comp__fac_name* private function
(c) *comp_g_fac_name* global variable
(d) *comp_s_fac_name* status codes
(e) *comp_c_fac_name* constants
(f) *comp_e_fac_name* enumeration values

If *name* is composed of more than one word, they should be separated by underscores.

Macros should be named in all uppercase unless they are specified by a standard to be otherwise (e.g., `getchar()`). If they are behaving as some other C object that has naming conventions, they should be followed, but should still be in uppercase.

The names of `typedef`'s and `struct`'s should be formed as follows. Always create a tag for a `struct`.

(a) *comp_fac_name*`_t` typedef
(b) *comp_fac_name*`_p_t` pointer to typedef
(c) *comp_fac_name*`_s_t` struct tag

For example:

```
typedef struct rpc_handle_s_t
{
    struct rpc_handle_s_t *next;
    int                    date;
} rpc_handle_t, *rpc_handle_p_t;
```

6.3. Comments

There are four types of comments:

(a) All files should have the file header comment at the beginning of the file. Define a new style or use one from DCE code.

(b) All functions should have the function header comment immediately before their definition. Define a new style or use one from DCE code.

(c) Block comments are used to document a section of code. They should start at the same level of indentation as the code they describe. Use them liberally.

(d) In-line comments appear on the same line as the code they describe. If the comment does not fit on the same line, then use a block comment above the line.

6.4. Functions

Keep the length of functions reasonable, in general under 100 lines. Always explicitly define type of the return value of a function, don't let it default to `int`. If there is no return value, define the function as returning `void`. Avoid side effects (implicit parameters) wherever possible.

Parameters should be ordered with all input parameters first, then all input-output parameters, and finally all output parameters. Any function which can fail should return an output status parameter.

Input parameters which are pointers to a `typedef`, should be defined with the type *type*`_p_t`. Input-output and output parameters which are pointers to a `typedef`, should be defined with the type *type*`_t *`. Aside from making the code easier to read (an input parameter will never be prefixed with a `*` while, input-output and output will), idl requires that input-output and output parameters have an explicit top-level `*`.

Some existing DCE code uses macros (e.g., EXTERNAL, GLOBAL, PUBLIC, PRIVATE, and INTERNAL) just before the function definition to define the scope of functions. Since C does support such a rich set of scope definitions, they are usually defined to be nothing and are used for commenting purposes only. Often these macros have the effect of confusing tools that try to parse C code (admittedly poorly). Do *not* use these macros, instead use the C storage class specifiers `static` or `extern`; if they do not provide enough specification, include more in a comment block.

6.5. Statements

Each line should contain at most one statement. If a statement must be broken at line boundaries, indent the second line to same level as the logic. So parameters that are on separate line should all line up vertically, or expressions that are broken onto separate lines all line up logically as in the following example:

```
foo_p->size = strlen (bufp->prefix)+
              strlen (bufp->text)+
              strlen (bufp->suffix)+
              strlen (bufp->prefix);
```

Use blank spaces between reserved works (or function names) and a following open parenthesis. For example:

```
if (condition)
```

instead of:

```
if(condition)
```

Avoid using the `goto` statement.

Switch statements should always have a `default` case. If it is not reachable then it should indicate a fatal error. All cases should have a `break` statement, do not let one case fall through to another. Several `case` labels can be associated with one block of code.

```
eow = 0;
while ((c = getchar ()) != EOF)
{
    switch(c)
    {
        case '\n':
        case '\t':
```

```
            case ' ':
                if (c == '\n')
                {
                    lines++;
                }
                words += eow;
                eow = 0;
                letters++;
                break;
            default:
                letters++;
                eow = 1;
                break;
        }
    }                   /* while */
    words += eow;
```

Use the brace style as shown below, with an indentation level of 4 spaces. Always use braces around the bodies of control structures, even if the body is only one line, as follows:

```
if (foo == 5)
{
    count +=7;
}
```

6.6. Expressions and Operators

Use parentheses whenever there are more than one operators involved in an expression to make things clear. Spaces should surround binary operators (e.g., `=`, `+`, `+=`), but there should be no spaces between a unary operator and its operand.

APPENDIX A. GCC 2.3 OPTIONS

The following descriptions of options is taken from the GNU C Compiler 2.3 documentation (with some reformatting) [GCC]. All language and warning options are listed, except those that are for C++ code only.

A.1. Options Controlling Dialect

The following options control the dialect of C or C++ that the compiler accepts:

(a) `-ansi`

Support all ANSI standard C programs.

This turns off certain features of GNU C that are incompatible with ANSI C, such as the `asm`, `inline` and `typeof` keywords, and predefined macros such as `unix` and `vax` that identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature, and disallows `$` as part of identifiers.

The alternate keywords `__asm__`, `__extension__`, `__inline__` and `__typeof__` continue to work despite `-ansi`. You would not want to use them in an ANSI C program, of course, but it useful to put them in header files that might be included in compilations done with `-ansi`. Alternate predefined macros such as `__unix__` and `__vax__` are also available, with or without `-ansi`.

The `-ansi` option does not cause non-ANSI programs to be rejected gratuitously. For that, `-pedantic` is required in addition to `-ansi`.

The macro `__STRICT_ANSI__` is predefined when the `-ansi` option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

The functions `alloca`, `abort`, `exit`, and `_exit` are not builtin functions when `-ansi` is used.

(b) `-fno-asm`

Do not recognize `asm`, `inline` or `typeof` as a keyword. These words may then be used as identifiers. You can use `__asm__`, `__inline__` and `__typeof__` instead. `-ansi` implies `-fno-asm`.

(c) `-fno-builtin`

Don't recognize built-in functions that do not begin with two leading underscores. Currently, the functions affected include `_exit`, `abort`, `abs`, `alloca`, `cos`, `exit`, `fabs`, `labs`, `memcmp`, `memcpy`, `sin`, `sqrt`, `strcmp`, `strcpy`, and `strlen`.

The `-ansi` option prevents `alloca` and `_exit` from being builtin functions.

(d) `-trigraphs`

Support ANSI C trigraphs. You don't want to know about this brain-damage. The `-ansi` option implies `-trigraphs`.

(e) `-traditional`

Attempt to support some aspects of traditional C compilers. Specifically:

(i) All `extern` declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.

(ii) The keywords `typeof`, `inline`, `signed`, `const` and `volatile` are not recognized. (You can still use the alternative keywords such as `__typeof__`, `__inline__`, and so on.)

(iii) Comparisons between pointers and integers are always allowed.

(iv) Integer types `unsigned short` and `unsigned char` promote to `unsigned int`.

(v) Out-of-range floating point literals are not an error.

(vi) String ‘‘constants’’ are not necessarily constant; they are stored in writable space, and identical looking constants are allocated separately. (This is the same as the effect of `-fwritable-strings`.)

(vii) All automatic variables not declared `register` are preserved by `longjmp`. Ordinarily, GNU C follows ANSI C: automatic variables not declared `volatile` may be clobbered.

(viii) In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.

(ix) In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.

(x) The predefined macro `__STDC__` is not defined when you use `-traditional`, but `__GNUC__` is (since the GNU extensions which `__GNUC__` indicates are not affected by `-traditional`). If you need to write header files that work differently depending on whether `-traditional` is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers.

You may wish to use `-fno-builtin` as well as `-traditional` if your program uses names that are normally GNU C builtin functions for other purposes of its own.

(f) `-traditional-cpp`

Attempt to support some aspects of traditional C preprocessors. This includes the last three items in the table immediately above, but none of the other effects of `-traditional`.

(g) `-fcond-mismatch`

Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

(h) `-funsigned-char`

Let the type `char` be unsigned, like `unsigned char`.

Each kind of machine has a default for what `char` should be. It is either like `unsigned char` by default or like `signed char` by default.

Ideally, a portable program should always use `signed char` or `unsigned char` when it depends on the signedness of an object. But many programs have been written to use plain `char` and expect it to be signed, or expect it to be unsigned, depending on the machines they were written for. This option, and its inverse, let you make such a program work with the opposite default.

The type `char` is always a distinct type from each of `signed char` or `unsigned char`, even though its behavior is always just like one of those two.

(i) `-fsigned-char`

Let the type `char` be signed, like `signed char`.

Note that this is equivalent to `-fno-unsigned-char`, which is the negative form of `-funsigned-char`. Likewise, `-fno-signed-char` is equivalent to `-funsigned-char`.

(j) `-fsigned-bitfields`

(k) `-funsigned-bitfields`

(l) `-fno-signed-bitfields`

(m) `-fno-unsigned-bitfields`

These 4 options control whether a bitfield is signed or unsigned, when the declaration does not use either `signed` or `unsigned`. By default, such a bitfield is signed, because this is consistent: the basic integer types such as `int` are signed types.

However, when `-traditional` is used, bitfields are all unsigned no matter what.

(n) `-fwritable-strings`

Store string constants in the writable data segment and don't uniquize them. This is for compatibility with old programs which assume they can write into string constants. `-traditional` also has this effect.

Writing into string constants is a very bad idea; ''constants'' should be constant.

A.2. Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error.

You can request many specific warnings with options beginning `-W`, for example `-Wimplicit` to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning `-Wno-` to turn off warnings; for example, `-Wno-implicit`. This manual lists only one of the two forms, whichever is not the default.

These options control the amount and kinds of warnings produced by GNU CC:

(a) `-fsyntax-only`

Check the code for syntax errors, but don't emit any output.

