IoT: Client Devices

A Sample Daemon

```
51
52 int main(void) {
53   // Open syslog. We want to write directly to the log with
54   // no wait, and we want to include the PID of the daemon process.
55   // We are running a daemon, so set the LOG_DAEOMON flag too.
56   //
57   // Why open here? well, if we have an error forking, we want to
58   // register the error to syslog. To do that, we need an open log.
59   // The child process will inherit this log, as we don't close it
60   // in the parent when it exits, this opening here is A+.
61   openlog(DAEMON_NAME, LOG_PID | LOG_NDELAY | LOG_NOWAIT, LOG_DAEMON);
62   syslog(LOG_INFO, "staring sampled");
63
64   // We really don't want to take over syslogd or initd, so
65   // fork.
66   pid_t pid = fork();
67
68   // Well something went wrong. fork() uses standard unix
69   // errno functionality, so let's log the problem and exit.
70   if (pid < 0) {
NORMAL  master  ./sample_daemon.c        unix < utf-8 < c  58%  70:1
```

Getting Started

```
64   // We really don't want to take over syslogd or initd, so
65   // fork.
66   pid_t pid = fork();
67
68   // Well something went wrong. fork() uses standard unix
69   // errno functionality, so let's log the problem and exit.
70   if (pid < 0) {
71     syslog(LOG_ERR, ERROR_FORMAT, strerror(errno));
72     return ERR_FORK;
73   }
74
75   // We receive a PID if we're the parent process. If we're then
76   // parent, let's just exit. We only care about the forked child.
77   if (pid > 0) {
78     return OK;
79   }
80
81   // I'd like to the the leader of the session. If I can't,
82   // I'm out.
83   if(setsid() < -1) {
NORMAL  master  ./sample daemon.c        unix < utf-8 < c  69%  83:1
```

Forking

```
80
81    // I'd like to the the leader of the session. If I can't,
82    // I'm out.
83    if(setsid() < -1) {
84      syslog(LOG_ERR, ERROR_FORMAT, strerror(errno));
85      return ERR_SETSID;
86    }
87
88    // We have console to write to anymore, so these file pointers
89    // are just silly. Close them.
90    close(STDIN_FILENO);
91    close(STDOUT_FILENO);
92    close(STDERR_FILENO);
93
94    // Are we creating files? Well, not in this example, but let's
95    // go ahead and set a sane UMASK anyway. This will give us
96    // read/write, and everybody else gets read permissions.
97    umask(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
98
99    // What's our working directory going to be? Let's make it the root
NORMAL  master  ./sample_daemon.c        unix < utf-8 < c  66%  80:1
```

Sessions and Output

```
 97   umask(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
 98
 99   // What's our working directory going to be? Let's make it the root
100   // directory just to make navigation easier if we needed it.
101   if (chdir("/") < 0) {
102     syslog(LOG_ERR, ERROR_FORMAT, strerror(errno));
103     return ERR_CHDIR;
104   }
105
106   // Now, we can only be controlled by signals as we have no interactive
107   // user session. So let's handle those signals, yeah?
108   signal(SIGTERM, _signal_handler);
109   signal(SIGHUP, _signal_handler);
110
111   // This is the daemon proces runloop. This could be a while statement,
112   // or some other construct, but daemons have specific responsibilities
113   // they need to handle, usually in a repeated way, and all that log
114   // would go here.
115   _do_work();
116
NORMAL  master  ./sample_daemon.c        unix < utf-8 < c  89%  108:1
```

Working Directories and Signals

```
25 } error_t;
26
27 // This is a signal handler. Signal is the signal passed to
28 // us to handle.
29 static void _signal_handler(const int signal) {
30   switch(signal) {
31     case SIGHUP:
32       break;
33     case SIGTERM:
34       syslog(LOG_INFO, "received SIGTERM, exiting.");
35       closelog();
36       exit(OK);
37       break;
38     default:
39       syslog(LOG_INFO, "received unhandled signal");
40   }
41 }
42
43 // This is where we do the work of the daemon. This example
44 // just counts and sleeps. Impressive!
```


Signal Handling

```
102     syslog(LOG_ERR, ERROR_FORMAT, strerror(errno));
103     return ERR_CHDIR;
104   }
105
106   // Now, we can only be controlled by signals as we have no interactive
107   // user session. So let's handle those signals, yeah?
108   signal(SIGTERM, _signal_handler);
109   signal(SIGHUP, _signal_handler);
110
111   // This is the daemon proces runloop. This could be a while statement,
112   // or some other construct, but daemons have specific responsibilities
113   // they need to handle, usually in a repeated way, and all that log
114   // would go here.
115   _do_work();
116
117   // This should actually never be reached. We should be looping in
118   // _do_work() until the daemon is killed by a signal, at which
119   // point we exit the process.
120   return ERR_WTF;
121 }
NORMAL  master  ./sample_daemon.c        unix < utf-8 < c  100%  121:1
```

Doing Actual Things!

```
38      default:
39        syslog(LOG_INFO, "received unhandled signal");
40    }
41  }
42
43  // This is where we do the work of the daemon. This example
44  // just counts and sleeps. Impressive!
45  static void _do_work(void) {
46    for(int i = 0; true; i++) {
47      syslog(LOG_INFO, "iteration: %d", i);
48      sleep(1);
49    }
50  }
51
52  int main(void) {
53    // Open syslog. We want to write directly to the log with
54    // no wait, and we want to include the PID of the daemon process.
55    // We are running a daemon, so set the LOG_DAEOMON flag too.
56    //
57    // Why open here? well, if we have an error forking, we want to
```


Run Loop