## Corpses (Darwin 15)

Although Darwin fully supports core dumps, there are several reasons why they are undesirable. Core dumps take up relatively large amounts of disk space (hundreds of megabytes or more per dump). This can be a concern, especially on i-Devices. Additionally, they may contain sensitive information (which is why there is a `sugid_coredump` `sysctl(2)` MIB). Core files also quickly grow stale, especially when created by suicidal processes which are frequently restarted. Crash reporters and debuggers often need access to a core only briefly, to perform post-mortem analysis tasks and file a "forensics reports" - after which the core file, no longer of value, becomes a nuisance.

Darwin 15 (XNU-3248) introduces the notion of **corpses**. As explained in XNU's osfmk/corpses/corpse.c, a corpse captures the process in its state post death, after all of its resources - descriptors and Mach ports - have all been released. The corpse only holds to the process memory image and runtime statistics, allowing the crash reporting frameworks to inspect the image for backtrace construction, leak detection, resource usage, and other analysis tasks. The corpse still takes up the memory of the original process, which is why the system limits the maximum number of concurrent corpses to the hard-coded value of `TOTAL_CORPSES_ALLOWED` (presently `#define`d to be 5 in task_corpse.h). Corpse functionality can also be entirely disabled via the `-no_corpses` boot-argument.

When a process is killed as a result of some exception, its exit processing logic (`proc_prepareexit` in bsd/kern/kern_exit.c) fires an `EXC_CRASH` exception message. If the exception is not handled, a corpse is automatically created and populated with `TASK_CRASHINFO_*` `kcdata` items (as defined in osfmk/kern/kcdata.h). The underlying task is marked as a `PENDING_CORPSE`, preventing it from being fully terminated. Another exception message, `EXC_CORPSE_NOTIFY` (see Listing 12-8) is sent, and the corpse is maintained until the message is handled.

The `kcdata` is a proprietary data format, described and explained in osfmk/kern/kcdata.h to be "a self-describing data serialization format". It is also used for other cases wherein the kernel needs to export data to user space, most notably stack snapshots, which have traditionally used a proprietary (and unstable) data format. Data items can be serialized not just with the standard type/length/value, but also with a textual description, enabling forward-compatibility with newer item-types, and providing clients with an ability to display items along with their semantics. The `kcdata` format is also suitable for serializing structures and container-based data.

The task MIG subsystem (from Table 12-7) has been updated as of XNU-3789 (Darwin 16) to support corpses: The `task_generate_corpse` (#3442), and the `task_map_corpse_info[_64]` calls (#3443/3450) can be used on a given task's send right, with the latter set useful for obtaining the corpse data. Apple's crash reporter (described later) can harvest the corpse information and embed in its its crash report, operating directly on the corpse data, saving it considerable time by already including much of the required crash data.

## Assisted Suicide

There are times when a program might want to voluntarily abort and create a crash dump. Traditionally, this is done with the C `assert()` macro, which takes a logical expression as an argument, and triggers a program `abort(3)` if the assertion has failed. Such assertions can also be set to `#define` to nothing in production code.

Darwin offers `_os_crash` as one of the many undocumented `os_*` exports from libSystem. The `os_crash` function (implemented in libsystem_c.dylib) allows an application to achieve similar functionality to `abort(3)` on failure. The trick is, however, that by default the function is effectively inert, and allows the program to complete execution. Callers are expected to override `os_crash_callback` first, providing their own function, which will in turn be called by `os_crash`. A good example of how Apple uses this internally can be found in `launchd(1)`, as shown in Listing 15-49:

