AN02051: How to deal with errors and exceptions in deployed systems

Publication Date: 2026/7/17
Document Number: XM-015392-AN v1.0.0

IN THIS DOCUMENT

When deploying an autonomous system based on an *XCORE* it will be desirable for the system to respond to unexpected behaviour. This application note lists examples of unexpected behaviour and how to handle them either by issuing a software-initiated reset or a hardware-initiated-reset.

1 Types of run time errors

In an ideal embedded design, every scenario the system may encounter is anticipated and handled. In practice, environmental factors outside the system's control—and sometimes design errors ("bugs")—can still cause faults. These issues typically surface in one of three ways:

- **Exceptions**: CPU or system exceptions raised at run time.
- **User tests**: Application-level checks that detect invalid states or inputs.
- **Timeouts**: Operations that fail to complete within an expected time window.

1.1 Exceptions

Exceptions are conditions in which the processor detects an error and an instruction cannot be completed normally. For example, an exception may occur when attempting to load a value from memory through a NULL pointer. The *XCORE* has more than ten exception types (see the ISA manual for the architecture). Approximately half of the instructions may cause an exception.

The terminology is as follows: when a hardware thread on an *XCORE* gets into an unrecoverable situation, it will *trap*, raising an *exception* and as a result the program-counter (PC) is set to address of that threads *exception-handler*. The exception handler is stored in a control register associated with that hardware thread.

Because each hardware thread maintains its own exception-handler address, exceptions are managed independently per thread. If one thread traps, other threads continue executing; they are not implicitly notified unless your software provides a mechanism (e.g. a shared flag or channel) to propagate the event.

This process allows exception handlers to deal with an exception. In general, exceptions are undesirable, and by default the tools configure exception handler to halt the offending thread. The handler clears all possible sources of events, executes a wait-for-event

instruction that never resumes. This is a safe default action, as the thread experiencing the error can not continue execution or corrupt state (e.g. memory), preventing further damage.

The default action is useful during development: the debugger places a breakpoint on the exception handler, so the designer can inspect system state at the moment the exception occurred.

In a deployed system this may not be ideal, as it may take time for other threads to notice that the offending thread has stopped. Strategies for handling this case are discussed later in this document.

1.2 User test or assertion

Programming using a *defensive* strategy is common place, and involves checking in software for exceptional situations. In particular where generic library components are developed, the first few statements of a function may involved checking that the parameters of the function are a legal combination.

The issue is what to do in the case that the parameters are not legal. It is not unusual to have a convention that a function returns *0* on success and non-zero on failure. Callers check the return value and, on failure, propagate an error or take corrective action.

This strategy allows the caller of a function (and the caller's caller) to take appropriate action. For example, a server handling commands from a many clients might, deep inside a set of function calls detect an error with the client's request, and at top level post a NACK to the client and wait for the next command.

In other software it may be simpler to replace the check returning an error with an *assert*. On an *XCORE*, assertions take a single clock-cycle and trap into the exception handler if the assertion fails.

This enables low-overhead checks, for example, array-bounds validation, that protect the system from buffer overflows. If an assertion triggers, control transfers to the exception handler as described earlier.

1.3 Timeouts

A third cause of errors arises when (part of) the program enters, an infinite loop for some reason - or otherwise stops making progress. This may be due to a programming error, or maybe something in the environment has caused the system to behave incorrectly. For example, brief operation outside the specification temperature range can cause undetermined behaviour in the processor.

In such cases, the system will no longer respond to requests; not dissimilar to the case where a hardware-thread has trapped into the exception handler.

2 Resetting the device

The system designer can choose to reset the device from software. No hardware is required, a single write to a node-configuration register will achieve this. Specifically a write to the *PLL_SETTINGS* register; see the device datasheet for the specification of this register.

Write the default boot value for the target device to this register. This value is silicon dependent, and in the case of *XCORE-200* dependent on the mode pins used:

Chip	Arch	Mode pins	Reg Number	Value to Write
xcore.ai	xs3a	N/A	6	0x01006300
xcore200	xs2a	00	6	0x0080a900
		01	6	0x00800f00
		10	6	0x00801f00
		11	6	0x00803f00

Writing to the *PLL_SETTINGS* register will configure the PLL back to its default state (per the table above), **and** issues a device reset. Although the PLL can be changed without resetting, the intent here is explicitly to reset.