(b) `-w`

Inhibit all warning messages.

(c) `-Wno-import`

Inhibit warning messages about the use of `#import`.

(d) `-pedantic`

Issue all the warnings demanded by strict ANSI standard C; reject all programs that use forbidden extensions.

Valid ANSI standard C programs should compile properly with or without this option (though a rare few will require `-ansi`). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected.

`-pedantic` does not cause warning messages for use of the alternate keywords whose names begin and end with `__`. Pedantic warnings are also disabled in the expression that follows `__extension__`. However, only system header files should use these escape routes; application programs should avoid them.

This option is not intended to be useful; it exists only to satisfy pedants who would otherwise claim that GNU CC fails to support the ANSI standard.

Some users try to use `-pedantic` to check programs for strict ANSI C conformance. They soon find that it does not do quite what they want: it finds some non-ANSI practices, but not all--only those for which ANSI C *requires* a diagnostic.

A feature to report any failure to conform to ANSI C might be useful in some instances, but would require considerable additional work and would be quite different from `-pedantic`. We recommend, rather, that users take advantage of the extensions of GNU C and disregard the limitations of other compilers. Aside from certain supercomputers and obsolete small machines, there is less and less reason ever to use any other C compiler other than for bootstrapping GNU CC.

(e) `-pedantic-errors`

Like `-pedantic`, except that errors are produced rather than warnings.

(f) `-W`

Print extra warning messages for these events:

(i) A nonvolatile automatic variable might be changed by a call to `longjmp`. These warnings as well are possible only in optimizing compilation.

The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp` cannot in fact be called at the place which would cause a problem.

(ii) A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

```
foo (a)
{
  if (a > 0)
    return a;
}
```

(iii) An expression-statement contains no side effects.

(iv) An unsigned value is compared against zero with > or <=.

(v) A comparison like `x<=y<=z` appears; this is equivalent to `(x<=y ? 1 : 0) <= z`, which is a different interpretation from that of ordinary mathematical notation.

(vi) Storage-class specifiers like `static` are not the first things in a declaration. According to the C Standard, this usage is obsolescent.

(vii) An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for `x.h`:

```
struct s { int f, g; };
struct t { struct s h; int i; };
struct t x = { 1, 2, 3 };
```

(g) `−Wimplicit`

Warn whenever a function or parameter is implicitly declared.

(h) `−Wreturn−type`

Warn whenever a function is defined with a return-type that defaults to `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.

(i) `−Wunused`

Warn whenever a local variable is unused aside from its declaration, whenever a function is declared static but never defined, and whenever a statement computes a result that is explicitly not used.

If you want to prevent a warning for a particular variable, you can use this macro:

```
#define USE(var) \
  static void * use_##var = (&use_##var, (void *) &var)

USE (string);
```

(j) `−Wswitch`

Warn whenever a `switch` statement has an index of enumeral type and lacks a `case` for one or more of the named codes of that enumeration. (The presence of a `default` label prevents this warning.) `case` labels outside the enumeration range also provoke warnings when this option is used.

(k) `−Wcomment`

Warn whenever a comment-start sequence `/*` appears in a comment.

(l) `−Wtrigraphs`

Warn if any trigraphs are encountered (assuming they are enabled).

(m) `−Wformat`

Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified.

(n) `−Wchar−subscripts`

Warn if an array subscript has type `char`. This is a common cause of error, as programmers often forget that this type is signed on some machines.

(o) `−Wuninitialized`

An automatic variable is used without first being initialized.

These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. If you don't specify `−O`, you simply won't get these warnings.

These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

These warnings are made optional because GNU CC is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
  int x;
  switch (y)
    {
    case 1: x = 1;
      break;
    case 2: x = 4;
      break;
    case 3: x = 5;
    }
  foo (x);
}
```

If the value of `y` is always 1, 2 or 3, then `x` is always initialized, but GNU CC doesn't know this. Here is another common case:

```
{
  int save_y;
  if (change_y) save_y = y, y = new_y;
  /* ... */
  if (change_y) y = save_y;
}
```

This has no bug because `save_y` is used only if it is set.

Some spurious warnings can be avoided if you declare as `volatile` all the functions you use that never return.

(p) `-Wparentheses`

Warn if parentheses are omitted in certain contexts.

(q) `-Wall`

All of the above `-W` options combined. These are all the options which pertain to usage that we recommend avoiding and that we believe is easy to avoid, even in conjunction with macros.

The remaining `-W...` options are not implied by `-Wall` because they warn about constructions that we consider reasonable to use, on occasion, in clean programs.

(a) `-Wtraditional`

Warn about certain constructs that behave differently in traditional and ANSI C.

(i) Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C.

(ii) A function declared external in one block and then used after the end of the block.

(iii) A `switch` statement has an operand of type `long`.

(b) `-Wshadow`

Warn whenever a local variable shadows another local variable.

(c) `-Wid-clash-LEN`

Warn whenever two distinct identifiers match in the first LEN characters. This may help you prepare a program that will compile with certain obsolete, brain-damaged compilers.

(d) `-Wpointer-arith`

Warn about anything that depends on the ‘‘size of’’ a function type or of `void`. GNU C assigns these types a size of 1, for convenience in calculations with `void *` pointers and pointers to functions.

(e) `-Wcast-qual`

Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a `const char *` is cast to an ordinary `char *`.

(f) `-Wcast-align`

Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries.

(g) `-Wwrite-strings`

Give string constants the type `const char[LENGTH]` so that copying the address of one into a non-`const char *` pointer will get a warning. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using `const` in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make `-Wall` request these warnings.

(h) `-Wconversion`

Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument except when the same as the default promotion.

(i) `-Waggregate-return`

Warn if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning.)

(j) `-Wstrict-prototypes`

Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)

(k) `-Wmissing-prototypes`

Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. The aim is to detect global functions that fail to be declared in header files.

(l) `-Wredundant-decls`

Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.

(m) `-Wnested-externs`

Warn if an `extern` declaration is encountered within an function.

(n) `-Winline` Warn if a function can not be inlined, and either it was declared as inline, or else the `-finline-functions` option was given.

(o) `-Werror`

Make all warnings into errors.

APPENDIX B. dce.h

```
/*
 *  @OSF_COPYRIGHT@
 */

#if !defined(_DCE_H)
```

```
#define _DCE_H

/*
 * Common definitions for DCE
 * This is a machine specific file that must be ported to each
 * platform.
 */

/*
 * Define the endianess of the platform, either BIG_ENDIAN or
 * LITTLE_ENDIAN.
 * Definitions for byte order, by byte significance from low address
 * to high.
 *   #define LITTLE_ENDIAN   1234   least-significant byte first
 *   #define BIG_ENDIAN      4321   most-significant byte first
 */
#define LITTLE_ENDIAN

/* Only one place needed for DCE to define these */
#define FALSE 0
#define TRUE 1

#if !defined(MIN)
#  define MIN(x, y)         ((x) < (y) ? (x) : (y))
#endif

#if !defined(MAX)
#  define MAX(x, y)         ((x) > (y) ? (x) : (y))
#endif

/*
 * Machine dependent typedefs for boolean, byte, and (un)signed
 * integers.
 * All DCE code should be using these typedefs where applicable.
 * The following are defined in nbase.h:
 *     unsigned8       unsigned  8 bit integer
 *     unsigned16      unsigned 16 bit integer
 *     unsigned32      unsigned 32 bit integer
 *     signed8           signed  8 bit integer
 *     signed16          signed 16 bit integer
 *     signed32          signed 32 bit integer
 * Define the following from idl types in idlbase.h (which is
 * included by nbase.h:
 *     byte            unsigned  8 bits
 *     boolean         unsigned  8 bits
 * Define (un)signed64 to be used with the U64* macros
 */
#include <dce/nbase.h>
typedef idl_byte        byte;
typedef idl_boolean     boolean;
typedef struct unsigned64_s_t {
    unsigned long hi;
    unsigned long lo;
} unsigned64;
```

```
typedef struct signed64_s_t {
    unsigned long hi;
    unsigned long lo;
} signed64;

/*
 * The following allows for the support of both old and new style
 * function definitions and prototypes.  All DCE code is required to
 * be ANSI C compliant and to use prototypes.  For those components
 * that wish to support old-style definitions, the following macros
 * must be used.
 *
 *  Declare a prototype like this (don't use variables):
 *      int foo _DCE_PROTOTYPE_((int, void *, struct bar *))
 *
 *  Define a function like this:
 *      int foo
 *      #if defined(_DCE_PROTO_)
 *              (
 *              int a,
 *              void *b,
 *              struct bar *c
 *              )
 *      #else
 *              (a, b, c)
 *              int a;
 *              void *b;
 *              struct bar *c;
 *      #endif
 */
#if defined(__STDC__)                   /* other conditionals
                                           can be tested */
#  define _DCE_PROTO_
#endif                                  /* defined(__STDC__) */

#if defined(_DCE_PROTO_)
#  define _DCE_PROTOTYPE_(arg) arg
#else                                   /* defined(_DCE_PROTO_) */
#  define _DCE_PROTOTYPE_(arg) ()
#endif                                  /* defined(_DCE_PROTO_) */

/*
 * For those components wishing to support platforms where void
 * pointers are not available, they can use the following typedef for
 * a generic pointer type.  If they are supporting such platforms
 * they must use this.
 */
#if defined(__STDC__)
#  define _DCE_VOID_
#endif                                  /* defined(__STDC__) */

