eheap™

User's Guide

Version 5.2

February 2024

by Ralph Moore

Micro Digital Associates, Inc.
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

Table of Contents

Chapter 1 Introduction

Embedded systems are 100 maybe 1000 times as diverse as desktop and server systems. Hence, they deserve a heap that is much more flexible and customizable than the typical OS heap.

eheap (embedded heap) is a generic, high-performance, configurable, self-healing heap, with enhanced safety and debug features. It has some similarities to *dlmalloc*, used in Linux, and to *tcmalloc*, used in Android, in that it is a *bin-type* heap. It differs from them in that its architecture is governed by the following embedded system requirements:

- Wide range of RAM sizes from very small to large.
- Good performance and deterministic operation are required.
- High priority tasks must be able to preempt and run quickly.
- Small code size is often necessary.
- Expected to run forever.
- Strong debug support is needed.
- Growing need for ruggedness and self-healing.
- Significant idle time is available.

Additional requirements for security are:

- Multiple heap support.
- Aligned allocations, including MPU region allocations.

Wide Range of RAM Sizes: An embedded heap must be efficiently adaptable to heap sizes from tens of kilobytes up to megabytes.

Good Performance: A general-purpose allocator, such as dlmalloc, must be ultra-fast because many applications that use it require tens of thousands of allocations and deallocations per second or even more. This is because they are typically written to use myriad tiny objects with very short lifetimes.

The situation is different for embedded systems, which typically are carefully coded to achieve maximum performance within limited constraints. Also, wherever extremely high allocation and deallocation rates are required, embedded applications have the option to use block pools, instead of the heap. Hence, extreme speed is not usually a primary embedded heap requirement.

Determinism: System determinism is a primary requirement. The heap must not cause mission-critical tasks to miss their deadlines. Too much indeterminism can also cause missed deadlines for those tasks using the heap. Hence an embedded heap must not use extravagant mechanisms, or it must make such mechanisms explicitly controllable by the programmer. In some cases, block pools may be the only solution to meet necessary deadlines.

Small Code is necessary in small embedded systems, but is not likely to be a primary requirement in systems having substantial heap usage.

Run Forever: This is typically not a requirement for desktop and enterprise systems because jobs are generally short, memory is plentiful, and the computer can be easily rebooted. However, embedded applications are typically unattended. Hence rebooting is undesirable and may be difficult to do. Therefore, the heap must run trouble-free for long periods.

Strong Debug Support is necessary to find heap-usage bugs before systems are shipped. These include features such as: time-stamping chunks, block overflow fences, block owner identification, block pattern filling to more easily see heap structure in memory, heap-aware debugger plug-in, and heap integrity scanning. If the heap can catch usage errors before other tasks are impacted, it is easier to find their causes and fix them.

Ruggedness and Self-Healing: A heap failure is not usually disastrous in a desktop or enterprise system because the system can simply be rebooted and/or the application re-run. Such is not the case in most embedded systems, which are expected to "keep marching on," whatever happens. eheap has features to help achieve this.

Idle Time: Embedded systems usually face large load variations, and even under heavy loads they must meet their deadlines. Hence there is usually significant idle time which can be used to improve heap performance and integrity.

Multiple Heap Support: Ideally software running in umode (unprivileged or user mode) should be divided into isolated partitions so that if one partition is penetrated others are still safe. This is not possible if the partitions share a single heap. Hence, a dedicated heap per partition using a heap is necessary for security.

Aligned Allocations are desirable for many reasons, but are especially important for MPU regions. Some MPUs further require blocks to be aligned on their sizes. See the SecureSMX User's Guide for more on this.

It is apparent from the above that a heap designed for embedded use must be configurable, since no one heap could meet all of those requirements. This was a basic design objective of eheap. In discussions that follow where a feature is optional, its configuration constant is specified, e.g. EH_ALIGN. If the configuration constant is OFF, the related code is not present. This typically reduces memory size and improves performance.

Other features may be dynamically switched ON or OFF. These are controlled by the mode field in the hv structure, e.g. mode.fl.debug. In these cases, if the mode is OFF, RAM usage may be reduced and performance improved, but there is no reduction in code size. Whereas configuration settings apply to all heaps in a multi-heap system, mode settings apply per heap. This permits, for example, debugging one heap while others run normally.

Chapter 2 Basics

physical structure

The memory area allocated to a heap is divided into *chunks* of various sizes. Each chunk has a *chunk control block* (CCB) and its remainder is available for use as a *data block*. Chunks are multiples of 8 bytes, in size and aligned on 8-byte boundaries. There are two main types of chunks: *inuse* and *free*. eheap provides an additional *debug* chunk type, which is discussed under **debug mode**, below.

An inuse chunk is one that has been allocated to an application via a malloc(). The application uses the data block of the chunk; it does not access the CCB. inuse chunks have CCBs of 8 bytes. If the average data block is small, say 24 bytes, the CCB represents a significant overhead of 33%. On the other hand, if average block size is 64 bytes, then the overhead is 12.5%.

The first word of all CCBs is a forward link (fl) to the next chunk, and the second word is a backward link (bl) to the previous chunk. These are used to doubly link all chunks in the heap, in physical order, regardless of chunk type. This provides the *physical heap structure*. The heap may be traversed in either direction for purposes such as splitting chunks, merging chunks, scanning chunks, and fixing broken CCBs. The physical structure produces a *linear heap*. The smxAware Memory Map Overview shows the physical heap structure graphically.

Figure 2.1 shows the physical structure of a typical heap. The heap starts with the Start Chunk, SC, which is an inuse chunk with no data block. This is followed by some inuse and free chunks, each starting with a CCB. The Top Chunk, TC, is a special free chunk that initially comprises the entire free heap and is the source of all allocated chunks. Eventually allocated chunks are freed and become the source for new allocations. The heap ends with the End Chunk, EC, which is an inuse chunk with no data block.

Heaps such as dlmalloc and its derivatives achieve greater memory efficiency by having only a chunk's chunk size in every chunk. If an inuse chunk is preceded by a free chunk, the size of the free chunk is put ahead of the inuse chunk's size; if an inuse chunk is preceded by an inuse chunk, the space is used for data. If it is assumed that inuse chunks are four times as numerous as free chunks, this achieves an average of 1.25 word overhead per chunk vs 2 for eheap. This advantage is significant for very small chunks, but of much less importance for larger chunks.

Having both forward and backward links, as in eheap, has the following advantages:

(1) Merging of free chunks is not mandatory, resulting in better bin utilization.

(2) Self-healing is possible by means of continual forward and backward scanning.

(3) Use of links rather than sizes makes manual heap tracing easier.

Hence, eheap may use bins more effectively and is a more rugged heap than dlmalloc – both important for embedded systems.

The emphasis that dlmalloc and its derivatives place upon small chunk efficiency is due to the fact that C++ and other object-oriented languages tend to create very large numbers of small objects from a heap. eheap takes a different approach to this problem by integrating 8-byte and 12-byte block pools. Block pools are faster and more memory-efficient than heaps. Block pool integration is done in such a way that if either pool becomes empty or a required alignment cannot be met, the block is allocated from the heap. When freed, blocks automatically go back to where they came from. This process is completely

transparent to the programmer; thus, block pools can be sized to meet average needs and peak needs are met by the heap.

For allocations, linear heaps must be searched from the first free chunk, chunk by chunk until a big-enough free chunk is found. As a heap is used, it tends to become divided into more and more chunks. For a linear heap, this means that allocations take longer and longer, and thus become less and less deterministic. Ultimately, the heap may become so *fragmented* that an allocation fails after searching the entire heap. This kind of heap is not useful for systems with large heaps and high heap activity.

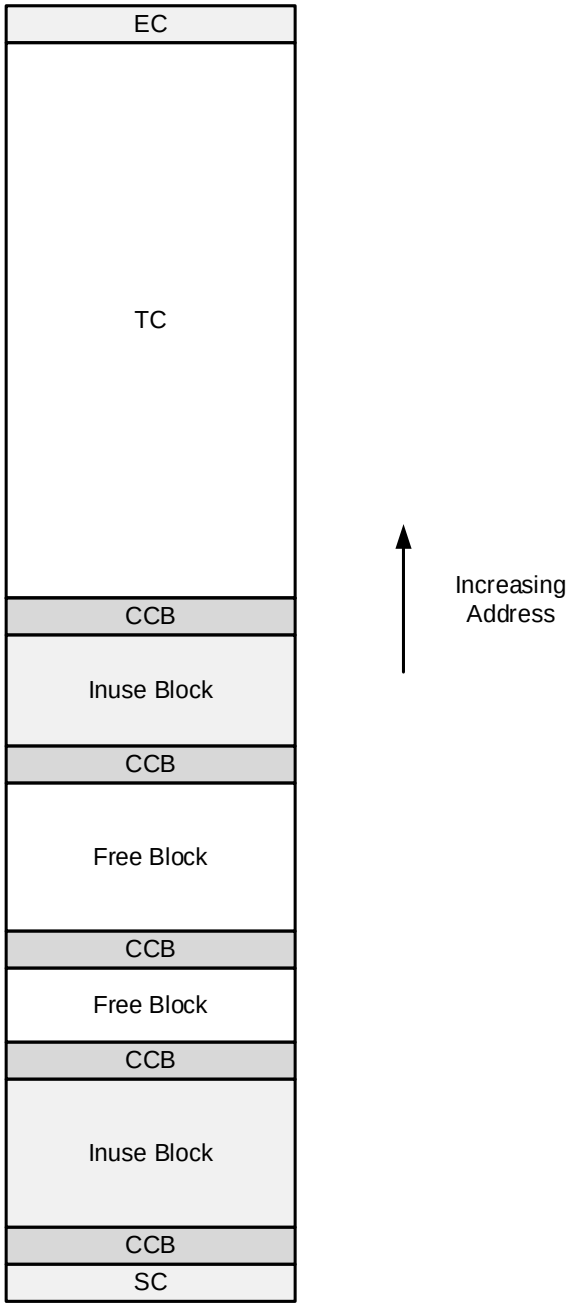

Fig. 2.1

logical structure

eheap superimposes upon the physical heap structure a *logical heap structure* consisting of *heap bins*. Heap bins "hold" free chunks of specified sizes. eheap bins are defined by two arrays: the array of bin sizes, binsz[], and the array of bins, bin[]. The size of each bin is the smallest chunk handled by that bin. A bin that holds a single size is called a *small bin*. For example, a small bin might handle only 24-byte chunks. A bin that holds a range of sizes is called a *large bin.* For example, a large bin might handle 128, 136, ..., 248-byte chunks (15 total sizes, spaced 8-bytes apart). Note that chunk sizes are multiples of 8-bytes.

This logical structure adds a second dimension to a heap, which allows plucking a big-enough chunk from anywhere in the heap with little or no searching. With eheap, the number of bins and the sizes of bins can be selected to best fit the needs of a specific heap. For example, a small heap dedicated to a particular function such as a USB host stack may require only a few bins to operate efficiently, whereas a large main heap may need many bins to operate efficiently.

Each bin consists of a *free forward link* (ffl) and a *free backward link* (fbl) to a doubly-linked list of free chunks. During allocation, a bin is selected by the desired chunk size. If it is a small bin, the first chunk is taken; if it is a large bin, the first large-enough chunk is taken. Provided that bins are not allowed to become empty, allocations from small bins can be very fast and allocations from large bins can be much faster than from a linear heap and sometimes as fast as a from a small bin. Methods to prevent bins from becoming empty, such as merge control and seeding, are presented in the Optimization chapter.

eheap also permits aligned allocations greater than 8 bytes. For these, the chunk selected must also have a block of sz bytes that is aligned on a 2^an boundary. This results in more searching and slower allocation times. See the **aligned allocation** section below.

data blocks vs. chunks

Data blocks are the user interface to a heap, but the heap, itself, is composed of chunks. This tends to cause confusion. For example, bin sizes are determined by chunk sizes, not by block sizes. Hence, a 160-byte small bin contains 160-byte chunks and an inuse chunk from this bin will hold a 152-byte data block. Bins must be thought of in terms of chunk sizes, not block sizes. Also, most pointers used within eheap are chunk pointers, not block pointers as are used within the application.

Another area of confusion is between physical and logical structure. The positions of chunks in bins have no relationship to their positions in the heap. For example, adjacent chunks in a bin are not likely to be adjacent in the heap and vice versa. This can be confusing, for example, when tracing bin links in a watch window.

small bin array, SBA

Most heaps start with a Small Bin Array, SBA. SBA bin sizes are 24, 32, 40, etc. up to the top SBA bin, with no missing 8-byte multiples. The SBA can have 0 to 31 bins, but usually it has just enough bins to cover the most frequently used small chunk sizes. The SBA can be omitted, but that is rare. In small systems, 5 SBA bins might be enough (24, 32, 40, 48, and 56), but 10 SBA bins might be needed (24, 32, 40, 48, 56, 64, 72, 80, 88, and 96) in a large system. As noted above, these are chunk sizes. For inuse chunks, corresponding block sizes are 8 bytes less – i.e.: 16 to 48 and 16 to 88. For debug chunks, the CCB is bigger, so the data block is even smaller.

An SBA has the following advantages: bin selection is very fast (binno = csize/8 – 3), the first chunk can be taken, unless required alignment is > 8 bytes.

Average chunk sizes used by some applications, can be very small (e.g. 32 bytes), so optimizing small chunk mallocs and frees is important for performance – especially for object-oriented code, such as Java

and C++. However, embedded systems that are written in C are likely to have larger average chunk sizes, and in some cases, the upper bins may be more important. eheap allows bin sizes to be adjusted to optimize performance for each specific heap. See the Optimization chapter for details.

upper bin array, UBA

Above the SBA, is the Upper Bin Array, UBA. The SBA plus the UBA can have up to 32 bins. The upper bin array may consist of any combination of large bins and small bins. For example, the sizes of a UBA might be defined as 104, 232, 360, 520, and 528. The first large bin is immediately above the 10-bin SBA (its top bin = 96). The 104 and 232 bins each cover 128 bytes and have 16 chunk sizes. The 360 bin covers 160 bytes and has 20 chunk sizes. The 520 bin is a small bin having only the 520-byte chunk size. It provides 512-byte blocks, which might be needed for a file system or a communication protocol. The 528 bin is called the *top bin*; it handles chunk sizes from 528 bytes and up.

An upper bin is located by doing a binary search on the binsz[] array vs. the needed chunk size. So, for example, if there are 15 upper bins, up to 4 searches are required. binsz[] should be located in fast memory in order to minimize search times. Once found, the bin's free chunk list is searched for the first big-enough chunk. In the case of a small bin this is the first chunk, unless an > 3. In the case of a large bin, it may be necessary to examine several chunks in the bin's free chunk list to find a big enough chunk. Bin sorting, during idle time, helps to ensure that a best fit is found.

Figure 2.2 illustrates the Bin Size Array, SBA, UBA, and free chunks in bins:

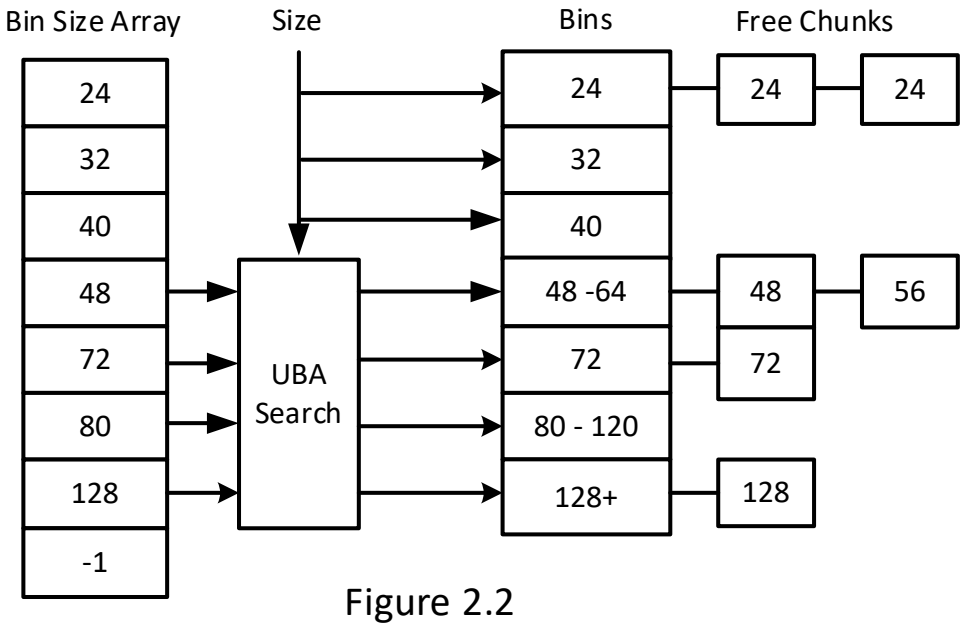

Figure 2.2

inuse and free chunks

An *inuse chunk* is one that has been allocated and is currently being used by an application. Its CCB consists of a forward link (fl) to the next chunk and a backward link + flags (blf) to the previous chunk:

```
fl      physical forward link
blf     physical backward link + flags
```

Since all chunks are 8-byte aligned, the 3 low bits of link pointers are available for flags. The flags in blf are: EH_SSP (bit 2), EH_DEBUG (bit 1) and EH_INUSE (bit 0). SSP is the Spare Space Pointer flag –

see the **spare space** section, below. DEBUG is the debug chunk flag – see the **debug chunk** section, below. INUSE is the inuse flag which indicates that the chunk is inuse vs. being free.

A *free chunk* is one that is available to be allocated. Its Chunk Control Block (CCB) has the following fields:

```
fl      physical forward link
blf     physical backward link + flags
sz      chunk size, in bytes
ffl     free forward link
fbl     free backward link
binx8   bin number times 8
```

The first two fields are the same as for inuse chunks and are for the heap physical structure. The chunk size, sz, is used by heap services. The last three fields are used by bins. ffl and fbl are used to link a free chunk into a bin free list and binx8 is the bin number x 8. This CCB requires 24 bytes and it establishes the minimum chunk size as 24 bytes. When a free chunk is allocated, the last four fields of the CCB (i.e. last 16 bytes) are overwritten by the data block. This establishes the minimum data block size for an inuse chunk of 16 bytes. Since free chunks are not in use, the CCB does not contribute to overhead.

Note: inuse chunk pointers are defined as CCB_PTR. Hence when looking at an inuse chunk via a debugger, it looks like it has the last 4 fields. This is not the case — they are data words.

special chunks

eheap has 4 special chunks: Start Chunk (SC), Donor Chunk (DC), Top Chunk (TC), and End Chunk (EC). These special chunks are never put into bins.

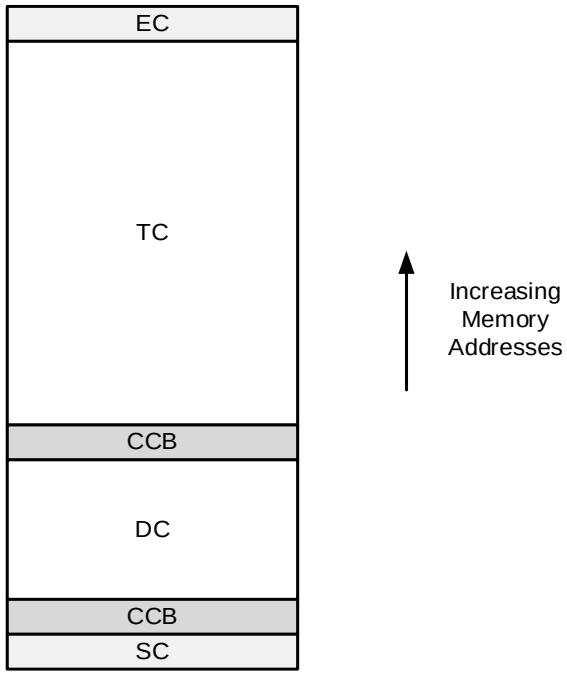

Fig 2.3

As shown in Figure 2.3, after heap initialization the heap consists of SC, EC, TC, and DC, if the dcsz parameter in eh_Init() is non-zero. SC and EC are permanently-allocated 8-byte, inuse chunks that have

no data. They mark the ends of the heap. The size of DC is specified by dcsz in eh_Init(). The size of TC is what is left after the other chunks. DC is usually much smaller than TC, but it must be at least 24 bytes. If DC == 24, mode.fl.use_dc is turned OFF and DC is not used.

Immediately after initialization, all free heap space is in DC and TC. Until bins begin filling up due to eh_Free() operations, smaller, SBA-size chunks will come from DC and larger chunks will come from TC. This results in small chunks being in lower heap and large chunks being in upper heap. This helps to reduce fragmentation caused by small, inuse chunks, getting between larger free chunks, thus blocking them from being merged.

It also helps *localization*. In heap theory, localization, is the attempt to ensure that chunks allocated close in time are physically close, which tends to increase cache hits. Of course, in time, there will be migration of chunks into the other region due to depletion of DC, merging of small chunks, and splitting of large chunks.

When DC becomes too small, it can be replenished by putting what is left into a bin, and then allocating a bigger chunk to it. The strategy for doing this is up to the user. Alternatively, mode.fl.use_dc can simply be turned OFF.

chunk comparisons

Figure 2.4 compares free (A), inuse (B), and debug chunks (C) of the same size. Some of the concepts in this figure have not yet been introduced, but it is convenient to see how the three types of chunks compare at this point. Notice that all have a forward link, fl, a backward link, bl, and flags s, d, and i, and these are all in the same positions. s represents the EH_SSP flag, which is set if there is spare space in the chunk; d represents EH_DEBUG, which is set for a debug chunk; and i represents the EH_INUSE flag which is set for an inuse or debug chunk.

Figure 2.4A is a free chunk. sz is its size, ffl and fbl are used to link it into a bin, binx8 is the bin number times 8 of the bin. Above this is free space that is available for allocation. The dotted line indicates spare space from the chunk above, which has been merged into the free space of this chunk.

Figure 2.4B is an inuse chunk. Note that data has overwritten the sz, ffl, blf, and binx8 fields of the free chunk. Thus, the data chunk has only 8 bytes of overhead. Above the data is spare space from the chunk above. Note that the spare space pointer, ssp, in the last word of the spare space points to its start.