**Listing 15-49:** `launchd(1)`'s use of the `os_crash()` facility

```
_my_crash_callback:
void my_crash_callback (char *CrashString) {
10000acf0  STP   X29, X30, [SP, #-16]!
10000acf4  ADD   X29, SP, #0      ; R29 = SP
10000acf8  BL    _will_abort_with_reason(c)       ; 0x100026910
  _will_abort_with_reason(CrashString);
 }
 ...
_main:
10000acfc  SUB   SP, SP, 288              ; SP -= 0x120 (stack frame)
   ...
 _os_crash_callback = my_crash_callback;
10000ad18  LDR   X8, #201576  ; X8 = *(10003c080) = libSystem.B.dylib::__os_crash_callback
10000ad1c  ADR   X9, #-44     ; _my_crash_callback  ; R9 = 0x10000acf0
10000ad20  NOP
10000ad24  STR   X9, [X8, #0] ; *libSystem.B.dylib::__os_crash_callback = my_crash_callback
...
 ; Example voluntary crash: failure to call proc_disable_wakemon
   if (proc_disable_wakemon() == -1) {
10000aebc  BL    libSystem.B.dylib::_proc_disable_wakemon        ; 0x10002dc30
10000aec0  CMN   W0, #1           ;
10000aec4  B.EQ  0x10000b174      ;
     _os_assert_log(* __error());
10000b174  BL    libSystem.B.dylib::___error      ; 0x10002d2c4
10000b178  LDRSW X0, [X0]         ; R0 = *(__error)()
10000b17c  BL    libSystem.B.dylib::__os_assert_log       ; 0x10002d330
     _os_crash();
10000b180  BL    libSystem.B.dylib::__os_crash    ; 0x10002d360
10000b184  HALT (self referential branch)
}
...
__will_abort_with_reason(c):
void _will_abort_with_reason(char *Reason) {
100026910  STP   X29, X30, [SP, #-16]!
100026914  ADD   X29, SP, #0      ; R29 = SP
    (void) abort_with_reason(OS_REASON_LIBXPC, // uint32_t reason_namespace
                             1,                // uint64_t reason_code
                             Reason,           // const char *reason_string
                             0);               // uint64_t reason_flags
100026918  MOV   X8, X0           ; X8 = X0 = ARG0
10002691c  ORR   W0, WZR, #0x7    ; R0 = 0x7
100026920  ORR   W1, WZR, #0x1    ; R1 = 0x1
100026924  MOV   X2, X8           ; X2 = X8 = ARG0
100026928  MOVZ  X3, 0x0          ; R3 = 0x0
10002692c  BL    libSystem.B.dylib::_abort_with_reason    ; 0x10002d3b4
}
```

Note the flow in the previous listing: When `_os_crash` is invoked at any time during program flow, `my_crash_callback` (0x10000acf0) gets called. It, in turn, goes to call a wrapper for the `abort_with_reason`, itself wrapping the `abort_with_payload` system call. The system call (#521) is a brand new one - added in Darwin 17, and allows its caller to specify any one of 21 presently defined "reason namespaces", a code and a reason, as well as flags, privately `#defined` in XNU's bsd/sys/reason.h. Listing 9-25 showed some the reason code used by Jetsam, which uses namespace #1. There are quite a few flags, but the ones allowed from userspace are masked to be `OS_REASON_FLAG_CONSISTENT_FAILURE` (0x40), `.._ONE_TIME_FAILURE` (0x80) and `.._NO_CRASH_REPORT` (0x2).

The system call is marked as `noreturn` - the calling process will be terminated with a `SIGABRT`, so in some respects this is in-line with the classic UN*X behavior. Apple no doubt will expand this mechanism to other daemons, likely enhancing its functions and further integrating it with the corpse facility.

The following listing demonstrates how the interested reader may make use of `os_crash` as well:

**Listing 15-50:** A simple example of using `os_crash`

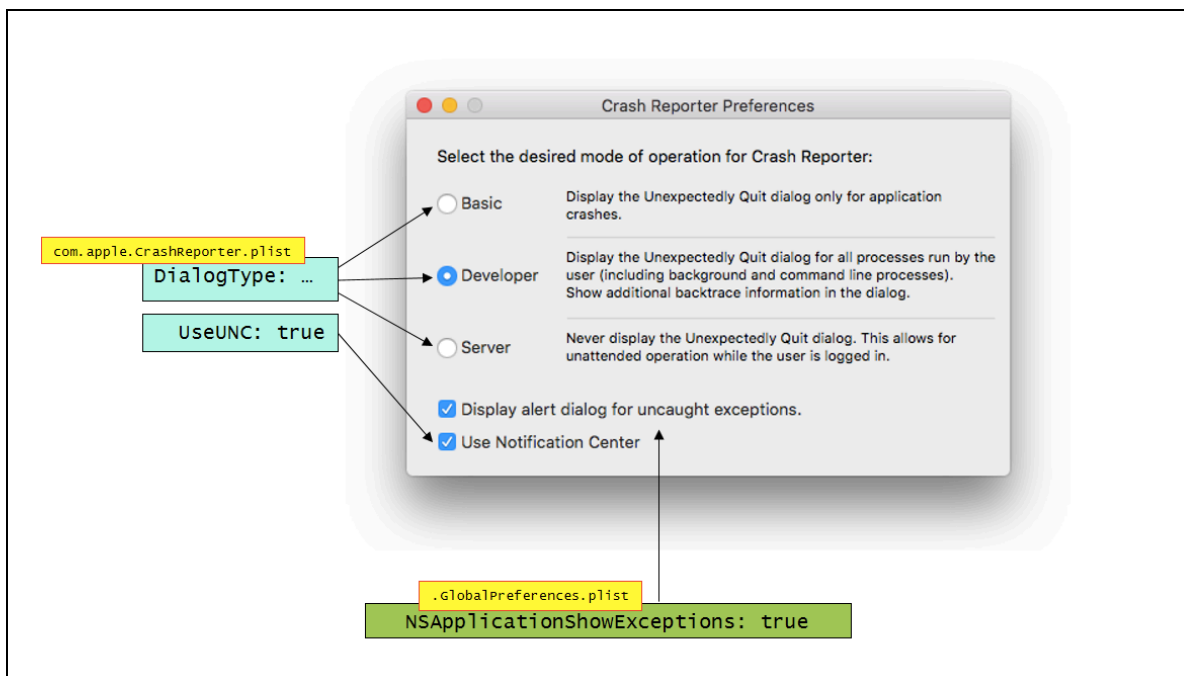
```
#include <stdio.h>

// Supply own defines here
typedef void (*crash_func)(char *);
extern void _os_crash(char *reason);
extern crash_func _os_crash_callback;

// From XNU's libsyscall/wrappers/terminate_with_reason.c
extern void abort_with_reason (uint32_t    reason_namespace,
                               uint64_t    reason_code,
                               const char *reason,
                               uint64_t    flags);

int death = 0;
void foo(char *ReasonString)
{
        printf("FOO! %s\n", ReasonString);
        if (death == 2) abort_with_reason(5,0xdead, ReasonString, 0);

}
void main (int argc, char **argv) {

   if (argc < 2) {
       fprintf(stderr,"Let's not die prematurely, eh? I need 0, 1 or 2\n");
       exit(1);}
    death = atoi(argv[1]);

    if (death) { _os_crash_callback = foo; }

    _os_crash("Oopsie?");
    printf("..I will survive..\n");

}

#if 0

  Example behaviors of running the above program:

Chimera morpheus$ ./test 0      # No callback = no crash
..I will survive..
Chimera morpheus$ ./test 1      # callback = user gets to trap, still no crash
FOO Oopsie?
..I will survive..
Chimera morpheus$ ./test 2      # callback + abort
FOO Oopsie?
Abort trap: 6

#endif
```

## Crash Reporting

As explained earlier in Chapter 12, one of Mach's most unique design features is exception handling via messages. This enables the installation of an exception handler at any one of three levels - the faulting thread, task or host, and can even be extended for handlers on remote nodes. This is used by Apple to trap program crashes, and produce Crash reports. The dedicated ReportCrash (internally, the "CrashCatcher") indicates in its Launch plist that is an `ExceptionServer`, which tells `launchd` to forward any messages on the host exception port to it. Upon receiving an exception message, ReportCrash analyzes the process (or, as of Darwin 16, process corpse), and generates a standardized crash report*.

Indications of crashes (`Path` of crashing program and `Date` of crash only) are kept in property lists in the user's $HOME/Library/Application Support/CrashReporter. The crash logs themselves are put in /Library/Logs/DiagnosticReports/*processName_timestamp_hostName*.crash. The crash reporter can be controlled by means of two per-user (~/Library/Preferences) property lists. Xcode's "Additional Tools" download provides a GUI front-end to these property lists in its Utilities/CrashReporterPrefs.app, shown in Figure 15-51:

**Figure 15-51:** XCode's Additional Tools CrashReporterPrefs




Another daemon - SubmitDiagInfo(8) picks up crash reports and submits them to Apple (if the user opted in to this feature. An interesting note on this daemon is that it can also SubmitToFolder (i.e. copy report to another local folder). This was exploited by Lokihardt to achieve an arbitrary directory modification - and an escalation of privileges (CVE-2016-1806, detailed in III/12).

Apple documents crash reporting in TN2123[9], although the crash report log format since that time - the technote details version 5 (from Darwin 9), but by Darwin 17 the report version is 12. Listing 15-52 shows a sample crash report:

---

* - Android has a similar facility is `debuggerd`, which generates "tombstones", but its implementation is different and involves locally catching the fatal signal, and sending an IPC message to the daemon.