Memory Management & Error Handling

# Introduction

This document describes the basics of memory management and error handling in InnoOS™. It includes brief descriptions on system heap memory allocation, memory reallocation, memory pools, freeing dynamically allocated memory, and reporting of fatal errors using available APIs. The accompanying code snippets demonstrate the use of memory management and error handling APIs.

# Memory Management and Error Handling

## System Heap

System heap provides drivers and applications with dynamically allocated memory buffers. It is initialized during system start-up and is available when the application enters main(). The size of the system heap is configurable with a boot argument heapsize (Refer Appendix A: Boot arguments in the API Reference Manual for more information).

The functions os\_alloc(), os\_zalloc(), and os\_realloc() are used to request memory buffers from the heap. A small overhead is added to the requested size. This overhead is used to track allocated and free buffers. The function os\_free() is used to free a buffer, returning it to the system heap.

## Memory Pools

Memory pools provide another way to dynamically allocate memory. Memory pools are useful when an application needs to allocate multiple small objects of a certain size. The use of a memory pool in such a scenario helps to minimize memory allocation overhead and internal fragmentation of the heap.

The pool allocator works by allocating large chunks of memory from the page allocator. These chunks are split into the exact size the pool allocator was initialized with (element\_size) when pool\_init() is called. These objects can now be allocated and freed using the pool\_alloc() and pool\_free()functions with minimal overhead in execution time and memory usage.

## Error Handling

Error handling refers to the anticipation, detection, resolution, and possible recovery from error conditions that may occur during the execution of a software application. It helps in maintaining the expected flow of program execution.

A run-time error takes place during the execution of a program. There are many types of runtime errors, which can be broadly categorized as recoverable errors and unrecoverable errors.

For example, the errors returned by functions, as to indicate the caller about the result through error codes, are recoverable.

Recoverable errors can be handled in many ways depending on the scenario. For example, an attempt to recover can be made by retrying (Ex: after a timeout) by reinitializing some components (Ex: drivers) or resetting a non-responding peripheral. Or they can be handled by passing the error to the other layer after releasing the resources.

The unrecoverable errors are fatal errors and can occur due to CPU exceptions due to illegal instructions or access to protected memory etc.

There are scenarios where some system-level checks are needed. And if they fail, the fatal errors are needed to be induced by executing an undefined instruction exception.

These fatal errors are handled by an exception handler which is used for getting important debug information and reason for failure, helping the problem to be solved using this information.

# Memory Management and Error Handling APIs

## System Heap APIs

### os\_alloc()

Allocates a block of memory at least the number of bytes in length asked as a parameter and returns a pointer to it or returns NULL if it fails in allocating.

|  |
| --- |
| void \* os\_alloc(size\_t size) |

Returned memory is aligned to the maximum natural alignment of the system (8 in this implementation).

### os\_zalloc()

Similar to os\_alloc()but with memory allocated is zero-initialized.

|  |
| --- |
| void \* os\_zalloc(size\_t size) |

### os\_realloc()

Attempts to change the size of an allocated block of memory.

|  |
| --- |
| void \* os\_realloc(void \*ptr, size\_t size) |

If the block can be resized in place, the same pointer passed in ptr is returned. If the block must be moved, a pointer different from ptr is returned.

### os\_avail\_heap()

Returns the size of the remaining heap.

|  |
| --- |
| size\_t os\_avail\_heap(void) |

This is the total size including internal overhead and may not represent the amount that can be allocated by an application.

### os\_free()

Free allocated memory.

|  |
| --- |
| void os\_free(void \*ptr) |

It returns the allocated memory to the heap. If the memory has more than one reference, the count is simply dropped by one.

## Memory Pool APIs

### pool\_init()

Initializes a memory pool.

|  |
| --- |
| void pool\_init(struct memory\_pool \*pool, size\_t element\_size) |

Pointer to memory\_pool and the size of the objects allocated from the pool is passed as input.

### pool\_alloc()

Allocates one object from the memory pool. If the pool is empty, the memory will be allocated from the heap.

|  |
| --- |
| void \* pool\_alloc(struct memory\_pool \*pool) |

Returns pointer to a new object.

