DCE 1.1 SERVICEABILITY PROPOSAL

1. INTRODUCTION

This document details OSF requirements for the serviceability support in DCE 1.1. It describes a message-reporting facility to be provided by OSF. All technology providers will be required to modify their code to use this facility at their existing message code-points. Additional messaging points will have to be added, so this document also presents criteria providers can use to determine where these code points are.

This document is OSF's response to the SIG requirements as detailed in [RFC11]. During the course of several meetings and discussions within OSF we prioritized the requirements and reviewed the design presented here so that reasonable goals could be set for DCE 1.1.

This document is not intended to be a complete description of the API's and facilities. The definitive reference is the manual pages and source that will be part of DCE 1.1; they are currently unavailable.

2. CHANGES SINCE LAST PUBLICATION

A number of changes have been made since this document was originally published [RFC24]. Most of the changes were made in response to comments made by suppliers. This section gives a high-level overview of the changes that were made, in no particular order:

(a) The word *sub-component* is now used instead of *facility*.

(b) More explanatory text was added, including a detailed detailed explanation of the message severities, an explanation of the migration path and future directions, and rationale for some design decisions.

(c) Some minor changes were made to a few of the API's and data structures. For example, the constants were changed from `SVC_XXX` and the like to `svc_c_xxx`. The *dce_log* program was renamed *dcesvc*.

(d) The *DCE_SVC_LOG* macro was added.

(e) The ''send message to console'' action flag was removed.

(f) References to security auditing and alarms were removed.

3. GOALS

The goal of serviceability is to make DCE easy to install, configure, and administer. The last area requires that DCE programs provide enough information so that an administrator can isolate faults and correct errors.

The primary method of accomplishing this is by having DCE programs detect error conditions and generate a message. All messages must be generated by calling a single routine. All components must be modified so that, *e.g.*, `printf()` calls are replaced by slightly different `dce_svc_printf()` calls. By requiring that all messages use this API, licensees can easily replace DCE serviceability facilities with their own, native, facilities.

All DCE messages must be consistent, providing a standard minimum amount of information. In order to guarantee this, all message calls will use a provided macro as their first argument. This macro will expand to several arguments. By using a macro, licensees can add additional information with minimal code changes to either DCE or customer source code.

All messages must meet internationalization requirements, as specified in [RFC23] (see also [Ogura], [XPG4], and [RFC34]). As a summary, however, all messages must be stored in message catalogs, they must not use fragmentation, and they may be displayed using the XPG4 language-independent `printf()` format string conventions. Messages will be uniquely identified by message number assigned from the status code space. No component should use the last 100 message numbers in its number space; these are reserved for licensee use. In general, messages should not be re-used unless the exact same error occurs in different places in the code. Because of the practical difficulty in achieving this, this is a guideline rather than a requirement.

Each message must have external explanatory text. This will be done using the message file described in [RFC23]. Each component will have a separate chapter in the *Problem Determination Guide* that is constructed primarily from the explanatory texts.

Only one process will write to each file, although the interface will be encapsulated so that licensees can extend this to allow different processes to write to the same file.

For DCE 1.1, we believe it is sufficient to provide only local log files written in human-readable or native binary format. We will provide an API to read the binary logs along with a sample log viewer. ''Remote Notification'' as described in [RFC11] is a non-goal for DCE 1.1. We might move to IDL network representation (that is, ''pickling'') after it is available. We expect to eventually migrate to EVS as described in [RFC36].

Because messages are written under the control of an arbitrary format string, the data within the message is, essentially, opaque. This makes it very difficult to do any pre-generation filtering (as opposed to *post-mortem* data reduction tools such as `grep`). A publicized hook is available so that licensees can add whatever run-time filtering they wish.

It must be possible to specify the disposition of messages at a fairly fine level of detail. In addition, it must be possible to do this consistently across all components, providing boot-time defaults with an override, as well as dynamic remote reconfiguration.

While we want to detect and report faults as soon as possible, we recognize that this can have performance implications. For example, ''before-the-fact'' data capture is not a goal for this release. Because error recovery in the simple case is difficult, and automated distributed recovery is a black art, cascading error processing and extended error processing as outlined in [RFC11] is not a goal. It is sufficient to generate an accurate, detailed message, and terminate.