#if defined(_DCE_VOID_)
  typedef void * pointer_t;
#else                                   /* defined(_DCE_VOID_) */
  typedef char * pointer_t;
```

```
#endif                                  /* defined(_DCE_VOID_) */

/*
 * Here is a macro that can be used to support token concatenation in
 * an ANSI and non-ANSI environment.  Support of non-ANSI
 * environments is not required, but where done, this macro must be
 * used.
 */
#if defined(__STDC__)
#  define _DCE_TOKENCONCAT_
#endif

#if defined(_DCE_TOKENCONCAT_)
#  define DCE_CONCAT(a, b)      a ## b
#else                                /* defined(_DCE_TOKENCONCAT_) */
#  define DCE_CONCAT(a, b)      a/**/b
#endif                               /* defined(_DCE_TOKENCONCAT_) */

/*
 * Define the dcelocal and dceshared directories
 */
extern const char *dcelocal_path;
extern const char *dceshared_path;

/* If DCE_DEBUG is defined then debugging code is activated. */
#define DCE_DEBUG

/*
 * Serviceability and perhaps other DCE-wide include files
 * will be included here.  This is a sample only.
 */
#include <dce/dce_svc.h>

/* from the editline package */
char *readline(char *);
void add_history(char *);

#endif                                  /* _DCE_H */
```

APPENDIX C. ANSI C LIBRARY FUNCTIONS

This appendix is included for reference purposes. It lists all routines (functions or macros) defined by [ANSI C]. The routines are grouped by the header file that declares the prototype, thus all math functions are grouped together, as are all string functions. Prototypes are given with variable names for ''documentation'' purposes.

C.1. assert.h

(a) void assert(int expression);

C.2. ctype.h

(a) int isalnum(int c);
(b) int isalpha(int c);
(c) int iscntrl(int c);
(d) int isdigit(int c);
(e) int isgraph(int c);

(f) int islower(int c);
(g) int isprint(int c);
(h) int ispunct(int c);
(i) int isspace(int c);
(j) int isupper(int c);
(k) int isxdigit(int c);
(l) int tolower(int c);
(m) int toupper(int c);

C.3. locale.h

(a) struct lconv * localeconv(void);
(b) char * setlocale(int category, const char *locale);

C.4. math.h

(a) double acos(double x);
(b) double asin(double x);
(c) double atan(double x);
(d) double atan2(double y,double x);
(e) double ceil(double x);
(f) double cos(double x);
(g) double cosh(double x);
(h) double exp(double x);
(i) double fabs(double x);
(j) double floor(double x);
(k) double fmod(double x, double y);
(l) double frexp(double value, int * exp);
(m) double ldexp(double x, int exp);
(n) double log(double x);
(o) double log10(double x);
(p) double modf(double value, double *iptr);
(q) double pow(double x, double y);
(r) double sin(double x);
(s) double sinh(double x);
(t) double sqrt(double x);
(u) double tan(double x);
(v) double tanh(double x);

C.5. setjmp.h

(a) void longjmp(jmp_buf env, int val);
(b) int setjmp(jmp_buf env);

C.6. signal.h

(a) int raise(int sig);
(b) void (*signal(int sig, void (*func)(int)))(int);

C.7. stdarg.h

(a) <type> va_arg(va_list ap, <type>)
(b) void va_end(va_list);
(c) void va_start(va_list ap, parmN)

C.8. stdio.h

(a) void clearerr(FILE *stream);
(b) int fclose(FILE *stream);
(c) int feof(FILE *stream);

(d)	int	ferror(FILE *stream);
(e)	int	fflush(FILE *stream);
(f)	int	fgetc(FILE *stream);
(g)	int	fgetpos(FILE *stream, fpos_t *pos);
(h)	char *	fgets(char *s, int n, FILE *stream);
(i)	FILE *	fopen(const char *filename, const char *mode);
(j)	int	fprintf(FILE *stream, const char *format, ...);
(k)	int	fputc(int c, FILE *stream);
(l)	int	fputs(const char *s, FILE *stream);
(m)	size_t	fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
(n)	FILE *	freopen(const char *filename, const char *mode, FILE *stream);
(o)	int	fscanf(FILE *stream, const char *format, ...));
(p)	int	fseek(FILE *stream, long offset, int whence);
(q)	int	fsetpos(FILE *stream, const fpos_t *pos);
(r)	long	ftell(FILE *stream);
(s)	size_t	fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
(t)	int	getc(FILE *stream);
(u)	int	getchar(void);
(v)	char *	gets(char *s);
(w)	void	perror(const char *s);
(x)	int	printf(const char *format, ...);
(y)	int	putc(int c, FILE *stream);
(z)	int	putchar(int c);
(aa)	int	puts(const char *s);
(ab)	int	remove(const char *filename);
(ac)	int	rename(const char *old, const char *new);
(ad)	void	rewind(FILE *stream);
(ae)	int	scanf(const char *format, ...);
(af)	void	setbuf(FILE *stream, char *buf);
(ag)	int	setvbuf(FILE *stream, char *buf, int mode, size_t size);
(ah)	int	sprintf(char *s, const char *format, ...);
(ai)	int	sscanf(const char *s, const char *format, ...);
(aj)	int	ungetc(int c, FILE *stream);
(ak)	int	vfprintf(FILE *stream, const char *format, va_list arg);
(al)	int	vprintf(const char *format, va_list arg);
(am)	int	vsprintf(char *s, const char *format, va_list arg));
(an)	FILE *	tmpfile(void);
(ao)	char *	tmpnam(char *s);

C.9. stdlib.h

(a)	void	abort(void);
(b)	int	abs(int j);
(c)	int	atexit(void (*func)(void));
(d)	double	atof(const char *nptr);

(e) int atoi(const char *nptr);
(f) long int atol(const char *nptr);
(g) void * bsearch(const void *key, const void *base,
size_t nmemb, size_t size,
int (*compar)(const void *, const
void *));
(h) void * calloc(size_t nmemb, size_t size);
(i) div_t div(int numer, int denon);
(j) void exit(int status);
(k) void free(void *ptr);
(l) char * getenv(const char *name);
(m) long int labs(long int j);
(n) ldiv_t ldiv(long int numer, long int denom);
(o) void * malloc(size_t size);
(p) int mblen(const char *s, size_t n);
(q) int mbtowc(wchar_t *pwc, const char *s, size_t
n);
(r) size_t mbstowcs(wchar_t *pwcs, const char *s, size_t
n);
(s) void qsort(void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void
*));
(t) int rand(void);
(u) void * realloc(void *ptr, size_t size);
(v) void srand(unsigned int seed);
(w) double strtod(const char *nptr, char **endptr);
(x) long int strtol(const char *nptr, char **endptr, int
base);
(y) unsigned long int strtoul(const char *nptr, char **endptr, int
base);
(z) int system(const char *string);
(aa) size_t wcstombs(char *s, const wchar_t *pwcs, size_t
n);
(ab) int wctomb(char *s, const wchar_t wchar);

C.10. string.h

(a) void * memchr(const void *s, int c, size_t n);
(b) int memcmp(const void *s1, const void *s2, size_t
n);
(c) void * memcpy(void *s1, const void *s2, size_t n);
(d) void * memmove(void *s1, const void *s2, size_t n);
(e) void * memset(void *s, int c, size_t n);
(f) char * strcat(char *s1, const char *s2);
(g) char * strchr(const char *s, int c);
(h) int strcmp(const char *s1, const char *s2);
(i) int strcoll(const char *s1, const char *s2);
(j) char * strcpy(char *s1, const char *s2);
(k) size_t strcspn(const char *s1, const char *s2);
(l) char * strerror(int errnum);
(m) size_t strlen(const char *s);
(n) char * strncat(char *s1, const char *s2, size_t n);
(o) int strncmp(const char *s1,const char *s2, size_t
n);
(p) char * strncpy(char *s1, const char *s2, size_t n);

(q)	char *	strpbrk(const char *s1, const char *s2);
(r)	char *	strrchr(const char *s, int c);
(s)	size_t	strspn(const char *s1, const char *s2);
(t)	char *	strstr(const char *s1, const char *s2);
(u)	char *	strtok(char *s1, const char *s2);
(v)	size_t	strxfrm(char *s1, const char *s2, size_t n);

C.11. time.h

(a)	char *	asctime(const struct tm *timeptr);
(b)	clock_t	clock(void);
(c)	char *	ctime(const time_t *timer);
(d)	double	difftime(time_t time1, time_t time0);
(e)	struct tm *	gmtime(const time_t *timer);
(f)	struct tm *	localtime(const time_t *timer);
(g)	time_t	mktime(struct tm *timeptr);
(h)	size_t	strftime(char *s, size_t maxsize, const char *format, const struct tm *timeptr);
(i)	time_t	time(time_t *timer);

APPENDIX D. POSIX.1 LIBRARY FUNCTIONS

This appendix is included for reference purposes. It lists all routines (functions or macros) defined by [POSIX.1] which are not defined in [ANSI C]. The routines are grouped by the header file that declares the prototype. Prototypes are given with variable names for ‘‘documentation’’ purposes.

D.1. dirent.h

(a)	int	closedir(DIR *dirp);
(b)	DIR *	opendir(const char *dirname);
(c)	struct dirent *	readdir(DIR *dirp);
(d)	void	rewinddir(DIR *dirp);

D.2. fcntl.h

(a)	int	creat(const char *path, mode_t mode);
(b)	int	fcntl(int fildes, int cmd, ...);
(c)	int	open(const char *path, int oflag, ...);

D.3. grp.h

(a)	struct group *	getgrgid(gid_t gid);
(b)	struct group *	getgrnam(const char *name);

D.4. pwd.h

(a)	struct passwd *	getpwnam(const char *name);
(b)	struct passwd *	getpwuid(uid_t uid);

D.5. setjmp.h

(a)	void	siglongjmp(sigjmp_buf env, int val);
(b)	int	sigsetjmp(sigjmp_buf env, int savemask);

D.6. signal.h

(a)	int	kill(pid_t pid, int sig);
(b)	int	sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
(c)	int	sigaddset(sigset_t *set, int signo);

(d)	int	sigdelset(sigset_t *set, int signo);
(e)	int	sigemptyset(sigset_t *set);
(f)	int	sigfillset(sigset_t *set);
(g)	int	sigismember(const sigset_t *set, int signo);
(h)	int	sigpending(sigset_t *set);
(i)	int	sigprocmask(int how, const sigset_t *set, sigset_t *oset);
(j)	int	sigsuspend(const sigset_t *sigmask);

D.7. stdio.h

(a)	char *	ctermid(char *s);
(b)	FILE *	fdopen(int fildes, const char *type);
(c)	int	fileno(FILE *stream);

D.8. sys/stat.h

(a)	int	chmod(const char *path, mode_t mode);
(b)	int	fstat(int fildes, struct stat *buf);
(c)	int	mkdir(const char *path, mode_t mode);
(d)	int	mkfifo(const char *path, code_t mode);
(e)	int	stat(const char *path, struct stat *buf);
(f)	mode_t	umask(mode_t cmask);

D.9. sys/times.h

(a)	clock_t	times(struct tms *buffer);

D.10. sys/utsname.h

(a)	int	uname(struct utsname *name);

D.11. sys/wait.h

(a)	pid_t	wait(int *stat_loc);
(b)	pid_t	waitpid(pid_t pid, int *stat_loc, int options);

D.12. termios.h

(a)	speed_t	cfgetispeed(const struct termios *termios_p);
(b)	speed_t	cfgetospeed(const struct termios *termios_p);
(c)	speed_t	cfsetispeed(struct termios *termios_p, speed_t speed);
(d)	speed_t	cfsetospeed(struct termios *termios_p, speed_t speed);
(e)	int	tcdrain(int fildes);
(f)	int	tcflow(int fildes, int action);
(g)	int	tcflush(int fildes, int queue_selector);
(h)	int	tcgetattr(int fildes, struct termios *termios_p);
(i)	int	tcsendbreak(int fildes, int duration);
(j)	int	tcsetattr(int fildes, int opt_acts, const struct termios *termios_p);

D.13. time.h

(a)	void	tzset(void);

D.14. utime.h

(a)	int	utime(const char *path, const struct utimbuf *times);

D.15. unistd.h

(a)	void	_exit(int status);
(b)	int	access(const char *path, int amode);
(c)	unsigned int	alarm(unsigned int seconds);
(d)	int	chdir(const char *path);
(e)	int	chown(const char *path, uid_t owner, gid_t group);
(f)	int	close(int fildes);
(g)	char *	cuserid(char *);
(h)	int	dup2(int fildes, int fildes2);
(i)	int	dup(int fildes);
(j)	int	execl(const char *path, const char *arg, ...);
(k)	int	execle(const char *path, const char *arg, ...);
(l)	int	execlp(const char *file, const char *arg, ...);
(m)	int	execv(const char *path, char *const argv[]);
(n)	int	execve(const char *path, char *const argv[], char *const envp[]);
(o)	int	execvp(const char *file, char *const argv[]);
(p)	pid_t	fork(void);
(q)	long	fpathconf(int fildes, int name);
(r)	char *	getcwd(char *buf, size_t size);
(s)	gid_t	getegid(void);
(t)	uid_t	geteuid(void);
(u)	gid_t	getgid(void);
(v)	int	getgroups(int gidsetsize, gid_t grouplist[]);
(w)	char *	getlogin(void);
(x)	pid_t	getpgrp(void);
(y)	pid_t	getpid(void);
(z)	pid_t	getppid(void);
(aa)	uid_t	getuid(void);
(ab)	int	isatty(int fildes);
(ac)	int	link(const char *existing, const char *new);
(ad)	off_t	lseek(int fildes, off_t offset, int whence);
(ae)	long	pathconf(const char *path, int name);
(af)	int	pause(void);
(ag)	int	pipe(int fildes[2]);
(ah)	ssize_t	read(int fildes, void *buf, size_t nbyte);
(ai)	int	rmdir(const char *path);
(aj)	int	setgid(gid_t gid);
(ak)	int	setpgid(pid_t pid, pid_t pgid);
(al)	pid_t	setsid(void);
(am)	int	setuid(uid_t uid);
(an)	unsigned int	sleep(unsigned int seconds);
(ao)	long	sysconf(int name);
(ap)	pid_t	tcgetpgrp(int fildes);
(aq)	int	tcsetpgrp(int fildes, pid_t pgrp_id);
(ar)	char *	ttyname(int fildes);
(as)	int	unlink(const char *path);
(at)	ssize_t	write(int fildes , const void *buf, size_t nbyte);

APPENDIX E. XPG4 LIBRARY FUNCTIONS

This appendix is included for reference purposes. It lists all routines (functions or macros) defined by [XPG4] which are not defined in either [ANSI C] or [POSIX.1] and are not marked as ''TO BE WITHDRAWN'' from [XPG4]. The routines are grouped by the header file that declares the prototype, thus all math functions are grouped together, as are all string functions. Prototypes are given with variable names for ''documentation'' purposes.

E.1. ctype.h

