DCE 1.1 CODE CLEAN-UP REQUIREMENTS

1. INTRODUCTION

A goal of DCE Release 1.1 is to improve portability by responding to issues raised by DCE engineers and DCE licensees. This RFC examines these issues. Emphasis in this list has been placed on items that have actually made DCE code difficult to port in the past. The DCE SIG (Special Interest Group) was surveyed in order to identify as many code clean-up issues as possible, and the results of that survey will be noted as occasion arises throughout this document.

1.1. Audience

The intended audience of this RFC is those familiar with DCE code. This RFC is not meant as a ''cookbook'' with all details of how to change old code or write new code meeting DCE 1.1 standards. It is meant to provide enough information to determine what existing code needs to be changed, and to some extents the scope of those changes. For example, in the ''Use stronger compiler switches'' section below, this RFC states that more compiler switches will be used, and lists all possible ones, but does *not* specify the set of switches that *will* be used, merely that the set will be determined by OSF. Complete details are specified wherever possible.

1.2. Priorities

The list of requirements is fairly long. The requirements are grouped into similar categories. Within each category they are generally listed from highest priority to lowest priority. This is not to mean that only some of these are requirements, merely that it is recognized that some are more important than others.

2. REQUIREMENTS

This section lists work items that are expected to be completed for DCE 1.1. Some of the items are required for both existing and new code, others for just new code. This is specified in each item.

2.1. ANSI C Compliance

DCE 1.1 will *require* an ANSI C compiler (see [ANSI C]) (except, names will be unique to 31 characters, not 8 characters). There will no longer be guaranteed support for the original Kernighan and Ritchie C [K&R]. Existing support may be left in the code; however, if it is not removed, it must be modified to use a single DCE-wide mechanism defined in `dce.h` (see Appendix below for a description of `dce.h`). Some mechanisms are currently defined in `dce.h`, any others required must be added to `dce.h` with OSF approval.

2.1.1. REQT: Use stronger compiler switches

OSF/1, running on a specified platform, will serve as one of the DCE 1.1 reference platforms. On that platform `gcc` will be the compiler used, with the `-ansi` switch. In addition, other switches will be turned on to indicate additional warnings which must be fixed in the DCE code.[1]

The complete list of `gcc` version 2.0 warning switches is given below from the `gcc` documentation [GCC2]. The `-pedantic` switch (which must be used with the `-ansi` switch) does not add as much checking as one might think. It will be used, but others must be used as well. The `-Wall` switch implies all other switches in the column above it.

---

1. An exception is any kernel code which will follow the conventions of the kernel build environment. 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.

```
-W                  -pedantic
-Wimplicit          -Wtraditional
-Wreturn-type       -Wshadow
-Wunused            -Wid-clash-31
-Wswitch            -Wpointer-arith
-Wcomment           -Wcast-qual
-Wtrigraphs         -Wcast-align
-Wformat            -Wwrite-strings
-Wuninitialized     -Wconversion
-Wall               -Wnoparentheses
```

The list of switches used will be determined by OSF after some experience with `gcc` version 2.0 is obtained.[2] All switches will be tried, however there may be problems using some of them. For example, there is a possibility that the use of the `TRY` and `CATCH` macros of the CMA exception handling interface will always generate a warning detected by the `-Wuninitialized` switch. If there are problems with a particular switch and the problem is not fixable, then either the switch will be turned off, or the warnings from those cases will be ignored.

2.1.2. REQT: Remove use of __STDC__

ANSI C requires the macro `__STDC__` be defined by the compiler. However, many compilers only define it in a strict ANSI mode, or support a subset of ANSI C features. Some DCE code makes use of explicit checks to see if `__STDC__` is defined to determine if a feature of ANSI C is to be used. These must either be removed completely, and the ANSI feature assumed to be present, or they must be changed to use a feature-specific macro defined in `dce.h`. This must be done for all existing and new code.

For example, the use of prototypes will be set on the flag `_DCE_PROTO_`, which will be set in `dce.h`. DCE code may not check if `__STDC__` is set to turn a prototype on or off. See the section ‘‘Function prototypes’’ in the ‘‘Style Guide’’ Appendix of this RFC for details.

Currently there are only two known uses of the `__STDC__` macro; to check for prototypes and for void pointers. Any other uses must be changed as well.

2.1.3. REQT: Use function prototypes

In addition to supporting ANSI C, DCE 1.1 will require prototypes for all functions.[3] 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.

---

2. `gcc` version 2.0 will probably be used for DCE 1.0.2.

3. An exception is any kernel code which will follow the conventions of the kernel build environment. It is not required that C code generated by `idl` or `mavros` contain function prototypes for all functions, however this should be a goal for future development work as resources allow.

The current use of a prototype macro as a selection method will not be required. However there will be one prototype macro defined in `dce.h` that can be used if desired. The use of all other prototype macros in existing or new code must either be removed or changed to this macro. It is described in the Style Guide Appendix of this RFC. It was chosen since it is very similar to the macro used in CMA, RPC, CDS, and GDS. DTS currently uses a macro of a different form.

To restate the requirement, all functions must use prototypes and new-style function definitions. Support for old-style function definitions can be included, but must use the common DCE-wide macro.

2.1.4. REQT: Use only comments after conditional compilation macros

All instances of `#ifdef` in both new and existing code must be ANSI C compliant. This means that there can be no text following a `#else` or `#endif` unless it is a comment. The following is *illegal*:

```
#ifdef AIX
    /* ... */
#else  not AIX
    /* ... */
#endif AIX
```

It should be changed as follows:

```
#ifdef AIX
    /* ... */
#else  /* not AIX */
    /* ... */
#endif /* AIX */
```

The text should not simply be removed. If the text is still correct, place it within comment delimiters.