There have been requests for expanded `printf` such as ''%m'' to expand to the text of a DCE message, and ''%p'' to print a pointer value. Unfortunately, our I18N goals require that the XPG/4 ''%n$'' format be supported. We do not have enough resources to provide an XPG/4 implementation that also understands additional format specifiers. A ''%b'' specifier is provided for binary log files; using this specifier on text log files will result in undefined behavior. This is a run-time check that will, perhaps unfortunately, not be enforced.

We will meet these goals by providing a new DCE serviceability component, and by requiring all existing code to use it. The functional outline and requirements it imposes are detailed below.

4. CODE POINT CRITERIA

In order to meet the DCE 1.1 serviceability goals, all providers will have to add messaging calls to their code. The following criteria should be used to determine these points:

(a) All servers must report when they are started, and when they have completed their initialization and are ready to perform work. They should also indicate when they are going off-line. These are informational messages and will use standard text to be supplied by OSF.

(b) All existing program exits must be reported as fatal errors. Similarly, all calls to `abort` must be replaced by calls to `dce_svc_printf` with the ''abort'' action.

(c) Errors which make it impossible to proceed must be detected as close as possible to the point of failure. This includes such ''impossible'' conditions such as failure to allocate (virtual) memory or failure to open a configuration file for reading, or a log file for writing.

(d) Conditions which might indicate system-level malfunction or poor performance must be reported. An example would be if the RPC runtime detects that it is making an excessive number of retransmits.

(e) Routine administrative actions should be reported as informational messages. This includes: creation, modification and deletion of tickets, threads, files, sockets, RPC endpoints, or other objects; message transfer, including name lookup, binding, and forwarding; directory maintenance, including caching, advertisting, skulking, and replication; and database maintenance, including replication or synchronization.

(f) It must be possible to determine that the system is operating reliably even if it is idle. Periodic ''ping'' messages are too expensive. We suggest a value-added program that collects and displays the statistics collected from the remote management interface (see below).

(g) It is *not* necessary for run-time programming errors to be detected and caught, as this could add unacceptable size and performance penalties. For example, internal subroutines need not validate their input parameters. Many routines in DCE are only called by other subroutines, and none of their input parameters is an arbitrary text string that would be entered by a user. These routines are not required to validate their input. DCE engineers might use the `dce_assert` macro (to be provided by OSF) to test for common obvious errors (such as an illegal null pointer); using assertions means that these checks can easily be removed from production code. Routines that are not part of the exposed API fall into this category and need not have their input validated.

(h) On the other hand, many other DCE routines process a string that could have been entered from a keyboard — directory name lookups are an obvious example. These routines must fail gracefully, and not core dump, for example, if given a syntactically incorrect path. Note that many components already address this to some extent; existing code should be converted to use the new service component and not removed.

It is not possible to use these criteria and assign an absolute ''serviceability value'' to some software. Nevertheless, we expect the providers to use them while making a best effort. At a minimum, if all existing message points have been converted, all C library routine failures have been captured, and all program exits and administrative requests have been captured, then the requirements have been meet. Additional guidelines and macros may be made available during DCE 1.1 development to help make the conversion easier.

5. FUNCTIONAL DEFINITION

We partition DCE into pieces known as *components*. This is similar the standard use of the term within the DCE development group, but at a finer grain. The DCE library is divided into the following components (name given in parentheses):

(a) DCE utilities library [RFC17] (`lib`).

(b) Directory/Naming (`cds`, `gds`, etc.).

(c) Filesystem (`dfs`).

(d) RPC runtime (`rpc`).

(e) Security runtime (`sec`).

(f) Threads (`thr`).

Component names are limited to three characters chosen from the DCE status code ''rad40'' alphabet.[1] The names of all components are set by OSF, and are not subject to I18N requirements. Note that not all parts of the DCE API have serviceability requirements. For example, there seems to be little need to include the `utc_*()` functions. It is also possible that providers may identify additional components not listed above; this would be handled as described in the next paragraph.

Some executables may share large portions of code that is in a private library and not part of the general DCE library. Providers can create a new component to capture this common code, after consultation with OSF to determine the name and scope of the component. For example, `cdsd` and `cdsclerk` might share a `cut` (''CDS utilities'') component.