For example, an *XCORE.AI* device is reset with the following statement:

```
.. code-block:: c
```

```
write_sswitch_register(get_local_tile_id(), 6, 0x01006300);
```

If there is more than one die in the system, they must all be reset. It is best to reset the last die in the topology first, and then work back towards earlier die.

3 Automating the process

This section shows, for each error type, how to automatically reset the device in response to a fault.

3.1 Resetting on an Exception, Assert, or User Test

In order to reset on an exception, change the exception handler to point to the function that performs the device reset. This involves a small assembly helper function `exception_handler_install_reset` shown in `app_an02051_exception_handler/src/exception_handler.S`.

```
    .align 4
exception_handler_install_reset:
    entsp 0
    ldap r11, node_reset
    set kep, r11
    retsp 0
```

This function just finds the address of the function to reset the node, and installs it as the `kep` (kernel entry point). The `node_reset` function itself is also quite small, as it is used as an exception handler it must be aligned on a 128 byte boundary:

```
    .align 128                          // The exception handler must be 128 byte aligned
node_reset:
    getr   r0, XS1_RES_TYPE_CHANEND
    freer  res[r0]
    shr    r0, r0, 16                    // Just keep the node ID
    ldc    r1, XS1_SSWITCH_PLL_CTL_NUM
    ldw    r2, cp[XCORE_AI_PLL_SETTING]
    bl     write_sswitch_reg
    clre                                 // This ought not return - but just in case it does
    waiteu                               // Hang up.
```

This function calls the `write_sswitch_reg` with the appropriate value. The two functions are declared in the header file:

```
// Copyright 2025 XMOS LIMITED.
// This Software is subject to the terms of the XMOS Public Licence: Version 1.

/**
 * Function that installs a reset on the exception handler
 * This function must be called by each thread when the thread is create.
 */
void exception_handler_install_reset();

/**
 * Function that can be called to forces the node to be reset.
 */
void node_reset();
```

The main code showing how to use this is in `app_an02051_exception_handler/src/main.c`. It contains a function to set-up the LEDS, and then it flashes leds every 500 ms for 100 ms The two key lines are in `main()` itself:

```
int main(void) {
    exception_handler_install_reset();
    led_setup();
    hwtimer_t tmr = hwtimer_alloc();
    while(1) {
        count ++;
        hwtimer_delay(tmr,40000000);     // Wait for 400 ms
        led_flash();
        xassert(count < 12);             // Creates an exception when count reached 12
    }
}
```

First, the first line of `main()` installs the exception handler. Each hardware thread must call this, because each thread has its own handler. Second, the `xassert` line raises an exception when the counter reaches 12. `xassert` is a single-instruction version of assert that raises an exception on failure.

When running this program on an XK-EVK-XU316 board with `xrun` you will see that the leds are flashing a binary counting sequence 0001, 0010, 0011, ..., 1010, 1011, 1100, and then it stops; as the assertion has trapped the processor has reset, and as it was started with `xrun` it will try and boot from flash which is empty.

The binary can be flashed with:

```
xflash bin/app_an02051_exception_handler.xe
```

In this case the same sequence 0001, 0010, 0011, ..., 1010, 1011, 1100, but then it restarts from 0001. As the device resets on each assert, the sequence loops indefinitely. It is good practise to erase the flash after this (or at least the first few blocks) so that later examples in this application note do not reboot into this program:

```
xflash --target XK-EVK-XU316 --erase
```

This example program will reset on any exception (including `xassert`) and will also reset if user code calls the function `node_reset` directly.

3.2 Resetting on a Timeout in a Single Thread or Tile using the Watchdog

XCORE.AI includes a built-in watchdog that can reset the device after a configured interval. By regularly resetting the watchdog, the program can keep the watchdog under control, until the program times out and the watchdog will reset the device.