Figure 2.4C is a debug chunk. sz is its size, time is when it was allocated (etime), onr is the task or LSR which allocated it, and fences are known patterns that surround and protect the data block. As shipped, the fence pattern is 0xAAAAAAA3. The last two bits must be 1. This enables a Free operation, which is given only bp, to determine if the CCB below is a debug CCB or an inuse CCB. The fence above onr is part of the debug CCB. The number of other fences is determined by EH_NUM_FENCES in eheap.h. Note how much the data space is reduced in the debug chunk vs in the inuse chunk.

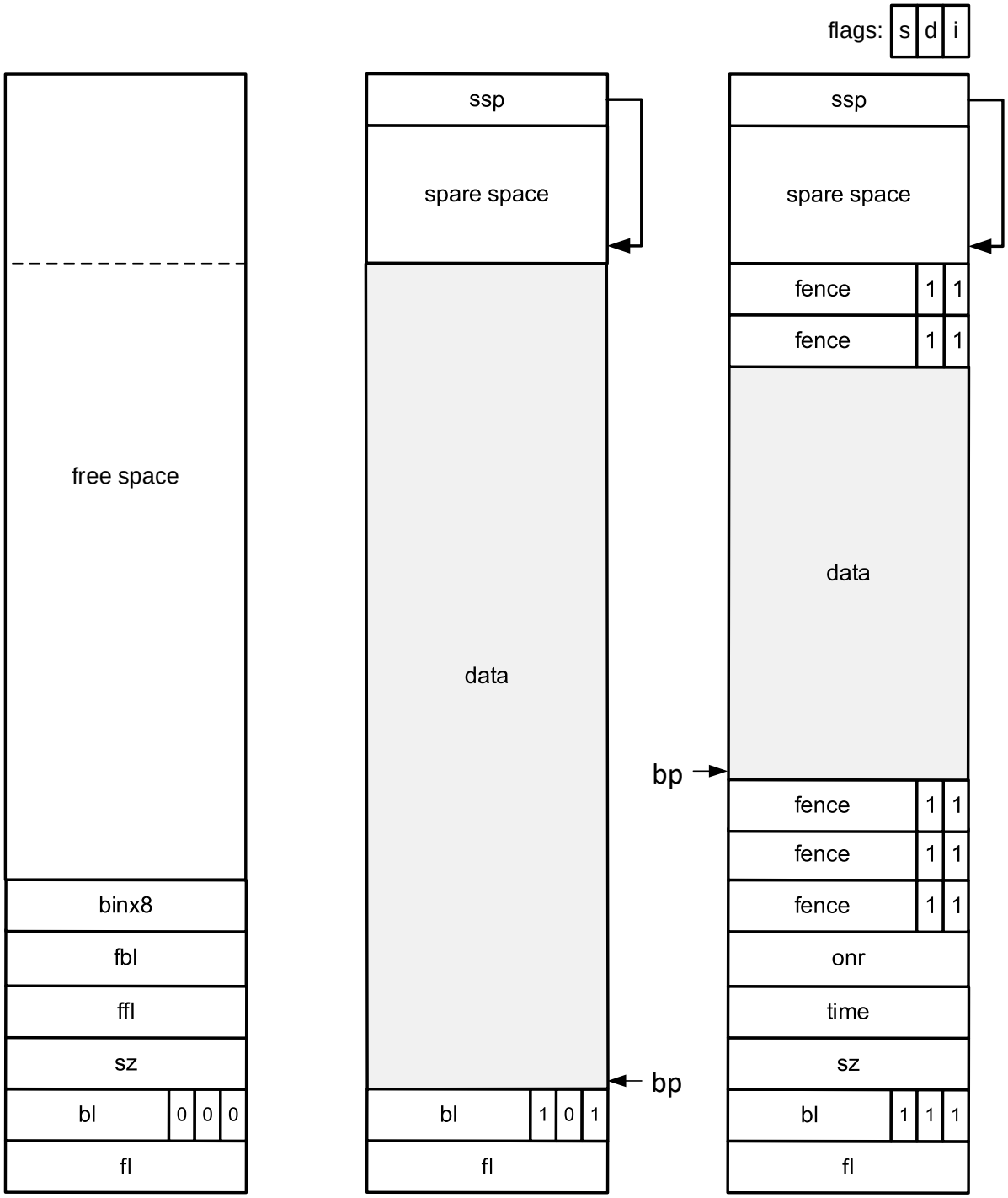

Fig 2.4A Fig 2.4B Fig 2.5C

Chapter 3 Setup

required structures and variables

For each heap the following need to be defined:

```
u32 const binsz[] =
/* bin   0   1   2   3   4   5   6   7   8   9   10   11   12 */
       {24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, \
/* bin  13   14   15   16   17   18   19   20    21    22    23 */
       128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, \
/* bin  24    25    26    27    28       end */
       1536, 1664, 1792, 1920, 2048, 0xFFFFFFFF};

#if defined(__IAR_SYSTEMS_ICC__)
#pragma data_alignment = SB_CACHE_LINE /* cache align in RAM */
#endif

HBCB  bin[(sizeof(binsz)/4)-1];   /* main heap bins */
EHV   hv; /* heap variable array */

#if EH_STATS
u32   bnum[(sizeof(binsz)/4)-1]; /* number of chunks per bin */
u32   bsum[(sizeof(binsz)/4)-1]; /* sum of chunk sizes per bin */
#endif

#if EH_BP
BPCB bpcb[2];
#endif
```

binsz[] defines the bin structure for the heap. In the above example, the upper numbers are bin numbers and the lower numbers are the bin sizes. Bins 0 through 12 comprise a small bin array, SBA, which has bin sizes from 24 bytes to 120 bytes, in 8-byte increments. Above the SBA, starting at bin 13, are 15 bins covering the range from 128 bytes to 2040 bytes (the last size in bin 27). Each of these bins covers a range of 128 bytes with 16 chunk sizes. The top bin starts at 2048 bytes and covers it and all larger sizes. The last entry, -1, terminates the bin size array. This is the standard bin configuration shipped with SMX, which is intended to provide a good starting heap for most systems.

Bins can easily be added, removed, or resized simply by changing the constants in binsz[] and recompiling. Then running eh_Init() initializes the bins and internal heap variables from it. Hence, the entire heap configuration can be changed by changing a few constants in binsz[]. This makes it easy to experiment with different bin configurations to see which produces the best performance. Bin arrays of 5 bins, even 1 bin are practical for small heaps — see **small bin arrays** in the Optimization chapter.

Note in the above example that binsz[] is put into SRAM for testing and into ROM for shipment. The latter is for improved reliability. However, if available ROM is too slow, binsz[] may need to be put into SRAM.

bin[] is the actual bin array. Its size is determined from binsz[]. Each entry consists of a free forward link, ffl, and a free backward link fbl. **hv** is the heap variable structure. It contains all variables used by a heap. These are defined in EHV in eheap.h. It requires about 124 to 140 bytes, depending upon options. **bnum[]** and **bsum[]** are required if heap statistics are enabled. **bpcb[2]** is an array of two pool control blocks

required if block pools are enabled. Note: These names are used in discussion that follows. Obviously, each heap requires unique names since all heaps and partitions are linked together.

initialization

The initialization code for each partition that requires a heap should contain the above plus code such as the following:

```
memset((void*)&hv, 0, sizeof(EHV));
hv.bszap = (u32*)binsz;
hv.binp  = (HBCB*)bin;
eh_Init(hsz, dcsz, haddr, &hv, (EH_CM | EH_FILL | EH_EDA | EH_EM | EH_PRE));
```

This code clears the hv structure, initializes the bzsap and binp fields, then calls eh_Init(), which initializes the heap and returns the heap number, hn. hsz is the size of the heap in bytes, dcsz is the size of the donor chunk in bytes, haddr is the starting address of the heap, and &hv is the address of the hv structure. It then enables chunk merge, fill, error display all, error manager, and preemption. Other mode flags are turned off. EH_PRE means that the heap is protected from preemption by some mechanism outside of eheap (see below). Space for the main heap is likely to be allocated by the linker command file. Space for smaller, dedicated heaps may be allocated from the main heap or by the linker command file.

eheap maintains an array of hv pointers, eh_hvp[EH_NUM_HEAPS] and hn is the index into it. Hence the hv structure is accessed as eh_hvp[hn]. See eh_Init() description in Appendix A for more information.

multitasking

eheap does not have inherent multitasking protection. If it is being used in a multitasking environment, shell functions must be added to provide preemption protection for each heap that requires it. The following is an example of a shell function from smx:

```
void* smx_HeapMalloc(u32 sz, u32 an, u32 hn)
{
   void* bp;
   if (!smx_HeapEnter(sz, an, hn, SMX_ID_HEAP_MALLOC))
      return NULL;
   bp = eh_Malloc(sz, an, hn);
   smx_HeapExit((u32)bp, hn, SMX_ID_HEAP_MALLOC);
   return bp;
}

BOOLEAN smx_HeapEnter(u32 p1, u32 p2, u32 hn, u32 id)
{
   MUCB_PTR mtx;
   if ((mtx = (MUCB_PTR)eh_hvp[hn]->mtx) != NULL)
   {
      if (!smx_MutexGet(mtx, smx_htmo))
         return FALSE;
   }
   return TRUE;
}
```

```
void smx_HeapExit(u32 rv, u32 hn, u32 id)
{
   if (eh_hvp[hn]->mtx != NULL)
      smx_MutexRel((MUCB_PTR)eh_hvp[hn]->acop);
}
```

If smx_HeapEnter() passes, eh_Malloc() is called, and then smx_HeapExit() is called to signal the mutex, thus allowing another task access to the heap. Note that MutexGet() and MutexRel() are skipped if mtx == NULL. This speeds up operation for heaps that are accessed by a single task or by tasks having the same priority and thus unable to preempt each other.

In the heap hn initialization code:

```
if (eh_hvp[hn]->mode.fl.pre)
{
   if (eh_hvp[hn]->mtx == NULL)
   {
      eh_hvp[hn]->mtx = smx_MutexCreate(1, 0, aco_names+(hn*ACO_NAME_LEN));
   }
   if (eh_hvp[hn]->mtx == NULL)
      hn = -1;
}
else
{
   eh_hvp[hn]->mtx = NULL;
}
smx_htmo = 10;
```

If the mode.fl.pre flag is on and if no mutex has been created, a mutex is created and its handle is loaded into eh_hvp[hn]->mtx. If the mode.fl.pre flag is off, no mutex is created and NULL is loaded into eh_hvp[hn]->mtx.

In the above example, tests for all heaps share a single timeout, smx_htmo = 10 ticks. If this timeout is exceeded, smx_HeapEnter() returns FALSE to smx_HeapMalloc(), which then returns NULL to the caller. Thus, it is very important to test the returned block pointer before using it. For smx, if NULL is returned, checking smx_ct->err will reveal if there was a timeout or an error such as insufficient heap. Other RTOSs, no doubt, work similarly. For simplicity, error handling code has been omitted in the above examples. See **error reporting** in the Reliability chapter for more information on error handling.

Chapter 4 Operation

normal block allocation

In the following discussion, *csize* is the minimum chunk size needed for the requested block size. It is determined by adding CHK_OVH to the requested block size. CHK_OVH is 8 bytes for an inuse chunk, but it could be 40 bytes, or more, for a debug chunk (see **debug mode** in the Debugging chapter). In what follows, a *small chunk* is a chunk of SBA size and a *large chunk* is larger than SBA size.

Allocation order for **small chunks** is: selected SBA bin -> DC -> larger bin -> TC. The correct SBA bin is quickly located via the simple formula: binno = csize/8 - 3. If occupied, the first chunk in this bin is dequeued and a pointer to its data block is returned. This is the fastest heap allocation possible with eheap.

If the selected SBA bin is empty, calving a chunk from DC is nearly as fast. The size of DC is set during initialization. It is desirable to make DC large enough to handle the expected maximum number of small chunks. However, there may not be enough RAM for all expected small chunks plus all expected large chunks, so a compromise may be necessary.

If DC is exhausted, the chunk is taken from the next-occupied larger bin. This bin most likely will be a larger SBA bin, but it might be a UBA bin. If that fails, the chunk is taken from TC. Following startup, most small chunks will come from DC and be freed to SBA bins. After running for a while, virtually all small chunks should come from the SBA. Allocations from the SBA, DC, and TC are exact fits. Allocations from larger bins may be larger than csize.

Allocation order for **large chunks** is: selected upper bin -> larger bin -> TC. The selected bin is found with a binary search on binsz[]. If the selected upper bin is occupied, the first big-enough chunk is taken. If the bin is a small bin[1], this will be the first chunk in it. If the bin is a large bin, the bin's free list is searched until a big-enough chunk is found. If one is not found, then the first chunk of the next larger occupied bin is taken (this is big enough, by definition). Failing this, the chunk is calved from TC.

Initially the size of TC is determined by sz - dcsz – 24 in eh_Init(). A goal of the above algorithms is to keep TC as the source of last resort for big chunks. If TC is not big enough for csize, auto-recovery will be attempted if mode.fl.auto_rec is ON. Otherwise, or if auto-recovery fails, EH_INSUFF_HEAP is reported and NULL is returned. Methods to deal with heap allocation failures are discussed in the Reliability chapter.

When a chunk is allocated from a large bin, it is likely to be larger than necessary. The unused space following the data block (or fences above it for a debug chunk) is called *spare space*. If the spare space is large enough (see **chunk splitting**, below), it is split off into a new free chunk. Otherwise, the Spare Space Pointer flag, EH_SSP, is set in blf (bit 2) and the spare space pointer, ssp, is loaded into the last word of the spare space (which is the last word of the chunk). See Figure 4.1.

aligned block allocation and spare space handling

All blocks from the heap are automatically 8-byte (an = 3) aligned. EH_ALIGN enables block allocations with larger alignments up to EH_MAX_AN. Only power-of-two alignments are implemented. In order to prevent errors, the alignment parameter in allocation services is the alignment number, an. It is the

[1] The UBA may contain small bins.

exponent of two corresponding to the alignment. For example, an = 5 corresponds to 2^5 = 32 bytes. As shipped, EH_MAX_AN is 12, which equivalent to 4096-byte alignment. If an > 3, but EH_ALIGN == 0, EH_INV_PAR is reported and NULL returned.

In the following steps and diagrams, *ICCB* is an *Inuse Chunk Control Block* and *FCCB* is a *Free Chunk Control Block*. Note that an FCCB is larger than an ICCB as shown in Figure 2 and the figures below. If EH_ALIGN and an > 3, an aligned block search is performed via the following steps:

1. Find the first large-enough free chunk for the desired block size, sz.
2. Find the first alignment boundary, 2^an, inside the potential inuse chunk's data block. (Note: alignment boundary could be inside of the current FCCB.)
3. Test if the remainder of the chunk is >= sz.
4. If not, go on to the next large-enough chunk.
5. When an acceptable chunk is found, put its ICCB below the boundary – i.e. just below the aligned data block.
6. The resulting space below the ICCB is called *free space*, and it is handled as follows:
    a. If the preceding chunk is free, combine the free space with it.
    b. Else, if the free space is large enough, make it into a free chunk.
    c. Else, combine the free space with free space at the end of the preceding inuse chunk, and if the result is big enough, make it into a free chunk.
7. Split off space after the block, if large enough for a free chunk, else make it free space.

The following diagrams illustrate the above operations. Spare space is merged into a free prechunk, as shown in Figure 4.1. BP points to the aligned data block. The spare space flag in the top ICCB is 0.

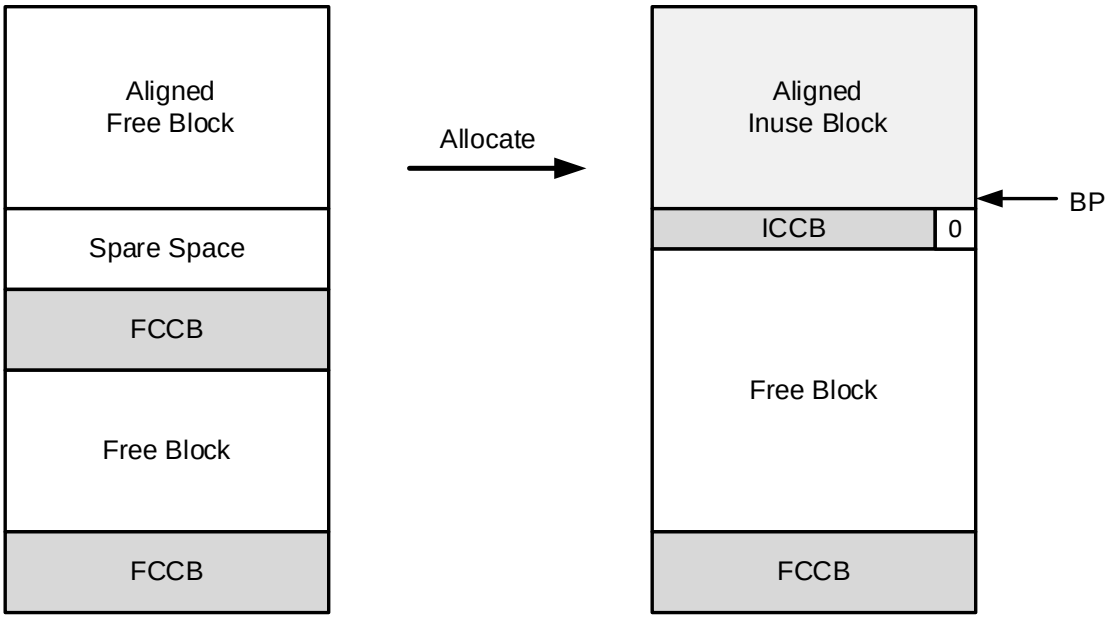

Fig. 4.1A

Fig. 4.1B

If the prechunk is inuse and the spare space is large enough, it is made into a new free chunk, as shown in Figure 4.2.

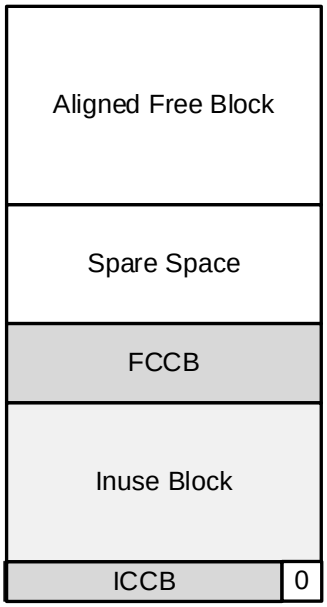

Fig. 4.2A

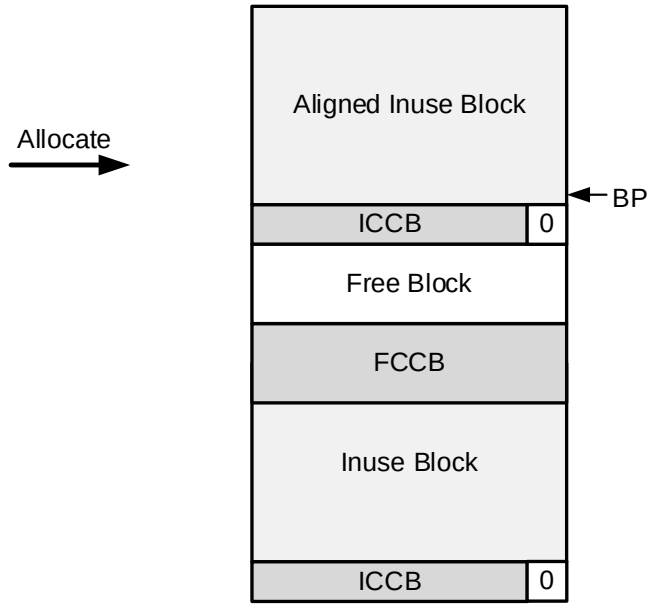

Fig. 4.2B

Otherwise spare space is added to the spare space, if any, in the prechunk, as shown in Figure 4.3. In this case there was no spare space in the inuse prechunk. After the allocation there is spare space in the inuse prechunk. Note that the spare space flag is now 1 and the spare space pointer, SSP, in the top word of the spare space points to the start of the spare space.

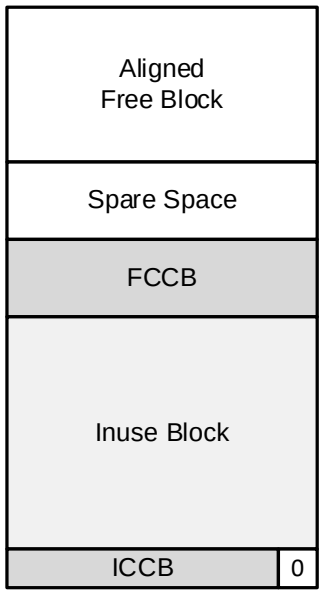

Fig. 4.3A

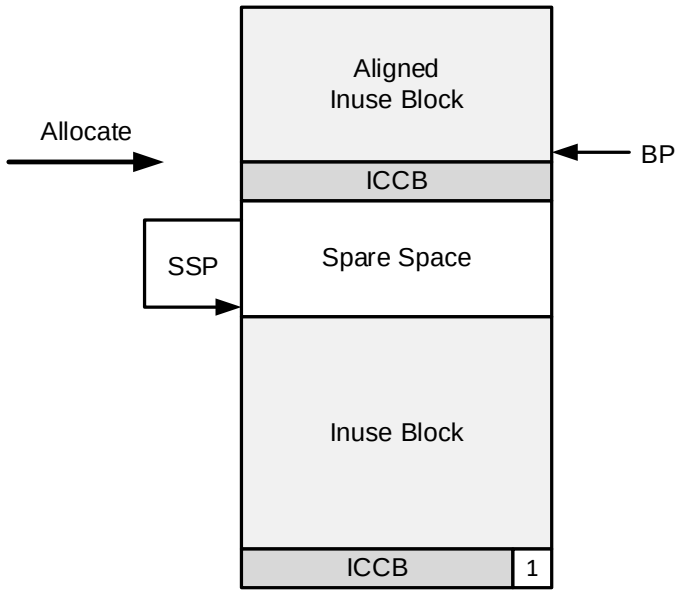

Fig. 4.3B

If the resulting spare space is large enough (> EH_MIN_FRAG), it is split off into a free chunk, as shown

in Figure 4.4. In this case, there was spare space in the inuse prechunk, which was combined with the spare space below the aligned block in the free chunk and the result was big enough for a new free block.