In addition to the library, each server executable has at least one component.

Component-level logging does not provide enough serviceability control. Components are divided into *sub-components*. Each sub-component corresponds to an architecturally-distinct part of the component. The number of

1. Source licensees can refer to the implementation of `dce_error_inq_text()` for the definition of ''rad40''.

sub-components will vary greatly by component, and will be determined by the provider. For example, threads might have a ''mutex'' sub-component and a ''general'' sub-component, and little else. The RPC runtime, on the other hand, currently uses a 28-element debug table, where each entry would become a sub-component. Most components should fall between these two endpoints.

The same sub-component name may appear in more than one component — for example, every component should have a catch-all sub-component named ''general''. The sub-component should be a single-word mnemonic chosen by the provider. It is not subject to I18N requirements. Each sub-component also has a (short, one-line) description that *is* stored in a message catalog. OSF will also review all the sub-components that are defined, and determine a standard name if any overlap is found.

Each message is uniquely part of a sub-component within a component.

Each message has a *severity level* attribute. The level is determined when the code-point is created and must be chosen from the following list (the manifest constant to be used is shown in parentheses):

(a) Fatal error exit (`svc_c_sev_fatal_error`). An unrecoverable error has occurred requiring special manual recovery actions to take place, such as database restoration. The program usually terminates immediately.

(b) Error detected (`svc_c_sev_error`). An unexpected event that is non-terminal or is correctable via human intervention has occurred, such as timeouts. The program continues although some functions or services may not be available. This may also be used to indicate that a particular request or action could not be completed.

(c) Correctable error (`svc_c_sev_warning`). An error occurred that was automatically corrected, such as a configuration file was not found so that defaults were used. This may also be used to indicate a condition that may be an error if the effects are undesireable, such as removing all files when a non-empty directory is removed. This may also be used to indication a condition that if not corrected will eventually result in an error, such as when a printer is running out of paper.

(d) Informational notice (`svc_c_sev_notice`). A predetermined major event has occurred, such as a server started.

(e) Verbose information notice (`svc_c_sev_notice_verbose`). A predetermined event has occurred, such as a directory entry was removed.

(f) Debug level 1 (`svc_c_debug1`) through Debug level 9 (`svc_c_debug9`). Messages in the nine Debug levels would not normally appear in production code. They are used to capture the current `#ifdef DEBUG` messages found in the DCE sources. There is also a `svc_c_debug_off` which means that no logging should be done for a sub-component.

In addition, some levels will be reserved for licensee-value-added use.

Within a given executable, messages can be routed according to their severity level. The reference implementation will only route to local files. This reasonably addresses end-user needs: ''send all fatal error messages to the console''.

Debug-level messages in an executable can be directed at the component level. For example, ''send all RPC runtime debugging messages to `/opt/dcelocal/adm/secd/foo` and all CMA debugging messages to `/opt/dcelocal/adm/secd/bar`''. This is an important capability for DCE developers and licensees.

It is also possible to vary the amount of debug information that is generated. This is done by attaching an ''active level'' to each sub-component. If a message is generated with a higher Debug Level than the current active level of the sub-component, then no output is generated.

6. DATA STRUCTURES AND DEVELOPER'S API

The following `typedef` defines a ''sub-component'':

```
typedef struct dce_svc_subcomp_s_t {
    char        *sc_name;        /* Sub-component name */
    char        *sc_descr;       /* Sub-component description */
    unsigned32  sc_descr_msgid; /* Message ID of sc_descr */
    unsigned32  sc_level;        /* Active level for the
                                    sub-component */
} dce_svc_subcomp_t;
```

The `sc_descr_msgid` field is used to translate the `sc_descr` field of the sub-component. It may be zero to indicate that the message should not be translated.

These sub-components are collected into the serviceability table. By convention this is named *comp*__svc_table, where *comp* is the name of the component. The table is expected to be globally known throughout the component. For example (where `NNN` is the message code):

```
dce_svc_subcomp_t thr__svc_table[] = {
    { "general", "General threads tracing", NNN,
      svc_c_debug_off },
    { "mutex", "Trace mutex calls", NNN,
      svc_c_debug_off },
    { NULL }
};

#define thr_svc_general 0
#define thr_svc_mutex   1
```

`Sams`, the message-text tool mentioned in [RFC23], will be able to generate this table from a higher-level specification.