```
(a) int          _tolower(int c);
(b) int          _toupper(int c);
(c) int          isascii(int c);
(d) int          toascii(int c);
```

E.2. dirent.h

```
(a) void         seekdir(DIR *dirp, long int loc);
(b) long int     telldir(DIR *dirp);
```

E.3. ftw.h

```
(a) int          ftw(const char *path,
             int (*fn)(const char *, const struct stat
                   *, int),
             int ndirs);
```

E.4. iconv.h

```
(a) size_t       iconv(iconv_t cd, char **inbuf,
                     size_t *inbytesleft, char **outbuf,
                     size_t *outbytesleft);
(b) int          iconv_close(iconvt_t cd);
(c) iconv_t      iconv_open(cont char *tocode,
                          const char *fromcode);
```

E.5. math.h

```
(a) double       erf(double x);
(b) double       erfc(double x);
(c) double       gamma(double x);
(d) double       hypot(double x, double y);
(e) double       isnan(double x);
(f) double       j0(double x);
(g) double       j1(double x);
(h) double       jn(int n, double x);
(i) double       lgamma(double x);
(j) double       y0(double x);
(k) double       y1(double x);
(l) double       yn(int n, double x);
```

E.6. monetary.h

```
(a) ssize_t      strfmon(char *s, size_t maxsize, const char
                     *format, ...);
```

E.7. nl_types.h.h

```
(a) int          catclose(nlcatd catd);
(b) int          catgets(nlcatd catd, int set_id, int msg_id,
                     const char *s);
```

```
(c) int             catopen(const char *name, int oflag);
```

E.8. search.h

```
(a) int             hcreate(size_t nel);
(b) void            hdestroy(void);
(c) ENTRY *         hsearch(ENTRY item, ACTION action);
(d) void *          lfind(const void *key, const void *base,
                        size_t *nelp, size_t width,
                        int (*compar)(const void *, const void
                                *));
(e) void *          lsearch(const void *key, void *base,
                          size_t *nelp, size_t width,
                          int (*compar)(const void *, const
                                  void *));
(f) void *          tdelete(const void *key, void **rootp,
                          int (*compar)(const void *, const
                                  void *));
(g) void *          tsearch(const void *key, void **rootp,
                          int (*compar)(const void *, const
                                  void *));
(h) void            twalk(const void *root,
                        void (*action)(const void *, VISIT,
                                const int));
```

E.9. stdio.h

```
(a) int             getw(FILE *stream);
(b) int             putw(int w, FILE *stream);
(c) char *          tempnam(const char *dir, const char *pfx);
```

E.10. stdlib.h

```
(a) double          drand48(void);
(b) double          erand48(unsigned short int xsubi[3]);
(c) long int        jrand48(unsigned short int xsubi[3]);
(d) void            lcong48(unsigned short int param[7]);
(e) long int        lrand48(void);
(f) long int        mrand48(void);
(g) long int        nrand48(unsigned short int xsubi[3]);
(h) int             putenv(char *string);
(i) unsigned short * seed48(unsigned short int seed16v[3]);
(j) void            setkey(const char *key);
(k) void            srand48(long int seedval);
```

E.11. string.h

```
(a) void *          memccpy(void *s1, const void *s2, int c,
                            size_t n);
```

E.12. sys/msg.h