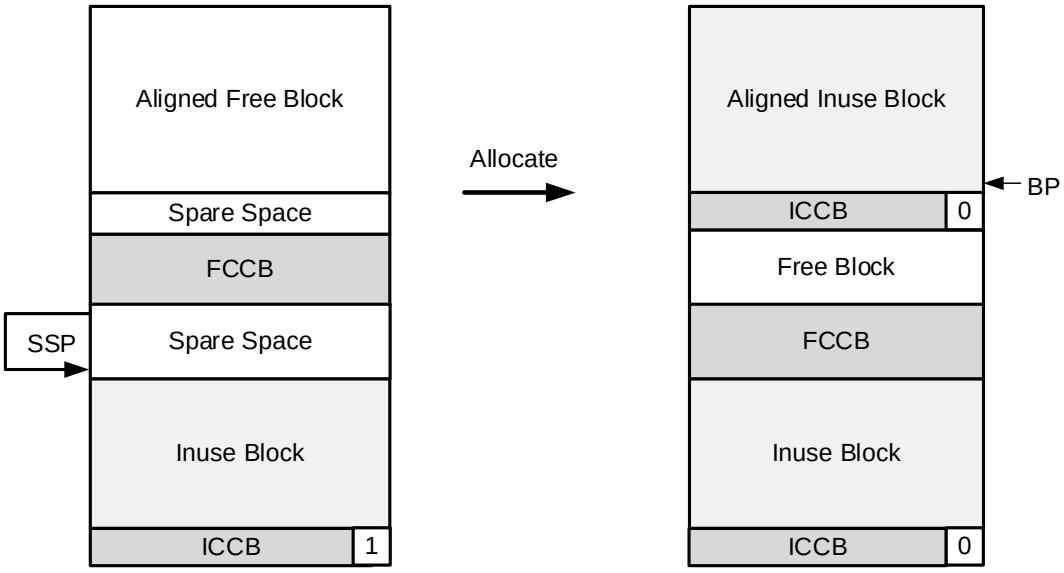

Fig. 4.4A Fig. 4.4B

Figure 4.5 shows what happens if an aligned inuse block is freed, the inuse prechunk has spare space, and EH_SS_MERGE in eheap.h is ON. In this case, the new free block has lost alignment and the inuse prechunk no longer has spare space. Note that its spare space flag is now 0.

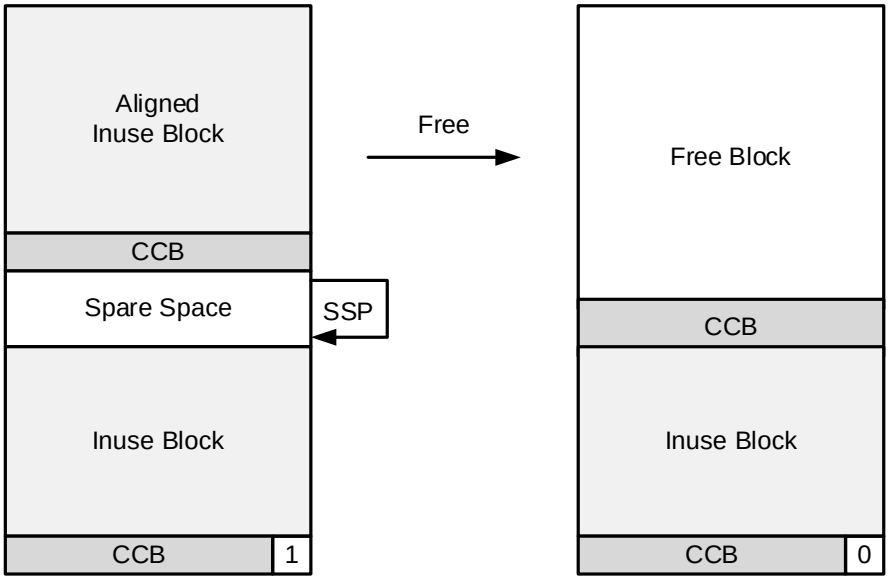

Fig. 4.5A Fig. 4.5B

Figure 4.6 shows what happens if EH_SS_MERGE is OFF in the same example. In this case the new free block is still “aligned”. (It is not actually aligned due to the larger size of its FCCB. However, if it is re-

allocated, the inuse data block would be aligned due to the smaller size of its ICCB, as shown in Figure 4.6A.

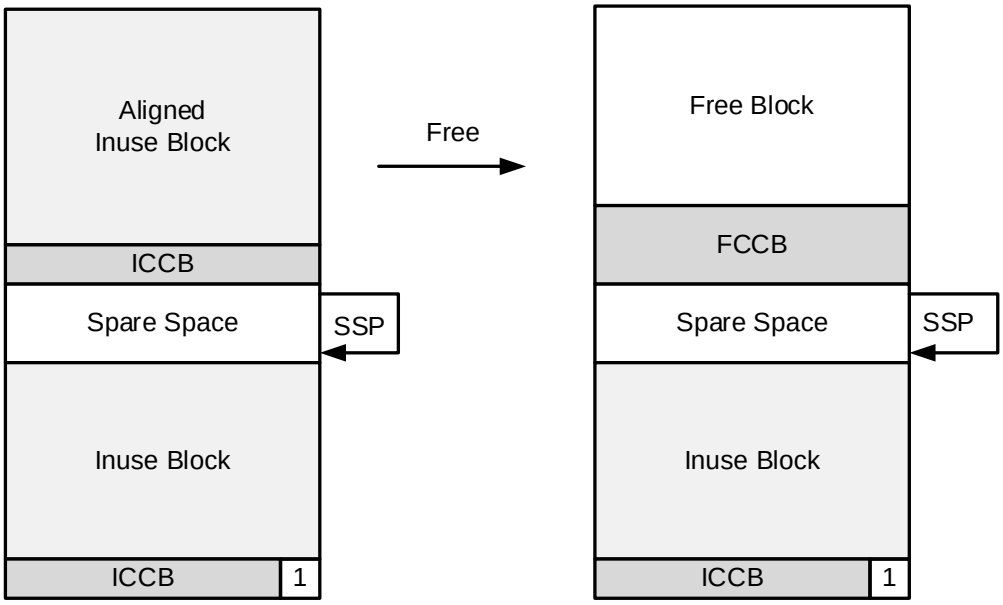

Fig. 4.6A Fig. 4.6B

EH_SS_MERGE OFF allows aligned blocks to remain aligned. This could improve performance for a system that frequently requires blocks with the same alignments. On the other hand, it potentially increases internal fragmentation, which may be a problem in tight heaps.

An example where aligned blocks may be beneficial is a heap in DRAM. In this case, cache-line aligned blocks could significantly reduce access times to data blocks, without greatly increasing block allocation times – especially if EH_SS_MERGE is OFF causing cache-line aligned data blocks to accumulate over time. Note that the ICCB for the aligned data block is guaranteed to be entirely within the previous cache line. Hence, heap operations, themselves, will be faster, on average.

MPU region block allocation

MPU *region block allocations* are used to allocate Cortex v7M MPU region blocks. A Cortex v7M MPU region must be a power of two in size and must be aligned on its size. A region block must be aligned on the subregion size, and all subregions necessary to hold the block must be within the region. To allocate a region block, EH_R is added to the alignment number to form the *an* parameter for eheap functions doing allocations.

The following steps are performed to allocate an MPU region block:

1. Determine the *region size* as the next larger power of two. For example, if sz = 630, then region size = 1024.
2. Determine the subregion size and the number of contiguous subregions needed. In the example, subregion size = 128, and 5*128 = 640 > 630, so N = 5.
3. Do aligned search steps 1 - 3 with alignment = 128 and size = 640.
4. Verify that all N subregions are in the same region – i.e. find the next region boundary (e.g. multiple of 1024) and verify that the last subregion ends before it.

5. Do aligned search steps 5 -7.

This results in a subregion-aligned v7M region block that is contained in contiguous subregions within a region, as shown in Figure 4.7:

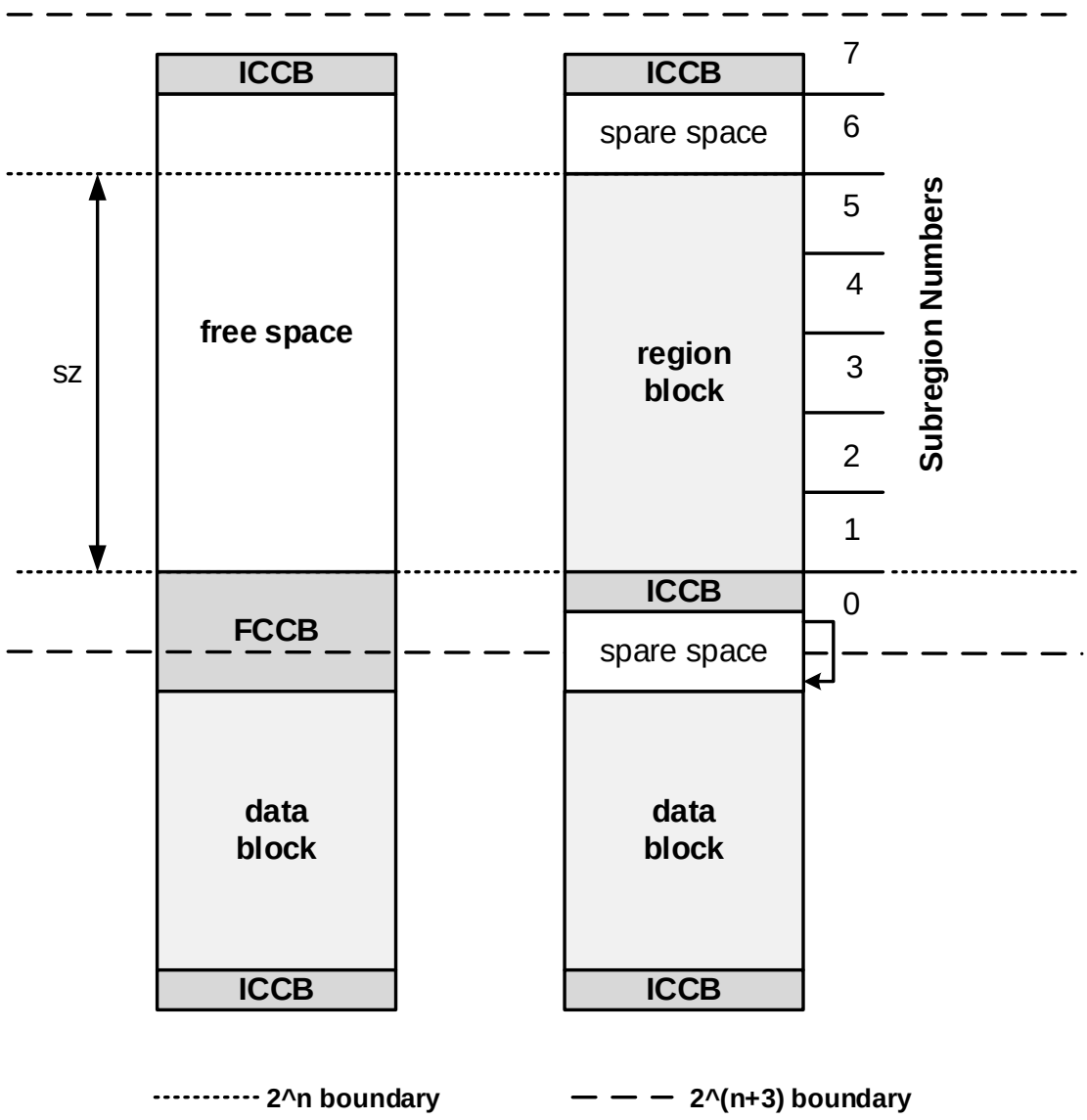

Fig. 4.7

In this example, subregions 1-5 will be enabled, and subregions 0, 6, and 7 will be disabled in the region block. Note how the disables protect the surrounding heap CCBs and spare space. Because the region block is only subregion-aligned, instead of region-aligned, it is much easier to find and causes less heap disruption. See the SecureSMX User's Guide for more discussion.

finding the next larger occupied bin

As discussed above, it is sometimes necessary to find the next larger occupied bin. eheap uses a 32-bit bin map, *bmap*, to do this. bmap has one bit per bin. When a chunk is loaded into a bin, the bin's bmap bit is set, and when a bin becomes empty, its bmap bit is cleared. A simple calculation magically converts bmap

into the bin number of the next larger occupied bin. If bmap is 0, there is no next larger occupied bin and the chunk is calved from TC.

eheap has one 32-bit bmap for all bins. Hence, the maximum number of bins is limited to 32. This can be increased to 64, or more, if needed. However, there is a small performance penalty, so bmap is being left at 32, for the time being, which may be enough for embedded systems.

chunk splitting

A found chunk may often have more space than is needed for the data block. The additional space is called spare space, as explained in the allocation sections above. If the spare space is greater than or equal to EH_MIN_FRAG (eheap.h) bytes, then it is split off into a new free chunk. This new free chunk is put into the correct bin for its size, unless the chunk above it is free and mode.fl.cmerge is ON. In that case, the two chunks are merged and put into the appropriate bin unless the merged chunk is DC or TC. If a merger was made with DC, dcp is updated; if a merger was made with TC, tcp is updated.

As shipped, EH_MIN_FRAG is 40 bytes for inuse chunks. The purpose of this constant is to avoid populating the heap with an excessive number of small chunks. The number of small chunks may exceed the needs of the application, and some sizes may not be needed at all. This can lead to allocation failure due to excessive *external fragmentation*. Also, chunk splitting adds overhead to chunk allocation, and it may also add overhead to chunk free due to merging back unused free fragments.

On the other hand, if EH_MIN_FRAG is too large, then excessive *internal fragmentation* may build up in the form of spare space, as described above. This too can lead to heap failure, since the spare space cannot be allocated. Experimentation may be necessary to find an optimum size. EH_MIN_FRAG must be at least as big as the smallest chunk size used in the heap. If necessary, a min_frag field will be added to EHV in the future so it can be customized to each heap.

Note that large bin sorting helps to reduce the fragmentation problem by achieving best-fit allocations from large bins. See the **bin sorting** section in the Optimization chapter.

block free

If merging is enabled (mode.fl.cmerge == ON), eh_Free() merges the chunk and its prechunk, if free, then merges the chunk and its postchunk, if free. Chunks to be merged, except DC or TC, are removed from their bins before merging them. Then the bin for the final free chunk is found and the chunk is put into that bin, unless it is DC or TC. If a merger was made with DC, dcp is updated; if a merger was made with TC, tcp is updated. Only upward mergers into DC or TC are permitted and those chunks are never put into bins. heap_used is reduced by the size of the freed chunk

For a small bin, the freed chunk is put at the start of the bin's free chunk list. For a large bin, if the freed chunk is smaller or equal to the first chunk in the bin, it is put ahead of it. Otherwise it is put at the end of the bin. This aids the large bin sort algorithm – see the **bin sort** section in Optimization chapter.

If EH_SS_MERGE (Spare Space Merge enable), the prechunk is inuse, and its EH_SSP (spare space pointer) flag is set, the spare space in the prechunk is merged with the freed chunk. This is done to reduce internal fragmentation, but it also spoils the freed chunk's alignment. This can be avoided by setting EH_SS_MERGE to 0. Then the freed chunk will retain its alignment.

If either hsp or hfp (heap scan or fix pointer) was pointing at the freed chunk, and it was merged with the prechunk or spare space, the pointer is backed up to the new chunk. If a chunk is put at the end of a large bin, the bsmap bit for that bin is set, indicating that the bin needs to be sorted. See **heap scan** and **bin sorting** sections below for more explanation.

deferred merging

Due to the double linking of chunks in eheap it is possible to defer merging of free chunks. Some heaps, such as dlmalloc, have only a chunk size field in all chunks and a free chunk size field in the last word of free chunks. This last word doubles as the first word of the following inuse CCB. For a free CCB, this word is the last data word of the preceding chunk, which must be an inuse chunk. As a consequence, free chunks cannot be adjacent and must always be merged, when they are. This undermines the utility of bins. eheap merging is controlled by mode.fl.cmerge, which is set by:

```
eh_Set(EH_ST_MERGE, ON/OFF);
```

To promote rapid bin filling, cmerge is OFF following heap initialization.

Keeping cmerge OFF is recommended to avoid *leaky bins.* For example, a 24-byte chunk is freed to bin 0, and a physically adjacent 48-byte free chunk resides in bin 3. If cmerge is ON, these chunks will be merged into a 72-byte chunk, which will be put into bin 6. Bins 0 and 3 have *leaked* up to bin 6. This is not conducive to best performance if the application needs 24 and 48-byte chunks and not 72-byte chunks.

Even when running with cmerge ON, bin leakage occurs due to chunk splitting. In this case, a larger-than-needed chunk is taken from a bin, an exact-fit chunk is split off, and the remnant is put into a lower bin. When the allocated chunk is freed, if the remnant is inuse, the chunk will probably be put into a lower bin. But it may be that the original size is what is most needed. Hence the larger bin has leaked down to smaller bins. As explained above in the **chunk splitting** section, this problem can be reduced by increasing the size of EH_MIN_FRAG.

However, the main problem with deferred merging is the possibility of excessive external fragmentation resulting in heap allocation failures. eheap does implement auto and manual recovery mechanisms (see **heap recovery** in the Reliability chapter), but it is best to prevent excessive fragmentation in the first place. As noted previously, mode.fl.cmerge can easily be turned ON and OFF, so all that is needed is an effective algorithm to do so. In cases of very tight heaps (i.e. little excess space) always OFF may be the best algorithm. This may also be the choice for very conservative designs. However, the downside is reduced performance.

For further discussion of merge control, see **merge control** in the Optimization chapter.

integrated block pools

Figure 4.8 shows how the 8-byte and 12-byte block pools are integrated into eheap.

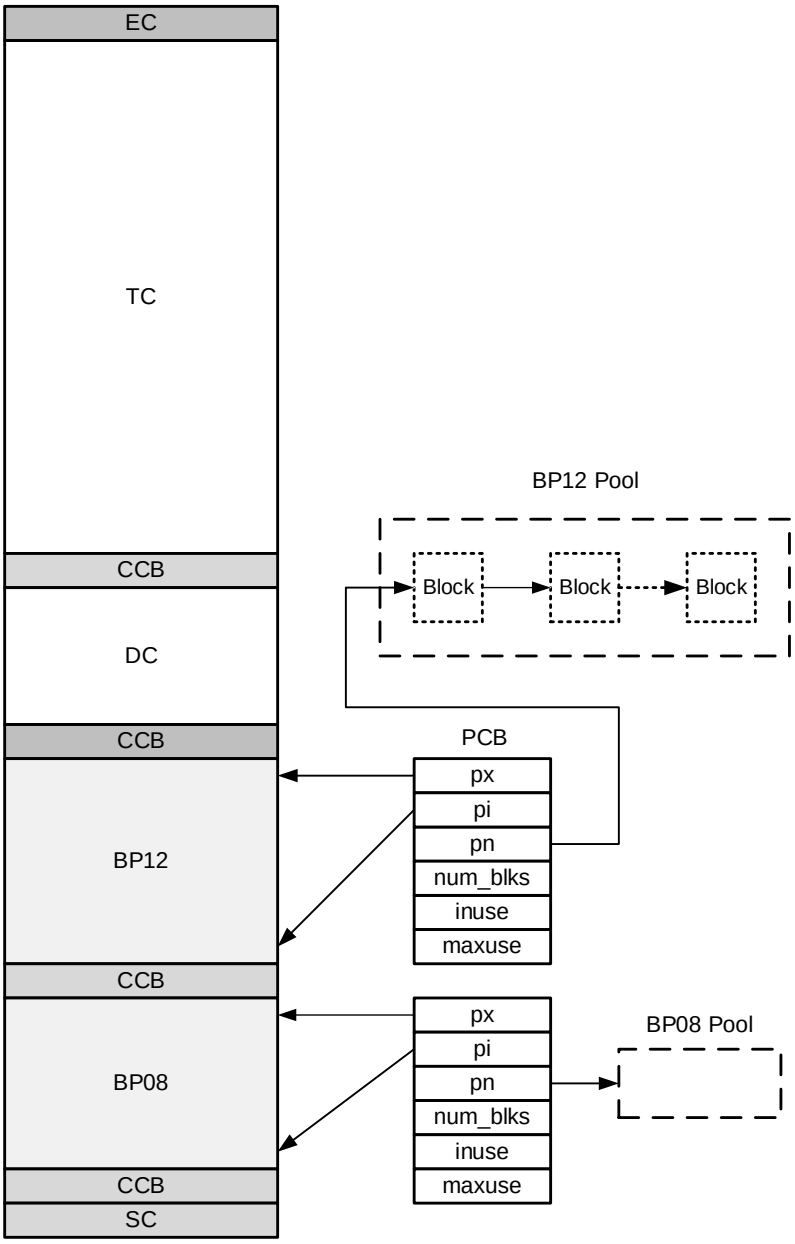

Fig. 4.8

If C++ is being used, there is a tendency to allocate a very large number of very small blocks from the heap for objects. For this reason, integrated block pools have been added to eheap for 8- and 12-byte blocks. Block pools are enabled by EH_BP = 1 in eheap.h. If EH_BP is 0, all block pool code is omitted. Block pools are faster and more memory efficient than heaps. The advantage of integrating block pools with eheap is that if a pool becomes empty or cannot satisfy an alignment, the block is taken from the heap. Thus, the block pool can be sized to meet normal demands with the heap backing it up for peak or unexpected demands.

As explained in the Setup chapter, it is necessary to allocate an array of two pool control blocks (PCBs) of type BPCB – one for the 8-byte block pool and one for the 12-byte pool. The fields are as follows:

```
u32      num_blks;   /* number of blocks in pool */
u32      inuse;      /* blocks in use of this size */
u32      maxuse;     /* maximum blocks in use from this pool */
void*    pi;         /* pointer to pool start block */
void*    pn;         /* pointer to next free block = NULL if none */
void*    px;         /* pointer to pool end block */
```

bpcbp points to the array of the two PCBs for heap hn. All block pool operations check that bpcbp is not NULL before proceeding.

The block pools are initialized by eh_Init(), which allocates space from the heap for 8- and 12-byte block pools at the bottom of the heap between SC and DC. Figure 4.8 shows a fully-configured heap following initialization with both block pools. Each PCB has a num blocks field. This is the only field that must be loaded by the user. If num is 0, no pool is created. Hence, block pools can be implemented only for heaps that need them and either block pool can be omitted. Otherwise, a pool of num blocks is created and its PCB is initialized. PCB.pn links to the first block, the first word of each block links to the next block, and the last block's first word == NULL.