### pool\_free()

Returns an object to the memory pool.

|  |
| --- |
| void pool\_free(struct memory\_pool \*pool, void \*ptr) |

The memory pool is used to allocate memory chunks suitable to fit a certain number of objects (of element\_size).

When an object is freed using the pool\_free(), the object is put on a free-list in the pool for reuse at later point in time. This allows for quick and efficient allocation of these objects.

The object is not returned to the heap until the destroy API is called.

### pool\_destroy()

Frees all memory claimed by the memory pool.

|  |
| --- |
| void pool\_destroy(struct memory\_pool \*pool) |

All the previously allocated objects must be freed using pool\_free()before destroying them. All memory chunks get released to heap on calling destroy.

## Error Handling APIs

### os\_error()

os\_error(uint32\_t errcode)is used to report a potential system failure scenario using a fatal error induced via an unidentified instruction exception. This instruction has an immediate field in the instruction encoding that is used to pass on supplied error code.

|  |
| --- |
| static inline void \_\_noreturn os\_error(uint32\_t errcode) |

This error is caught by the exception handler which dumps the CPU register states and parts of the current stack on the debug console and halt the system.

### os\_error2()

os\_error2(uint32\_t errcode, uint32\_t extra) does the same thing as os\_error() with an extra argument parameter. This extra argument can be used to pass some additional information about the error.

For example, an extra argument can be used to pass the ID of an invalid argument, or the message to be passed when an assertion fails, or the exception vector and, so on.

|  |
| --- |
| static inline void \_\_noreturn os\_error2(uint32\_t errcode, uint32\_t extra) |

### enum os\_err\_t

OS error codes are defined in this enum in error.h and are as follows:

|  |
| --- |
| /\*\*  \* OS error codes  \*/  typedef enum {  /\*\* Error in application \*/  OS\_ERR\_APPLICATION = 0x00,  /\*\* Heap is out of memory \*/  OS\_ERR\_HEAP\_EXHAUSTED = 0x01,  /\*\* Failed to initialize virtual memory \*/  OS\_ERR\_VM\_INIT\_FAILED = 0x02,  /\*\* Invalid argument in os function call \*/  OS\_ERR\_INVALID\_ARGUMENT = 0xfa,  /\*\* An spurious event was triggered \*/  OS\_ERR\_SPURIOUS\_EVENT = 0xfb,  /\*\* OS internal error \*/  OS\_ERR\_INTERNAL\_ERROR = 0xfc,  /\*\* Timer callback missing \*/  OS\_ERR\_INVALID\_TIMER = 0xfd,  /\*\* Assertion failure \*/  OS\_ERR\_ASSERTION\_FAILED = 0xfe,  /\*\* Unexpected exception \*/  OS\_ERR\_UNEXPECTED\_EXCEPTION = 0xff,  } os\_err\_t; |

### macro OS\_ERR\_HEAP\_EXHAUSTED()

OS memory allocation functions can be put under macro OS\_ERROR\_ON\_NULL() which returns the error OS\_ERR\_HEAP\_EXHAUSTED if a NULL is returned by the allocation function. Heap is out of memory.

# Code Walkthrough

**Note**: All the applicable ELF are available in the following location of the SDK release package: sdk\_x.y\examples\innoos\_memory\_mgmt\bin.

x and y in sdk\_x.y refer to the SDK release version. For example: *sdk\_2.4\examples\innoos\_memory\_mgmt\bin*.

## Example memory\_management\_1.c

### Overview

The sample code in the path: examples/innoos\_memory\_mgmt/src/memory\_management\_1.c

is a simple application that demonstrates the addition of two integers using dynamic memory allocation and showcases os\_alloc() , os\_zalloc(),os\_mem\_incref() and os\_free().

### Sample Code Walkthrough

Consider the following section of the example:

|  |
| --- |
| int a=10, b=20, \*num\_1, \*num\_2, \*sum; |

These pointers will be used to address memory that is dynamically allocated to hold each of our integers and their sum.