```
(a) int             msgctl(int msqid, int cmd, struct msqid_ds
                          *buf);
(b) int             msgget(key_t key, int msgflg);
(c) int             msgrcv(int msqid, void *msgp, size_t msgsz,
                          long int msgtyp, int msgflg);
(d) int             msgsnd(int msqid, void *msgp, size_t msgsz,
                          int msgflg);
```

E.13. sys/sem.h

(a) int semctl(int semid, int semnum, int cmd, ...);
(b) int semget(key_t key, int nsems, int semflg);
(c) int semop(int semid, struct sembuf *sops,
    unsigned int nsops);

E.14. sys/shm.h

(a) void * shmat(int shmid, void *shmaddr, int shmflg);
(b) int shmctl(int shmid, int cmd, struct shmid_ds
    *buf);
(c) int shmdt(const void *shmaddr);
(d) int shmget(key_t key, size_t size, int shmflg);

E.15. time.h

(a) char * strptime(const char *buf, const char *format,
    struct tm *tm);

E.16. ulimit.h

(a) long int ulimit(int cmd, ...);

E.17. unistd.h

(a) char * crypt(const char *key, const char *salt);
(b) char * encrypt(char block[64], int edflag);
(c) int fsync(int fildes);
(d) int nice(int incr);
(e) void swab(const void *src, void *des, size_t
    nbytes);

E.18. wchar.h

(a) wchar_t * fgetws(wchar_t *, int, FILE *);
(b) wint_t fgetwc(FILE *stream);
(c) wint_t fputwc(wint_t c, FILE * stream);
(d) int fputws(const wchar_t *s, FILE *stream);
(e) wint_t getwc(FILE *stream);
(f) wint_t getwchar(void);
(g) wchar_t * getws(wchar_t *s);
(h) int iswalnum(wint_t wc);
(i) int iswalpha(wint_t wc);
(j) int iswcntrl(wint_t wc);
(k) int iswctype(wint_t wc, wctype_t prop);
(l) int iswdigit(wint_t wc);
(m) int iswgraph(wint_t wc);
(n) int iswlower(wint_t wc);
(o) int iswprint(wint_t wc);
(p) int iswpunct(wint_t wc);
(q) int iswspace(wint_t wc);
(r) int iswupper(wint_t wc);
(s) int iswxdigit(wint_t wc);
(t) wint_t putwc(wint_t c, FILE *stream);
(u) wint_t putwchar(wint_t c);
(v) int putws(const wchar_t *s);
(w) wint_t towlower(wint_t wc);
(x) wint_t towupper(wint_t wc);

(y)	wint_t	ungetwc(wint_t c, FILE * stream);
(z)	wchar_t *	wcscat(wchar_t *ws1, const wchar_t *ws2);
(aa)	wchar_t *	wcschr(const wchar_t *ws, wchar_t wc);
(ab)	int	wcscmp(const wchar_t *ws1, const wchar_t *ws2);
(ac)	int	wcscoll(const wchar_t *ws1, const wchar_t *ws2);
(ad)	wchar_t *	wcscpy(wchar_t *ws1, const wchar_t *ws2);
(ae)	size_t	wcscspn(const wchar_t *ws1, const wchar_t *ws2);
(af)	size_t	wcsftime(wchar_t *wcs, size_t maxsize, const char *fmt, const struct tm *timptr);
(ag)	size_t	wcslen(const wchar_t *wc);
(ah)	wchar_t *	wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);
(ai)	int	wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
(aj)	wchar_t *	wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
(ak)	wchar_t *	wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
(al)	wchar_t *	wcsrchr(const wchar_t *ws, wchar_t wc);
(am)	size_t	wcsspn(const wchar_t *ws1, const wchar_t *ws2);
(an)	double	wcstod(const wchar_t *nptr, wchar_t **endptr);
(ao)	wchar_t *	wcstok(wchar_t *ws1, const wchar_t *ws2);
(ap)	long int	wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
(aq)	unsigned long int	wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);
(ar)	wchar_t *	wcswcs(const wchar_t *ws1, const wchar_t *ws2);
(as)	int	wcswidth(const wchar_t *pwcs, size_t n);
(at)	size_t	wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);
(au)	int	wcwidth(const wchar_t wc);
(av)	wctype_t	wctype(const char *property);

APPENDIX F. ROUTINES WRAPPED BY DCE THREADS

This appendix lists all the wrappers that DCE Threads provides and the header file in which they are defined. Wrappers need to be added to DCE Threads for any library routines called which are not in this list. Those wrappers marked with a ‘‘*’’ are only wrapped on an SVR4 platform. Some wrappers that are not marked are not on the SVR4 platform as they use reentrant libraries for thread-safety.

* (a)	_sleep	[cma_ux.h]
(b)	accept	[cma_ux.h]
* (c)	atfork	[cma_ux.h]
(d)	calloc	[cmalib_crtlx.h]
(e)	catclose	[cma_stdio.h]
(f)	catgets	[cma_stdio.h]
(g)	catopen	[cma_stdio.h]
(h)	cfree	[cmalib_crtlx.h]

(i) close [cma_ux.h]
(j) connect [cma_ux.h]
(k) creat [cma_ux.h]
(l) ctermid [cma_stdio.h]
(m) cuserid [cma_stdio.h]
(n) dup [cma_ux.h]
(o) dup2 [cma_ux.h]
(p) fclose [cma_stdio.h]
(q) fcntl [cma_ux.h]
(r) fdopen [cma_stdio.h]
(s) fflush [cma_stdio.h]
(t) fgetc [cma_stdio.h]
(u) fgets [cma_stdio.h]
(v) fopen [cma_stdio.h]
(w) fork [cma_ux.h]
(x) fprintf [cma_stdio.h]
(y) fputc [cma_stdio.h]
(z) fputs [cma_stdio.h]
(aa) fread [cma_stdio.h]
(ab) free [cmalib_crtlx.h]
(ac) freopen [cma_stdio.h]
(ad) fscanf [cma_stdio.h]
(ae) fseek [cma_stdio.h]
(af) ftell [cma_stdio.h]
(ag) fwrite [cma_stdio.h]
(ah) getc [cma_stdio.h]
(ai) getchar [cma_stdio.h]
(aj) gets [cma_stdio.h]
(ak) getw [cma_stdio.h]
(al) isatty [cma_stdio.h]
(am) malloc [cmalib_crtlx.h]
(an) mktemp [cma_stdio.h]
(ao) open [cma_ux.h]
(ap) pclose [cma_stdio.h]
(aq) pipe [cma_ux.h]
(ar) popen [cma_stdio.h]
(as) printf [cma_stdio.h]
(at) putc [cma_stdio.h]
(au) putchar [cma_stdio.h]
(av) puts [cma_stdio.h]
(aw) putw [cma_stdio.h]
(ax) read [cma_ux.h]
(ay) readv [cma_ux.h]
(az) realloc [cmalib_crtlx.h]
(ba) recv [cma_ux.h]
(bb) recvfrom [cma_ux.h]
(bc) recvmsg [cma_ux.h]
(bd) rewind [cma_stdio.h]
(be) scanf [cma_stdio.h]
(bf) select [cma_ux.h]
(bg) send [cma_ux.h]
(bh) sendmsg [cma_ux.h]
(bi) sendto [cma_ux.h]
(bj) setbuf [cma_stdio.h]

(bk)	setbuffer	[cma_stdio.h]
(bl)	setlinebuf	[cma_stdio.h]
(bm)	setvbuf	[cma_stdio.h]
(bn)	sigaction	[cma_px.h]
(bo)	sigwait	[cma_sigwait.h]
* (bp)	sleep	[cma_ux.h]
(bq)	socket	[cma_ux.h]
(br)	socketpair	[cma_ux.h]
(bs)	sprintf	[cma_stdio.h]
(bt)	sscanf	[cma_stdio.h]
(bu)	system	[cma_stdio.h]
(bv)	tempnam	[cma_stdio.h]
(bw)	tmpfile	[cma_stdio.h]
(bx)	tmpnam	[cma_stdio.h]
(by)	ttyname	[cma_stdio.h]
(bz)	ttyslot	[cma_stdio.h]
(ca)	vfprintf	[cma_stdio.h]
(cb)	vprintf	[cma_stdio.h]
(cc)	vsprintf	[cma_stdio.h]
(cd)	write	[cma_ux.h]
(ce)	writev	[cma_ux.h]

APPENDIX G. STANDARD SYSTEM DEFINED CONSTANTS

This appendix lists constants defined by [ANSI C], [POSIX.1], and [XPG4] that a programmer might wish to check for. It does not list all macros that are defined, but rather those that might be of interest. For example, the macros used to define flags to routines, signals, and error cases are *not* list here. However, those that define the maximum size of an `unsigned int` and the number of bits in a `char` are.

G.1. limits.h

G.1.1. Numerical limits

(a)	`CHAR_BIT`	Number of bits in a `char`
(b)	`CHAR_MAX`	Max value for a `char`
(c)	`CHAR_MIN`	Min value for a `char`
(d)	`INT_MAX`	Max value for an `int`
(e)	`INT_MIN`	Min value for an `int`
(f)	`LONG_BIT`	Number of bits in a `long int`
(g)	`LONG_MAX`	Max value for a `long int`
(h)	`LONG_MIN`	Min value for a `long int`
(i)	`MB_LEN_MAX`	Max number of bytes in a multibyte character, for any supported locale
(j)	`SCHAR_MAX`	Max value for a `signed char`
(k)	`SCHAR_MIN`	Min value for a `signed char`
(l)	`SHRT_MAX`	Max value for a `short int`
(m)	`SHRT_MIN`	Min value for a `short int`
(n)	`UCHAR_MAX`	Max value for an `unsigned char`
(o)	`UINT_MAX`	Max value for an `unsigned int`
(p)	`ULONG_MAX`	Max value for an `unsigned long int`
(q)	`USHRT_MAX`	Max value for an `unsigned short int`
(r)	`WORD_BIT`	Number of bits in a word or `int`

G.1.2. Run-time invariant values

(a)	`ARG_MAX`	Max length of arg to `exec` (including env data)

(b)	`CHILD_MAX`	Max number of simultaneous processes per real uid
(c)	`OPEN_MAX`	Max number of files that a process can have open
(d)	`STREAM_MAX`	Max number of streams that a process can have open
(e)	`TZNAME_MAX`	Max number of bytes supported for name of a time zone

G.1.3. Pathname variable values

(a)	`LINK_MAX`	Max number of links to a single file
(b)	`MAX_CANON`	Max number of types in a terminal canonical input file
(c)	`MAX_INPUT`	Min number of bytes available in a terminal input queue
(d)	`NAME_MAX`	Max number of types in a filename (not including the null)
(e)	`PATH_MAX`	Max number of bytesin a pathname (including the null)
(f)	`PIPE_BUF`	Max number bytes that is guaranteed to be atomic when writing to a pipe

G.1.4. Other invariant values

(a)	`CHARCLASS_NAME_MAX`	Max number of bytes in a character class name
(b)	`NL_ARGMAX`	Max value of digits in calls to `printf` and `scanf`
(c)	`NL_LANGMAX`	Max number of types in a `LANG` name
(d)	`NL_MSGMAX`	Max message number
(e)	`NL_NMAX`	Max number of bytes in an N-to-1 collation mapping
(f)	`NL_SETMAX`	Max set number
(g)	`NL_TEXTMAX`	Max number of bytes in a message string
(h)	`NZERO`	Default process priority

G.2. float.h

(a)	`DBL_EPSILON`	1ulp when exponent = 0
(b)	`DBL_MANT_DIG`	Number of base-FLT_RADIX digits in the floating-point significand
(c)	`DBL_MAX_10_EXP`	Largest base 10 exponent of NORMALIZED `double`
(d)	`DBL_MAX_EXP`	Exponent of largest NORMALIZED `double`
(e)	`DBL_MIN_10_EXP`	Min base 10 exponent of NORMALIZED `double`
(f)	`DBL_MIN_EXP`	Exponent of smallest NORMALIZED `double`
(g)	`FLT_EPSILON`	1ulp when exponent = 0
(h)	`FLT_MANT_DIG`	Number of base-FLT_RADIX digits in the floating-point significand
(i)	`FLT_MAX_10_EXP`	Largest base 10 exponent of NORMALIZED `float`
(j)	`FLT_MAX_EXP`	Exponent of largest NORMALIZED `float`
(k)	`FLT_MIN_10_EXP`	Min base 10 exponent of NORMALIZED `float`
(l)	`FLT_MIN_EXP`	Exponent of smallest NORMALIZED float number

(m)	`FLT_RADIX`	Radix of exponent representation
(n)	`FLT_ROUNDS`	Macro that returns current rounding mode value
(o)	`LDBL_DIG`	Number of decimal digits of precision
(p)	`LDBL_EPSILON`	1ulp when unbiased exponent = 0
(q)	`LDBL_MANT_DIG`	Number of base-FLT_RADIX digits in the floating-point significand
(r)	`LDBL_MAX`	Largest NORMALIZED `long double`
(s)	`LDBL_MAX_10_EXP`	Largest base 10 exponent of NORMALIZED `long double`
(t)	`LDBL_MAX_EXP`	Exponent of largest NORMALIZED `long double`
(u)	`LDBL_MIN`	Smallest NORMALIZED `long double`
(v)	`LDBL_MIN_10_EX`	Min base 10 exponent of NORMALIZED `long double`
(w)	`LDBL_MIN_EXP`	Exponent of smallest NORMALIZED `long double`

G.3. stddef.h

(a)	`NULL`	An implementation defined null pointer constant

G.4. math.h

(a)	`HUGE_VAL`	A positive double expression, not necessarily representable as a `float`
(b)	`MAXFLOAT`	Max non-infinite single-precision floating point number
(c)	`M_1_PI`	Value of $1/\pi$
(d)	`M_2_PI`	Value of $2/\pi$
(e)	`M_2_SQRTPI`	Value of $2 \times \pi^{-½}$
(f)	`M_E`	Value of e
(g)	`M_LN10`	Value of $\log_e(10)$
(h)	`M_LN2`	Value of $\log_e(2)$
(i)	`M_LOG10E`	Value of $\log_{10}(e)$
(j)	`M_LOG2E`	Value of $\log_2(e)$
(k)	`M_PI`	Value of π
(l)	`M_PI_2`	Value of $\pi/2$
(m)	`M_PI_4`	Value of $\pi/4$
(n)	`M_SQRT1_2`	Value of $2^{-½}$
(o)	`M_SQRT2`	Value of $2^{½}$

G.5. stdio.h

(a)	`BUFSIZ`	The size of the buffer used by `setbuf`
(b)	`EOF`	Negative integral constant indicating end-of-file
(c)	`FILENAME_MAX`	Max number of `char`'s in a filename
(d)	`FOPEN_MAX`	The min number of files that can be open simultaneously
(e)	`L_tmpnam`	Max number of `char`'s in a filename generated by `tmpnam`
(f)	`TMP_MAX`	Min number of unique filenames generated by `tmpnam`

G.6. stdlib.h

(a)	`EXIT_FAILURE`	Can be passed to `exit` to indicate failure
(b)	`EXIT_SUCCESS`	Can be passed to `exit` to indicate success
(c)	`MB_CUR_MAX`	Max number of bytes in a multi-byte character for the extended character set specified by the current locale, and whose value is never greater than `MB_LEN_MAX`
(d)	`RAND_MAX`	The max value returned by `rand`

G.7. time.h

(a)	`CLOCKS_PER_SEC`	The number per second of the value returned by `clock`

G.8. unistd.h

(a)	`STDIN_FILENO`	File number of stdin
(b)	`STDOUT_FILENO`	File number of stdout
(c)	`STDERR_FILENO`	File number of stderr

APPENDIX H. dce/assert.h