2.2. REQT: Better Machine Dependency Handling

Provider companies will be responsible to ensure that target machine dependencies are in fact target machine dependent. Code that can be isolated based on a feature such as endianess will be handled as follows:

```
#if defined(BIG_ENDIAN)
    /* ... */
#else   /* LITTLE_ENDIAN */
    /* ... */
#endif  /* defined(BIG_ENDIAN) */
```

That is, not in a target machine specific manner. One of `BIG_ENDIAN` or `LITTLE_ENDIAN` will be defined in `dce.h`. The solution and the different macros used must be documented in the *DCE Porting and Testing Guide* [PORTG].

All macros used in conditional compilation (i.e., `ifdefs`) must be documented in the *DCE Porting and Testing Guide*. Provider companies are responsible for supplying this information to OSF so it can be documented.

The current list of supported target machines must be checked as well. The extra code needed to support non-references platforms often obfuscates the code. Support for non-DCE systems must be removed (e.g., DECNET) in both `#ifdefs` and in target machine specific subdirectories. It is believed that in DFS there are a lot of target machine and architecture conditionals that go back to platforms supported in AFS 3. These must be removed as well.

All remaining machine dependencies must be isolated and moved to machine dependent files. There will be no `#ifdef` in any existing or new DCE code to check what machine it is on. This will make porting to a new platform easier, particularly when dealing with updates. Every machine-dependent function will be in a machine-dependent file (preferably of the same name). These functions (both existing and new) must be documented in the *DCE Porting and Testing Guide*. This work will be done by provider companies.

2.3. Consistency

There were many complaints about DCE code being inconsistent and having many duplicate definitions of functionality. A major complaint was that DCE seems to use every known system interface, making it difficult to port to some systems. The following requirements address these concerns.

2.3.1. REQT: Place system wide definitions in dce.h

Lack of centralization of DCE-wide definitions is a very common complaint about DCE 1.0. Much of the code that requires porting is spread throughout DCE, and often is duplicated. A new public header file, `dce.h`, will provided by OSF and will be included by most, if not all, DCE C source files. This file will include all typedefs, macro definitions, and possibly common includes, etc., that span components. It will be machine specific; it must be ported to each platform. OSF will ‘‘own’’ this file, providers are responsible for including it in their code.

See Appendix A in this RFC for the current version of `dce.h` for the PMAX platform. Providers who have typedefs, macros, prototypes, etc., that need to be included, must identify them for inclusion by OSF. Examples of typedefs and macros found in this header file include:

(a) `boolean`
(b) `byte`
(c) `signed8`
(d) `signed16`
(e) `signed32`
(f) `unsigned8`
(g) `unsigned16`
(h) `unsigned32`
(i) `BIG_ENDIAN`
(j) `LITTLE_ENDIAN`

Also, other DCE-wide declarations such as serviceability macros and prototypes, and debugging macros will be obtained by a module by including `dce.h`.

It will be 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.)

2.3.2. REQT: Standardize system and library calls

One of the complaints about DCE is that it uses many different routines that do the same thing. In an effort to reduce this problem, DCE 1.1 defines the following hierarchy of standards to be used to determine the appropriate system or library routine to call. Much of this work will be accomplished during the SVR4 merge as part of DCE 1.0.2.

All new code *must* conform to the following. All existing code *should* as well, but where this is not practical, at a minimum all non-standard usages 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()`).

(b) POSIX.1
Routines defined in [POSIX.1] (e.g., `sigaction()` instead of `signal()`).

(c) XPG4
Routines defined by X/Open in [XPG4] (e.g., message catalog routines `catopen()`, `catgets()`).

(d) AES
The Operating System Programming Interfaces Volume of the Application Environment Specification [AES] from OSF must be consulted first. Its organization is similar to the other items on this list.

(e) Finally, if none of the above standards are applicable, the extensions need to be localized 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).

2.3.3. REQT: Use stdarg not vararg

All uses of the `vararg` facility to handle variable length argument lists must be changed to use the ANSI C defined `stdarg` facility. No support of the `vararg` facility will be in DCE 1.1 since its similarity with the `stdarg` facility can make it confusing for the reader. Provider companies will be responsible for doing any conversions necessary.

2.3.4. REQT: Eliminate repetitive code between components

A utility library which would hide some of the utility/OS specific code from the rest of the system would aid in porting. OSF will provide (or approve) an implementation of the following items to be used by all DCE components.

(a) Common 64 / 128 bit arithmetic utilities.

(b) Getting unique IDs, e.g., ethernet addresses. There are currently two routines that do this; one in RPC and one in CDS.

(c) Hash table functions particularly in DFS.

(d) A single memory management scheme (allocate and free) with debugging switches which would simplify finding memory leaks.

OSF will create a new ‘‘common library’’ that will include the above mechanisms.

2.3.5. REQT: Use upper-case for all macro names

In order to make the DCE code easier to read and follow, all macro names defined by DCE will be in all upper-case. Providers are responsible for changing all existing code to follow this convention, and for ensuring all new code follows this convention. Macros that implement standard routines, will follow whatever conventions the standard requires (e.g., the pthreads routines defined by CMA must be in lower-case).