All components will be required to register their table with the serviceability facility using the following function:

```
dce_svc_handle_t
dce_svc_register (
    dce_svc_subcomp_t *table,
    const idl_char    *component_name,
    error_status_t    *st );
```

For example:

```
dce_svc_handle_t thr__svc_handle;

thr__svc_handle = dce_svc_register(thr__svc_table, "thr", &st);
```

A compile-time macro is also available to create a global handle variable from a seviceability table:

```
DCE_SVC_DEFINE_HANDLE(handle, table, component_name);
```

A companion call, `dce_svc_unregister`, can be called to free the resources associated with a handle. It is normally not necessary to call this routine as the regular process exit handling will, *e.g.*, close all open files.

Once the table has been registered, the following routine is used for all service-related messages:

```
void
dce_svc_printf(
    DCE_SVC(handle, print_args),
    unsigned32 table_index,
    unsigned32 attributes,
    unsigned32 message_index,
    ... );
```

Note that no status code is returned. The `table_index` specifies the offset into the service table associated with the registered `handle` parameter. `Sams`, can be used to create ‘‘boilerplate’’ macros that expand to the initial fixed argument list.

`DCE_SVC` is an opaque macro that provides initial arguments. Its first argument is the registered handle; the second describes the print arguments. The expansion of this macro is variable; for example, it might expand to the text string ‘‘`handle, __FILE__, __LINE__`’’. The ‘‘`...`’’ arguments indicate that this is a `printf()`-style routine. A sample call would be:

```
dce_svc_printf(DCE_SVC(thr__svc_handle, "%d"), thr_svc_mutex,
    svc_c_sev_notice, thr_s_created_mutex, &mutex);
```

This could generate a message like the following (terminal ‘‘\’’ denotes line-continuation/screen-wraparound that would normally appear as a single line on output):

```
1992-2-10-17:01:00+05:00 NOTICE thr mutex 0x10430000 \
Created mutex 0x8088be62
```

Additional action can also be performed by OR’ing in any of the following with the ‘‘attributes’’ parameter:

(a) Abort program (`svc_c_action_abort`).

(b) Exit with failure status (`svc_c_action_exit_bad`).

(c) Exit with succesful status (`svc_c_action_exit_ok`).

(d) Do not print standard prolog (`svc_c_action_brief`).

In addition specialized routing can be used by OR’ing in any of the following:

(a) Do not send message to log (`svc_c_route_nolog`).

(b) Send message to standard error (`svc_c_route_stderr`).

For example, the following will log an error message, display it to the user, and then exit:

```
dce_svc_printf(DCE_SVC(thr__svc_handle, "%ld"), thr_svc_create,
    svc_c_sev_fatal_error | svc_c_route_stderr
        | svc_c_action_exit_bad,
    /* thr_s_no_malloc: "can't allocate %ld bytes for stack" */
    thr_s_no_malloc, size);
```

```
1992-2-10-17:01:02+05:00 FATAL thr create 0x10430000 \
can't allocate 102400 bytes for stack
```

For debugging messages, the following two macros can also be used:

```
DCE_SVC_DEBUG((handle, table_index, debug_level, format, ...));
DCE_SVC_LOG((handle, table_index, debug_level, message_index,
    arg, ...));
```

These macros can turn into null code in a production environment; in a debugging environment they decide if the message should be generated. Because the C pre-processor does not allow macros with varying numbers of arguments, the ‘‘double parentheses trick’’ must be used here. The first version takes a text string, while the second resembles `dce_svc_printf()` in that it takes a message ID and a format specifier. Note that `dce_svc_printf()` can also be used for debugging statements.

It is very important that debug messages not be used for errors that can occur during ordinary operation. For example, if the RPC runtime receives a corrupted packet this should be reported as a correctable error.