Blocks are allocated by eh_Malloc(). Prior to searching the heap, block pool allocation proceeds as follows: If 0 < sz <= 8, there is a block in the 8-byte pool (i.e. pn != NULL), and an <= 3, the block is taken. If 8 < sz <= 12, there is a block in the 12-byte pool, and an <= 2 or bp is 8-byte aligned, the block is taken. The reason for the latter complexity is that some 12-byte blocks are 8-byte aligned and others are only 4-byte aligned. To avoid surprises, specify an <= 2 for 12-byte block allocations. When a block is taken, PCB.inuse is incremented, PCB.maxuse is updated, if PCB.inuse is larger, PCB.pn is set to point to the next block, and bp is returned. Otherwise, sz is increased to 16, and allocation comes from the heap.

Blocks are freed by eh_Free(bp), which proceeds as follows: if bp is less than fhcp (first heap chunk pointer), the block pointed to by bp is freed either to the 8-byte pool or to the 12-byte pool, depending upon which pool bp points at. The freed block is put at the start of the pool's free list, pn is updated to point to it, and TRUE is returned. If bp >= fhcp, the block is freed to the heap.

Hence, if a 16-byte block were allocated from the heap because the desired block could not be allocated from a block pool, due to the pool being empty or due to misalignment, it will look to the application as though it came from the pool. The application may thus be dealing with a mixture of 8-, 12-, and 16-byte blocks. When freed, blocks will be transparently returned to where they came from. If, for some reason, 8- or 12-byte blocks with alignments of 16 or greater are needed, they will come from the heap, not from block pools.

heap modes

Throughout this manual there is frequent mention of heap modes. These are single bit flags in hv.mode.fl. eheap provides direct access to and control of heap modes. See HMODE in eheap.h for a list of all modes. As opposed to configuration constants which apply to all heaps in a multi-heap system, modes are heap-specific. Hence one heap may have a mode on while another heap has it off.

eh_Peek(par, hn) is used to get heap modes. par is of type EH_PK_PAR. See eh_Peek() in Appendix A for a list of parameters. The return values are ON (1) and OFF (0). eh_Peek() returns -1 and reports EH_INV_PAR, if par is not valid.

eh_Set(par, val, hn) is used to set heap modes. par is of type SMX_ST_PAR. See eh_Set() in Appendix A for a list of parameters. val == ON or OFF.

Heap modes are discussed in detail in places where they are used.

heap statistics

The following are accumulated as the heap runs:

hhwm Heap high water mark. Records maximum number of bytes used.

hused Number of bytes of the heap currently allocated.

These are in hv and accessible via a debugger or via smxAware. If hhwm becomes nearly equal to the heap space, the latter should be increased, if possible. It is advisable to do long system runs in order to make this adjustment.

If EH_STATS (eheap.h) is 1, the following heap statistics are accumulated:

smx_bnum[] Number of chunks in each bin

smx_bsum[] Sum of chunk sizes in each bin

These arrays are allocated in application memory for heap hn and their addresses are put into bnump and bsump in hv for heap hn during initialization. They can be viewed through a debugger watch window to provide some of the same information as the smxAware heap window. (See smxAware User's Guide.) These statistics could be used for heap seeding and for cmerge control – e.g. if all bins are adequately populated, cmerge could be turned ON.

Chapter 5 Debugging

Debugging often makes or breaks projects and heap usage may frequently be the crux of the problem. Understanding how the heap works and having good visibility into its operations is important, because heap operations can be complicated and difficult to understand.

debug mode

Debug mode is intended to help find problems such as buffer overflows and memory leaks. When mode.fl.debug is ON in hv heap hn is in *debug mode*. In this mode, all chunks allocated are debug chunks rather than inuse chunks. This includes not only eh_Malloc() and eh_Calloc(), but also eh_Realloc(), even if the initial chunk was an inuse chunk.

A debug chunk is a special form of an allocated chunk (in fact, its INUSE flag, blf bit 0, is set). Its format is as follows:

```
fl      physical forward link
blf     physical backward link + flags
sz      chunk size, in bytes
time    time of allocation (etime)
onr     task or LSR that allocated this chunk
fence   first fence

fences  pre-block fences

data block

fences post-block fences
```

The part above the pre-block fences is the Chunk Debug Control Block, CDCB. fl and blf in it are common to all chunks, except that its DEBUG flag, blf bit 1, is set. sz is the size of the chunk, not of the block. time is the time when the chunk was allocated. It is set by the callback function, eh_time(). For smx this is:

```
u32 eh_time(void)
{
   return smx_etime;
}
```

onr is the task or other actor that allocated the chunk. It is set by the callback function, eh_onr(). For smx, this is:

```
u32 eh_onr(void)
{
   return (smx_clsr != NULL ? (u32)smx_clsr : (u32)smx_ct);
}
```

Equivalent callback functions are needed for the RTOS or standalone code being used.

The final field in the CDCB is the first fence. A fence is a one-word pattern, EH_FENCE_FILL (0xaaaaaaa3) defined in eheap.h. Any pattern may be used, but bits 1 and 0 in the pattern must be 1, because they are the alternate DEBUG and INUSE flags, which are used when a chunk is accessed via its

data block pointer (e.g. in eh_Free(bp)). These flags are necessary to determine the chunk type in order to determine where the chunk starts.

The number of pre-block and post-block fences is the same and is specified by EH_NUM_FENCES in eheap.h. The purpose of fences is to prevent overflows from damaging the heap and to make overflows easier to spot in the debugger memory window. If EH_NUM_FENCES is odd, the data block will be 4-byte aligned, instead of 8-byte aligned. This could cause a problem for code that expects the block to be 8-byte aligned. It could also alter cache performance. Hence, a problem could show up for the debug version of a chunk and not for the inuse version.

Pre-fences and post-fences are useful to:

(1) Detect block and stack overflows and block underflows.

(2) Show the overflow footprint, in order to help identify its source.

(3) Protect CCBs so a system can continue running if the overflow does not exceed the extent of the fences.

For large block overflows, it might be necessary to surround the blocks with 20 or more fences in order to keep the system running and to see the footprint of the intruder. This could result in 200 bytes of overhead per debug chunk, which may not allow normal heap operation in the RAM available for the heap. In this case, debug mode would probably be turned ON only for one or two suspected tasks or functions.

One way to do this is to turn debug ON at the start of the suspected task or suspected function in the task. In the case of an RTOS, like smx, mode.fl.debug can be turned OFF in the *exit routine* hooked to the task and ON in the *enter routine* hooked to the task. Then debug is ON only when the task is actually running. If the same function is used in other tasks, then those tasks must also be hooked to the same enter and exit routines. The debug mode can be safely set or reset directly in hooked routines, since they cannot be preempted. For example:

```
eh_hvp[hn]->mode.fl.debug = ON/OFF;
```

To find memory leaks, EH_NUM_FENCES can be set to 0, in which case the debug chunk overhead is only the 24 bytes of the CDCB. The time and onr fields in the CDCB are useful to track down memory leaks. For example, the following chunks would be suspect:

(1) An old chunk, unless it has a permanently allocated block.

(2) A chunk allocated by a task that has been deleted or stopped.

(3) An old ISR or LSR chunk since it should have been passed on to a task and freed by the task after it processed the data.

Even if the above are live chunks, they may indicate poor coding practices that should be corrected.

Since debug chunks are typically much larger than inuse chunks, it may not be possible for all allocated chunks to be debug chunks – especially in small heaps. For that reason, they can be selectively controlled by turning the mode.fl.debug flag ON to create debug chunks and OFF to create inuse chunks, using eh_Set() – see Appendix A. This allows using debug chunks in new sections of code that are being debugged and inuse chunks in old sections of code that have already been debugged.

Debug chunks can be freely intermixed with inuse and free chunks. Their main difference is their size. This may cause some different behavior during debugging:

(1) A debug chunk may come from a higher bin than the corresponding inuse chunk.

(2) Allocation of debug chunks is slower than inuse chunks, due to the need to fill in fences and extra fields.

(3) Allocation could be slower due to higher bins not being populated or coming from a large bin vs. coming from a small bin for the inuse chunk.

(4) If an inuse chunk is mistaken for a debug chunk, the fields after blf will be garbage. Be sure to check the DEBUG flag, blf bit 1.

These are not show-stoppers, but they may cause confusion when looking at bins via a debugger window — e.g. chunks not being in their expected bins. This could cause wasted time trying to find a problem that does not exist. Also, debug chunks could alter system behavior. It is expected that they will be used in sections of code that are being debugged, not during system integration. One should be aware of the above the above drawbacks and use debug chunks with caution. On the positive side, the heap fences stand out in a debugger memory window and clearly delineate data blocks.

fill mode

Sometimes there is no alternative but to look directly at the heap in order to track down heap usage problems. eheap has a fill mode, which fills unique patterns into data blocks when allocated, free blocks when freed, and DC and TC when the heap is initialized or a block is freed to them. These are helpful when viewing a heap in a debugger memory window. It is easier to see free, inuse, and debug chunks and to also see how much of an inuse data block has been used. It is also helpful to see how DC and TC are faring – i.e. plenty left or almost gone?

mode.fl.fill can be turned ON or OFF using eh_Set(). Thus, heap fill is selective, like debug mode. It can be applied only to chunks of interest. The heap fill patterns are defined in eheap.h:

```
EH_DATA_FILL     0xDDDDDDDD
EH_FREE_FILL     0xEEEEEEEE
EH_FENCE_FILL    0xAAAAAAA3
EH_DTC_FILL      0xCCCCCCCC
```

eheap is shipped with these values. They can be changed to whatever is preferred, except that the low nibble of EH_FENCE_FILL must be 0x3.

If EH_BT_DEBUG, eh_Init() fills DC and TC with the EH_DTC_FILL pattern. This is helpful during debug to easily find DC and TC in the debugger memory window and to see how they are faring, as the system runs. This is not done for released systems because filling DC and TC is equivalent to filling the entire heap and can increase boot time significantly.

When fill mode is ON, the data blocks of inuse and debug chunks, are filled with EH_DATA_FILL when they are allocated. The fl and blf fields of an inuse chunk or the CDCB of a debug chunk are followed with the data fill pattern. Spare space above the data block, except ssp, is filled with EH_FREE_FILL. These are helpful to see what chunks are allocated, how much of each data block is being used, how much spare space is in the chunk, and whether or not data has overflowed the spare space. This information is useful for upsizing or downsizing blocks. Note that if the chunk is a debug chunk, the onr field identifies the task or actor that allocated it.

If a chunk is freed with mode.fl.fill mode ON, EH_FREE_FILL fills the rest of the free chunk after its CCB. So, in the memory window, the CCB will be followed by the free fill pattern, making it easier to identify free chunks and their CCBs.

These patterns are helpful when looking at a heap through a debugger memory window — they make it easier to understand what is being seen. Having different fills enables quickly spotting what is free and what is inuse, while tracking down heap-related problems. Filling, of course, does reduce performance and is not recommended for released systems. However, since it is selective, it is helpful for debugging heap problems in new code without impacting other parts of a system.

error checks

As a further aid for debugging and system reliability, all heap service parameters are checked and invalid parameters reported. In most cases, services abort if an invalid parameter is found. eheap has a 3-level error reporting system controlled by hmode.fl.ed_en =

```
0none
1all but allocation and free
2all
```

During debugging, the error reporting level should be set at 2. It is convenient to load eh_hvp[hn] into the debugger watch window and pay attention to the errno field. Seeing an error pop up can save a great deal of debug time chasing the wrong problem. Even better, if using an RTOS like smx, put a breakpoint at the start of the error manager to catch heap errors the moment they occur. Then, using the call stack window it is easy to pinpoint the cause of the error. A typical error might be eh_Free(bp) where bp -> garbage. This will cause an EH_INV_PAR error.

Using built-in error detection can save wasted debug time chasing what appear to be serious errors, which in fact are just uninitialized pointers, wrong sizes, etc.

heap information

When facing difficult heap debug problems, it may be helpful to write routines that take snapshots of the heap and report abnormalities. This is made easier by the services discussed below, which allow getting useful information about chunks and bins.

eh_ChunkPeek(vp, par, hn) can be used to get information about chunks. vp is a chunk pointer in all cases, except for par == SMX_PK_CP, in which case it is a block pointer. If vp is out of heap range or not 4-byte aligned eh_ChunkPeek() reports EH_INV_PAR and returns 0. The parameter, par, is of type SMX_PK_PAR. Available parameters are:

```
SMX_PK_BINNO     Chunk bin number (0 if not free, dc, or tc).
SMX_PK_BP        Data block pointer from cp (0 if free).
SMX_PK_CP        Chunk pointer from bp (0 if free).
SMX_PK_NEXT      Address of next chunk in the heap.
SMX_PK_NEXT_FREE Address of next chunk in this bin (0 if last chunk, dc,
                 tc, or not free).
SMX_PK_ONR       Chunk owner (0 if not debug chunk).
SMX_PK_PREV      Address of previous chunk in heap.
SMX_PK_PREV_FREE Address of previous chunk in bin (0 if first chunk, dc,
                 tc, or not free).
SMX_PK_SIZE      Chunk size.
SMX_PK_TIME      Time chunk allocated (0 if not debug chunk).
SMX_PK_TYPE      Chunk type (free == 0, inuse == 1, debug == 3).
```

eh_ChunkPeek() reports EH_INV_PAR and returns -1, if par is not one of the above. If the chunk is SC, PREV will return 0; if the chunk is EC, NEXT will return 0. If a chunk is the last chunk in a bin, NEXT_FREE will return 0. Similarly, if the chunk is the first chunk in the bin, PREV_FREE will return 0. If a chunk is inuse, it cannot be in a bin, thus 0 is returned. Since 0 is a valid bin number, the chunk should be tested for free.

As shown above, eh_ChunkPeek() returns 0 if the chunk type is invalid for the parameter requested. It usually is advisable to read the chunk type first to make sure that the expected chunk information is actually available. It also is advisable to check that the return value is not -1 or 0 before using it, except for bin number and chunk type, where 0 returns are valid.

The following example shows finding the bin number of a free chunk:

```
int  bin_num;
void *bp;
CCB_PTR cp;

bp = eh_Malloc(100);
cp = eh_ChunkPeak(bp, EH_PK_CP);
eh_Set(EH_ST_MERGE, OFF);
eh_Free(bp);
bin_num = eh_ChunkPeek(cp, EH_PK_BINNO);
```

In this example, a block is allocated from the heap, and the chunk pointer, cp, is obtained from the block pointer. Chunk merging is turned off so eh_Free() does not merge the chunk with another free chunk, but rather puts it into the correct bin for its size. Then eh_ChunkPeek() is used to find out which bin the chunk was put into. This would not be a trivial exercise for a debug chunk.

eh_ChunkPeek() is useful for heap integrity checking and heap maintenance. For example, if a debug chunk is owned by a task that has been deleted or stopped then a heap leak has been found. If a block that is about to be freed is already free then a double free has been detected. Implementing tests like these can significantly reduce debug time – especially when adding *SOUP* to projects. This kind of testing may also be of value in released systems to improve their reliability.

eh_BinPeek(binno, par, hn) can be used to obtain information concerning bins. binno is the bin number. If it is not in the range 0 to top_bin, EH_INV_PAR is reported and 0 is returned. The parameter, par, is of type EH_PK_PAR. Available parameters are:

```
EH_PK_COUNT    Number of chunks in bin.
EH_PK_FIRST    Address of first chunk in bin, NULL if empty.
EH_PK_LAST     Address of last chunk in bin, NULL if empty.
EH_PK_SIZE     Minimum chunk size for bin.
EH_PK_SPACE    Free space in bin.
```

HeapBinPeek() reports EH_INV_PAR and returns -1, if par is not one of the above. It returns 0 if the bin is empty, except for SIZE. Determining chunk counts in bins is useful in order to keep bins from becoming empty, as in the example in the bin seeding section of the Optimization chapter, or too full. Totaling up free space in all bins or in all small bins might be used to control cmerge. See also **heap statistics** in the Operation chapter.

debugging problems

Both inuse and free chunks use the CCB type. This is valid for free chunks, but only the fl and blf fields are valid for inuse chunks. Thus, in the debugger watch window, the sz, ffl, fbl, and binx8 fields are either data or data fill, not metadata as the debugger suggests. The debug chunk uses a CDCB type and thus all fields shown for it are valid.

eh_Realloc() poses a complexity, as follows: The resulting chunk type, when eh_Realloc() is called, is determined by the value of debug mode then, not the value when the original chunk was allocated. Hence, an inuse chunk might be reallocated as a debug chunk or vice versa. Then, depending upon the debug chunk overhead, reallocating a block in a debug chunk to a larger size block in an inuse chunk may not require changing chunks.

eh_Free() makes a best attempt to detect a double free. However, it cannot detect a situation where a block is freed by task A, allocated to task B, freed again by task A, then allocated to task C. As a result, tasks B and C will unknowingly be sharing a block. Only careful coding will avoid this problem. Use of debug chunks will help to identify the problem, because the debug chunk's owner will be task C, not B.

Thus, if looking in task B to see why data is spuriously changing, check the chunk owner – B may not own the chunk!

debugging techniques

It is generally helpful to look at the actual heap in the debugger memory window. Search on the block pointer of interest. Its CCB is immediately above. It is fairly easy to use CCB.fl to find the next chunk and CCB.bl (sans bits 2-0) to find the previous chunk. The fills and fences help quite a bit to delineate chunks and blocks. Looking at a chunk or block of interest may help to figure out what is wrong or to correct a misapprehension.

It is also helpful to put hv and bin[] for heap hn in the watch window (**see required structures and variables** in the Setup chapter). The first shows all variables for heap hn and the latter shows its bins. monitoring dcp and tcp, in hv helps to see if chunks are coming from DC or TC. As noted before, it is important to watch errno for errors. Other helpful fields are bmap, hused, hhwm, mode, and retries. It can be particularly useful to see if chunks are in expected bins or come from them and if bins are populated or not.

Hopefully you will not find errors in eheap, itself. However, looking at how it works will help you to better understand it and to better use it. Then, of course, all of the suggestions made in this chapter should be brought to bear.

using smxAware

smxAware has many features to make heap debugging much easier. See the smxAware User's Guide for more information.

Chapter 6 Optimization

need for tuning

As shipped, eheap is configured for normal heap requirements. However, if performance is not adequate, tuning a heap for its intended use will improve it.

Theoretically, it is not possible to design one heap that will work well for all applications. For every allocation strategy, some applications can be found that will cause excessive fragmentation or other serious problems. A general-purpose heap, such as dlmalloc, is good at satisfying the needs of most applications. It has the advantage of a large amount of memory and it can usually get more memory, if needed.

This situation is different for most embedded systems. Typically, memory is in short supply, which is exacerbated by the need for multiple heaps to achieve security requirements. In addition, there is a wide variation of requirements from one embedded system to another and from one partition to another.

Although there is a wide variation of characteristics from one embedded system to the next, a given embedded system is a typically a single application with constrained characteristics. A given partition will be even more constrained. Thus, it is feasible to tune its heap to get good performance without serious risk of fragmentation failure or other serious problems. Also, variable structure and tunability help to shoe-horn heaps into tight spaces, while achieving necessary performance. Tunability is a valuable characteristic for embedded system heaps.

optimizing bin arrays

The bin structure of eheap is adjustable to suit a wide range of requirements. The best plan is to initially run with the standard bin configuration in order to get the application software running in its final form. Then record the sizes being used and optimize the bin structure accordingly.

For a system using a large variety of chunk sizes, an evenly spaced bin array, such as the standard bin configuration, is probably the best solution. However, if a system uses certain large sizes much more frequently than others, creating large bins that start with those sizes can greatly improve performance. For example, say a system uses predominantly 200, 400, 800, and 1200-byte blocks and a scattering of other sizes. Then the standard heap bins could be optimized, as follows:

```
/* bin   0   1   2   3   4   5   6   7   8   9   10   11 */
       {24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, \
/* bin  12   13   14   15   16   17   18   19   20    21    22    23 */
       120, 128, 208, 408, 512, 640, 768, 808, 1024, 1152, 1208, 1408, \
/* bin  24    25    26    27    28  end */
      1536, 1664, 1792, 1920, 2048, -1};
```

Bold numbers indicate the bin sizes that have been changed (remember that bin sizes are chunk sizes). Due to the way that free() works, chunks of these sizes will always be put in the front of their bins. Hence, the next access for one of these chunk sizes is as fast as a small upper bin, even though these bins also contain other sizes. Due to taking the last-freed chunk first, cache hits for accesses to the blocks in these chunks also improve.

However, if there are a large number of 808-byte chunks in bin 19, for example, then search times for larger chunks in the bin (e.g. 816, 1016, etc.) may become too long. This can be solved by making bin 19

a small bin containing only 808-byte chunks and starting bin 20 at 816. If this is too many chunk sizes for bin 20, bin 21 could be started lower, thus reducing the number chunk sizes in bin 19. An alternative solution would be to turn cmerge on while excess 808 chunks were being freed. This should merge them into larger chunks that are put elsewhere.

smaller bin arrays

smaller bin arrays It is likely that partitions requiring dedicated heaps will require only a small number of block sizes and not require high performance. In these heaps, simple bin arrays should be adequate, such as the following:

```
/* bin  0   1     2     3     4   end */
      {24, 512, 1024, 1536, 2048, -1}
```

This covers the same range, as the standard array, with only 5 bins and the top bin (4) covers 2048 bytes and up, as in the standard array. Note that these are all large bins and that there is no SBA. Also, DC is only 24 bytes and is not used. However, TC still exists. Bins 1 thru 3 each cover 512 bytes and have 64 chunk sizes. Bin 0 is slightly smaller. Bins 0 thru 4 act like five small heaps. Finding the right bin takes up to 3 comparisons. Compared to the previous heap, it saves 288 bytes of memory for the bin array. If the bin sizes are chosen to be equal or slightly less than frequently used chunk sizes, then combined with bin sorting, this heap should be much faster and more deterministic than a simple linear heap.

To go even smaller, consider:

```
/* bin  0  end */
      {24, -1}
```

This defines a one bin heap. bin 0 handles all chunk sizes from 24 bytes and up. This heap would be appropriate for a very small partition, with a tiny heap space, such as 10KB. Like the previous heap, there is no SBA nor is DC used, but TC still exists. A one-bin heap has many of the eheap advantages over a simple linear heap:

(1) Only free chunks are linked into the bin, so it is not necessary to search through both inuse and free chunks. This alone should produce 5x faster allocations.