2.3.6. REQT: Use consistent debug macros

There are 58 different macros in DCE that have the text `DEBUG` in them. Some use positive logic, others negative to turn debugging on or off. Some are set only at compile time and others allow for runtime setting. A single mechanism, to be used across all DCE components, is specified as part of the DCE 1.1 serviceability project. It can be set on a component by component basis, at both compile time and at runtime, and will be obtained by including `dce.h`.

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.[4] For example, the following should appear in a `cds.h` file:

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

All new and existing debug code must use or be changed to use this mechanism.

2.3.7. REQT: Remove inappropriate dependencies

No DCE code should make any calls directly to CMA, Kerberos or any other inappropriate interface. An inappropriate interface is one which is not exported by the service for use by a DCE component. For example, `cma_*()` calls are not to be used by other components, `pthread_*()` calls should be used. Also,

---

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

`cdspi_*()` calls are allowed to be called by the NSI (and XDS), but not by other components, they should use the NSI.

It is believed that DFS makes calls directly to Kerberos, and CDS makes calls directly to CMA. These need to be checked and fixed. Each component must be reviewed for the appropriateness of the calls it makes to other components in both existing and new code.

2.3.8. REQT: Remove dependency on uid of 0

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 module, to isolate these issues. All existing and new code must call these routines where appropriate.

```
sec_priv_proc_enable() {
    return seteuid(0);
}

sec_priv_proc_disable() {
    return seteuid(geteuid());
}

sec_priv_proc_is_appropriate() {
    return geteuid() ? TRUE : FALSE;
}
```

Hence:

```
if ( seteuid(0) == 0 )
```

becomes:

```
if ( !sec_priv_proc_enable() )
```

And:

```
euid = geteuid();
/* ... */
if ( euid != 0 ) /* ... */
```

becomes:

```
euid = geteuid();
/* ... */
if ( !sec_priv_proc_is_appropriate() ) /* ... */
```

2.3.9. REQT: Allow changeable DCE root directory

It needs to be possible to change the DCE root directory without having to recompile nearly all the files. `DCE_LOCAL` will be a variable instead of a macro. It will not be an environment variable for security reasons.

The header file `dce.h` will declare a variable defined to give the path of the root of the dcelocal directory on the installed system. This is by default `/opt/dcelocal`:

```
char *dcelocaldir = "/opt/dcelocal";
```

All references to this path in both existing and new code must be determined by the use of this variable.

2.3.10. REQT: Better tty handing in admin programs

The control programs, as well as all other DCE programs, will use the termios interface from [POSIX.1] and [XPG4] as specified in the [AES] for handling terminal I/O. Again, a single interface used for this function will make the code easier to port. All code both existing and new must be changed to use these interfaces as opposed `ioctl()` or anything else.

2.4. Include Files

There were also some complaints about the usage of include files in DCE. Simple solutions to these problems are presented below.

2.4.1. REQT: Use system include files

There are places in DCE where system and library calls are prototyped, as opposed to the prototype being obtained by including a system header file. These must be removed and replaced with the appropriate `#include` in all existing and new code.

If systems do not properly define the prototypes in these header files, it is considered a machine specific problem and should be handled in a machine specific manner in `dce.h` or in another appropriate header file.