Here in the example, heap memory is allocated for the three variables using os\_alloc() and os\_zalloc():

|  |
| --- |
| /\* Memory allocation \*/  num\_1 = (int\*) os\_alloc(sizeof(int));  num\_2 = (int\*) os\_zalloc(sizeof(int)); |

The memory for the variable num\_2is allocated using os\_zalloc(). This will initialize the variable with zero.

Memory for num\_1 and sum is allocated with os\_alloc().This function, like os\_zalloc(), allocates memory of a given size, but the difference is that the allocated memory will not be initialized with zero.

Consider the following section of the example:

|  |
| --- |
| os\_printf("\nprinting initial values num1(allocated using os\_alloc):\  [%d](garbage)\n", \*num\_1);  os\_printf("\nprinting initial values num2(allocated using os\_zalloc)\  :[%d]\n", \*num\_2); |

The os\_avail\_heap() function reveals how much space is available on the heap.

The snippet above first displays the available space on the heap. The statement os\_mem\_incref(num\_1)increments the reference count of the memory allocated for num\_1 by one. The second call to os\_avail\_heap() will reveal that increasing the reference count did not change the amount of available heap space.

The example contains the following:

|  |
| --- |
| os\_free(num\_1);  os\_free(num\_2);  os\_free(sum); |

os\_free()is used to free allocated memory after use. For num\_2 and sum, only a single call to os\_free() is needed to free the allocated memory. Two os\_free() calls are necessary to free the memory allocated for num\_1 because the reference count for this piece of allocated memory was 2, the reference count was 1 when the memory was allocated with os\_alloc(), and it increased by one with the call to os\_mem\_incref().

### Running the Application

Program memory\_management\_1.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the memory\_management\_1.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

memory\_management\_1.elf is created when compiling the code which provides the following console output when programmed to Talaria TWO.

|  |
| --- |
| UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-fdfd20079 $  $App:git-b1ab1153  SDK Ver: sdk\_2.4  Memory Management Demo App 1  initial heap size:449944  printing initial values num1(allocated using os\_alloc): [329236](garbage)  printing initial values num2(allocated using os\_zalloc) :[0]  heap after allocation:[449896]  Sum = 30  available heap before incref = [449896]  available heap after incref = [449896]  available heap after all free =[449928]  final heap size:[449944] |

## Example memory\_management\_2.c

### Overview

The sample code in the path examples/innoos\_memory\_mgmt /src/memory\_management\_2.c is a simple application that demonstrates the memory reallocation using os\_realloc().

### Sample Code Walkthrough

Consider the following section of the example:

|  |
| --- |
| char \*str; |

This defines a pointer that will be used to address memory that is dynamically allocated to hold the string.

In the example, heap memory is allocated using os\_alloc():

|  |
| --- |
| /\*Initial memory allocation\*/  str = (char \*) os\_alloc(10); |

The memory for stris allocated using os\_alloc(10), and the string can store nine characters plus a null terminator.

The example then checks the remaining memory available on the heap:

|  |
| --- |
| /\* Heap check after memory allocation\*/  os\_printf("\nheap after allocation:[%d]\n", os\_avail\_heap()); |

os\_avail\_heap() returns the amount of space available on the heap.

Next, we set the contents of the string:

|  |
| --- |
| strcpy(str, "INNOPHASE");  os\_printf("String = %s address[%p]", str, str);  /\* Reallocating memory \*/  str = (char \*) os\_realloc(str, 20);  strcat(str, " Talaria"); |

Here, INNOPHASEis first copied to the character array. By using os\_realloc(str, 30)**,** the size of the memory previously allocated is changed to 30 characters. Post this, there is space to concatenate Talariato the existing string.

Once the dynamically allocated memory is no longer needed, it is freed:

|  |
| --- |
| os\_free(str); |

os\_free() is used to free dynamically allocated memory. In this program, os\_free(str) frees the memory previously allocated for str.