It is possible to add per-component filtering. This is done using the following data types and API:

```
typedef struct dce_svc_prolog_s_t {
    dce_svc_handle_t    handle;
    int                 version;
    utc_t               t;
    const char          *argtypes;
    unsigned32          table_index;
    unsigned32          attributes;
    unsigned32          message_index;
    const char          *format;
    char                *file;
    int                 line;
} *dce_svc_prolog_t;

/* Return true or false if message should be output or not. */
typedef boolean (*svc_filter_proc_t) (
    const dce_svc_prolog_t prolog,
    va_arg                 args );

/* External filter routine; parses byte string according to its
   own semantics, used to modify the filter_proc behavior. */
typedef void (*svc_filterctl_proc_t) (
    idl_long_int   arg_size,
    unsigned char  *arg,
    error_status_t *status );

/* Attach a filter and a filter-control to a component. */
void
dce_svc_define_filter (
    svc_handle_t         *handle,
    svc_filter_proc_t    filter_function,
    svc_filterctl_proc_t filter_ctl_function,
    error_status_t       *status );
```

The `filter_function` points to a function that takes a pointer to a prolog structure followed by the `printf()` format string and arguments. A structure is used so that licensee-made changes in the macro need not affect filtering routine sources.

An additional hook is used to provide remote control of filtering by registering a control function. This function takes an array of bytes and parses it according to the licensee-supplied semantics to control the filtering.

7. MESSAGE DISPOSITION AND MANAGEMENT API

Each non-debug level can be routed separately. When a component first calls `dce_svc_register()`, a file is consulted to determine where each level is routed. Then an environment variable is consulted (see below).

In addition to these initialization mechanisms, the following routine can be called to specify where messages get routed:

```
void
dce_svc_routing (
    const char     *where,
    error_status_t *st );
```

The `where` parameter is divided into three parts, separated by colons: the level, a routing identifier and a routing parameter. (A single parameter is used, rather than three, because it will typically come from a single command-line flag or an environment variable.) OSF will implement at least the routing identifier `FILE`, where the parameter specifies the filename, as in `FILE:/tmp/log`. Licensees can provided value-added features such as `IPC:syslog` to indicate that the local `syslog` daemon should be used. A full specification would be

`FATAL:FILE:/dev/console`. For this routine only, if the first part is omitted, then the routing is set for all levels.

All components will have a specified environment variable that can be used as the default `where` parameter. The variable will be named `SVC_`*`COMP`*, where *`COMP`* is the name of the component in all uppercase. All server executables should have a standard flag — we propose `-w` — that can be specified multiple times to set multiple routine requirements. For example:

```
cdsd -wFATAL:FILE:/dev/console -wNOTICE:FILE:/tmp/log
```

Debug-level messages are routed using the following routines:

```
void
dce_svc_debug_routing (
    const char     *where,
    error_status_t *st );

void
dce_svc_debug_set_levels (
    const char     *flags,
    error_status_t *st );
```

The first routine sets both the debugging levels and the routing. The second routine sets only the levels, typically after the routing has been set. These routines are typically used to process command-line flags.

Debug flags are specified as a comma-separated list of sub-component name and numeric level separated by a period. An asterisk indicates all sub-components. The settings are parsed in order, so `*.1,cnauth.9` could be used to obtain minimal debugging within the RPC component, with extensive detail for the AUTH/CN sub-component. Again, after the config file an environment variable `SVC_`*`COMP`*`_DBG`, where *`COMP`* is the name of the component, specifies the level and disposition of debugging messages.

The remote interface (described below) will call these routines to dynamically change message disposition.

8. REMOTE INTERFACES

All servers will be required to export the OSF-written management routines that allow for queries and redirection of messages. If GDS is not converted to use threads then GDS will not be required to support the remote interface. A reasonable alternative would be to have an external program that exports the interface and forwards all requests to GDS using appropriate IPC.

All components will be under a single ACL that must be implemented by the manager. If a server already has the concept of a management ACL, then that can be used to control this interface; otherwise OSF will provide a default ACL manager that servers can use.

Note that this puts an additional requirement on all servers: each server must change to its own working directory upon start-up. (Related servers such as `cds` and `cdsclerk` can share a directory.) The default directory should be something like `/opt/dcelocal/var/server`.