2.4.2. REQT: Consistent prevention of duplicate inclusion

Most, if not all, DCE header files currently have some mechanism to prevent duplicate inclusion of the same header file. However, different mechanisms are used. It will be standardized in DCE 1.1 so that all header files, in both existing and new code, begin and end with:

```
#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.

Some files check `HEADER_INCLUDED` or some other macro. Ideally, checks should be done at the point of the inclusion instead of within the file to avoid having to read the whole file if it has already been included. This will *not* be required for DCE 1.1, but having consistently generated macros to check would facilitate such a change if a licensee deems it necessary.

Providers are responsible to check and if necessary change all headers so that they check for duplicate inclusion in the proper manner.

3. BUILD ENVIRONMENT REQUIREMENTS AND ISSUES

One response to the Code Clean-Up Survey was phrased very succinctly:

> ‘‘Simplify the build environment! The makefiles are too complex.’’

Build environment issues are all grouped in this section since typically they do not involve DCE code, but rather the makefiles and source tree organization.

3.1. Make Clean-Up

There are some simple clean-up issues that need to be done:

3.1.1. REQT: Use RIOS method for libdce.so

When DCE is ported to OSF/1 1.1 the complicated method of building libdce on a PMAX can be removed and both the PMAX and RIOS can use the same method of getting the objects for libdce from the component libraries. This work will be done by OSF engineers.

3.1.2. REQT: Use install tree properly

All the components must be reviewed so that everything that is needed is placed in the install tree not the object tree. The tests are the worst offenders. The goal of a build is to populate the install tree, the object tree is just an intermediary point. The results of the build, including all executables, datafiles, scripts, and anything else needed to run DCE or the tests, will be in the install tree. This work will be done by DCE engineers at provider sites.

3.1.3. REQT: Use libraries where needed

If an object is to be included in more than one executable it must be in a library. Any places where this is not true must be corrected in both existing and new code. This work will be done by provider companies.

3.1.4. REQT: Build DCE in network 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.

3.1.5. REQT: Build kernel using standard method

The method of building the kernel used by DFS must be corrected so that the passes match as close as possible the passes used in other areas of the DCE build process. Currently each pass must be invoked by hand, they must be automated by ODE. This work will be done by the OSF Release and Tools Group.

3.2. REQT: Common Makefile Clean-Up

The DCE common makefiles need to be cleaned up as well. There have been many complaints that they are too complicated. In addition, they need to be documented. These files define the variables used in DCE makefiles. The documentation must include a usage section, explaining how DCE makefiles should be written.

The OSF Release and Tools Group will be responsible for the clean up of the common makefiles and for the writing of this documentation. Provider companies must modify both existing and new makefiles to meet these guidelines. In general, this will mean makefile cleanup on the order of using the `CFLAGS` variable correctly, and the proper use of `INCFLAGS`.

3.3. REQT: Better Machine Dependency Handling

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.

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 Release and Tools Group will be responsible for this work.

3.4. REQT: Better IDL Handling in ODE

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.

3.5. REQT: Incremental Builds

The build environment in DCE has been a problem for a long time. Two of the problems are the length of builds and the inability to do incremental builds. Incremental builds in DCE are rebuilds of all of DCE, in a sandbox that already contains a previously built DCE. Header file and library dependencies are not included in the makefiles, so if these are changed, it is up to the programmer to know what needs rebuilding. This is not always possible with a project the size of DCE, so full rebuilds are done. ODE has a tool called makedepend (`md`) that can generate header files dependencies by parsing the C source files. The current method of building `libdce` on a PMAX prohibits this from being run, but this will be corrected when DCE is ported to OSF/1 1.1. Library dependencies are important as well, but since these are less numerous and less mutable, they will be included by hand in the makefiles. `md` works by building static files with dependencies, this is a process that will be run during a nightly build, so any changes in dependencies might not be immediately reflected in these dependency files. The OSF Release and Tools group will be responsible for this work.

3.6. Partial Builds

Incremental builds are not the only problem. If the above is implemented, there is still a restriction that builds happen from the top `src` directory and be allowed to progress through *all* the subdirectories in each pass. This is because there are many unspecified inter-directory dependencies. One directory will assume that another directory has already been built and try to find objects there via the `VPATH` construct. This creates two problems. First, it is hard to rebuild one component (known as a partial build) since you might have to rebuild in another directory to get objects it depends on, and two, it makes debugging build problems more difficult. If you find that something did not build, you must find out from which directory it was trying to build, and from where it was trying to get sources. Use of the `VPATH` construct should be avoided, unfortunately there are 127 occurrences of it in the DCE makefiles. The solution to this problem is a tree reorganization.

3.7. Tree Reorganization

This involves changing the tree structure of the DCE source tree including directory names and organization. The goal is to simplify the makefiles and allow partial builds. Many of the complexities come from the fact that there is more than one build target in most of the DCE directories. This exploited a bug in ODE for which the fix involved invoking `make` once for every pass (there are 7 passes) in each directory, whether it is needed or not. It also makes the makefiles more complex since each make variable must be prepended with the target.

To solve this, the tree should be restructured in the following manner. (Much of this comes from a document describing the DME tree structure.)

`comp`	
`comp/h`	all headers that are component wide
`comp/idl`	IDL files
`comp/lib`	directory for libraries
`comp/lib/lib`*`A`*	contains everything for library *`A`*
`comp/lib/lib`*`B`*	contains everything for library *`B`*
`comp/`*`foo`*`d`	all sources for the *`foo`* daemon
`comp/`*`foo`*`cp`	all sources for the *`foo`* control program
`comp/`*`foo`*`clerk`	all sources for the *`foo`* clerk

Additional rules would include:

(a) Use of `VPATH` is discouraged (not sure if it can be made illegal).

(b) Library dependencies for executables must be included in makefiles.

(c) Header dependencies will be generated using `md`.

Certainly there are some aesthetic improvements as well. The code is more uniform and it is easier to find something in the tree. (How fast can you find `acl_edit` in the tree? It is not in `src/security/client/acl` as you might think, it is in `src/security/client/admin/acl_edit`. `sec_clientd` gets its own directory, but `secd` is found in `src/security/server/bin`.)

Unfortunately, a tree reorganization would cause many problems for continuing development and perhaps for licensees. Before a decision can be made, a prototype will done. One component will be reorganized, the changes noted, and then the affected parties conferred with to determine the impact of such changes. OSF will prototype and make the decision based on cost, benefit, and resources. Providers should plan to convert to the new structure during DCE 1.1.

4. COMPONENT SPECIFIC REQUIREMENTS

This section is derived mostly from responses received from the Code Clean-Up Survey. These items are specific to a particular component. Often they are restatements of other requirements, but are included here where a particular component requires more than general work to meet the requirement. Included here as well, are items received from the Code Clean-Up Survey that will *not* be requirements of DCE 1.1. These are clearly identified and are included for informational reasons only.

4.1. REQT: RPC

Requirements:

(a) The RPC runtime has lots of old status codes, aliased with `#define`. They need to be removed for DCE 1.1.

(b) There is some debate on this, but remove the use of `memcmp()` from `uuid_equal()`.

(c) Defining types with IDL.

As a rule, do not define types via IDL and then send them over the net without going via marshalling / unmarshalling code — doing this for rpc message headers would be a big expense, so in cases like that, do not use IDL defined types as part of the message headers.

Do not define system types via IDL and then pass them to system calls (e.g., `sockaddr_t` from `rpc/sys_idl/lbase.idl`, was a ‘‘truly painful experience’’ for some licensees).

(d) Aligning pointers to word boundaries / treating pointers as integers. (Actually, 1.0.1 gets rid of many of these.) These are cases where pointer arithmetic is done by casting pointers to integers:

(i) cnbuf.c: at the end of `rpc__cn_fragbuf_alloc()`

(ii) cnpkt.c: in `force_alignment()`

(iii) cnrcvr.c: in `receive_packet()`:

```
iov.base = (byte_p_t)((unsigned32)(fbp->data_p) +
                      fbp->data_size);

memcpy ((*ovf_fragbuf_p)->data_p,
        (pointer_t)((unsigned32)(fbp->data_p) +
                    fbp->data_size),
        (*ovf_fragbuf_p)->data_size);
```

(iv) cnsassm.c: in `do_alter_cont_req_action_rtn()`:

```
sec_addr = (rpc_cn_port_any_t *)
           ((unsigned32)(resp_header) +
            header_size);

pres_cont_list = (rpc_cn_pres_cont_list_t *)
                 ((unsigned32) req_header +
                  RPC_CN_PKT_SIZEOF_ALT_CTX_HDR);
```

```
pres_result_list = (rpc_cn_pres_result_list_t *)
                   ((unsigned32)(resp_header)
                    + header_size);
```

(v) cnsassm.c: in `do_assoc_req_action_rtn()`:

```
sec_addr = (rpc_cn_port_any_t *)
           ((unsigned32) (resp_header) +
            header_size);

pres_result_list = (rpc_cn_pres_result_list_t *)
        ((unsigned32) resp_header + header_size);

pres_cont_list = (rpc_cn_pres_cont_list_t *)
                 ((unsigned32) req_header +
                  RPC_CN_PKT_SIZEOF_BIND_HDR);
```

Non-Requirements:

(a) If there is a system definition of a type, do not declare your own version, use the system one. An example of this is `rpc_addr_t` in the rpc runtime library, which is used instead of `struct sockaddr`. This is not included since this example is in error. The `rpc_addr_t` structure is an abstraction so that something other than sockets could be used by RPC. If there are other, valid examples of such problems, they will be corrected.

(b) The `uuid_t` type is defined via IDL, and is also sent down the wire as part of RPC message headers. Using some compilers, the C structure could end up being the wrong size. Licensees with this problem solved it by defining a new type using bit fields, and adding pack/unpack code to move between `uuid_t` structures and all RPC message headers. This also broke `uuid_hash()` in `uuid.c`.

This is not a problem with code but rather documentation. There are hooks in the code to correct this problem. On those machines where the format of the `uuid_t` type will cause problems, code must be added in the `#ifdefs` marked with `MISPACKED_HEADER`. There are only twon or three places where this occurs. Porting instructions need to be added to the *DCE Porting and Testing Guide*.

4.2. REQT: CDS

Requirements:

(a) Eliminate deprecated `utc_` function calls.[5]

CDS is using one or more of the following `libutc` routines: `utc_strtorel()`, `utc_strtouc()`, `utc_mkutcstr()`, `utc_mkrelstr()`. These routines are obsolete, and will be removed from `libutc`. CDS is the only component using these routines. The supported routines are: `utc_ascgmtime()`, `utc_asclocaltime()`, `utc_ascanytime()`, `utc_ascreltime()`, `utc_mkasctime()`, `utc_mkascreltime()`.

(b) Delete DECnet/DECdns code.

(c) Eliminate `all_ptr` union by doing proper casts where needed.

(d) Eliminate CMA dependencies.

(e) `cdscp`-specific:

(i) Reduce number of modules. A number will be agreed on in the future.

---

5. The text from this item came from CR Number 5164 in the DCE OpenTrack database.

(ii) Portable, configurable and documented command line recall.

(iii) Eliminate tools, e.g., `msg`, `prs`, and `parser_aid`.

(iv) Clean up startup code, particularly regarding removing VMS specific portions of the command line interface.

(f) Use `fcntl.h` instead of `sys/file.h`.

The contents of `fcntl.h` are regulated by POSIX (assuming a POSIX-compliant machine) whereas the contents of `sys/file.h` are not. Some licensees, for example, has had several namespace conflicts because their version of `sys/file.h` defines some structures that DCE tried to redefine, like `thread_t`.

This problem has been most common in CDS and GDS. Here is a sample list of files (probably not exhaustive):

(i) `src/directory/cds/adver/adver_clerk_event.c`
(ii) `src/directory/cds/adver/adver_main.c`
(iii) `src/directory/cds/child/clerk_main.c`
(iv) `src/directory/cds/control/format_tower.c`
(v) `src/directory/cds/control/startup_file.c`
(vi) `src/directory/cds/gda/gda_main.c`
(vii) `src/directory/cds/server/dump_files.c`
(viii) `src/directory/cds/server/mgmt_file.c`
(ix) `src/directory/cds/server/server_main.c`
(x) `src/directory/cds/server/unix_net.c`
(xi) `src/directory/cds/server/unix_stubs.c`
(xii) `src/directory/cds/server/db_unix.c`

(g) Pointer Casts and Dereferences.

There are places in CDS, where a byte pointer is legally cast to a struct pointer and then dereferenced. This is perfectly legal in ANSI C, but ANSI C also allows this to result in the generation of an alignment error. The suggested solutions range from ‘‘insert macro calls wherever this occurs in order to align the structure’’, to the cleaner, ‘‘use RPC and its marshalling capabilities, that is what it is there for’’. This casting problem is very pervasive throughout the CDS code.

The extreme solution of converting everything to use the IDL generated `cds_` structures code rather than those in `dns_record.h` would allow:

(i) The elimination of most of the macros

(ii) The elimination of the packing/unpacking of RPC calls (the current design defeats the purpose of IDL).

(iii) The elimination of dependencies on structure/field sizes

(iv) The elimination of word alignment problems (if done correctly).

(v) The code to be a lot more efficient in BOTH size and speed.

It is not believed that it would involve a protocol change since these problems occur in the communication between the `libcds` and the `cdsclerk`. The problem needs to be analyzed, a solution devised, and implemented.

4.3. REQT: GDS

Requirements:

(a) Uses of the following macro definitions that make C ‘‘look like’’ some other programming language must be removed.

(i) `#define Entr {`
(ii) `#define End }`
(iii) `#define Begin`
(iv) `#define Bend`
(v) `#define For for`
(vi) `#define While while`
(vii) `#define Do {`
(viii) `#define Od }`
(ix) `#define If if`
(x) `#define Then {`
(xi) `#define Else } else {`
(xii) `#define Elsif } else if`
(xiii) `#define Fi }`
(xiv) `#define Switch(x) switch (x) {`
(xv) `#define Case case`
(xvi) `#define Default default`
(xvii) `#define Esac }`
(xviii) `#define ADDR     &`
(xix) `#define begin    {`
(xx) `#define end      }`
(xxi) `#define GT       >        /* greater than */`
(xxii) `#define LT       <        /* less than    */`
(xxiii) `#define GE       >=       /* greater equal*/`
(xxiv) `#define LE       <=       /* less equal   */`

(b) The ANSI defined `stdlib.h` will be used instead of `malloc.h`.

(c) Translate all German comments into English. There are still German comments in the modules comprising `libshth.a`, `cmaxddr.h`, `malgcfe.h`, etc. A search of *all* the code must be done.

(d) Correct usage of target machine specific files. A target machine directory (e.g., PMAX) should not have any subdirectories.

(e) GDS must build in a sandbox mounted via NFS. This will affect how the initial database is created.

(f) Currently `mavros` generated code must be compiled with `gcc` using the `−traditional` switch. This must be corrected so that C files generated by `mavros` can be compiled in ANSI mode.

(g) Change the prototype type macro `FUPTYPE` in the file `d23ipc.h` and the other files it is used in, to follow prototype requirements stated in this RFC. The use of `TSTYPE` as defined in `tnsx.h` must be changed as well. These items are called out specifically so they are not lost; since a search for the string ‘‘`PROTO`’’ will not find them.

Non-Requirements:

(a) The code in directory `src/test/directory/xds` does not compile in ANSI mode because of function argument type clashes. These arise because of the use of logging functions that serve as wrappers around a `printf` statement; since the number and type of arguments tend to vary, the real solution would be to use `stdarg.h` or figure out if they really need a wrapper in the first place. These wrapper routines are defined towards the bottom of `xt_file.c`. This is a non-requirement since it will be addressed by new logging functions defined as part of serviceability.

4.4. REQT: DFS

Requirements:

(a) Remove conditionals for obsolete machines and operating systems.

(b) Decide on a consistent set of macro functions for handling `asfHyper` data types (64 bit integers). There are at least two sets of macros that perform the same operations!

(c) Consistent use of naming conventions. For example, `tkm_byterangePair_FREE_RANGES(oldCellP)` and `TKM_BYTERANGEPAIR_PRINT(stream, pair)` are macros, while `tkm_FidHash_LOOKUP_INSERT()` is a function.

(d) Consistent use of structure definitions, and use of typedefs. For example, the following can be found in the code:

```
typedef struct struct_name { ... } struct_name_t;
typedef struct struct_name { ... } struct_name;
struct struct_name { ... }
```

The first is used most consistently, and should be followed.

(e) Consistent use of typedefs as opposed to macros. For example, the following can be found in the code:

```
typedef struct lock_data  tkm_lock_data_t;
#define tkm_lock_data_t   struct lock_data
```

(f) Use ANSI C prototypes for all functions (except non-DFS kernel sources).

(g) Consistent error handling and error propagation. For example, some use value return from function, some use errno variable.

(h) Linked-list handling is reinvented every time it is used. General purpose linked-list functions should be implemented and would result in considerable savings in maintenance and testing.

The following requirements are open ended. Transarc must address these issues and present work plans to OSF. OSF and Transarc must agree on what are satisfactory exit criteria for this work.

(a) Makefiles have many problems that need to be cleaned up: use of obsolete libraries, including `machdep.mk` incorrectly, incorrect attempts to compile stubs differently, use of [BSD] functions, ...

(b) There are far too many conditionals in far too many files. There are 80 files with OSF/1 conditionals and a lot more with AIX conditionals. Things which are feature-related should have feature related conditionals, not AIX or OSF/1 conditionals.

(c) Reduce the use of macros where functions should be used, this makes debugging difficult. For example, in the token manager `TKM_INTERNALTOKEN_NEW_FROM_INT_TKN` is a macro with the following definition:

```
MACRO_BEGIN                                             \
 tkm_TokenID_INIT(&((dstP)->tokenID));                  \
 (dstP)->state = TKM_TOKEN_INIT;                        \
 (dstP)->sleepers = 0;                                  \
 (dstP)->pendingRPCCount = 0;                           \
 (dstP)->hostPossessingToken = (hostP);                 \
 if ((dstP)->hostPossessingToken != (struct hs_host *)NULL) { \
   (void)HS_HOLD((dstP)->hostPossessingToken);          \
 }                                                      \
 (dstP)->expiration = 0L;                               \
 (dstP)->timeQueued = 0L;                               \
 tkm_TokenDomain_Init(&((dstP)->tknDomain));            \
 (dstP)->nextP = (tkm_internalToken_p)NULL;             \
 (dstP)->prevP = (tkm_internalToken_p)NULL;             \
 (dstP)->headP = (struct tkm_tokenListHead *)NULL;      \
 \
 if ((srcP) == (tkm_token_t *)NULL) {                   \
   tkm_TokenSet_MAKE_EMPTY(&(dstP)->types);             \
   TKM_BYTERANGE_MAKE_RANGE_EMPTY(&((dstP)->            \
                               tknDomain.tokenBytes)); \
```