```
/*
 *  @OSF_COPYRIGHT@
 */

/*
 *  The dce_assert() macro; like ANSI C's assert().
 */

#if     defined(dce_assert)
#undef dce_assert
#endif  /* defined(dce_assert) */

#if     defined(DCE_ASSERT)

extern void dce___assert(
    struct dce_svc_handle_s_t *, const char *, const char *, int
);

#define dce_assert(h, EX) \
        ((EX) ? ((void)0) \
              : dce___assert(h, # EX, __FILE__, __LINE__))

#else

#define dce_assert(h, EX)       ((void)0)

#endif  /* defined(DCE_ASSERT) */
```

APPENDIX I. DCE ODE DOCUMENTATION

This appendix describes ODE from a user's perspective. It does not provide full documentation on building backing tree or sandboxes or in using the ODE commands. It does provide detailed information on variables used in

DCE `Makefile`'s and some background information on ODE tree organization and the ODE system of passes to build a product.

I.1. Overview of the Build Tree Layout

ODE organizes the build tree into several main parts:

(a) `src`

All source files reside here. Beneath this directory are subtrees for each component of DCE.

(b) `obj`

An internal building area. All compiles are run here, and all object and executables are created here.

(c) `export`

Another internal building area. All inter-component dependencies are resolved by exporting files (usually headers and libraries) here, and then having other components look here instead of in the src subtree of other components.

(d) `install`

The final resting place of user-visible programs, libraries, headers, datafiles, tests, etc. The format of this tree mirrors the format of an installed DCE system (i.e., programs are found in a `bin` directory, etc.).

(e) `tools`

A directory storing all tools needed during the build process. Both ODE tools (e.g., `make`) and DCE tools (e.g., `idl`, `mavros`, etc.) reside here.

(f) `rc_files`

This directory contains files (known as rc_file's) used by ODE to control the environment of ODE commands. They describe sandbox and backing tree information.[21] See below for a short description of the rc_files and their syntax.

The rc_files define the environment used by the ODE tools. In particular, these include `workon`, `build`, and `sbinfo`. This last one is used to display information on these environment variables. The `workon` creates a new shell with various ODE variables exported to the shell environment. The `build` command is a wrapper around `make` that ODE uses. All variables described in the rc_files are defined in the environment of ODE tools (e.g., `build`), only some are exported to the environment of the shell by the `workon` command.

Each sandbox and backing tree contain two basic rc_files, they are: `shared` and `local`. The rc_files support an `include` command to include the contents of other rc_file's. The rc_file's are chained together using this command.

All ODE commands that read the rc_file's do so by reading the `local` file in the sandbox. This file includes the `shared` file in the sandbox. The `shared` file includes the `shared` file in the backing tree (however many levels back there are, i.e., any shared sandboxes have their `shared` rc_file included as well).

The rc_files contain the following types of commands (square brackets denote optional keywords):

