Copyright ©1994 Christopher J. Kane. Version 1.0.

Notes on the Miscdaemon routines

Module Description

The Miscdaemon C module provides some basic facilities useful to nearly any daemon or daemon-like application. These facilities include: global variables for static and interesting values, memory allocation checking functions, message logging, daemon assertions, and, perhaps most importantly, a mechanism to transform a process into a daemon. The function and variable descriptions give the details needed to use the routines; understanding how they work should only be necessary if you want to do something fancy (or tricky).

You'll note that this is not a class, nor a category, nor anything Objective-C, but just pure C code. To me, at least, it seems to be a tricky problem thinking about how this functionality (and code) should be integrated into the Objective-C class hierarchy. A daemon doesn't seem to be an object in its own right, such as is a tree or a chair, but rather it is a "flavor" of process. The best place would seem to be a category of Application. A method call such as

[NXApp **log:**LOG_INFO **message:**"opening file %s", filename];

seems quite natural. But what if a program doesn't link with the AppKit, and uses the Listener **-run** method, or one of the NXConnection **-run** methods instead? So, perhaps a subclass of Object...but this doesn't fit well into the class hierarchy either, as I mentioned earlier.

So I have opted to not "classify" the functionality provided by these routines, and leave them as C code. A developer should be able to simply drop the two files **Miscdaemon.h** and **Miscdaemon.c** into a project, and gain the utility. But, additionally, the debug flag should also be defined somewhere in a project. If you don't want to use it, add a line to the top of **Miscdaemon.c** declaring it there. And a call to **daemonize**() should be added to the project's **main**() routine, as one of the first things it does.

A typical call to daemonize(), the most complex of the functions, might look like this:

```
daemonize("myDaemon", "/usr/adm/myDaemon.log", LOG_DAEMON,
                "/", "/usr/adm/", NULL, SIGHUP, NULL);
```

The **safe_dir** need not be a directory that the daemon can write it, it's just the directory that should be the daemon's current working directory. Since "/" always exists, it is a typical choice. The lock file, which in this example will be "/usr/adm/myDaemon.pid", prevents more than one copy of the daemon from running at a time. The **ignfds** and **ignsigs** are typically NULL to provide default behavior; since **daemonize**() will usually happen as (nearly) the first thing done in a program, there won't usually be file descriptors that you wish to protect (except perhaps the stdio descriptors).

Imported Variables

debug

int **debug**

This is a global variable which must be defined outside this module. It is used to modify the behavior of the code in the module, depending on whether or not debugging behavior is desired. Note that this debugging of the application, not this module. Note also that this flag can be set and unset at runtime to dynamically switch between debugging and non-debugging modes. See the descriptions of the individual functions for the behavioral differences when debugging is on or off.
The use of an external flag/variable for this functionality shows questionable judgement.

Exported Variables

daemon_host
char ***daemon_host**

Points to, after the call to **daemonize**(), the string name of the local host.

daemon_name
char ***daemon_name**

Points to, after the call to **daemonize**(), the string name of the daemon, as given to **daemonize**() as its first parameter.

daemon_pid
int **daemon_pid**

Contains, after the call to **daemonize**(), the numerical ID of the daemon process.

Exported Functions

daemonize
void **daemonize**(char ***prog_name**, /* name for the daemon to be used in the log */

```
char *log_fn,       /* file name for the log file, or special string */
int log_facility,   /* syslog facility constant */
char *safe_dir,     /* directory where the daemon can put a core file */
char *lockf_dir,    /* directory in which to put the lock file */
fd_set *ignfds,     /* an fd_set indicating file descriptors to ignore */
int loghup_sig,     /* a signal constant */
int *ignsigs);      /* pointer to array of signal constants to ignore */
```

Transforms the process into a typical daemon, returning nothing. Because of its actions, this function should be one of the first things called in a daemon's **main**() routine, and it may only be called once. **prog_name** may be empty or NULL; if NULL, the default name "daemon" is used. If **safe_dir** is NULL, the daemon's current directory isn't changed.

The value of **log_fn** indicates the type of message logging that is desired. There are four options:

1. No logging (pass **log_fn**==NULL)
2. Log to syslog (pass **log_fn**=="syslog")
3. Log to stderr (pass **log_fn**=="stderr")

4. Log to a file (pass **log_fn**==any other string)

In the case of option 4, the string is interpreted as the path and name for the log file. The log file will be created if necessary, and appended to. Note that, if **log_fn** is NULL but the **debug** flag is set when this function is called, then the log file will be stderr.

The **lockf_dir** parameter specifies the directory in which to store the daemon's lock file. The lock file is given the name "%s.pid", where the string passed in **prog_name** is substituted for "%s". The process id of the daemon is written to the lock file. If **lockf_dir** is NULL, or the **debug** flag is set, no lock file is created. If the lock file already exists or cannot be created, the daemon exits.