```
     }                                                   \
     else {                                              \
       tkm_TokenSet_COPY(&((dstP)->types), &((srcP)->type));\
       hset64((dstP)->tknDomain.tokenBytes.lowBnd,       \
              (srcP)->startPositionExt, (srcP)->startPosition); \
       hset64((dstP)->tknDomain.tokenBytes.upBnd,        \
              (srcP)->endPositionExt, (srcP)->endPosition);\
     }                                                   \
    MACRO_END
```

(d) Reduce the level of unnecessary function nesting. For example, `call_syscall` → `afs_syscall` → `kafs_syscall`. The difference between `call_syscall` and `afs_syscall`, is one additional parameter, which is added by `call_syscall` in the call to `afs_syscall`. Also, this function nesting is not consistently applied for rest of the code.

(e) Add block comments to each major module.

5. NON-REQUIREMENTS

This section discusses items that were received as part of the Code Clean-Up Survey sent to the DCE SIG, but are *not* expected to be completed (or necessarily begun) in DCE 1.1.

5.1. 16/64 Bit Clean

This is not a requirement for DCE 1.1 since it would be very difficult for some providers to test their code for 16 or 64 bit cleanliness. OSF will certainly accept corrected code, and wherever possible will include it in DCE code or documentation.

5.2. Reduce Code Complexity

Make use of analysis tools to determine the McCabe Complexity Metric of the code and simplify any code with too high a value. This work has been started here at OSF. DME has chosen 25 as the upper limit for acceptable code. Other experiences are that 10 is a better value. From a sample in July, 316 function in CDS (including some IDL generated code) had McCabe complexities above 10. This sounds worse than the situation really is. Many were machine generated functions, 167 were below 20 and 224 were below 25, but there is still real work to be done here.

Work of this nature is not considered Code Clean-Up but rather part of the Quality work scheduled for DCE 1.1.

5.3. Remove Slashes from Names

The following is common in DCE code:

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

For some machines the use of slashes in the name of the include file could cause a problem. However, since many compilers on these machines can handle slashes in names, and since it would be a large effort to change all DCE existing code, this is not a requirement for DCE 1.1.

5.4. Remove Use of BSD 4.3 Passwd Structures

The `passwd` structure in DCE (`pwd.h`) is based on BSD 4.3 and contains two fields (`pw_passwd` and `pw_gecos`) which are not defined by [POSIX.1]. Consequently, a compliant implementation of POSIX which is not required to support these fields, especially in the order defined in `pwd.h`, may produce unexpected results.

However, considering that many existing Unix systems, applications, and users rely upon the non-POSIX information in the `passwd` structure, the following was proposed:

> ‘‘DCE will continue to support the BSD 4.3 `passwd` structure internally. However, since POSIX-compliant system calls depend upon a `pwd.h` file which defines the POSIX `passwd` structure, it is recommended that the existing `pwd.h` file be renamed (`bsdpwd.h`?) for internal use only and that the `pwd.h` file provided by the platform vendor be used by system-specific DCE code for system calls.’’

There is currently a defect[6] open on this for DCE 1.0.2. It is scheduled to be fixed in DCE 1.0.2 by fixing the return of local OS data from sec_login to use platform dependent functions (as documented). The code will stop making any assumptions that a BSD 4.4 `passwd` structure is used by the local system.

5.5. Unify Control Program Syntax

Unifying the syntax of the control programs would be very nice from a user's point of view. It certainly would make things more logical. However this is out of the bounds of code clean-up for DCE 1.1. It might be included in other work areas.

5.6. Centralize Inclusion of System Include Files

There is a suggestion to have `dce.h` do work to centralize the `#includes` of system header files. Where system header files are needed (e.g., for the flags used by `open()`, `fseek()`, etc.) centralize the way they are included. One possible means of doing this is to have `dce.h` have sections along the lines of:

```
#ifdef _NEEDS_TCPIP_

#ifdef BSD42
# include /* ... */
#endif

#ifdef SYSV
# include /* ... */
#endif
#endif
```

Then in the actual source files you have the following:

```
#define _NEEDS_TCPIP_
#include <dce.h>
```

This would mean that only one place would need changing to get the correct header files for your operating system.

In order for common utilities to be mapped onto a single name, the same file could also do things like the following:

```
#if    BSD42
#  define block_copy(src, dest, len)    bcopy(src, dest, len)
#  define block_zero(dest, len)         bzero(dest, len)
#  define block_cmp(m1,m2,len)          bcmp(m1,m2,len)
#endif BSD42