The interface will include the following:

```
[
    uuid(000cf72e-0688-1acb-97ad-08002b12b8f8),
    version(1),
    pointer_default(ptr)
]

interface serviceability
{
```

```
typedef [ref, string] char *dce_svc_string_t;

typedef struct dce_svc_stringarray_s_t {
        long                    tab_size;
    [size_is(tab_size)]
        dce_svc_string_t        *table;
} dce_svc_stringarray_t;

typedef struct dce_svc_subcomparray_s_t {
        long                    tab_size;
    [size_is(tab_size)]
        dce_svc_subcomp_t       *table;
} dce_svc_subcomparray_t;


/*
** Calls dce_svc_routing() to actually set the routing.
*/
void dce_svc_set_route (
    [in]        handle_t        handle,
    [in, string]
                char            *where,
    [out]       error_status_t  *status );

/*
**  Calls dce_svc_debug_routing() to actually set the routing.
*/
void dce_svc_set_dbg_route (
    [in]        handle_t                handle,
    [in, string]
                char                    *where,
    [out]       error_status_t          *status );

/*
**  Calls dce_svc_debug_levels() to actually set the levels.
*/
[idempotent] void dce_svc_set_dbg_levels (
    [in]        handle_t                handle,
    [in, string]
                char                    *flags,
    [out]       error_status_t          *status );

/*
** Returns an array of the names of all components that have
** been registered with the dce_svc_register() routine.
*/
[idempotent] void dce_svc_inq_components (
    [in]        handle_t                handle,
    [out]       dce_svc_stringarray_t   *table,
    [out]       error_status_t          *status );
```

```
/*
** Returns the table registered with the indicated component
** name.
*/
[idempotent] void dce_svc_inq_table (
    [in]        handle_t                handle,
    [in]        dce_svc_string_t        component,
    [out]       dce_svc_subcomparray_t  *table,
    [out]       error_status_t          *status );

/*
** Calls the filter control routine for the component.
*/
void dce_svc_filter_control (
    [in]        handle_t                handle,
    [in]        dce_svc_string_t        component,
    [in]        long                    arg_size,
    [in, size_is(arg_size)]
                byte                    *argument,
    [out]       error_status_t          *status );

/*
**  Operating statistics.  Mostly a place-holder.
*/

/*
**  The "unknown" statistics.
*/
const long dce_svc_stats_unknown = 0;
typedef struct dce_svc_stats_unknown_s_t {
        long        data_size;
    [size_is(data_size)]
        byte        *data;
} dce_svc_stats_unknown_t;

/*
**  A union of all possible statistics.
*/
typedef union switch (long stats_type) dce_svc_stats_u_t {
    case dce_svc_stats_unknown:
        dce_svc_stats_unknown_t stats_unknown;
} dce_svc_stats_t;

/*
**  Return some operating statistics.
*/
void dce_svc_inq_stats(
    [in]        handle_t                handle,
    [out]       dce_svc_stats_t         *stats,
    [out]       error_status_t          *status
);

}
```

Control programs should probably be enhanced to access this interface, although this is not a requirement. The DCE administrative shell [RFC30] will probably have commands to access this interface.

Using this API, a new command, `dcesvc`, will be provided. It can be used to remotely get and set the disposition of all messages for any DCE server. Typical uses are as follows:

(a) List all components in an executable:
`dcesvc <entity>`

(b) List the sub-components within the threads component:
`dcesvc -l <entity> thr`

(c) Set the threads ‘‘mutex’’ sub-component to Debug Level 9:
`dcesvc -d <entity> thr:mutex.9`

(d) Send all fatal errors to the console:
`dcesvc -w <entity> FATAL:FILE:/dev/console`

Entities are names within the namespace. All servers will be required to export a UUID and enter it into an OSF-specified place within the namespace. This UUID must have an ACL attached to it.

APPENDIX A. PROVIDER CHECK-LIST

The following list is intended as a guide for providers so that they can estimate the amount of work that will be involved in converting their offering to meet the requirements described here:

(a) Determine if any new components need to be created; work with OSF to identify and name them and determine their scope.

(b) Divide the source into sub-components, and create the service component table.

(c) Review all debugging, diagnostic, and logging messages. Assign them to a sub-component and give them the appropriate level.

(d) Create the message catalog and convert all text strings to use it. Add explanatory text to be used in the Problem Determination Guide.