```
include filename
[on machine_type] [replace] [setenv] variable value
```

where:

(a) The `include` command includes the named file. It is similar to the `#include` command that the C preprocessor understands.

---

21. For an explanation of these terms see ODE documentation. [DUG] This section of this appendix will assume understanding of these terms.

(b) If a command is prefaced with `on` *machine_type*, then that command is only run on machines of that type. The *machine_type* value is compiled into the ODE tools for each architecture. They are currently: `pmax`, `rios`, etc.

(c) The `setenv` keyword means that `workon` will set this variable in the environment of the new shell. If `setenv` is not included in the command, then the variable is *only* set in the environment of ODE commands (e.g., `build`).

(d) If the `replace` keyword is *not* given on a command line, then that variable is only set if it previously had no value. This applies to commands with or without the `setenv` keyword. If the `replace` keyword is given on a command line, then that variable is set regardless of any previous value.

I.2. Overview of Build Passes and Makefiles

ODE organizes the DCE build process into several passes. Passes have numbers (e.g., `FIRST`, `FIRSTA`, etc.) and tags (e.g., `SETUP`, `EXPINC`, `EXPLIB`, etc.) associated with them. The tags then have actions (e.g., export, build, comp, clobber, etc.) associated with them.

(a) setup

This pass builds tools needed during the export pass. The primary example is `idl`, which is built during this pass so that headers generated by it can be exported during the export pass.

(b) export

This pass builds headers (if necessary) and libraries needed by one component from another component. Most programs need headers and functions contained in `libdce.a`, these are built and placed the `export` tree during this pass.

(c) comp

This pass builds all other programs and datafiles needed. Any libraries that are not exported are built during this pass as well.

(d) install

This pass takes built programs libraries and datafiles and places them into the `install` tree.

There are other passes defined as well. These allow `lint` and various tags creating programs (e.g., `ctags` or `etags`) to be run on the source files, as well as several cleanup passes that remove objects from the `obj` tree.

ODE and DCE define several different types of makefiles that are meant, by convention, to contain different types of build information.

(a) common makefiles

The common makefiles are all located in `src/lbe/mk` and are considered to be a part of ODE, although DCE uses its own version of them. All of their names are of the form `osf.*.mk`. They define the variables and rules that can be used in other makefiles. All the built in rules are defined in the common makefiles, and are accessible in other makefiles that include the common makefiles by using the line:

```
.include <${RULES_MK}>
```

This line *always* appears *after* all variable defines in the `Makefile`. These files are owned by the OSF RATs, DCE developers should have little reason to modify any of these files.

(b) `Makefile`'s

These are most analogous to standard `Makefiles`. They appear in each directory of the `src` tree, and make use of variables and rules defined in the common makefiles. DCE developers create and modify these as required during the development of a component.

(c) `machdep.mk` files

These files are used to handle machine dependencies in the source code. Whenever a function is machine dependent, it is placed in a file in a target machine subdirectory. In the `Makefile` for the program which contains this function, there is a command to include this file from whichever target machine subdirectory is appropriate. It usually looks like:

```
.if exists(${TARGET_MACHINE}/machdep.mk)
.include "${TARGET_MACHINE}/machdep.mk"
.endif
```

DCE developers create and modify these as required during the development of a component.

(d) `src/Makeconf`

This file is used internally by ODE to locate the top of the `src` tree particularly in relationship to the other subtrees in the build tree (e.g., `obj` and `export`). DCE uses this file to define various platform-specific build variables that apply to all of DCE. DCE developers may have need to modify this file, however it is generally considered to be own by the OSF RATs.

(e) component `.mk` files

These are makefiles that are located in the top-most directory of each component, and named with the name of the component followed by `.mk` (e.g., `src/rpc/rpc.mk`). They are found by the build process in the following manner. The common makefile `osf.dce.mk` determines what component make is being run in, and includes the appropriate component makefiles with a `.include` command.

These are a creation of DCE (they are not a part of ODE) and were meant to capture all build dependencies that were machine specific for an entire component. The theory being that by editing this one file during a port, any build switches that needed to be changed or anything else, could be done in one place as opposed to several. This turned out not be true and added yet another file to look at to find out how a particular build line was generated. They must not be used for any new or existing code. Instead the information should be in the `Makefile`'s and `machdep.mk` files.

I.3. ODE Variables

The following are the variables that can be defined in DCE `Makefile`'s to control the build process. They are specified by category and by the passes in which they are used.

(a) Subdirectory Variables

The following variables can be used in Makefiles to define subdirectories to visit during specified passes.

(i) `SETUP_SUBDIRS`

This variable defines a list of subdirectories to visit during the setup pass. This pass is used to define tools that are needed during the export passes

(ii) `EXPINC_SUBDIRS`

This variable defines a list of subdirectories to visit during the export include file pass. This pass is used to place headers needed by other components into the `export` tree.

(iii) `EXPLIB_SUBDIRS`

This variable defines a list of subdirectories to visit during the export libraries pass. This pass is used to place libraries needed by other components into the `export` tree.

(iv) `EXPSHLIB_SUBDIRS`

This variable defines a list of subdirectories to visit during the export shared libraries pass. This pass is used to place shared libraries needed by other components into the `export` tree. Currently it is only used for `libdce.so`.

(v) `SUBDIRS`

This variable defines a list of subdirectories to visit during all the other passes. Principally, this means the comp pass which is used to build programs and the install pass, but it is also used for the clean, rmtarget, clobber, lint, and tags passes as well.

(b) ODE Variables Used in the setup Pass

These variables are used by ODE during the setup pass to determine various targets and dependencies. The goal of the setup pass is to build various tools needed during the export pass. Currently, this list of tools is: `compile_et`, `idl`, `idl.cat`, `mavcod`, `mavros`, `msg`, and `prs`. The built tools are placed in the `tools` subtree by default.

To build the tools, run the setup pass by typing: `build setup_all`. Note that `DEFTOOLBASE` must be set in some way. It can be defined on the build command line, added to the `local` rc_file, or the `-sb_rc` can be used so that the `setup` rc_file is read instead of the `local` rc_file (this file includes `local` and then sets `DEFTOOLBASE`).

(i) `SETUP_INCLUDES`

This variable defines a list of include

(ii) `SETUP_PROGRAMS`

This variable defines a list of programs to be built during the setup pass.

(iii) `SETUP_SCRIPTS`

This variable defines a list of scripts to be built during the setup pass.

(iv) `SETUP_OTHERS`

This variable defines a list of other things to be built during the setup pass.

(v) `DEFTOOLBASE`

This variable defines the directory where tools built during the setup pass are placed, and where these tools are looked for during other passes. Note that the path assigned to this variable must begin and end with a `/`.

(c) ODE Variables Used in the export Passes

These variables are used by ODE during the export pass to determine various targets and dependencies. The goal of the export pass is to place any dependency needed between components into the `export` tree.

(i) `INCLUDES`

This variable defines a list of include files to be exported to the `export` tree.

(ii) `EXPLIB_TARGETS`

This variable defines a list of libraries to be built and exported to the `export` tree during the `EXPLIB` pass of the build. If this variable is not set, then all libraries listed in the variable `LIBRARIES` (see below) will be exported. If this variable is set, then only the libraries listed in this variable will be exported, regardless of the values of `LIBRARIES`.

The value of this variable is a list of libraries; each library name must be preceded with `export_`. Any targets in the `EXPLIB_TARGETS` list that do not have the `export_` prefix will be built, but not exported. Ensuring the prefix is present can be done in two ways: either by including it by hand, or by including just the library names without the `export_` and then including a substitution command *after* the definition to add the prefix as follows:

```
EXPLIB_TARGETS = liba.a libb.a libc.a
EXPLIB_TARGETS = ${EXPLIB_TARGETS:S/^/export_/g}
```

(iii) `EXPSHLIB_TARGETS`

Same as `EXPLIB_TARGETS` above, but use `SHARED_LIBRARIES`, etc.

(iv) `EXPDIR`

This variable defines the directory into which include files and libraries will be exported. It is relative to the `export/`*`machine`* directory in the sandbox. Note that the path assigned to this variable must begin and end with a `/`.

(v) `IDLFILES`

This variable defines a list of `.idl` files to run the `idl` compiler on. This is done during an export pass to generate `.h` files for the `export` tree.

(vi) `SAMSFILES`

This variable defines a list of `.sams` files to run the `sams` processor on.

Some ODE documentation describes the variables `EXPLINKS` and `EXPBIN`. Do *not* use these as they are not supported by the DCE common makefiles.

(d) ODE Variables Used in the build Pass

These variables are used by ODE during the build pass to determine various targets and dependencies. The build pass is when programs and non-exported libraries are built. It is the main build pass.

(i) `LIBRARIES`

This variable defines a list of libraries which need to be compiled and archived. See the definition of `EXPLIB_TARGETS` for how to export libraries. If `EXPLIB_TARGETS` is *not* defined then all libraries listed here will be exported during the `EXPLIB` pass. Libraries have dependencies on objects defined in `OFILES` and headers defined in `HFILES`, both described below.

(ii) `SHARED_LIBRARIES`

This variable defines a list of shared libraries which need to be compiled and archived. See the definition of `EXPSHLIB_TARGETS` for how to export shared libraries. If `EXPSHLIB_TARGETS` is *not* defined then all libraries listed here will be exported during the `EXPSHLIB` pass. Shared libraries have dependencies on objects defined in `OFILES` and headers defined in `HFILES`, both described below.

There are differences in building shared libraries on different platforms. The best place to find out more about these differences is Chapter 10 of the *DCE Porting and Testing Guide* for DCE 1.0.2.

(iii) `PROGRAMS`

This variable defines a list of programs to compile and link. Programs have dependencies on objects defined in `OFILES` and headers defined in `HFILES`, both described below.

(iv) `OBJECTS`

This variable defines a list of objects that need to be compiled. Do *not* use this to build programs, use this only for special cases where objects must be explicitly placed into the `obj` tree. Some `Makefile`'s that build `libdce` use this.

(v) `SCRIPTS`

This variable defines a list of shell scripts that get built. If this variable is defined then rules for files ending in `.csh` and `.sh` are created by including the common makefile `osf.script.mk`. In this case ‘‘built’’ means the following three things happening:

[a] Shell scripts are copied to the `obj` tree and are stripped of their extension. The elements of the `SCRIPTS` list must not have a `.sh` or `.csh` extension, however the corresponding files in the `src` tree must have one of the extensions. So the correct way to specify a script to the build is to have in the `Makefile`:

```
SCRIPTS = foo
```

which will result in `make` looking in the `src` tree for either `foo.sh` or `foo.csh` and then it will create a file called `foo` in the `obj` tree.

[b] All of the execution bits are set on the file created in the `obj` tree.

[c] Runs `sed` on the file to change all occurrences of `@SOURCEWARNING@` into:

```
THIS IS NOT A SOURCE FILE - DO NOT EDIT
```

If the variable `SED_OPTIONS` is set then its value is passed to `sed` as well. A possible use of this variable is to strip all the comments out of scripts.

Use this variable only if scripts need some processing before being installed. If the script in the `src` tree is what should be installed on a system, then use the `ILIST`, *et al* variables described below and not the `SCRIPTS` variable. Do *not* use `SCRIPTS` as a convenient name for a variable and then include it in some other variable list such as `DATEFILES` or `ILIST`; only use it for its intended purpose.

(vi) `DATAFILES`

This variable defines a list of files that get copied from the `src` tree to the `obj` tree, and keep the same name. Defining them with this variable creates a rule that `make` can use to check if the file in the `obj` tree is up-to-date. This need only be used for files that need to be in the `obj` tree during some later parts of the build (files can go directly from the `src` tree to the `install` tree), and which have the same name in the object tree as in the source tree.

An example of the need for this variable is a tool that is compiled (so it is in the `obj` tree which does not have some equivalent of `cpp`'s `-I` switch. In this case, ODE has no way to tell it about search paths in the `src` tree. Therefore, to use this tool, the file will have to be copied to the `obj` tree, but what if it is then updated in the `src`? If the file is just copied, `make` has no way knowing that the file changed. If however, it is defined with `DATAFILES`, then `make` can check if it is up-to-date, and copy it to the "obj" tree only when necessary.

(vii) `OTHERS`

This variable defines a list of targets to be built which do not fall into another categories. These targets are built by `make` but the common makefiles do not have any rules on how to build `OTHERS`. To build others, specific make rules will have to be specified in the `Makefiles`.

(viii) `OFILES`

This variable defines a list of object files that comprise a program or library. If there is only one program or library described in the `Makefile`, then just use this variable name, if there are more than one, then use the syntax *`target`*`_OFILES`, where *`target`* is the name of the program or library.

(ix) `HFILES`

This variable defines a list of header files on which a program or library depends. If there is only one program or library described in the `Makefile`, then just use this variable name, if there are more than one, then use the syntax *`target`*`_HFILES`, where *`target`* is the name of the program or library.

(e) ODE Variable Conventions,

ODE has several conventions for its variable naming scheme. The suffix of variable names defines the manner in which the variable should be set.

(i) `ARGS` — Variables to be set on the command line.

(ii) `ENV` — Variables to be set in the environment

(iii) `FLAGS` — Variables to be set in `Makefile`'s

(iv) `DIRS` — Variables to be set in `rc_files`'s

For example, there are four variables used to generate the list of `-I`'s to a compile line. They are `INCARGS`, `INCENV`, `INCFLAGS`, and `INCDIRS`.

(f) The ODE Flag Variables

The following is a list of the variables that ODE uses to determine flags to set on various command lines. The flags given below should be set in `Makefile`'s. Those that end with the suffix `FLAGS` follow the ODE variable naming conventions described above, and therefore have alternate forms to be set in different manners.

(i) `CFLAGS` (also `ARGS` and `ENV`)

This variable defines a list of flags to pass to the C Compiler.

(ii) `INCFLAGS` (also `ARGS`, `ENV` and `DIRS`)

This variable defines a list of directories to be included in the C compiler invocation line with the `-I` switch.Each directory included on this list is used as an argument to `genpath` so that the equivalent directory in the sandbox and backing tree (and any intermediary shared sandboxes) are included in the compile line.

(iii) `LDFLAGS` (also `ARGS` and `ENV`)

This variable defines a list of flags to pass to the linker.

(iv) `LIBFLAGS` (also `ARGS`, `ENV` and `DIRS`)

This variable defines a list of directories to be included in the linker invocation line with the `-L` switch. The values in this list are of the form `-L`*`path`* where *`path`* is a relative path to the directory where the library should be found.

Each directory included on this list is used as an argument to `genpath` so that the equivalent directory in the sandbox and backing tree (and any intermediary shared sandboxes) are included in the link line.

(v) `IDLFLAGS` (also `ARGS` and `ENV`)

This variable defines a list of flags to pass to `idl`.

(vi) `IDLINCFLAGS` (also `ARGS` and `ENV`)

This variable defines a list of directories to be included in the `idl` compiler invocation line with the `-I` switch. The values in this list are of the form `-I`*`path`* where *`path`* is a relative path to the directory where the library should be found.

Each directory included on this list is used as an argument to `genpath` so that the equivalent directory in the sandbox and backing tree (and any intermediary shared sandboxes) are included in the compile line.

(vii) `LFLAGS` (also `ARGS` and `ENV`)

This variable defines a list of flags to pass to `lex`.

(viii) `YFLAGS` (also `ARGS` and `ENV`)

This variable defines a list of flags to pass to `yacc`.

(ix) `OPT_LEVEL`

This variable is meant to define debugging and optimization switches. It is included in the both the compile and link invocation lines.To specify optimization for just one of these lines use the following variables:

[a] `CC_OPT_LEVEL` — Used to specify optimization for just the compiler.

[b] `LD_OPT_LEVEL` — Used to specify optimization for just the linker.

(g) Variables Used in the install Pass.

These variables are used by ODE during the install pass to determine various targets and dependencies. The install pass populates the `install` tree with targets found in either the `obj` tree or the `src` tree. ODE will look first in the `obj` tree and then in the `src` tree to find targets.

(i) `ILIST`

This variable defines a list of targets to place in the `install` tree.

(ii) `TOSTAGE`

This variable defines the base directory to install all target listed in `ILIST`. It is usually set by the rc_files and is set to include `${sandbox_base}`.

(iii) `IDIR`

This variable defines the directory (relative to `TOSTAGE`) in which to install the targets listed in `ILIST`. Targets are installed in the directory `${TOSTAGE}/${IDIR}`. Note that the path assigned to this variable is terminated with a `/`.

(iv) `IMODE`

This variable defines the permissions to set on installed targets.

(v) `IGROUP`

This variable defines the group to set on installed targets.

(vi) `IOWNER`

This variable defines the owner to set on installed targets.

REFERENCES

[AES] Application Environment Specification — Operating System Programming Interfaces Volume, Revision A, Open Software Foundation, 1989.

[ANSI C] ANS X3.159-1989, Programming Language C (which is identical to ISO/IEC 9899:1990, Programming Languages — C).

[BSD] UNIX Programmer's Reference Manual 4.3 Berkeley Software Distribution (BSD), April 1986.

[DUG] OSF Development Environment User's Guide, ODE Release 2.2, Open Software Foundation, January 1993.

[GCC] R. M. Stallman, ''The GNU C Compiler''.

[K&R] B. W. Kernighan and D. M. Ritchie, ''The C Programming Language'', 1st ed., Prentice Hall, 1986.

[I18N] T. Ogura, ''DCE 1.1 I18N Workbook'', DCE Document, October 1992.

[PORTG] DCE Porting and Testing Guide, Open Software Foundation, 1992.

[POSIX.1] ISO/IEC 9945-1:1990, Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] (which is identical to IEEE Std 1003.1-1990).

[POSIX.2] ISO/IEC DIS 9945-2:1990, Information technology — Portable Operating System Interface (POSIX) — Part 2: Shells and Utilities (which is identical to IEEE Std 1003.2-Draft).

[RFC 17.0] H. Melman, DCE-RFC 17.0, ''DCE Code Clean-Up Requirements'', December 1992.

[RFC 23.0] R. Mackey, DCE-RFC 23.0, ''DCE 1.1 Internationalization Guide'', January 1993.

[RFC 24.0] R. Salz, DCE-RFC 24.0, ''DCE 1.1 Serviceability Proposal'', November 1992.

[RFC 34.0] H. Melman, ''DCE 1.1 Coding Style Guide'', April 1993.

[RFC 34.1] H. Melman, ''DCE 1.1 Coding Style Guide'', June 1993.

[RFC 35.0] H. Melman, DCE-RFC 35.0, ''Location of DCE files'', January 1993.

[XPG4] X/Open Portability Guide Issue 4, Volume 2 — 1992 XSI System Interface and Headers.

AUTHOR'S ADDRESS

Howard Melman
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA

Internet email: melman@osf.org
Telephone: +1-617-621-8989