#if    SYSV
#  define block_copy(src, dest, len)    memcpy(dest, src, len)
#  define block_zero(dest, len)         memset(dest, ' ', len)
#  define block_cmp(dest, len)          memcmp(m1,m2,len)
#endif SYSV
```

This suggestion would *only* be used in places where there are known machine differences, and if the methods described above in the ''Standardize System and Library Calls'' section do not result in finding an appropriate function to use.

5.7. REQT: Pretty Print Sources

This requirement is to determine a DCE style and to enforce it by running a pretty printer on all DCE sources. This will not make the code more portable, however, it will make it more consistent to read, which might be very

6. See CR Number 5978 in the DCE OpenTrack database for details.

important to licensees, particularly where one person is responsible for more than one component.

The major drawback of doing this to all DCE code is the effect it will have on tracking differences between versions of code as well as problems providers will have dealing with code in a new format. For these reasons this will not be done for DCE 1.1.

6. ISSUES

The items in this section require some further investigation or decision on their inclusion or exclusion from DCE 1.1 requirements. Issues raised in the Requirements section above are listed here for completeness.

6.1. Licensing Issues with Common Utilities

In selecting code for a common dce utilities library, there are licensing issues when these routines are used in other components. It is common practice that providers use the same code tree for their DCE product as well as their proprietary product. All DCE code is subject to the DCE license, except that which providers already owned. The use of common utilities covered under a DCE license can make this more difficult.

6.2. Retain RCD Ids in Code

It has been requested that the RCD Ids of files not be removed from the code to aid in version tracking of bug reports. It is common OSF practice to remove these Ids before shipping. Changing this practice will be investigated by OSF.

6.3. Use Better Locking Schemes

As recent 1.0.1 experience shows, DCE must provide better documentation on the locking schemes employed, both within each component and particularly between components. This documentation will go into the *DCE Porting and Testing Guide*. A simple layer above `pthread_mutex` calls could be implemented to enforce the lock hierarchy and aid in debugging.

Goals:

(a) Prevent inter-thread deadlock by a layer built on top of existing mutex/condition mechanisms.

Non-goals:

(a) Preventing inter-process deadlocks. (Historically insignificant and there is no simple place to hook in a mechanism.)

(b) Post deadlock detection/recovery. This is a much bigger project (backing out of transactions, state recovery, etc.).

APPENDIX A. SAMPLE DCE.H

The following is a sample version of `dce.h` for the PMAX platform. It is expected to be expanded.

A.1. dce.h

```
/*
 * COPYRIGHT NOTICE
 * Copyright (c) 1990, 1991, 1992, 1993 Open Software Foundation,
 * Inc.  ALL RIGHTS RESERVED (DCE).
 */

#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
 */
#include <dce/nbase.h>
typedef idl_byte        byte;
typedef idl_boolean     boolean;

/*
 * 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 directory, it will be "/opt/dcelocal"
 * The variable is defined in dceldir.c
 */
extern char *dcelocaldir;

/* 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>

#endif                                  /* _DCE_H */
```

APPENDIX B. STYLE GUIDE

The goals of this section are to capture items received in response to the Code Cleanup-Up Survey, which are not requirements, but are still recommended; and to describe methods of implementing requirements. It is a first step towards a DCE Code Clean-Up Cookbook.

B.1. Conditional Compilation

It has been suggested that comments following an `#else` and an `#endif` that describe the conditional be required in all DCE code. This will *not* be required for DCE 1.1 but is recommended.

There have been problems in DCE involving the differences between `#ifdef` and `#if`. An `#ifdef` test will return true if the value is defined, even if it is defined to be zero. An `#if` test will return true if the tokens evaluate to non-zero.

As a result of these problems and of problems when adding additional cases to an existing `#ifdef`, it has been suggested that `#ifdef` not be used at all in DCE and that it be replaced with the more explicit, C-like, clearer, and more easily extended:

```
#if defined(FOO)
```

This will *not* be required in DCE 1.1 code, as the difference is quite minor when adding conditionals, and `#ifdef` is quite different from `#if`. Mistakes that have been made confusing the two, could just as easily be made in including the `defined()` in the `#if`.

It has also been requested that a nested `#ifdef` be indented to match the level of indentation. ANSI C allows spaces between the leading `#` (which must be in the leftmost column) and the token. The following is a common style of indentation:

```
#if A
#  if B
     /* ... */
#  else   /* B */
     /* ... */
#  endif  /* B */
#else     /* A */
#  if B
     /* ... */
#  else   /* B */
     /* ... */
#  endif  /* B */
#endif    /* A */
```

While this style is recommended, it will *not* be required for DCE 1.1. It might make the code simpler to read, however, it does not aid in its portability. Also, there will be an effort to remove as many occurrences of `#ifdef` and `#if` in the code as possible, so it is expected that the above cases will occur infrequently.

B.2. Indicating Porting Problems

Whenever possible indicate porting problems at compile time. This is most easily accomplished with the `#error` pre-processor command. For example, if a section of code is dependent on a feature macro that needs to be defined in `dce.h` a conditional test before its use is appropriate:

```
#if !defined(BIG_ENDIAN) && !defined(LITTLE_ENDIAN)
#error Endianess must be defined in dce.h
#endif /* check for endianess macro */
```

B.3. Function Prototypes

The following prototype macro will be defined in `dce.h` and needs to be used for all functions that wish to support old-style function definitions. Support of old-style function definitions is *not* required, but if it is included in the code it must be done using the following macro definitions.

```
#if defined(__STDC__)  /* other conditionals can be tested */
#define _DCE_PROTO_
#endif

#if defined(_DCE_PROTO_)
#define _DCE_PROTOTYPE_(arg)    arg
#else
#define _DCE_PROTOTYPE_(arg)    ()
#endif
```

It is used in the following manner:

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

int foo
#if defined(_DCE_PROTO_)
        (
        int a,
        void *b,
        struct bar *c
        )
#else  /* no prototypes */
        (a, b, c)
        int a;
        void *b;
        struct bar *c;
#endif /* _DCE_PROTO_ */
```

This is very similar to the macros currently used by the CMA, RPC, and CDS components. A simple `sed` script could be run to change those components to use these macros.

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.

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

[GCC2] 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.

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

[POSIX.1] IEEE Std 1003.1 — 1988, Portable Operating System Interface for Computer Environments.

[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