(2) TC provides a fast start up for allocations and is the source of last resort.

(3) Bin sorting and deferred merging ensure that small chunks can be found faster than large chunks.

(4) Debug and safety mechanisms are available.

With regard to (3) it can be argued that longer allocation times for large blocks do not necessarily reduce system performance because processing large blocks takes longer than processing small blocks. eh_Free() puts very small chunks at the start and all other chunks at the end of the bin free list. Then, smx_BinSort() will quickly move small- and medium-size chunks back to where they belong. Hence, operation of the one bin heap could be pretty good.

merge control

As noted in **deferred merging** in the Operation chapter, eheap implements deferred free chunk merging. Merging is controlled by mode.fl.cmerge, which can be turned ON or OFF via eh_Set(). As also noted, for some heaps it may be best to leave cmerge continuously ON. In other heaps, bin leakage up due to free chunk merging may hurt performance. Unfortunately running with cmerge OFF increases the risk of allocation failures due to excessive external fragmentation. As noted previously, some bin leakage down is caused by chunk splitting, even if cmerge is ON. Increasing EH_MIN_FRAG helps to reduce this, but

increases internal fragmentation, which may also cause allocation failures. Hence, avoiding excessive bin leakage while also avoiding heap failure due to fragmentation is not easy.

eheap provides mode.fl.amerge, which if ON can be used to implement automatic merge control. The followin is an example of a possible algorithm for this:

```
if(eh_hvp[hn]->mode.fl.amerge == ON)
{
   tblcp = bin[eh_hvp[hn]->top_bin].fbl;
   tblcsz = (tblcp == NULL ? 0 : tblcp->sz);
   tcsz = eh_hvp[hn]->tcp->sz;

   if ((eh_hvp[hn]->hused > HEAP_USE_MAX) || (tblcsz < HEAP_CSZ_MAX &&
                                              tcsz < HEAP_CSZ_MAX))
         eh_hvp[hn]->mode.fl.cmerge = ON;

   if ((eh_hvp[hn]->hused <= HEAP_USE_MIN) && (tblcsz >= HEAP_CSZ_MAX ||
                                             tcsz >= HEAP_CSZ_MAX))
         eh_hvp[hn]->mode.fl.cmerge = OFF;
}
```

This is just turning cmerge ON when hused is too large and turning it OFF when hused is below a lower limit. In addition, there is code to make that the largest possible chunk can always be allocated. HEAP_USE_MAX = ¾ heap size is probably a good starting point, and:

```
HEAP_USE_MIN = HEAP_USE_MAX – HEAP_BAND;
```

where HEAP_BAND is a few hundred bytes to provide stability. HEAP_USE_MAX may need to be ½ heap size or even less for tight heaps.

This algorithm is probably good for most heaps, but it may not work for other heaps. Other algorithms may be more effective, such as those based upon factors such as number of free chunks, average size of free chunks, total space in bins, etc. Long-run testing of these alternatives on the target embedded system is necessary to determine which works best and how much heap margin (i.e. total space / used space) is necessary to ensure that heap failure does not occur. Time delays and calculations should be compressed, if possible, so that test run times are equivalent to much longer actual run times. This gives confidence that the heap will not fail in practice.

Dynamic merge control can also be implemented at the task level. For example, cmerge can be turned OFF by tasks, which are heavy heap users. Tasks written in object-oriented languages such as C++ and Java are likely to be such tasks, since they often do frequent mallocs and frees of small blocks as objects go in and out of scope. Inhibiting merging while these tasks run can improve performance by avoiding leaky bins, particularly in the SBA. One way to do this is to turn cmerge OFF when the task is running and ON when it is not. This is possible with RTOSs that permit hooking entry and exit routines to tasks. When the task is about to stop running, it can turn cmerge ON so that the small chunks it allocated will be merged and become available for larger allocations by other tasks. This scheme may not avoid heap failure unless heavy heap users are prevented from running simultaneously.

Allocation failure is not necessarily catastrophic – see **recovery** in the Reliability chapter.

bin seeding

Bin seeding provides an alternative to deferred merging that may be helpful in tight heaps. The **eh_BinSeed(num, bsz, hn)** service is used to directly seed num chunks into the bin for the block size, bsz. The bin is not specified, because it depends upon the chunk size, which, in turn, depends upon the

debug mode. This service allocates a big-enough chunk, splits it into num chunks the right size for blocks of bsz, and frees the chunks to their correct bin. While in operation, cmerge is ignored.

Bin seeding, combined with monitoring bin populations, may be the best way to populate bins, in some systems. For example:

```
void FillBins(u32 hn)
{
   u32 bn, bsz;
   for (bn = 0; bn <= eh_hvp[hn]->sba_top; bn++)
   {
      if (eh_BinPeek(bn, SMX_PK_COUNT, hn) < 2)
      {
         bsz = eh_BinPeek(bn, SMX_PK_SIZE, hn);
         eh_BinSeed(2, bsz, hn);
      }
   }
}
```

FillBins() could be called from the idle task in order to keep 1 to 2 chunks in each SBA bin. This would ensure fast allocations for small chunks. Since the new chunks are not separated by inuse chunks they may soon leak out and unite with other free chunks if cmerge is ON. For this reason, it is probably best to only seed few chunks at a time and to do so frequently. Two is a good number because the second chunk will be allocated first and will provide an inuse barrier to protect the first chunk.

Debug mode must be OFF for the above code to function as expected. Otherwise chunks will be put in higher bins than expected. Furthermore, the last chunk may be larger than the other chunks due spare space left in it, and thus it may be put into a higher bin, than the other chunks.

Another use for eh_BinSeed() is to populate bins during initialization. This gets the heap off to a faster start and is an alternative to using DC.

bin sorting

Bin sorting is done by increasing size in large bins and improves their performance. (Bin sorting is not necessary for small bins.) Since a large-bin allocation takes the first big-enough chunk, it will be the *best-fit chunk* available in the bin. As a consequence, there will be less splitting of chunks, which improves performance and reduces fragmentation. In addition, choosing bin sizes that are equal or slightly less than the most frequently used chunk sizes will result in faster allocations, on average.

Embedded applications must have significant average idle time to deal with peak loads caused by simultaneous asynchronous events. Also, they must be designed with spare processing time in order to handle future added features. Idle time is not characteristic of server or desktop applications, which simply run at full speed until done. eheap takes advantage of idle time to sort large bins. If there is enough idle time, large bins should almost always be well-sorted. Though poorly sorted bins may cause suboptimal performance, they do not cause heap failures and thus should be acceptable during periods of peak loading, when there is no idle time. It is worth noting that lower-priority tasks that use a heap will probably not be able to run during peak-load periods, anyway.

The sort algorithm is bubble sort with last-turtle insertion. The last turtle is the last chunk in the bin free list that may be smaller than some chunk ahead of it. When starting a bin sort, the last chunk put at the end of the bin by the last free operation is the last turtle. It is called a “turtle” because it moves forward very slowly in a normal bubble sort. Last-turtle insertion fixes this problem. During each pass, the current last turtle is inserted ahead of the first larger chunk found. The chunk that was ahead of the last turtle becomes the new last turtle. A bubble sort requires multiple passes through the bin free list. After k passes, the last k chunks are sorted. However, last turtle insertion can considerably speed up the sort.

Each pass ends at the current last turtle and if n turtles were moved during this and previous passes, then the current last turtle will be n + k chunks from the end of the free list. When no chunks have been moved during a pass, the sort is complete.

When free() puts a large chunk at the end of a sorted bin, last turtle insertion will immediately move it to its most likely position during the first pass. If bin sort is running frequently, this may be all that is required to sort the bin.

Bin sorting is done by calling **eh_BinSort(binno, fnum, hn)** each time the idle task runs. eh_BinSort() should not be interrupted by another heap service during a run. A run consists of testing fnum chunks and moving those that are smaller. Dividing a sort into runs is intended to allow higher priority tasks needing heap hn services to run without missing their deadlines. Of course, the smaller fnum is, the longer a bin sort will take, so there is a tradeoff to be made.

A bin sort map, *bsmap*, similar to *bmap* used by malloc() and free(), determines which bins to sort. The bsmap bit of a bin is set if eh_Free() puts a chunk at the end of that bin's free list. This happens whenever the freed chunk is larger than the first chunk in the bin. Otherwise the chunk is put at the start of the bin, in which case, no sort is needed and the bsmap bit is not set. Small bins never need sorting, hence their bsmap bits are never set. Therefore, bsmap shows only those bins that need sorting.

A bin's bsmap bit is reset when a sort begins for the bin. A global variable, *smx_csbin*, keeps track of the current bin being sorted, and a static variable, *ccp*, keeps track of the next chunk to start the next run. If a preempting eh_Free() sets the bin's bsmap bit, due to putting a chunk at the end of the bin, the sort is restarted for the next run. If a preempting eh_Malloc() takes a chunk from the bin, it also sets the bin's bsmap bit, causing the sort to start over on the next run. Starting over is not detrimental to a sort, because any sorting already done is preserved. Otherwise each run starts from where the last run left off. Note that one pass may require multiple runs or one run may cover multiple passes, depending upon the size of fnum vs. the length of the pass, which steadily declines for a sort.

The *binno* parameter in eh_BinSort() specifies the bin to sort. If it is a valid bin number and the bsmap bit is set for the bin, then the binno bin is sorted. If binno is greater than the top bin number, all bins with bsmap bits set are sorted, smallest first. This favors smaller large bins, because they are likely to be more active. eh_BinSort() returns TRUE when a bin has been sorted, or if no bins needed to be sorted. Note that unless bin sorting is frequently preempted, all chunks in the bin free list are likely to be in the data cache and chunk accesses will be fast. Hence large runs (i.e. large fnums) may be possible without causing high-priority tasks to miss their deadlines.

If bins are not being adequately sorted, fnum can be increased or eh_BinSort() can be called from a higher priority task. It might make sense to call it from the priority level of the lowest priority task using the heap. Then it will run ahead of other idle functions. Because it is expected to run frequently, eh_Sort() makes no entries in the event buffer, other than those due to reported errors or fixes.

Chapter 7 Reliability

Most embedded systems are expected to run indefinitely without supervision. In addition, some are placed in hostile environments subject to extreme temperatures, voltage transients, high fluxes of high energy particles, etc. And then there is hacking and malware, which is an increasing problem. eheap provides several features to minimize damage from errors and to achieve self-healing.

error reporting

eheap reports the following errors:

```
EH_OK
EH_ALREADY_INIT
EH_HEAP_BRKN
EH_HEAP_FIXED
EH_HEAP_ERROR
EH_HEAP_FENCE_BRKN
EH_INSUFF_HEAP
EH_INV_CCB
EH_INV_PAR
EH_RECOVER
EH_TOO_MANY_HEAPS
EH_WRONG_HEAP
```

See the API discussion in Appendix A for a description of what each error means relative to each heap service. eheap does extensive error detection and reporting. This is great during debugging and is highly recommended during normal operation to detect and record latent bugs and to help detect and thwart hacking. However, reporting may hurt performance of code that does intensive block allocations and frees, such as object-oriented code. To deal with this problem, eheap has a 3-level error reporting system controlled by hmode.fl.ed_en =

```
0 none
1 all but allocation and free
2 all
```

For released software, level 2 is recommended for best reliability and security. However, if performance is more important, level 1 skips error reporting for allocation and free operations and level 0 skips all error reporting. It is important to note that error detection is still performed and appropriate actions taken. For example, in eh_Malloc() if sz = 0, operation is aborted and NULL returned.

It may be thought that error detection should also be skipped in order to improve performance when application code is very solid. However errors are still possible due to environment factors, such as high-energy particles causing bit flips and due to hacking. Not detecting and protecting against such errors is likely to result in a fragile system wherein heap integrity may be lost and spurious MMFs (see the **SecureSMX User's Guide**) and other faults occur.

Heaps are especially vulnerable compared to other memory objects because of their large concentrations of control information in the data area. For example, if the average block size is 32 bytes then, about 25% of heap memory consists of control information (primarily forward and backward links). This is a large target for energetic particles. A block pool, by comparison, it has about 8%. The reason this is important is that one bit flip in a pointer or size field is likely to put the system into the weeds and cause a system crash. A bit flip in a data word is not likely to be so catastrophic because it may make little difference (e.g. in the least significant bits) or is easily caught by a range check and rejected.

Previously, in **multitasking** in the Setup chapter, the use of shell routines to adapt eheap functions to the multitasking RTOS mechanism for access protection was discussed. The same shell routines can be used to map the eheap errors to RTOS errors. For example, for smx:

```
void* smx_HeapMalloc(u32 sz, u32 an, u32 hn)
{
   void* bp;
   if (!smx_HeapEnter(sz, an, hn, SMX_ID_HEAP_MALLOC))
      return NULL;
   bp = eh_Malloc(sz, an, hn);
   if (eh_hvp[hn]->errno != 0 && eh_hvp[hn]->mode.fl.em_en)
   {
      smx_ERROR((SMX_ERRNO)xerrno[eh_hvp[hn]->errno], SMX_ERRH_UNS);
   }
   smx_HeapExit((u32)bp, hn, SMX_ID_HEAP_MALLOC);
   return bp;
}

/* eheap error to smx error mapping table */
const u32 xerrno[] = {SMXE_OK,
                      SMXE_HEAP_ALRDY_INIT,
                      SMXE_HEAP_BRKN,
                      SMXE_HEAP_FIXED,
                      SMXE_HEAP_ERROR,
                      SMXE_HEAP_FENCE_BRKN,
                      SMXE_INSUFF_HEAP,
                      SMXE_INV_CCB,
                      SMXE_INV_PAR,
                      SMXE_HEAP_RECOVER,
                      SMXE_TOO_MANY_HEAPS,
                      SMXE_WRONG_HEAP,
                      };
```

These smx errors are in the same order as the eheap errors shown previously. smx_ERROR() is a standard smx macro used to call the smx error manager, smx_EM(), which records errors and may perform other functions, depending upon the error.

If eheap error reporting is suppressed, then eh_hvp[hn]->errno = 0 and smx_ERROR() is not called. If mode.fl.em_en is OFF, all eheap error reporting by the RTOS is suppressed. In this case, hmode.fl.ed_en might be set to 2 so eheap errors are recorded in eh_hvp[hn]->errno, but the RTOS error manager is not called. This way, if an eheap operation fails, it is possible to determine why, yet the RTOS error manager will not be invoked.

fragmentation

When a heap becomes excessively *fragmented*, small inuse chunks may prevent medium chunks from being merged to form larger chunks that are needed for allocations. When this happens, *heap failure* is the result and EH_INSUFF_HEAP is reported. This is not necessarily catastrophic. For example, the task requesting the large block could be rescheduled to run at a later time or at a lower priority and try again, or less important tasks could be stopped and their heap blocks freed with merging enabled. eheap tries to hold reserve space in the top bin and in TC for large chunk allocations. However, over time, both are likely to be used up, unless there is ample RAM available.

Deferred merging, as in the Optimization chapter, is generally viewed as increasing fragmentation, thus making heap failure more likely. If all methods of merge control are resulting in fragmentation leading to heap failures, then it may be necessary to keep cmerge permanently ON. This is more likely to occur in a

tight heap due to insufficient RAM than in a system where RAM is abundant. The downside is that cmerge continuously ON will reduce performance due to leaky bins.

Another potential problem with deferred merging is *stuck chunks* in large bins. This can happen in a system that allocates random large sizes, some of which are seldom used. When freed, these chunks will be put into large bins. If such a chunk is larger than the chunks normally allocated from that bin, it may sit in the bin for a long time until the bin runs out of smaller sizes, then the too-large chunk is taken and split. Turning cmerge ON will help to remedy this problem.

If merge control fails to stem an occasional heap failure, an eheap recovery service is provided to search the heap to find and merge adjacent free chunks in order to satisfy a failed allocation. If this fails, an eheap extension service can be used to extend the heap into reserve memory, such as a slower RAM area or one reserved for future expansion. Heap recovery and heap extension services are discussed near the end of this chapter.

Unfortunately, reserve memory may not be available in a specific embedded system. If this and other strategies, such as those suggested above, do not work, then in the worst case it may be necessary to reinitialize the heap and restart all tasks using it. Obviously, mission-critical tasks should use block pools instead of the heap. Recovery methods are discussed more later in this chapter.

self-healing

With increasing IoT deployment, self-healing is becoming more important. General-purpose systems are generally housed within concrete buildings that provide protection against environmental factors. In contrast, embedded systems are often deployed at high altitudes or high latitudes, where high-energy particle fluxes are large. Also embedded systems are likely to be less protected, possibly right out in the open, subject to temperature extremes and EMI from thunderstorms, sunspots, etc. And ever-smaller semiconductor feature sizes are exacerbating these problems.

In addition to bit flips, discussed above, heaps are highly vulnerable because control information (i.e. CCBs) is sandwiched between data blocks. Data block overflows damage the control information, again sending the system into the weeds. This kind of damage usually results from programming errors or malware. Typically, data buffers overflow in the up direction and stacks overflow in the down direction, so neither end of a data block is safe.

We are accustomed to reloading our applications or rebooting our desktop computers whenever misbehaviors occur. This not an option for most embedded systems because they are unattended and expected to run forever. Hence, heap self-healing is desirable for embedded systems. eheap accomplishes self-healing by continuously scanning the heap and bins, fixing broken pointers, wrong sizes, etc., whenever possible, or sounding an alarm if not possible. The latter helps to achieve a soft landing by fixing the problem before a malloc() or free() encounters it and crashes.

A modest heap of 10,000 chunks has 2 pointers per inuse chunk and 4 pointers per free chunk. Assuming 75% of the heap is inuse there are 27,500 pointers in this heap. An average malloc() or free() will use about 6 heap pointers, so the probability of it encountering a damaged pointer or size field is 6 in 27,500 = .022%. If 100 mallocs and 100 frees occur in the time needed for one scan, the probability of failure is about 4.3%, or broken links will be fixed 22 out of 23 times – clearly a big MTBF improvement!

With increased emphasis on security comes multiple heaps per system – i.e. one per partition that needs a heap. The upside to this is that if a heap is damaged and cannot be repaired, then the partition can be rebooted without bringing the whole system down. In addition, dedicated partition heaps reduce the dynamic load on the main heap by quite a bit compared to a one-heap system. It may be used primarily for static allocations, such as dedicated heaps, task stacks, and large arrays and structures. As a consequence, the main heap may be able to sustain considerable damage before failing.

heap scanning

eh_Scan(cp, fnum, bnum, hn) is like a night watchman – a slow, but trusted patrol looking for trouble. It is intended to perform continuous forward heap scans and to fix heap problems, or report ones it cannot fix. To accomplish this, it can be called once per idle task pass, so that it will not consume valuable processing time. It scans each chunk from the start of the heap to the end, fixing broken backward links, flags, sizes, and fences, as it goes.

In the debug version (EH_BT_DEBUG defined), the scan stops on a broken fence so that the fence can be studied for clues to what happened. In the release version, the fence is fixed. Fixes are reported so they can be saved for analysis. This can be valuable for systems in the field, in order to monitor stresses and behaviors. During scans, all pointers are heap-range tested before use, in order to avoid MMFs and data abort exceptions, or equivalent, if they are broken.

If a broken forward link is found and the chunk is either a free chunk or a debug chunk, the sz field is used to attempt a fix. If sz + fl does not point to a valid chunk or if this chunk is inuse, a backward scan is started. It proceeds from the end of the heap until the break is found and the forward link is fixed. Then the forward scan is resumed until the end of the heap is reached.

While backward scanning, other broken forward links are fixed, if encountered. However, a broken backward link stops the backward scan and a heap bridge is formed between the chunk with the broken forward link and the chunk with the broken backward link. When this happens, many chunks may be bridged over. Bridging serves to allow the scan to complete and may give the system a chance to limp along for a while. The bridged chunks might be permanent inuse chunks so operation could continue indefinitely. But if a bridged chunk is freed or allocated, a failure (e.g. MMF or data abort exception) is likely to occur. Therefore EH_HEAP_BRKN is reported, so that error recovery code can take action.

It is assumed that in a multitasking system, eh_Scan() will be access protected, so it could block a higher-priority task needing access to this heap for a long time, thus causing it to miss its deadline. Consequently eh_Scan() operates incrementally. When forward scanning, it moves forward fnum chunks per call; when backward scanning, it moves backward bnum chunks per call. These are called runs. Thus, a full heap scan normally requires many runs. bnum is usually larger than fnum because backward scanning is faster and fixing a break is more urgent. So, fnum might be 2 and bnum might be 100 or -1.

A run will go the specified number of chunks then return FALSE, unless it is done or encounters an error that it cannot fix. Thus eh_Scan() can be called repetitively, as follows:

```
while (!eh_Scan(NULL, 2, 100, hn)){}
```

This will scan from the start of the heap to the end of the heap, one run at a time, unless an unfixable error is encountered, in which case it stops and returns TRUE. It keeps going for all fixable errors. The NULL parameter means to start from hsp (heap scan pointer), for each run. At the end of each run, hsp, is set to point to the next chunk to scan. Should a preempting free() merge the chunk pointed to by hsp with a lower chunk, the free() changes hsp to point to the lower chunk. Then the next run will start with it, rather than with potential garbage. Other frees and mallocs do not affect hsp. During initialization and when a scan is completed, hsp is set to SC, causing the next eh_Scan() to start from the beginning of the heap.

eh_Scan() is normally called once per pass through idle. Hence, heap scanning is a slow, continuous process that makes use of idle time to increase heap reliability. fnum can be adjusted upward to ensure a higher probability that a break will be found by the scan before it is found by a heap service. Yet it should be possible to adjust fnum low enough that the impact on task latency is negligible. A value of 1 or 2 will probably suffice, in most situations. In order to not consume too much idle time, it may be desirable to space heap scans out – e.g. one per second or one per minute. This requires consideration of expected error rates vs. desired reliability. Security may also be a factor.

eh_Scan() can be called directly from an application task, as well as from idle. This might be done when another heap service reports an error, as follows:

```
eh_Scan(cp, 1, 10000, hn);
while (!eh_Scan(NULL, 10000, 10000, hn))
```

The scan will start from cp, which might point at the chunk in question. As shown, fnum and bnum are likely to be high values so scanning will finish quickly. eh_Scan() will report EH_HEAP_FIXED, if it succeeds. An alternative to the above code is:

```
eh_Scan(cp, 1, 10000));
if (eh_hvp[hn]->errno != EH_HEAP_FIXED)
{
   while (!eh_Scan(NULL, 100, 100) && eh_hvp[hn]->errno !=
EH_HEAP_FIXED){}
}
```

This code makes shorter runs and stops after the problem is fixed.

bin scanning

Whereas bins are in local memory and thus may be less susceptible to bit flips, most of the bin free list pointers are in the free chunks, themselves, in the heap, and thus are just as susceptible to damage as the heap forward and backward links. So, bin free list scanning is necessary to provide a consistent level of heap protection. **eh_BinScan(binno, fnum, bnum, hn)** performs this function. It is similar to eh_Scan(): it incremental, scans doubly-linked chunk lists, and fixes broken links, when it can. It has four parameters: binno, the bin number, fnum the forward run limit and bnum the backward run limit, and hn. Like heap scan it returns FALSE until it is done with the bin or an unfixable error is encountered. See eh_BinScan() in Appendix A for a usage example.