This is demonstrated in `app_an02051_jobs/src/main.c` with the watchdog shown in `watchdog.c`. The watchdog is used in `main()`. In this case it is assumed that there are five tasks `f0`, `f1`, ..., `f4` that are executed concurrently in a while loop:

```
int main(void) {
    led_setup();
    watchdog_setup_ms(1005);           // Set watchdog going with initial time of 1.005 second
    while(1) {
        PAR_JOBS(
            PJOB(f0,()),
            PJOB(f1,()),
            PJOB(f2,()),
```

(continues on next page)

(continued from previous page)

```
            PJOB(f3,()),
            PJOB(f4,()));
        watchdog_reset_timer_ms(1005); // Reset watchdog for another 1.005 second
    }
}
```

The watchdog is set up in the first line of `main()`, and is reset every time that all five tasks complete (remember that PAR_JOBS only complete when all tasks complete). If any of the tasks time out, in this case take more than 1.005 seconds, the watchdog will time out and reset the device. In this case the first four jobs are empty, and the fifth one flashes LED 0 with ever increasing pauses: 100 ms, 200 ms, 300 ms, ..., 1000 ms, and then the watchdog will intervene because the task took too long. You can see this by running the program using `xrun` and LED zero will flash 10 times with larger and larger pauses. Then it stops, because the device has been reset.

> **Note**
>
> If it does not stop and instead you see binary counting or something else it is likely that the flash has not been erased; see the end of the previous section.

As before, the binary can be flashed with:

```
xflash bin/app_an02051_jobs.xe
```

The same sequence occurs until LED 3 turns on and the pattern restarts. Illuminating LED 3 is intentional:

```
void led_setup() {
    unsigned has_triggered;
    port_enable(p_leds);
    has_triggered = watchdog_get_has_triggered();
    if (has_triggered) {
        leds_value = 0x8;
    } else {
        leds_value = 0x0;
    }
    port_out(p_leds, leds_value);
}
```

During LED setup, the program checks whether reset was due to the watchdog (using `watchdog_get_has_triggered`). If this function returns 1 the chip was reset due to a watchdog, and we turn LED 3 on. After this, it will go on forever with LED 3 on and LED 0 blinking slower and slower until it is more than a second and it will repeat.

Again, it is good practice to erase the flash afterward (or at least the first few blocks) so that later examples do not reboot into this program:

```
xflash –target XK-EVK-XU316 –erase
```

Note that fork-join PAR_JOBS only works within a tile, and the next section shows a more general case.

3.3 Resetting on a Timeout in a Multi-Tile Program using the Watchdog

This variant shows another way to use the watchdog. It uses the same watchdog code, but instead of assuming that there is one or more threads that are forked and join, it assumes that there is multiple threads that execute forever, possibly on a multi-tile system.

This is shown in `app_an02051_channels/src/main.c`. The code using the watchdog is shown in a function `g0`. The `main()` function forks 5 tasks `g0`, `g1`, ..., `g4`, and connects them using a ring of channels. These tasks could be anywhere on the system, and it is assumed that the tasks never return and continue forever. Instead of a ring one can have other topologies depending on your application. Or, you may not need to monitor all threads but just two threads will do:

```
int main() {
    channel_t a[5];
    for(int i = 0; i < 5; i++) {
        a[i] = chan_alloc();
    }
    PAR_JOBS(
        PJOB(g0,(a[0].end_a, a[1].end_b)),
        PJOB(g1,(a[1].end_a, a[2].end_b)),
        PJOB(g2,(a[2].end_a, a[3].end_b)),
        PJOB(g3,(a[3].end_a, a[4].end_b)),
        PJOB(g4,(a[4].end_a, a[0].end_b)));
    for(int i = 0; i < 5; i++) {
        chan_free(a[i]);
    }
    return 0;
}
```