Normally, **daemonize**() closes all file descriptors except stdin, stdout, and stderr (and possibly a file descriptor for the log file). stdin and stdout are pointed at /dev/null. If stderr is not the log file, it is also pointed at /dev/null, the process is placed in its own process group, and it is disconnected from its controlling TTY. The daemon's file creation mask is set to 033.

This behavior can be modified with the **ignfds** fd_set. File descriptors for which the corresponding element of ignfds is set are ignored±files that are open remain open, files that

are closed remain closed. The elements for the stdin, stdout, and stderr files may also be set to leave them alone. If the **log_fn** parameter is "stderr", stderr is automatically ignored; you do not need to set its element in the fd_set. A NULL **ignfds** parameter will provide the default behavior.

Finally, some signal handlers are set, and some signals are explicitly set to be ignored by the daemon. Generally, if there is already a (non-default) signal handler set for a signal, it is left unmolested. There exception to this is SIGTERM. SIGTERM is always set to an internal signal handler that cleans up and shuts the daemon down (via **daemon_exit**()). The **loghup_sig** specifies the signal which will activate another internal signal handler that closes and reopens the log file, if the log file is an ordinary file, so that the file may be trimmed. This parameter may be zero if this functionality is not desired. Also, if no handler has been set before the call to **daemonize**(), SIGCHLD is set to a handler that reaps the children of the daemon that have terminated.

Signals may be set to be ignored with the **ignsigs** parameter. If it is NULL, a default set of signals (SIGINT, SIGPIPE, SIGALRM, SIGTSTP, SIGTTIN, SIGTTOU, SIGVTALRM, SIGPROF, SIGUSR1, and SIGUSR2) are ignored, unless the **debug** flag is set when this

function is called. Otherwise, the parameter is assumed to be a variable-length array of signal constants, with the last element having the sentinel value zero, which are then ignored if no handler has already been set.

daemon_alloc

void ***daemon_alloc**(int **size**)

This function, along with **daemon_free**(), provide a simple dynamic memory allocation checking facility to a daemon. **size** is the number of bytes to allocate, and a pointer to the allocated, zeroed block (of at least **size** bytes) is returned. If the allocation fails, the program is aborted with logged messages. When the **debug** flag is set, a message is logged to the daemon log noting the allocation and the **size**, returned pointer value, and the source file and line where the allocation occurred. When used with the output from **daemon_free**() (with **debug** set), this information can be used to track down memory leaks.

daemon_exit

void **daemon_exit**(int **exit_val**)

Does whatever cleanup is necessary, and terminates the daemon. **exit_val** is the exit value for the daemon, which, unless the **debug** flag was set when **daemonize**() was called, is inaccessible to other processes (due to the "disconnection" that daemonize does). In the current implementation, a message is logged to the daemon log, the lock file (if it was created) is deleted, the syslog file is closed, and **exit**() called.

daemon_free

void ***daemon_free**(void ***block**)

This function, along with **daemon_alloc**(), provide a simple dynamic memory allocation checking facility to a daemon. **block** is a pointer to a block of memory previously allocated with **daemon_alloc**(). The function always returns NULL. When the **debug** flag is set, a message is logged to the daemon log noting the deallocation, pointer value, and the source file and line where the deallocation occurred. When used with the output from **daemon_alloc**() (with **debug** set), this information can be used to track down memory leaks.

daemon_log

void **daemon_log**(int **prio**, char ***msg**, ...)

Provides a message logging facility to a daemon. **prio** is one of the syslog priority constants (such as LOG_ERR, LOG_INFO, etc.), **msg** is a **printf**()-style format string, and a variable number of arguments supply the values for the message string. Messages with priority LOG_DEBUG are not logged if the **debug** flag is not set. Messages with a priority lower than LOG_DEBUG (greater integer value) are never logged. Because **daemonize**() sets up the log file, this function is not guaranteed to work until after **daemonize**() has been called. Note, too, that no newline should be included at the end of the message format string; that is taken care of in the function.

The length limit on the message, after "format string expansion", is about 4K. If this limit is exceeded, some memory has probably been clobbered, and the daemon will (probably) terminate. Termination is not guaranteed, however, since the most likely bits to get clobbered in the current implementation are this function's call stack area, and the stack area of

vsprintf() (called to do the "expansion"). However, 4K should be ample for every need.

Exported Macros

daemon_assert
void **daemon_assert**(EXPR)

A debugging assert macro for daemons. **EXPR** is the expression to evaluate; no value is returned. If the expression evaluates to false, a message is logged to the daemon log and the daemon exits. Note that the code of this macro is always compiled into an executable, but the expression is only evaluated if the debug flag is set. This allows debugging assertions to be turned on and off at runtime.