If binno is greater than the top bin number or fnum or bnum are 0, EH_INV_PAR is reported and eh_BinScan() returns with TRUE. If the parameters are valid, bin scanning begins. If the bin is empty, its free back link, fbl, is set to NULL, if necessary, and TRUE is returned. Otherwise, the bin free forward link, ffl, is checked. If it is out of heap range, both bin links are set to NULL, the bin's bmap bit is set to 0, EH_HEAP_BRKN is reported and TRUE is returned. The bin is now effectively empty and the chunks that were in its free list cannot be allocated. However, normal operation can continue and, if cmerge is ON, the lost free chunks may eventually be recovered through merging with other chunks.

Barring the foregoing, a bin scan starts at the beginning of the bin free list and checks fnum chunks. As with a heap scan, a global pointer, bsp (bin scan pointer) maintains the place to resume the next run. Runs continue until the end of the bin free list reached, then TRUE is returned. Broken fbls are fixed as encountered. If a broken ffl is found, eh_BinScan() scans backward to fix it and bfp (bin fix pointer) maintains the place to resume the next run.

Like eheap scan, double breaks are bridged. In this case, the bridged chunks are no longer available to be allocated, but heap operation can continue normally, if cmerge is OFF. Bridging is a partial solution, but EH_HEAP_BRKN is still reported. If cmerge is ON, free() may fail when it attempts to merge a bridged chunk with a broken pointer, but. in some cases, the merge will proceed ok.

If eh_Free() preempts between runs it may add a chunk to the start or to the end of the bin free list. Since this affects neither bsp nor bfp, no special action is required. If eh_Malloc() preempts between runs, it may take the chunk pointed to by either bsp or bfp. In this case, the scan is aborted and a new forward scan is started from the beginning of the bin free chunk list.

Whenever a fix is made, EH_HEAP_FIXED is reported. This can be used to monitor how often problems are being found and fixed.

MTBF improvement

A modest heap of 10,000 chunks has 2 pointers per inuse chunk and 4 pointers and 1 size per free chunk. Assuming 75% of the heap is inuse there are 7500*2 + 2500*5 = 27,500 control words in the heap. Ignoring chunk splitting, assume an average malloc searches for 2 chunks then dequeues the chunk found in a bin. This requires 4 pointer accesses. Ignoring chunk merging, an average free needs to access one pointer and a size to compare the chunk, then to access either 1 more pointer or 2 more pointers to enqueue the chunk, each half the time = 3.5 control words, average. If 100 mallocs and 100 frees occur in the time needed for one scan, 4*100 + 3.5*100 = 750 control word accesses are required. Hence the probability of a bad pointer or size access is 100*750/27500 = 2.7%. Thus, broken control words will be fixed before they are used 36 out of 37 times that a bit flip occurs – clearly a big MTBF improvement!

broken heap

If EH_HEAP_BRKN is reported, the heap is pretty much kaput. It can be dealt with by stopping all tasks using the heap, reinitializing the heap, then restarting all tasks that use the heap. For a dedicated heap this will impact only one partition. Naturally, this will cause a large hiccup for functions requiring the heap. However, if high-priority, mission-critical tasks use only block pools, this can be a workable, worst-case solution – valuable data and processing might be lost, but the system will continue performing its critical mission. By taking action before eh_Malloc() or eh_Free() encounter broken links the system is saved from going into the weeds and failing.

The amount of time and code that should be devoted to ruggedizing the heap is application dependent. It may not be of much importance for applications in protected environments that can be conveniently rebooted. It may be of extreme importance in harsh, remote environments where rebooting has serious consequences. The foregoing services provide some basic tools, but additional tools may be needed.

Note that as object-oriented languages make further inroads into embedded systems, ruggedized heaps will become more important, because these languages are heavy heap users.

heap recovery

Heap failure, reported by EH_INSUFF_HEAP, is likely to be due to a situation where too much free space is allocated to small free chunks and insufficient space is allocated to larger free chunks. This is called fragmentation. A recovery service, **eh_Recover(sz, num, an, hn)** , is provided to deal with this problem. It searches the heap to find and merge adjacent free chunks, in order to create a big-enough free chunk to satisfy the failed allocation.

eh_Recover() starts the scan from SC for small chunks or from DC for large chunks. All scans go to the end of the heap at EC before quitting. It searches for adjacent free chunks to merge. If a big-enough chunk can be formed by merging adjacent free chunks, it removes the free chunks (except DC and TC) from their bins and merges them. If the merged chunk is not DC nor TC, it puts the merged chunk into its proper bin, else it updates dcp or tcp, then returns TRUE.

This service does not merge chunks that it cannot use nor that it does not need. mode.fl.cmerge is ignored. If successful, eh_Recover() should be followed by retrying the allocation that failed. If mode.fl.auto_rec is ON, this is done automatically and the allocation returns a block, if one is found. In this case, recovery is transparent to the application, except that the allocation will take much longer than normal and EH_RECOVER will be reported by it. Returns FALSE if a big-enough chunk is not found and the allocation fails.

If eh_Recover() is called directly (mode.fl.auto_rec == OFF), it will search for num chunks and return FALSE if nothing is found. This is intended to put a limit on search times for very large heaps; it allows application recovery code to try another approach or to simply move on. If num expires on a free chunk,

the scan continues until a big-enough free space is found, an inuse chunk is found, or the end of the heap is reached. If a big-enough free space is found, the chunks are merged and TRUE is returned.

eh_Recover() does not do iterative runs like the scans and sort because it is not reentrant – i.e. if eh_Malloc() preempted and also failed, eh_Recover() would be called with a different size and the first size would be lost.

eh_Recover() should be followed by retrying the allocation service that failed, as in this simplified example:

```
while (1)
{
   if (bp = eh_Malloc(size))
      /* use bp */
   else
      if (!eh_Recover(size, 2000, 0, hn))
         break;
}
/* try another heap recovery action */
```

In the above example, if eh_Malloc() fails, eh_Recover() is called. If it finds a big-enough chunk, eh_Malloc() is called again. If eh_Recover() fails after 2,000 chunks, the while loop is exited and another approach is tried. Allocation failure is most likely to occur for large blocks while the heap is still usable for smaller blocks. In time, the large block allocation might be tried again and succeed. The while loop insures that eh_Malloc() is called if eh_Recover() succeeds. If a higher priority task takes the chunk just recovered, eh_Malloc() would fail again and eh_Recover() would be called again.

This example is just for illustration. Generally, it is not practical to implement all calls to eh_Malloc() this way. See Appendix A for a more practical example, where a common recovery task is the only task using eh_Recover(). However, it may not be desirable to add recovery code to every eh_Malloc(), as shown in that example. An alternative approach shown for smx is as follows:

```
TCP_PTR StoppedTask;

void TaskA_main(void)
{
   while (bp = smx_HeapMalloc(size, 0, hn))
   {
      /* use bp and continue normally */
   }
   StoppedTask = self;
   BrokenHeap = hn;
} /* auto stop on a smx_HeapMalloc() failure */

void smx_EMHook(SMX_ERRNO errnum, u32 par)  /* par = block size needed */
{
   switch (errnum)
   {
      ...
      case SMXE_INSUFF_HEAP:
         smx_TaskStartPar(RecoveryTask, par);
         break;
      ...
   }
}
```

```
void RecoveryTaskMain(u32 bsz)
{
     if (smx_HeapRecover(bsz, 2000, BrokenHeap))
       smx_TaskStart(StoppedTask);
  else
     /* try another heap recovery action */
}
```

In this example, TaskA runs normally unless there is an allocation failure, in which case an SMXE_INSUFF_HEAP error is reported, the error manager, smx_EM() is invoked, and it calls smx_EMHook(), which has error management code for SMXE_INSUFF_HEAP. This code starts the RecoveryTask, which should run at a lower priority than TaskA so that smx_EM() can return to TaskA and TaskA can save its handle in StoppedTask, hn in BrokenHeap, then autostop. Other high priority tasks can preempt and run while RecoveryTask runs, unless they also need access to heap hn. num = 2000 is picked to ensure that no mission-critical task will miss its deadline.

If a big-enough free chunk is found, taskA is restarted. Otherwise RecoveryTask tak**es** some other action. With this method, the recovery code for heap allocation failures is separate from the application code and transparent to it. Notice that taskA is structured such that it will try to allocate the needed chunk again, when it is restarted. For more sophisticated task structures, see the ideal task structure section in the Tasks chapter of the smx User's Guide.

The is not a completely practical example because other tasks, with failing heap allocations, could restart the recovery task before it finished, resulting in preempting eh_Recover(), which, as noted above, is not preemptible by itself. A better design would be for RecoveryTask to wait at an exchange for error messages sent by smx_EMHook(). If multiple allocation failures occur, heap recovery messages would be queued up at the exchange and processed in priority order.

When it receives a recovery message, the recovery task would call eh_Recover(). If successful, it would restart the stopped task that needed the block and that task would allocate the block and continue. Hence operation is simple and avoids one task tripping over another.

If eh_Recover() fails and other possible means of recovery are:

(1) Wait and retry.

(2) Free blocks from lower priority tasks with cmerge ON.

(3) Extend the heap with eh_Extend() (discussed next).

(4) Stop all tasks using heap hn, reinitialize it, and restart the tasks.

(5) Reboot the system.

To avoid conflicts with eh_Scan(), eh_Recover() should be used from a higher priority task. eh_Recover() restarts eh_Scan() when it is done, to avoid a possible scan error.

heap extension

If heap recovery fails, the heap can be extended into reserve memory using **eh_Extend(xsz, xp, hn)** . xsz is the size of the extension and xp is its location. The extension may be adjacent to the present heap or elsewhere in memory. The only requirement is that the extension must be above the current heap. If there is a gap, it is covered by an artificial inuse chunk and the extension becomes the new TC. If there is no gap, the extension is added to the current TC.

A heap extension might be to less desirable memory, such as slower DRAM, in which case access to chunks in the extension would be slower, but preferable to system failure. The need for heap extension might happen when operating with debug chunks and might not occur in a released system using only inuse chunks. If it did, it is likely that there would be only a few slow chunks, so performance might

suffer for only a few unlucky tasks. Note that doing cache-line-aligned accesses could greatly improve data block access times if in DRAM.

Reserve memory could also be memory that is available in some systems and not in others. For example, the system used for debugging might have more memory than production systems. In this case, if no reserve memory is used after long runs it could be assumed that production systems have enough memory.

Unfortunately, reserve memory may not be available in a specific embedded system. Therefore, one of the other solutions suggested in the above heap recovery section would need to be used.

Appendix A API

These services meet the ANSI C/C++ Standard for malloc(), free(), realloc(), and calloc() and offer many additional services. eheap supports multiple heaps. Each heap has its own EVH structure, which is defined in eheap.h. This structure contains all static variables needed by eheap. eh_hvp[hn] is the heap variable pointer for heap hn. Space for eh_hvp must be allocated for each heap. For systems without multiple heaps, the heap number, hn, defaults to 0. "eh_hvp[hn]->" has been omitted ahead of heap variables in the following descriptions, for clarity, but it is required in code that accesses them.

eh_BinPeek

u32 eh_BinPeek (u32 binno, EH_PK_PAR par, u32 hn=0)

Summary Allows obtaining information concerning the heap bin specified by binno.

Parameters

	binno	Bin number.
	par	Desired information.
	hn	Heap number.

Returns

value	Value of par.
-1	Error.

Errors EH_INV_PAR Invalid parameter.

Descr Used to obtain information about heap bins. binno is the bin number. The parameter, par, is of type EH_PK_PAR. Available parameters are:

```
EH_PK_COUNT    Number of chunks in bin.
EH_PK_FIRST    Address of first chunk in bin, NULL if empty.
EH_PK_LAST     Address of last chunk in bin, NULL if empty.
EH_PK_SIZE     Minimum chunk size for bin.
EH_PK_SPACE    Free space in bin.
```

eh_BinPeek() reports EH_INV_PAR and returns -1, if par is not one of the above or if binno is not in the range 0 to top_bin. It returns 0 if the bin is empty. This service is recommended over directly reading bin parameters, because the latter can result in incorrect readings due to preemption by other tasks.

Example

```
CCB_PTR cp;

cp = (CCB_PTR)eh_BinPeek(14, EH_PK_FIRST);
```

This returns a pointer to the first chunk in bin 14.

eh_BinScan

BOOLEAN eh_BinScan (u32 binno, u32 fnum, u32 bnum, u32 hn=0)

Summary Scans forward through free bin list of binno for broken links and fixes what it can. Scans backward to fix broken forward links.

Parameters

	binno	Bin to scan.
	fnum	Number of chunks to scan forward per run.
	bnum	Number of chunks to scan backward per run.
	hn	Heap number.

Returns

TRUE	Done or unfixable error encountered.
FALSE	Call again to continue scanning.

Errors

EH_HEAP_BRKN	The free bin list is broken and cannot be fixed.
EH_HEAP_FIXED	A broken link in the free bin list has been fixed.
EH_INV_PAR	Invalid parameter.

Descr eh_BinScan() scans the free-bin list of bin binno and fixes broken links that it finds or reports EH_HEAP_BRKN if a link is unfixable. Normally it is called once per pass of heap manager and scans fnum chunks forward, per run. It must not be interrupted during a run by another heap service for the same heap. Scans are broken into runs, to permit higher priority tasks to access the heap and not miss their deadlines. If binno is out of range, or if either fnum or bnum is 0, EH_INV_PAR is reported and TRUE is returned.

A global pointer, bsp, points at the next chunk to scan, at the start of a run. If it is NULL, a new scan begins from the bin free forward link, ffl. bsp is set to NULL by eh_Init() or when a bin scan completes. Repetitively calling eh_BinScan() each time it returns FALSE, results in moving through the bin’s free chunk list, fnum chunks at a time, until the end of the list is reached and TRUE is returned.

If the bin is empty, TRUE is returned immediately. If broken, the bin’s free back link, fbl, is fixed first. If the bin's free forward link, ffl, is out of heap range, it and fbl are set to NULL and the bin’s bmap bit is cleared, causing the bin to be empty. Then EH_HEAP_BRKN is reported and TRUE is returned. In this case, the chunks that were in the bin are no longer available for allocation, however the heap can continue operating. If cmerge is ON, these chunks may eventually be merged with other chunks, as they are freed, and thus their free memory becomes available, again. Therefore, it may not be necessary to take further action.

If the bin has only one-chunk, TRUE is returned after fixing any broken links and binx8 in the chunk, if necessary.

If the bin has more than one chunk, cp is advanced one chunk at a time until fnum chunks have been checked or the end of the bin free list has been reached. The free forward link of each chunk is heap-range checked before use. If it fails, mode.fl.bs_fwd is turned OFF, bfp (bin fix pointer) is set to the end of the binno free list, and FALSE is returned. Thereafter, when eh_BinScan() is called, it will scan backward bnum chunks at a time, fixing broken ffl’s, as it goes, until it reaches bsp. Then mode.fl.bs_fwd is turned back ON and FALSE is returned. Thereafter, when eh_BinScan() is called, forward scan will resume from bsp. Normally, only the one broken ffl will need to be fixed – i.e. the one at bsp. If no further

broken links are found, forward scan will continue, fnum chunks per run, until the end of the bin is reached. Then the scan stops and TRUE is returned.

bsp and bfp are automatically corrected by eh_free() and eh_Malloc().

If the backward scan finds a broken back link before it reaches bsp, then it is not possible to fix the broken forward link at bsp. So, instead, the gap from bsp to bfp is bridged and EH_HEAP_BRKN is reported. The bridge allows the scan to finish and the heap to continue operating. This is like the broken bin ffl, above, but only part of the bin free list has been lost. See the Reliability chapter for more information.

Note (1) Whenever a fix is made, EH_HEAP_FIXED is reported, and the scan continues.

Example

```
void HeapMgr(void)
{
static u32 i = 0;
..
if (eh_BinScan(i, 10, 20))
   i = (i == eh_hvp[hn]->top_bin ? 0 : i+1);
...
}
```

This is an example of bin scanning from a heap manager. eh_BinScan() is called once per pass and it scans 10 chunks, at a time – probably enough for an average bin. Note that the backward scan number is twice as big. This is because backward scan is both faster and more urgent since a broken forward link has been found. When a bin is finished, eh_BinScan() returns TRUE, and i is incremented to scan the next larger bin. If the top bin has just been scanned, i is cleared and scanning starts over at bin 0.

If eh_BinScan() cannot fix a break, it reports EH_HEAP_BRKN. This should be treated as an irrecoverable error and appropriate action taken.

eh_BinSeed

BOOLEAN eh_BinSeed (u32 num, u32 bsz, u32 hn=0)

Summary Gets a big enough chunk to divide into num chunks for blocks of size bsz and puts them into the correct bin for their size.

Parameters

	num	Number of blocks.
	bsz	Size of each block, in bytes.
	hn	Heap number.

Returns

TRUE	Blocks seeded.
FALSE	Block not seeded due to error.

Errors Same as eh_Malloc() and eh_Free().

Descr This service is used to seed a bin with num chunks having block size, bsz. The bin is not specified because it depends upon the chunk size, which in turn depends upon debug mode being ON or OFF. If ON, debug-size chunks will be generated; if OFF inuse-size chunks will be generated. eh_BinSeed() shares internal subroutines with eh_Malloc() and eh_Free() and thus returns the same errors that they do.

eh_BinSeed() calculates the necessary chunk size for bsz and debug mode, multiplies it by num, and malloc's a big-enough chunk from the heap for that much space. It then splits the chunk into num chunks, physically links them together into the heap. Debug information is loaded into each chunk if debug mode is ON. cmerge mode is turned OFF, the new chunks are freed to the right-size bin, cmerge is restored, and TRUE is returned.

This service, combined with monitoring bin populations, may be a good way to maintain effective bin populations.

Note Due to the way malloc() works, the big chunk may be slightly bigger than necessary. As a consequence, the last chunk may be larger than the others and could be put into a higher bin.

Example

```
for (bin = 0; bin <= top_bin; bin++)
{
   if( (n = eh_BinPeek(bin, EH_PK_COUNT)) <= trgt_num[bin])
   {
      bsz = eh_BinPeek(bin, EH_PK_SZ);
      num = trgt_num[n] – n;
      eh_BinSeed(num, bsz);
   }
}
```

This function compares bin contents to a target size for each bin from 0 to the top_bin and seeds the bin if necessary, to bring it up to the target size. If applied to the SBA, this might help to improve performance for small block allocations. Bin seeding like this might be used during initialization to get the heap bins off to a good start.

eh_BinSort

BOOLEAN eh_BinSort (u32 binno, u32 fnum, u32 hn=0)

Summary Sorts a large bin's free chunk list by increasing chunk size.

Parameters

	binno	Bin number.
	fnum	Number of chunks to sort per run.
	hn	Heap number.

Returns

TRUE	Bin sort has been completed, was not needed, or was aborted due to an error.
FALSE	Call again to continue sorting this bin.

Errors EH_INV_PAR fnum is 0.

Descr eh_Malloc() and other eheap allocation services take the first large-enough chunk from a large bin. If the bin's free chunk list is sorted by increasing size, this will be the best-fit chunk in the bin. Thus, large-bin sorting helps to reduce fragmentation and to improve average allocation times for smaller chunks.

This service is used to put chunks in order by increasing size in large-bin free lists. A run consists of fnum sorting loops. It must not be interrupted during a run by another heap service for the same heap. fnum is chosen to be small enough so that higher priority tasks needing access to this heap do not miss their deadlines, yet large enough so that bins will generally be sorted when needed. Bin sorting is normally done during idle time.

The bin sort map, bsmap, has a bit per bin. The bit for a bin is set if eh_Free() puts a chunk at the end of the bin's free list. This happens when the chunk is larger than the first chunk in the bin. Otherwise the chunk is put at the start of the bin's free list, in which case no sort is needed and the bin's bsmap bit is not set. Small bins never need sorting, hence their bsmap bits are never set. Thus, bsmap shows only those large bins that need sorting.

There are two methods for using eh_BinSort(): First, if binno is less than or equal to the top bin number and its bsmap bit is ON, csbin = binno and sorting of this bin is started. Else, if its bsmap bit is OFF, no sorting is performed and TRUE is returned. Application code can pick any large bin and call eh_BinSort() each time FALSE is returned. When TRUE is returned, sorting of this bin is complete and application code can pick another bin to sort. This method gives exact control.

The second method is to call eh_BinSort() with binno greater than the top bin number. In this case, sorting of the smallest unsorted bin is started and csbin = that bin number. Application code continues calling eh_BinSort() each time FALSE is returned. When TRUE is returned application code calls eh_BinSort with binno > top bin in order to sort the next smallest unsorted bin. This method gives preference to smaller large bins.

A bin's bsmap bit is reset when a sort begins and csbin stores the bin number being sorted, between runs. If a preempting free sets the bit, due to putting a chunk at the end of the bin, the sort is aborted and restarted on the next run. If a preempting malloc takes a chunk from the bin, it also sets the bin's bsmap bit, causing the sort to start over. Starting over is not detrimental to a sort, because any sorting already done is preserved. Otherwise each run starts from where the last run left off.

Notes

(1) Heap sorting need not be perfect. Allocating a somewhat larger chunk than necessary due to imperfect sorting is not likely to be significant for heap operation.
(2) Because it is expected to run frequently, eh_BinSort() makes no entries in the event buffer, other than those due to reported errors or fixes.