The watchdog is set up in the first line of the `g0` task program, and is reset every time that `g0` has received confirmation along the ring that all other essential tasks are still alive:

```
void g0(chanend_t in, chanend_t out) {
    watchdog_setup_ms(1005);           // Set watchdog going with initial time of 1.005 second
    chanend_out_control_token(out, 1);  // Somebody has to set the token going, its g0
    while(1) {
        // Do a lot of work
        chanend_out_control_token(out, 1);         // Now pass the token around
        chanend_check_control_token(in, 1);        // Check if token has come around
        watchdog_reset_timer_ms(1005);             // Reset watchdog for another 1.005 second
    }
}
```

In this example `g0` does no work, but typically it would do work and occasionally check that all is well to reset the watchdog. (One can use all sorts of different topologies or methods of receiving tokens, including in `SELECT_RES` constructs that are used for other purposes.) If one of the tasks fails to pass the token on, `g0` will not get the token in time and the watchdog will reset the device. As before, a LED is flashed (in `g4()`) and the delay is increased. You can test the program with `xrun` and `xflash` as in previous sections.

4 Persistent state over a reboot

The *XCORE* memory is not cleared on a reset or reboot. As long as memory remains powered, it will retain it state. However, some sections of the memory will be overwritten during boot and startup. The memory comprises six areas in this order:

- A *text* segment where code resides
- A *data* segment where global variables reside that are initialised to a non-zero value
- A *bss* segment where global variables reside that are initialised to zero
- A *heap* that is used by `malloc()`, and by extension by functions that may call `malloc()` such as `printf()`.
- A *stack* where all the stacks of the hardware threads are allocated
- The *debug stack* used by the debugger as a scratch memory

The sections that are overwritten across a reset/reboot cycle are:

- On boot the *text* and *data* sections are reloaded from flash, and memory between *0xffc00* and *0xfff80* is overwritten.
- On startup, the *bss* section is cleared, and the stack section is overwritten.
- Whilst the program is running, the *heap* may be overwritten by allocations

To pass state across a reset/reboot while power remains on, store data in one of the following areas:

- On *XCORE.AI* memory between 0xfffe8 and 0xfffff is not used, and can be used to save a small amount of state across a reset. Note that some libraries and tools may use this area, for example the Device Firmware Upgrade (DFU) in `lib_xua` and the debug mode in `xflash` both use this.

- For larger state, an area between `_bheap` and `_eheap` can be used, but beware that the use of `malloc()` allocates from this space. A safe pattern is to perform a `malloc()` as the first action in `main()` to obtain a pointer to memory that will persist across subsequent resets.

5 Example applications

5.1 Building the examples

This section assumes that the XMOS XTC Tools have been downloaded and installed. The required version is specified in the accompanying `README`.

Installation instructions can be found here.

Special attention should be paid to the section on Installation of Required Third-Party Tools.

The application is built using the xcommon-cmake build system, which is provided with the XTC tools and is based on CMake.

The `an02051` software ZIP package should be downloaded and extracted to a chosen working directory.

To configure the build, the following commands should be run from an XTC command prompt:

```
cd an02051
cmake -G "Unix Makefiles" -B build
```

All required dependencies are included in the software package. If any dependencies are missing, they will be retrieved automatically during this step.

The application binaries should then be built using `xmake`:

```
xmake -j -C build
```

Binary artifacts (.xe files) will be generated under the appropriate subdirectories of the relevant application `/bin` directory for example, `app_an02051_jobs/bin` — one for each supported build configuration.

For subsequent builds, the `cmake` step may be omitted. If `CMakeLists.txt` or other build files are modified, `cmake` will be re-run automatically by `xmake` as needed.

5.2 Running the examples

From an XTC command prompt, the following command should be run from the `an02051/app_an02051_jobs` directory:

```
xrun ./bin/app_an02051_jobs.xe
```

Alternatively, the application can be programmed into flash memory for standalone execution:

```
xflash ./bin/app_an02051_jobs.xe
```


6 Further reading

- *XMOS* XTC Tools Installation Guide
  https://xmos.com/xtc-install-guide
- *XMOS* XTC Tools User Guide
  https://www.xmos.com/view/Tools-15-Documentation
- *XMOS* application build and dependency management system; *xcommon-cmake*
  https://www.xmos.com/file/xcommon-cmake-documentation/?version=latest