(e) Add new messaging code points using the criteria described above.

(f) Add command-line support for routing messages, using the flags suggested above.

(g) Export a UUID for the remote interface and make sure it has an ACL.

APPENDIX B. SAMPLE CODE

The following library routine is intended to show how the various parts of the serviceability component are used. The `mem_s_*` values are message IDs defined elsewhere.

```
#include <dce.h>

dce_svc_subcomp_t mem_svc_table[] = {
    { "general", "General memory use", 0, svc_c_debug_off },
    { "alloc", "Memory allocations", 0, svc_c_debug_off },
    { "free", "Memory de-allocations", 0, svc_c_debug_off },
    { "copy", "String copying", 0, svc_c_debug_off },
    { NULL }
};

#define mem_svc_general 0
#define mem_svc_alloc   1
#define mem_svc_free    2

dce_svc_handle_t mem__svc_handle;
```

```
/*
**  Initialization routine.
*/
init(void)
{
    error_status_t      st;

    mem__svc_handle = dce_svc_register(mem__svc_table, "mem",
                              &st);
}

/*
**  Allocate memory.
*/
void *
allocate(int bytes)
{
    char        *new;

    DCE_SVC_DEBUG((mem_svc_table, mem_svc_alloc,
        svc_c_debug2, "Allocating %d bytes", bytes));
    if ((new = malloc(bytes)) == NULL) {
        dce_svc_printf(DCE_SVC(mem__svc_handle, "%d"),
            mem_svc_alloc, svc_c_sev_fatal_error,
            mem_s_noalloc, strlen(p) + 1);
        exit(1);
    }
    DCE_SVC_DEBUG((mem__svc_handle, mem_svc_alloc,
        svc_c_debug1, "Allocated %d bytes at 0x%x", bytes,
        new));
    return new;
}

/*
**  Copy a string to allocated space.
*/
char *
strcopy(char *p)
{
    char        *new;

    DCE_SVC_DEBUG((mem__svc_handle, mem_svc_alloc,
        svc_c_debug1, "Copying string |%s|", p));
    if ((new = allocate(strlen(p) + 1)) == NULL) {
        dce_svc_printf(DCE_SVC(mem__svc_handle, "%d"),
            mem_svc_alloc, svc_c_sev_fatal_error,
            mem_s_noalloc, strlen(p) + 1);
        exit(1);
    }
    return strcpy(new, p);
}
```

```
/*
**  Free memory.
*/
void
unallocate(void *p)
{
    if (p == NULL) {
        dce_svc_printf(DCE_SVC(mem__svc_handle, NULL),
            mem_svc_free, svc_c_sev_warning, mem_s_free_null);
        return;
    }
    DCE_SVC_DEBUG((mem__svc_handle, mem_svc_alloc,
        svc_c_debug1, "Freeing memory at 0x%x", p));
    free(p);
}
```

REFERENCES

[Ogura] T. Ogura, ‘‘DCE 1.1 I18N Workbook’’, DCE Document, September 1992.

[RFC11] M. Hubbard, DCE-RFC 11.0, ‘‘DCE SIG Serviceability Requirements’’, August 1992.

[RFC17] H. Melman, DCE-RFC 17.0, ‘‘DCE Code Clean-Up Requirements’’, December, 1992.

[RFC23] D. Mackey, DCE-RFC 23.0, ‘‘DCE 1.1 Internationalization Guide’’, February, 1993.

[RFC24] R. Salz, DCE-RFC 24.0, ‘‘DCE 1.1 Serviceability Proposal’’, November 1992.

[RFC30] H. Melman, DCE-RFC 30.0, ‘‘DCE 1.1 Administrative Improvements: Requirements for DCE Shell’’, December, 1992.

[RFC34] H. Melman, DCE-RFC 34.0, ‘‘DCE 1.1 Code Cleanup Requirements’’, April, 1993.

[RFC36] R. Salz, DCE-RFC 36.0, ‘‘Migrating DCE 1.1 Serviceability to EVS’’, January, 1993.

[XPG4] *X/Open XPG4 Portability Guide*, message catalog routines.

AUTHOR’S ADDRESS

Rich Salz
Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142
USA

Internet email: rsalz@osf.org
Telephone: +1-617-621-7253