Example 1

```
void eh_Manager(void)
{
   for(i = first_large_bin, i <= top_bin, i++)
   {
      while (!eh_BinSort(i, 4) {}
   }
   delay(n);
}
```

Example 2

```
void eh_Manager(void)
{
   while (!eh_BinSort(top_bin + 1, 4)
   {
      while (!eh_BinSort(top_bin + 1, 4) {}
   }
delay(n);
}
```

Example 1 one shows method 1 going methodically through all large bins every n time units. Example 2 shows method 2 where the first eh_BinSort() finds the smallest unsorted bin, if any, calls eh_BinSort() repetitively to sort that bin, finds the next smallest unsorted bin, if any, continues until all bins have been sorted, then waits n time units to start over. Method 2 is clearly more efficient if the objective is to maximize sorted large bins, especially the smaller ones. In both cases preemption is possible every 4 runs.

eh_Calloc

void* eh_Calloc (u32 num, u32 sz, u32 an=0, u32 hn=0)

Summary Allocates space for an array of num elements of sz bytes from the heap and clears all elements. See eh_Malloc() for details concerning allocations.

Compl eh_Free()

Parameters

num	Number of elements.
sz	Size of each element, in bytes.
an	Alignment number (block alignment = 2^an bytes).
hn	Heap number.

Returns

pointer	to allocated array.
NULL	Array not allocated due to error.

Errors Same as eh_Malloc()

Descr Allocates a single block of memory from the heap of (num * sz) bytes with fill mode OFF. The contents of the block are cleared, fill mode is restored, and a pointer to the block is returned. This service shares internal subroutines with eh_Malloc()and thus returns the same errors that it does.

Example

```
#define NUM_RECS 10

typedef struct
{
   u32 field1;
   u32 field2;
} REC;
```

```
REC *rp;
u32  i, error;

void array_op(void)
{
   if (rp = (REC*)eh_Calloc(NUM_RECS, sizeof(REC)))
      for (i = 0; i < NUM_RECS; i++, rp++)
      {
         rp->field1 = i;
         rp->field1 = 2*i;
      }
   else
      /* report error */
}
```

eh_ChunkPeek

u32 eh_ChunkPeek (void* vp, EH_PK_PAR par, u32 hn=0)

Summary Returns the information specified by par concerning a chunk in the heap, given a pointer to either the chunk or to the block in it.

Parameters

vp	Chunk or block pointer.
par	Desired information.
hn	Heap number.

Returns

value	Value of par.
-1	Error.

Errors

EH_INV_PAR	Invalid par.
EH_WRONG_HEAP	vp is not in heap n range or is not 4-byte aligned.

Descr Used to return information about heap chunks. The parameter, par, is of type EH_PK_PAR. Permitted values are:

```
EH_PK_BINNO      Chunk bin number (0 if not free, or in dc or tc).
EH_PK_BP         Data block pointer from cp (0 if free).
EH_PK_CP         Chunk pointer from bp (0 if free).
EH_PK_NEXT       Address of next chunk in the heap.
EH_PK_NEXT_FREE  Address of next chunk in this bin (0 if last chunk,
                   in dc or tc, or not free).
EH_PK_ONR        Chunk owner (0 if not debug chunk).
EH_PK_PREV       Address of previous chunk in heap.
EH_PK_PREV_FREE  Address of previous chunk in bin (0 if first chunk,
                   in dc or tc, or not free).
EH_PK_SIZE       Chunk size.
EH_PK_TIME       Time chunk allocated (0 if not debug chunk).
EH_PK_TYPE       Chunk type (free == 0, inuse == 1, debug == 3).
```

eh_ChunkPeek() returns -1 and reports EH_INV_PAR, if par is not one of the above. If a chunk is inuse, it cannot be in a bin, thus 0 is returned. Since 0 is a valid bin number, the chunk should be tested for free. Care must be taken that vp is a valid chunk pointer in all cases, except par == EH_PK_CP, in which case it must be a valid data block pointer.

Using this service is recommended over directly reading chunk parameters. The latter may result in incorrect readings, due to preemption by another task or due to attempting to read an invalid field for the chunk type. As shown above, eh_ChunkPeek() returns -1 in such a case. Also, chunk parameters cannot be directly read in umode. It usually is advisable to read the chunk type first to make sure that the expected chunk information is actually available. It also is advisable to check that the return value is not -1 or 0 before using it, except in the cases of bin number and type, where 0 returns are valid.

Example

```
u8*   bp;
CCB_PTR   cp;
int time = 0;
#define DEBUG 3

if (cp = (CCB_PTR)eh_ChunkPeek(bp, EH_PK_CP))
   if (eh_ChunkPeek(cp, EH_PK_TYPE) == DEBUG)
      time = eh_ChunkPeek(cp, EH_PK_TIME);
```

Starting with a block pointer, this example shows how to get the chunk pointer, cp, then determine when the block was allocated, if it is in a debug chunk.

eh_Extend

BOOLEAN eh_Extend (u32 xsz, u8* xp, u32 hn=0)

Summary Adds a memory extension to the heap.

Parameters

xsz	Extension size, in bytes.
xp	Extension pointer.
hn	Heap number.

Returns

TRUE	Heap extended.
FALSE	Heap not extended due to error.

Errors EH_INV_PAR xsz is zero or xp is not above current heap.

Descr eh_Extend() is used to extend the heap to additional memory space. xsz is the size of the additional space and xp is a pointer to the start of it. The space can come from other memory (e.g. DRAM), but it must be above the current heap. If not, eh_Extend() reports EH_INV_PAR, and returns FALSE. This is also the case if xsz == 0. Otherwise, xsz is increased to 16 or set to the next 8-byte boundary and xp is moved up to the next 8-byte boundary, if necessary.

eh_Extend() handles both the case where the extension is adjacent to the top of the current heap and the case where there is a gap in between. In both cases, EC (end chunk) is moved to the top of the extension. In the adjacent case, the extension is merged with the top chunk, TC, and the merged chunk becomes the new TC. In the gap case, an artificial inuse chunk is created from the old EC to cover the gap and the extension becomes the new TC. The old TC is freed to a bin. TC and the freed chunk are filled if fill mode is ON. Then tcp and hsz are updated and TRUE is returned.

Example

```
#define HEAP_EXT  4096
#pragma section="spare_space"
u8*  xp = __section_begin(spare_space);
BOOLEAN  ok;

if (smx_errno = EH_INSUFF_HEAP)
{
   ok = eh_Extend(HEAP_EXT, xp);
}

if (ok)
   /* retry allocation */
```

This example shows extending the heap by 4096 bytes in order to recover from an EH_INSUFF_HEAP error. The extension is taken from spare_space defined by the linker command file (for EWARM).

eh_Free

BOOLEAN eh_Free (void* bp, u32 hn=0)

Summary Frees a block to the heap that was previously allocated from the heap.

Compl eh_Malloc(), eh_Calloc(), and eh_Realloc()

Parameters

bp	Pointer to block to free.
hn	Heap number.

Returns

TRUE	Block freed or already free.
FALSE	Block not freed due to an error.

Errors

EH_HEAP_ERROR	Block is already free.
EH_INV_CCB	Forward or backward link is out of range.
EH_INV_PAR	Derived cp is out of range or not 8-byte aligned

Descr Frees the block pointed to by bp back to the heap. If bp is NULL, no operation is performed and TRUE is returned, per the ANSI C/C++ standard. If bp is not in heap hn range, EH_INV_PAR is reported and FALSE is returned.

If EH_BP (Block Pool enable) and bp is less than first heap chunk pointer, fhcp, then the block pointed to by bp is freed to either the 8-byte pool or to the 12-byte pool, depending upon bp. This operation is aborted and FALSE return if bpcbp, block pointer control block pointer, is NULL. bpcbp should point to an array of two pool control blocks allocated by the user. The pn field in each PCP points to the first block in the free block linked list. The freed block is put at the start of this list, pn is updated to point to it, and TRUE is returned. This operation is very fast compared to a normal free.

An attempt is made to detect a double free by testing the inuse bit of the word before bp. If 0

EH_HEAP_ERROR is reported and FALSE is returned. This test is not 100% effective because the chunk may have already been reallocated and thus pass the test. If not double free, bp is converted to its corresponding chunk pointer, cp and its prechunk pointer, PCP is derived. If either is out of heap hn range EH_INV_CCB is reported and FALSE returned.

If EH_SS_MERGE (Spare Space Merge enable), the prechunk is inuse, and its EH_SSP flag is set, the spare space in the prechunk is merged with the freed chunk. This is done to reduce internal fragmentation.

If merging is enabled (mode.fl.cmerge == ON), eh_Free() merges the prechunk if it is free and merges the postchunk if it is free. Chunks to be merged, except DC or TC, are removed from their bins before merging them. Then the bin for the final free chunk is found and the chunk is put into that bin, unless it is DC or TC. If a merger was made with DC, dcp is updated; if a merger was made with TC, tcp is updated. Only upward mergers into DC or TC are permitted and those chunks are never put into bins. heap_used is reduced by the size of the freed chunk

If chunk filling is enabled (mode.fl.fill == ON) a free block is loaded with the EH_FREE_FILL pattern; DC or TC is loaded with the EH_DTC_FILL pattern. This greatly increases the time required to free a block and should be used only to assist debugging.

If either hsp or hfp was pointing at the freed chunk and it was merged with the prechunk or spare space, the pointer is backed up to the new chunk. If a chunk is put at the end of a large bin, the bsmap bit for that bin is set, indicating that the bin needs to be sorted.

Example

```
void function(void)
{
   void *dp;

   dp = eh_Malloc(100);

   /* access block of memory via dp */
   eh_Free(dp);
}
```

This example gets a block of 100 bytes from the heap, uses it, then frees it back to the heap.

eh_Init

u32 eh_Init (u32 sz, u32 dcsz, u8* hp, EHV_PTR vp, u32 mode)

Summary Initializes a heap.

Parameters

sz	Size of the heap, in bytes.
dcsz	Donor chunk size. 0 means no donor chunk.
hp	Start of heap pointer.
vp	Pointer to heap variable structure.
mode	User-modifiable heap mode flags.

Mode Flags

EH_CM	chunk merge*	
EH_DBM	debug mode*	
EH_FILL	fill*	
EH_AM	automerge*	
EH_HFR	heap failure report	
EH_AR	auto recover	
EH_ED	error detection excluding allocation and free	
EH_EDA	error detection includeing allocation and free	
EH_EM	error manager	
EH_PRE	preemption protection	
EH_NORM	(EH_AM | EH_EDA | EH_EM | EH_PRE)	normal operation
EH_DBOP	(EH_NORM | EH_FILL | EH_HFR	debug operation

Returns

hn	Heap number (means heap has been initialized.)
-1	Heap not initialized due to an error or it was already initialized.

Errors

EH_ALREADY_INIT	This heap has already been initialized.
EH_INV_PAR	sz or hp is invalid.
EH_TOO_MANY_HEAPS	EH_NUM_HEAPS have already been initialized.

Descr

A heap must be initialized before it can be used. If the heap is used by C++ initializers, then eh_Init() must be called by the first allocation from that heap. See eh_Malloc() to see how this is done. Otherwise, eh_Init() should be called by startup code before the first heap allocation. hp can point anywhere in RAM and sz can be any desired size >= 32 bytes. Typically, a main heap is allocated from SRAM or DRAM and small dedicated heaps may be allocated from it. In some systems the main heap may be allocated from DRAM and a small fast heap may be allocated from SRAM

eheap maintains an array of pointers, eh_hvp[EH_NUM_HEAPS]. Each pointer points to the heap variable structure (EHV) for heap hn, where hn is the index into eh_hvp. Hence, eh_hvp[hn]-> is used to access heap hn variables. "eh_hvp[h]->" is omitted when referring to heap variables in this manual, but it must be included in code. Space for each heap variable structure is allocated by application code and certain fields must be set before calling eh_Init() — see the example below. When a heap has been initialized, its number, hn, is returned. Thereafter, this number must be used for all accesses to that heap.

If vp is NULL, -1 is returned and nothing is done. If sz < 32 or hp is NULL, EH_INV_PAR is reported; if eh_hvp[] is full, EH_TOO_MANY_HEAPS is reported; if mode.fl.init is ON, EH_ALREADY_INIT is reported. In all cases -1 is returned and nothing is done.

If they are not multiples of 8, sz is adjusted to the next lower multiple of 8 and hp is adjusted to the next higher multiple of 8. This is done so that the heap and all chunks in it will be 8-byte aligned. After the heap has been initialized, mode.fl.init is turned ON.

Following initialization, the heap consists of four chunks: start chunk (SC), donor chunk (DC), top chunk (TC), and end chunk (EC). SC and EC are inuse chunks with no data. They are each 8 bytes in size. DC is a free chunk, which initially contains dcsz bytes. TC is a free chunk, which initially contains the remaining free space of the heap = sz – dcsz - 16. DC normally is much smaller than TC; it is the source for small chunks. If dcsz < 24 DC becomes

a free chunk with no space for data and mode.fl.use_dc is turned OFF. TC is the source for large chunks. eh_Init() links all chunks together.

If EH_BP and bpcbp is not NULL, space is allocated from the heap for 8-byte and 12-byte block pools at the bottom of the heap between SC and DC and the pools are initialized. bpcbp points at an array of pool control blocks. Each pool control block has a num field, which is loaded by the application. If num is 0, no pool is created. Otherwise, a num pool is created and its pool control block is initialized.

eh_Init() loads pi = SC and px = EC. It initializes the mode field so that cmerge, debug and fill flags are OFF and other flags are ON. It also initializes the bins and other heap variables. If hmode.fl.fill is ON, DC and TC are filled with the EH_DTC_FILL pattern.

The mode flags shown are user-modifiable. They enbable corresponding heap operations, if set. If a flag is present in the mode argument, it is set in eh_hvp[n]->mode; otherwise, it is reset in eh_hvp[n]->mode. The flags marked with * can also be set and reset with eh_Set(). The EH_PRE flag is set if eheap calls are protected from preemption by a mutex or similar means. This flag is 0 if preemption protection is not present. This would be case, for example, if the heap is used by only one task. EH_PRE off results in faster performance.

See the Setup chapter for detailed information on setting up and initializing heaps.

Example

```
u32 hmain;  /* main heap */
memset((void*)&hmv, 0, sizeof(EHV));
hmv.bszap = (u32*)hm_binsz;
hmv.binp  = (HBCB*)hm_bin;
#if DEBUG
hmv.mode.fl.fill = ON;
#endif
hmain = eh_Init(hm_sz, hm_dcsz, hm_addr, &hmv);
eh_hvp[hmain]->mode.fl.amerge = OFF;
eh_hvp[hmain]->mode.fl.cmerge = ON;
```

As shown in this example, it is a good idea to clear the heap value structure, hmv, to ensure all fields start at 0. It is necessary to set the bin size array and bin pointers. Turning fill on ensures that DC and TC will be filled during debug. After initialization, it may be desirable to change some modes, as shown.

NOTE: Because heap numbers depend upon the order in which heaps are initialized, it is best to store heap numbers in variables named after heaps and to use these names in all heap operations, as shown above. If heaps are initialized by C++ initializers, there is no programmer control over order of initialization.

eh_Malloc

void* eh_Malloc (u32 sz, u32 an=0, u32 hn=0)

Summary Allocates a block of at least sz bytes from heap hn, aligned on at least a 2^an-byte boundary. Also can perform region block allocations.

Compl eh_Free()

Parameters

	sz	Size of block to allocate in bytes.
	an	Alignment number (block alignment = 2^an bytes).
	hn	Heap number.

Returns

bp	Block pointer.
NULL	Insufficient space or error.

Errors

EH_INV_PAR	Invalid parameter.
EH_INSUFF_HEAP	Insufficient space in heap.
EH_RECOVER	Automatic recovery has succeeded.

Descr Allocates a block of at least sz bytes and aligned on a 2^an-byte boundary from heap hn. A chunk is allocated from the heap to contain the block. If debug mode is OFF, an inuse chunk is allocated; if debug mode is ON, a debug chunk is allocated. The minimum block size that can be allocated from the heap is 16-bytes. The block size may be larger than sz, if an exact-fit chunk was not found.

Prior to searching the heap, If EH_BP and sz <= 12 a **block pool** allocation is attempted. If an > 3 or bpcbp == NULL, block pool allocation is skipped and heap allocation continues. If sz <= 8 and there is a block in the 8-byte pool, it is taken. If sz <= 12, there is a block in the 12-byte pool, and an <= 2 or bp is 8-byte aligned, it is taken. If a block is found, the PCB inuse field is incremented, the PCB maxuse field is incremented, if smaller, and bp is returned. Otherwise heap allocation continues.

If sz is 0, EH_INV_PAR is reported and NULL is returned. If sz is less than 16, it is rounded up to 16. If sz is not a multiple of 8, it is adjusted to the next higher multiple of 8. For example, if sz = 27, it will be adjusted to 32.

The search for the needed chunk progresses as follows until it is found: a small chunk is taken from the right-size bin in the small bin array (SBA), the donor chunk (DC), the next larger occupied bin, or the top chunk (TC). DC is used ahead of the next larger occupied bin in order to populate the SBA as quickly as possible. A large chunk is taken from the upper bin for its size, or the next occupied bin, or TC. If allocation fails automatic recovery is attempted if mode.fl.auto_rec is ON. If mode.fl.auto_rec is OFF or if auto recovery fails, EH_INSUFF_HEAP is reported and NULL is returned. Auto recovery may be disabled in order to implement a custom recovery process that calls eh_Recover() directly.

All blocks from the heap are automatically 8-byte (an = 3) aligned. If greater alignment is needed an aligned block search is performed. This requires EH_ALIGN; if not, EH_INV_PAR is reported and NULL returned. The aligned search is like the above search. For each candidate chunk, the distance, d, to the next 2^an boundary is added to sz in order to determine if a large-enough chunk has been found. (Of course, if the block in the chunk is

already aligned, then d == 0.) When a big-enough chunk is found, its CCB is moved up under the new aligned block, creating spare space below the new chunk. This space is merged into a free prechunk or added to the spare space in an inuse or debug prechunk. If EH_R is added to the alignment number to form the an parameter, a region block allocation will be performed. See Section 4 for more information on both of these types of allocation.

The found chunk is marked inuse and split if its spare space is greater than or equal to EH_MIN_FRAG. The upper part becomes a free chunk, which is merged into a free postchunk if mode.fl.cmerge is ON. If the found chunk's spare is less than EH_MIN_FRAG, its EH_SSP flag is set (bit 2 in blf) and its spare space pointer, ssp, is loaded into the last word of the spare space (= last word of the chunk).

If mode.fl.debug is ON, the chunk is converted to a debug chunk. In the searches above, the chunk overhead, CHK_OVH, used to find a big enough chunk is 8 if mode.fl.debug is OFF or sizeof(CDCB) + 8*EH_NUM_FENCES if mode.fl.debug is ON. EH_BP_OFFS is handled similarly. Hence the found chunk is big enough to put a CDCB (Chunk Debug Control Block) + EH_NUM_FENCES ahead of the data block, even if it is aligned, and EH_NUM_FENCES after the data block, ahead of spare space, if any. See the Debug section for more information.

The final chunk size is added to hsize and to hused, if necessary. hsize may be used to control merging. hused records the high-water mark of heap usage and is useful to determine if the heap needs more memory.

If mode.fl.fill is ON, the data block is filled with EH_DATA_FILL (d's) and spare space, except ssp, is filled with EH_FREE_FILL (e's). For a debug chunk, the fences were previously filled with EH_FENCE_FILL (0xaaaaaaa3). A's are chosen to show overwrites better than, for example f's or 0's. 3 is necessary for operation.

If allocation fails, NULL is returned and EH_INSUFF_HEAP is reported. This is a good reason for always checking the return value before using it.

If either bin scan pointer, bsp or bfp, was pointing at the chunk allocated, the bin scan is restarted. Also, the smx_bsmap bit for the bin is set, which results in the bin sort being restarted. If the either heap scan pointer, hsp or hfp, was pointing to a CCB that was moved or eliminated, the pointer is moved to the previous chunk that it tested.

Example

```
void* bp;

if (bp = eh_Malloc(204, 5, mheap))
{
   /* access block using bp */
   eh_Free(bp);
}
```

Since 204 is not a multiple of 8, the size is increased to 208. A block of 208 bytes, aligned on a 32-byte boundary, is allocated from the main heap. If the main heap is in DRAM and the cache line size is 32 bytes, this alignment will improve access times to the block. The block could be larger if an exact-fit chunk could not be found. When no longer needed, the block is released back to the heap by eh_Free().

eh_Peek

u32 eh_Peek (EH_PK_PAR par, u32 hn=0)

Summary Returns information concerning the heap mode.

Compl eh_Set()

Parameters

par	Desired information.
hn	Heap number.

Returns

value	Value of par.
-1	Error.

Errors EH_INV_PAR Invalid parameter.

Descr Used to obtain information about the heap. The parameter, par, is of type EH_PK_PAR. Permitted values are:

```
EH_PK_AUTO     Automatic chunk merge control is enabled.
EH_PK_BS_FWD   Bin scan forward.
EH_PK_DEBUG    Allocate debug chunks.
EH_PK_FILL     Fill blocks, spare space, dc, and tc with unique
               patterns.
EH_PK_HS_FWD   Heap scan forward.
EH_PK_INIT     Heap has been initialized.
EH_PK_MERGE    Merge chunks when freed.
EH_PK_USE_DC   Allocation from donor chunk is enabled.
```

eh_Peek() returns -1, and reports EH_INV_PAR, if par is not one of the above. Otherwise, it returns the value of the mode (ON or OFF). Using this service is recommended over directly reading heap modes, because the latter can result in incorrect readings due to preemption by other tasks.

Example

```
if (eh_Peek(EH_PK_MERGE) )
/* chunks are being merged, when freed */
else
/* chunks are not being merged, when freed */
```

This might be used to monitor how automatic merge control is doing or to decide what action to take if a heap failure has occurred.

eh_Realloc

void* eh_Realloc (void *cbp, u32 bsz, u32 an=0, u32 hn=0)

Summary Allocates a new size block from an existing heap block. Preserves existing contents and conforms to the ANSI C/C++ Standard. See eh_Malloc() for details concerning allocations.

Compl eh_Free()

Parameters

	cbp	Pointer to block to reallocate.
	bsz	New block size.
	an	Alignment number (block alignment = 2^an bytes).
	hn	Heap number.

Returns

	nbp	New block pointer.
	NULL	Insufficient space or error.

Errors Same as eh_Malloc() and eh_Free().