### Running the Application

Program memory\_management\_2.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the memory\_management\_2.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

memory\_management\_2.elf is created when compiling the code which provides the following console output when programmed to Talaria TWO.

|  |
| --- |
| UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-fdfd20079 $  $App:git-b1ab1153  SDK Ver: sdk\_2.4  Memory Management Demo App 2  initial heap available:448920  heap after allocation:[448896]  String = INNOPHASE address[0x000bfb60]  heap after reallocation:[448880]  String = INNOPHASE Talaria TWO address[0x000bfb60]  available heap after all free =[448920] |

## Example pool.c

### Overview

The sample code in the path examples/innoos\_memory\_mgmt/src/pool.c is a simple application that demonstrates the steps required to use a memory pool including initialization of the pool, allocation and freeing of memory from the pool, and cleanup of the pool when it is no longer needed.

### Sample Code Walkthrough

The example program declares a structure struct Data, that represents the kind of data that the application needs to manage. A memory pool will be used to allocate memory for instances of this structure.

The structure declaration is as follows:

|  |
| --- |
| struct Data  {  void \*addr;  uint16\_t length;  uint16\_t flags;  void \*next;  }; |

The memory pool is defined as follows:

|  |
| --- |
| /\* Definition of a memory pool \*/  static struct memory\_pool dd\_pool; |

The function os\_avail\_heap() reveals the amount of memory available on the heap. The example displays this information at various points to show when the memory is allocated and freed on the heap when using the memory pool.

Before the memory pool can be used, it needs to be initialized with pool\_init(). The second argument of pool\_init()specifies the size of objects that the pool will be used to allocate.

|  |
| --- |
| /\* Get available heap memory before allocation \*/  os\_printf("\ninitial heap available:[%d]\n", os\_avail\_heap());  /\* Initialize memory pool\*/  pool\_init(&dd\_pool, sizeof(struct Data)); |

Once the pool is initialized, memory can be allocated from the pool using pool\_alloc() as follows:

|  |
| --- |
| /\* Allocate memory from the pool \*/  struct Data \*pdata1 = pool\_alloc(&dd\_pool); |

The example displays the amount of memory available on the heap after the first allocation from the pool, as well as the pointer address of the allocated memory and its size:

|  |
| --- |
| /\* Check the amount of memory available on the heap again \*/  os\_printf("\nheap available after 1st allocation:[%d] allocated address[%p] size[%d]\n", os\_avail\_heap(), pdata1, sizeof(struct Data)); |

Memory that was allocated from the pool can be freed with pool\_free():

|  |
| --- |
| /\* Free memory allocated from the pool \*/  pool\_free(&dd\_pool, pdata1); |

When the memory pool is no longer needed, it can be destroyed using pool\_destroy(). Note that all memory previously allocated from the pool must be freed before the pool is destroyed.

|  |
| --- |
| /\* Destroy the pool \*/  pool\_destroy(&dd\_pool); |

### Running the Application

Program pool.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the pool.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

pool.elf is created when compiling the code and gives the following console output when programmed to Talaria TWO.

|  |
| --- |
| UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-fdfd20079 $  $App:git-b1ab1153  SDK Ver: sdk\_2.4  Mem Pool Demo App  initial heap available:[449680]  heap available after memory pool init: [449680]  heap available after 1st allocation:[449520] allocated address[0x000bf570] size[12]  heap available after 1st free:[449520]  heap available after 2nd allocation:[449520] allocated address[0x000bf580] size[12]  heap available after 2nd free:[449520]  final heap available:[449680] |

## Example error\_handling.c

### Overview

The sample code in the path: examples/innoos\_memory\_mgmt/src/error\_handling.c

is a simple application that demonstrates how fatal errors are reported by calling APIs which induce an undefined instruction exception, in the scenarios where the critical system level checks fail.

For example, the cases where system-level potential failures are needed to be identified are, if the heap is getting exhausted if some undefined event has occurred or if there is an invalid argument in the OS function call, and so on.

These exceptions are handled internally by the default exception handler os\_unexpected\_exception(), which prints an informative message on the console with details of the CPU registers and state at the time of the exception and the stack trace.

### Sample Code Walkthrough

This section of the program demonstrates heap exhaustion. os\_error(OS\_ERR\_HEAP\_EXHAUSTED) is used to report a fatal error with an error code representing heap exhaustion, which means that the heap is out of memory.

|  |
| --- |
| data\_packet = os\_alloc(1000000);  if (!data\_packet)  {  os\_printf("\nthere is not enough memory to allocate the data packet. Available heap:[%d]", os\_avail\_heap());  os\_error(OS\_ERR\_HEAP\_EXHAUSTED);  return;  } |

In the following code, os\_error(OS\_ERR\_INVALID\_ARGUMENT)is used to report that a function argument is invalid.

|  |
| --- |
| if (!input\_data)  {  os\_error(OS\_ERR\_INVALID\_ARGUMENT);  } |

The following example reports a spurious event:

|  |
| --- |
| os\_error(OS\_ERR\_SPURIOUS\_EVENT); |

By using os\_error(OS\_ERR\_SPURIOUS\_EVENT), a spurious event will be handled as a fatal error.

An application-specific error can be reported as follows:

|  |
| --- |
| /\*Application specific error\*/  os\_error(OS\_ERR\_APPLICATION); |

Additionally, the sample program reports an unexpected exception as:

|  |
| --- |
| os\_error2(OS\_ERR\_UNEXPECTED\_EXCEPTION, 3); |

For unexpected exception conditions, os\_error2()is used.

Since each of these errors is fatal, the program will be terminated as soon as one of them occurs. To experiment with various types of errors in the sample program, disable earlier errors so that recent ones can be seen. This can be accomplished by commenting out calls to functions that produce errors.

### Running the Application

Program error\_handling.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the error\_handling.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

error\_handling.elf is created when compiling the code and provides the following console output when programmed to Talaria TWO.

|  |
| --- |
| UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-fdfd20079 $  $App:git-b1ab1153  SDK Ver: sdk\_2.4  Error Handling Demo App  Sample started. Available heap:[450456]  there is not enough memory to allocate the data packet. Available heap:[450456]  OS\_ERROR: HEAP EXHAUSTED  OS\_UNEXPECTED\_EXCEPTION 0x6  R0=00000058 R1=0004b728 R2=00fc0d00 R3=00000000  R4=04444444 R5=05555555 R6=06666666 R7=07777777  R8=08888888 R9=09999999 R10=0aaaaaaa R11=0bbbbbbb  R12=000b2bf2 SP=000b2c94 LR=001049fd PC=001049fc  xPSR=21000000 CONTROL=00000000 CFSR=00010000 BFAR=e000ed38  STACK:  0x000b2cd8: 03333333 000463b9 6e69616d 2189fa00  0x000b2ce8: 001049d1 07f83201 00000002 000b2c94  0x000b2cf8: 000b24e8 a5631209 000b2d00 000b2d00  0x000b2d08: 000b2d08 000b2d08 0004000c 0004000c  0x000b2d18: 0004005c 000bed20 000b2d20 000b2d20  0x000b2d28: 00000000 a0906362 6df329e6 7309ec7f  0x000b2d38: 9ae8e4cd 04b5564b 145d426c b9c0484e  0x000b2d48: ffc8daa3 ebfbaff5 8fa90474 1f15fa67  0x000b2d58: 7e719464 5ba8eb37 7635912d fd979a53  0x000b2d68: cbadc8d9 601b8df6 0b4e4084 679732f3  0x000b2d78: 297325ec 11401758 e51f63df 0b6a56b6  0x000b2d88: 70d31ab7 0e0912cd aafd1662 5df9b7f7  0x000b2d98: e349cfe6 8343f8b2 cd865e0a 33f21cc5  0x000b2da8: 6c98c51e dce15390 84f8e847 351cfd2a  0x000b2db8: e50d5dac b8202066 bdb87b4e 9974b068  0x000b2dc8: 0dc94b10 266dde76 4bb652e3 74264fbe |