Descr Reallocates an existing block pointed to by cbp to a new block of size, bsz, and returns a new block pointer, nbp. Can be used to either downsize or upsize the current block @cbp. eh_Realloc() is more complex than the other two heap allocation services. However, it uses eh_Malloc() and eh_Free(), so the same discussion for them concerning size, errors, etc. applies to it.

Per the ANSI C/C++ Standard: if cbp == NULL, a block of bsz bytes is allocated from the heap; if bsz == 0, cbp is freed to the heap. Otherwise, if cbp is not within heap hn range or not 8-byte aligned, EH_INV_PAR is reported and NULL is returned. If bsz is greater than 0, but less than 16, it is automatically rounded up to 16 and if bsz is not a multiple of 8, it is rounded up to the next multiple of 8.

The current chunk size is determined and the necessary new chunk size is determined. If mode.fl.debug is OFF the latter will be for an inuse chunk, else it will be for a debug chunk. This is true, regardless of the type of the current chunk, which is being reallocated. Hence, eh_Realloc() can be used to convert an inuse chunk to a debug chunk or vice versa, without losing data in the data block. eh_Realloc() can also be used to increase the alignment of the block. Either of these is likely to require a new chunk.

There are two possibilities for reallocation, due to relative chunk sizes:

current chunk is big enough, then it is split into a new, exact-fit chunk and a new free chunk[2]. The new free chunk is merged with the chunk after[3], if it is free and cmerge is ON. The block pointer returned, nbp, is the same as cbp and the block size is equal to or slightly larger than bsz[4]. Note that data up to the new size is preserved and that data above that size is lost.

[2] There is a limitation on chunk splitting. See discussion in the chunk splitting section.
[3] When discussing chunks, “before” and “after” or “lower” and “upper” refer to physical chunk positions.
[4] See discussion in eh_Malloc().

current chunk is not big enough, then the current chunk is freed. This may result in its being merged with a lower free chunk or an upper free chunk, or both, which could result in a chunk that is now big enough for the new block. However, the odds of that occurring are small, so the new free chunk is put into a bin, and eh_Malloc() is called to get the best-fit chunk that can be found. Then data is copied from the current block to the new block, if necessary[5], and the new block pointer, nbp, is returned. Also, the unused upper portion of the chunk is split off into a new free chunk, if it is big enough[2].

If a big-enough chunk cannot be found, the preceding free, merge, and bin load operations are reversed, and eh_Realloc() fails, EH_INSUFF_HEAP is reported, and NULL is returned. In this case, the initial block is undisturbed and can continue being used via the cbp pointer. Means to recover from this failure are the same as described for eh_Malloc().

In all cases, data is preserved up to the end of the current block or to the end of the new block, whichever is smaller. To ensure this, fill mode is turned OFF, then restored at the end of this service. Thus, heap fill is suspended for all eh_Realloc() operations.

Example

```
void *bp, *nbp;

bp = eh_Malloc(200);
/* use 200-byte block via bp */

/* need another 200 bytes */
nbp = eh_Realloc(bp, 400);
/* use 400-byte block via nbp */
```

This example allocates 200 bytes from the heap, uses it for a while, then increases the block size to 400 bytes. When a block is being increased in size, the most likely scenario is that a larger chunk will be allocated elsewhere in the heap, the data from the old block will be copied to the new block, then the old chunk will be freed. In the above example, nbp is unlikely to be the same as bp. Hence, care must be exercised to update any secondary pointers (e.g. read pointer, write pointer, etc.). The contents from byte 0 to byte 199 of the original block are guaranteed to be unchanged, even though the block may have been moved.

eh_Recover

BOOLEAN eh_Recover (u32 sz, u32 num, u32 an=0, u32 hn=0)

Summary Tries to find enough adjacent free chunks that can be merged to create a chunk large enough for a block of sz bytes with alignment an. See eh_Malloc() for details concerning allocations.

Parameters

sz	Block size needed.
num	Maximum number of chunks to scan.
an	Alignment number (block alignment = 2^an bytes).
hn	Heap number.

[5] It is possible that the chunk and data block do not move, even though they are larger, in which case block contents are not copied.

Returns

TRUE	Chunk is now available to allocate.
FALSE	Chunk not found.

Errors

EH_INV_PAR	Invalid parameter: sz or num = 0.

Descr

This service is intended to recover from a situation where a large chunk cannot be allocated because this heap has been fragmented into too many smaller free chunks. Recovery is possible only if enough free space is found in adjacent free chunks. Otherwise, this service fails and some other means must be used to allocate the needed chunk.

eh_Recover() starts the scan from SC for small chunks or from DC for large chunks. All scans go to the end of the heap at EC before quitting. It searches for adjacent free chunks to merge. If a big-enough chunk can be formed by merging adjacent free chunks, it removes the free chunks (except DC and TC) from their bins and merges them. If the merged chunk is not DC nor TC, it puts the merged chunk into its proper bin, else it updates dcp or tcp, then returns TRUE.

This service does not merge chunks that it cannot use nor that it does not need. mode.fl.cmerge is ignored. If successful, eh_Recover() should be followed by retrying the allocation that failed. If mode.fl.auto_rec is ON, this is done automatically and the allocation returns a block, if one is found. In this case, recovery is transparent to the application, except that the allocation will take much longer than normal and EH_RECOVER will be reported by it. Returns FALSE if a big-enough chunk is not found and the allocation fails.

If eh_Recover() is called directly (mode.fl.auto_rec = OFF), it will search for num chunks and return FALSE if nothing is found. This is intended to put a limit on search times for very large heaps; it allows application recovery code to try another approach or to simply move on. Allocation failure is most likely to occur for large blocks while the heap is still usable for smaller blocks. In time, the large block allocation might be tried again and might succeed.

If num expires on a free chunk, the scan continues until a big-enough free space is found, an inuse chunk is found, or the end of the heap is reached. If a big-enough free space is found, the chunks are merged and TRUE is returned.

Example

```
void* bp;
TCP_PTR StoppedTask;

void ProcessTaskMain()  /*for mode.fl.auto_rec = OFF */
{
   while (1)
   {
      if (bp = eh_Malloc(1000, 0, fheap))
      {
         /* process data using bp */
         eh_Free(bp);
      }
      else
         break;
   }
   smx_TaskStartPar(RecoveryTask, 1000);
   StoppedTask = self;
}
```

```
void RecoveryTaskMain(u32 size)
{
   if (eh_Recover(size, 10000, 0, fheap))
     smx_TaskStart(StoppedTask);
else
   /* use alternate recovery */
}
```

In the above example, if eh_Malloc() fails in ProcessTask, RecoveryTask is started with the needed size as a parameter, ProcessTask's handle is saved in StoppedTask, and ProcessTask autostops. When RecoveryTask runs, it calls eh_Recover(), which tests up to 10,000 chunks. If it finds a big-enough chunk it returns TRUE, which restarts ProcessTask. If not, ProcessTask remains stopped while alternate recovery techniques are tried, such as extending fheap, using a different heap (e.g. mheap), releasing unneeded blocks, restarting ProcessTask, or rebooting the system

eh_Scan

BOOLEAN eh_Scan (CCB_PTR cp, u32 fnum, u32 bnum, u32 hn=0)

Summary Scans forward through heap hn for errors and makes fixes when it can. Scans backward through the heap to fix broken forward links.

Parameters

cp	Chunk pointer to start scan. Start at hsp, if cp == NULL.
fnum	Number of chunks to scan forward per run.
bnum	Number of chunks to scan backward per run.
hn	Heap number.

Returns

TRUE	Stop scanning – done or unfixable error encountered.
FALSE	Continue scanning.

Errors

EH_HEAP_BRKN	Heap cannot be fixed.
EH_HEAP_FENCE_BRKN	Broken fence found (fixed in release version).
EH_HEAP_FIXED	A heap fix was made.
EH_INV_PAR	Invalid parameter: fnum or bnum == 0.
EH_WRONG_HEAP	cp does not point within heap hn.

Descr eh_Scan() is intended to perform frequent heap scans and to fix or report heap problems that it finds. Normally it is called once per pass of the idle task and scans fnum chunks forward or bnum chunks backward. It should not be interrupted by another heap service during a scan.

cp can be set to start a scan at a specific chunk in the heap, however, it is usually set to NULL, in which case, the scan starts from hsp (heap scan pointer), which is where the last scan left off. Repetitively calling eh_Scan() with cp == NULL, results in forward scanning through the entire heap, fnum chunks at a time, until the end of the heap is reached. Then TRUE is returned and hsp is set to SC.

When a chunk scanned, its forward link (fl) is first checked that it points after the current chunk and before EC. If not, an attempt is made to fix fl, using the chunk's size, if it is a free or debug chunk (inuse chunks have no size field). If this fails, then mode.fl.hs_fwd is turned OFF, hfp (heap fix pointer) is set to the end of the heap, and FALSE is returned. As a

consequence, the next time eh_Scan() is called, it will scan backward bnum chunks, per run, until it reaches hsp. It then fixes the broken fl, EH_HEAP_FIXED is reported, mode.fl.hs_fwd is turned back ON, and FALSE is returned. The next time eh_Scan() is called, the forward scan will resume.

While forward scanning, if fl is ok, the back link of the next chunk is checked that it points to the current chunk. If not, it is fixed. If bl was wrong, it is assumed that the flags were also wrong and an attempt is made to fix them. Then flags of the current chunk are tested and if wrong an attempt is made to fix them. In this case, the SSP flag will be lost. For a free or debug chunk, size is checked and fixed, if wrong. For a debug chunk, the lower and upper fences are checked. If a broken fence is found for the debug version of smx (EH_BT_DEBUG == 1), eh_Scan() reports EH_HEAP_FENCE_BRKN and returns TRUE. This stops the scan so that the broken fence can be inspected. In the release version, broken fences are fixed, and the scan continues.

Whenever a fix is made, EH_HEAP_FIXED is reported and the scan continues. FALSE means to continue and TRUE means to stop. If free() preempts between runs and merges the chunk pointed to by hsp or hfp with a lower free chunk, it backs up smx_hsp or smx_hfp to point to the new chunk. Malloc() operations do not affect these pointers.

If the backward scan finds a broken back link before it reaches hsp, then it is not possible to fix either link. So, instead, the gap is bridged from the chunk at hsp to the chunk at hfp and EH_HEAP_BRKN is reported. This is done by setting hsp->fl = hfp and hfp->flb = hsp + flags. The bridge allows the scan to finish and may allow the system to limp along, but stronger measures are needed. More frequent scanning will reduce the likelihood of double breaks.

See Reliability chapter for more information on heap scanning.

Note Because it is expected to run frequently, eh_Scan() makes no entries in the event buffer, other than those due to reported errors or fixes.

Example

```
void IdleMain(void)
{
   ...
   eh_Scan(NULL, 2, 100);
   ...
}
```

This example shows heap scanning in the idle task. eh_Scan() is called once per pass through IdleMain() and will continuously scans 2 chunks, per run, starting over when it reaches the end of the heap. It will fix what it can and report what it can’t.

If the heap has 200,000 chunks it will take 100,000 passes to scan. This might be too often; if slowed down to once per tick, it would take 1000 seconds (about 17 minutes) to complete a pass. Note that a backward scan will cover 100 chunks at a time. This is because the backward scan is both faster and more urgent. The frequency of heap scanning depends upon the expected frequency of heap damage, which depends upon the system's environment.

If eh_Scan() cannot fix a break, it reports EH_HEAP_BRKN. This should be treated as an irrecoverable error by the error manager.

eh_Set

BOOLEAN eh_Set (EH_ST_PAR par, u32 val, u32 hn=0)

Summary Sets the specified heap mode to ON or OFF.

Compl eh_Peek()

Parameters

	par	Parameter to set.
	val	Value to set.
	hn	Heap number.

Returns

TRUE	Parameter has been set.
FALSE	Parameter has not been set due to error.

Errors EH_INV_PAR Invalid parameter

Descr Used to control heap modes. par is of type EH_ST_PAR. Available parameters are:

```
EH_ST_AUTO     Automatic free chunk merge control.
EH_ST_DEBUG    Debug mode control.
EH_ST_FILL     Block fill mode control.
EH_ST_MERGE    Free chunk merge control.
```

and the available values are ON and OFF. These modes are discussed in detail in UG sections. Briefly: EH_ST_AUTO enables automatic control of chunk merge (cmerge) implemented in the idle task. EH_ST_DEBUG controls debug mode, which causes allocations to create debug chunks. EH_ST_FILL controls fill mode, which enables filling blocks with patterns, when allocated or freed. It also enables filling DC and TC with patterns. EH_ST_MERGE control cmerge mode, which applies to free operations. If par is not recognized, returns FALSE and reports EH_INV_PAR.

Using this service is highly recommended over directly setting internal heap modes, which may result in incorrect settings due to preemption of the current task. Also, direct heap mode setting is not possible in umode.

Example

```
eh_Set(EH_ST_MERGE, ON);
```

This example turns on cmerge mode so that blocks being freed will be merged with adjacent free blocks.

Appendix B Glossary

Terms used in this manual are defined below. All heap variables are fields in the eheap variable structure, EHV, defined in eheap.h. Heap variables are referred to by field name, e.g. dcp, but in code they must accessed via the pointer returned by eh_Init(), e.g. eh_hvp[hn]->dcp, where hn is the heap number. mode is a bit field structure, HMODE, defined in eheap.h. Herein modes are referred to as bit fields, e.g. mode.fl.cmerge. Prefixes such as EH_ and eh_ are omitted below.

allocation policy means specifying how a best-fit chunk is found and also specifying the minimum remnant size for splitting a new chunk from a larger chunk that has been found. The allocation policy effects performance vs. memory efficiency.

automatic merge is controlled by the **mode.fl.amerge** flag. It starts ON and can be turned OFF or ON via eh_Set(). In the ON state, automatic merge control is enabled. In the OFF state, chunk merging can be manually controlled.

best-fit chunk is the chunk in a large heap bin, which is the smallest chunk that is big enough to satisfy an allocation request. If the bin is sorted, by increasing size, this will be the first large-enough chunk found in the bin.

bin See heap bin**.**

bin leak occurs when cmerge is ON and chunks freed are merged with adjacent free chunks and the resulting larger free chunks are moved to larger bins. Also occurs when cmerge is OFF and chunks are split and remnants are moved to smaller bins.

bin-type heap A heap that uses bins to "store" free chunks. Chunks are not actually moved from the heap to the bins. Rather they are linked to the bins. Each bin stores one or more chunk sizes.

block pool A pool of equal-size blocks controlled by a Pool Control Block (PCB).

bmap **bin map** has one bit per bin. If the bit is set, the bin contains at least one chunk.

BPCB **block pool control block** controls an eheap block pool. It has the number of blocks in the pool, number inuse, maximum number inuse, pointers to the first and last blocks of the pool, and the free block list pointer.

bridge is formed when heap links cannot be fixed by eh_Scan(). When this happens, the chunk with a broken forward link is linked to the chunk with a broken backward link. Thus, many chunks may be bridged over.

bs_fwd mode is controlled by the **mode.fl.bs_fwd** flag. It starts ON and controls the direction of heap bin scans. It is an internal mode and is not user controlled.

bsmap **bin sort map** has one bit per bin. If the bit is set, the bin needs to be sorted.

CCB **chunk control block** is placed at the start of a free chunk. It provides information necessary to manage the free chunk. A CCB contains 24 bytes for a free chunk and 8 bytes for an inuse chunk.

Appendix B

CDCB **chunk debug control block** is placed at the start of a debug chunk. It provides information necessary to debug heap problems. A CDCB contains 24 bytes.

chunk A block of memory used by the heap. A chunk consists of a chunk control block (CCB) used by the heap code and a data block used by the application. A chunk is thus larger than the data block, which it contains. The smx heap supports three types of chunks: free, inuse, and debug.

current chunk is the chunk that is currently being processed.

cmerge mode is controlled by the **mode.fl.cmerge** flag. Normally it starts OFF. In this state, chunk merging by eh_Free() is inhibited, which helps to maintain bin populations. When ON, chunks are merged with adjacent free chunks when freed. This helps to avoid allocation failures by reducing fragmentation. Can be turned ON or OFF via eh_Set(). See also **automatic merge**.

DC See **donor chunk.**

debug chunk is a heap chunk, which is currently inuse and which contains debug metadata. The debug metadata consists of a Chunk Debug Control Block, CDCB and fences surrounding the data block. The number of fences is user-specified.

debug mode is a flag in **mode.fl.debug.** It starts OFF and can be turned ON or OFF via smx_HeapSet(). When ON, allocations produce debug chunks; when OFF, allocations produce inuse chunks.

donor chunk is located between the lower heap and the upper heap. Initially it is located immediately after start chunk. It supplies small chunks for the lower heap, which are of SBA size. If the SBA bin for the desired size is empty, the chunk is taken from DC. This helps to maintain locality of SBA chunks. The minimum size for DC is 24 bytes = free CCB.

double free occurs when eh_Free() attempts to free a chunk, which has already been freed. If the chunk has not already been reallocated, this is detected and EH_HEAP_ERROR is reported. Double free is not 100% detectable.

dynamic merge control Control of heap chunk merging that is implemented via an algorithm to achieve good performance without causing fragmentation failure.

EC See **end chunk.**

end chunk is the last chunk in the heap. It is an 8-byte, inuse chunk with no data block. px points to it.

errno **eh error number** is a field in the EHV structure. See eheap.h for error types.

fence is a known pattern, such as 0xAAAAAAA3, in a debug chunk before and after the data block. The pattern is determined by EH_FENCE_FILL in eheap.h. This can be any pattern as long as bits 1 and 0 are 1's. (These are the *alternate* DEBUG and INUSE flags.) The number of fences after the data block is EH_NUM_FENCES (eheap.h) and before is EH_NUM_FENCES + 1.

fill mode is controlled by **mode.fl.fill** flag. It can be turned ON or OFF by eh_Set(). When ON, all blocks freed or allocated, DC, TC, and new fences are filled with unique patterns. When OFF they are not.

fragmentation as applied to a heap means that chunks become smaller and smaller and thus less useful. Severe fragmentation may result in failure to be able to allocate larger chunks. More accurately, this is known as **external fragmentation.** There also is **internal fragmentation,** which is the spare space after blocks in chunks. This also can cause allocation failure.

free() Generic heap free operation that frees inuse chunks to the heap.

free chunk A heap chunk that is not in use and thus free to be allocated. A free chunk consists of a 24-byte Chunk Control Block, CCB and free space.

free chunk list Doubly-linked list of free chunks in a heap bin. Free forward links (ffl's) and free backward links (fbl's) in the bin and in each chunk are used to create the list. All chunks in the list are of the correct size for the bin.

heap A heap is a region of memory from which variable-size blocks can be dynamically allocated and to which they can be dynamically freed, when no longer needed.

heap failure Inability for the heap to supply a desired size block. Usually caused by excessive fragmentation. This is indicated by the EH_INSUFF_HEAP error.

heap bin A heap bin is the head of a free list of doubly-linked chunks of a certain size or range of sizes.

heap block is a data block allocated from the heap. It is contained within a chunk.

heap range test is a test of a chunk pointer to verify that it is within the range of the heap, i.e.: pi <= cp <= px. eheap tests all pointers, before use, in order to find broken pointers and to avoid MMFs and data abort exceptions. If the heap has been extended over a gap, this test will be less effective. Note: If EH_SAFE is OFF, some heap range tests are disabled in order to improve performance.

hfp **heap fix pointer** points to the starting chunk for the next eh_Scan() backward run.

hhwm **heap high-water mark** is the largest value of hused since the heap was last initialized.

hs_fwd mode is controlled by the **mode.fl.hs_fwd** flag. It starts ON and controls the direction of heap scans. It is an internal mode, not user controlled.

hsp **heap scan pointer** points to the starting chunk for the next eh_Scan() forward run.

hused **heap used** is the total heap space currently allocated, including chunk overhead.

init mode is controlled by the **mode.fl.init** flag. It starts OFF and is set ON when the heap has been initialized. It can be turned ON or OFF by smx_HeapSet(). It must be turned OFF to reinitialize the heap.

internal fragmentation refers to spare space in a chunk due to it being larger than necessary for the block it contains.

inuse chunk A heap chunk, which is currently being used. It contains 8-bytes of metadata (CCB) plus the data block being used by the application.

large bin A heap bin that stores a range of chunk sizes, which are 8-byte aligned and multiples of 8 bytes.

large chunk A large chunk is one that fits into an upper bin.

last turtle is the last chunk in a large heap bin free list that might be smaller than a chunk before it. It is called a *turtle* because it moves forward very slowly in a bubble sort.

linear heap has only a physical structure and must be searched sequentially to find large-enough chunks to allocate.

localization means to allocate chunks, which are close in time, to be physically close in order to increase cache hits.

logical structure A heap structure that provides a more efficient means of searching for block allocations than the physical structure. eheap provides an array of heap bins for this purpose.

malloc() Generic name for heap allocation service.

memory leak normally occurs in a heap due to failure to free blocks when no longer needed. Reallocating the blocks, when needed again, results in steady loss of free heap space.

The debug chunk helps to identify leaked blocks by recording time of allocation and owner.

MIN_FRAG Configuration constant in eheap.h that defines the minimum fragment (remnant) that can be split off of a larger chunk during an allocation. This should be at least as large as the minimum chunk size that an application needs, in order to prevent accumulation of unusable small chunks.

physical heap structure consists of all chunks in the heap doubly-linked together in physical address order. Every chunk has a forward link, fl, and a backward link + flags, blf, for this purpose. The flags are PPC (bit2), DEBUG (bit 1) and INUSE (bit 0). Adding flags to the back link is possible because all chunks are 8-byte aligned, hence address bits 0, 1, and 2 are always 0 and not needed for addressing. The flags must be stripped from blf before using it as a pointer to the previous chunk.

postchunk is the chunk the follows the current chunk.

prechunk is the chunk that precedes the current chunk.

remnant is the remainder of a chunk after splitting a chunk. It must be at least MIN_FRAG (eheap.h) bytes or the initial chunk will not be split. It will always be above the allocated chunk. It will be merged with a fee postchunk if cmerge is ON.

SBA See **small bin array.**

SC See **start chunk.**

small bin A heap bin that stores a single chunk size.

small bin array (SBA) is an array of small heap bins in the bin[] array for heap hn, starting at size 24 and consisting of consecutive bin sizes that are multiples of 8 (e.g. 24, 32, 40, ...) up to sba_top bin. SBA bins can be accessed very quickly by converting the desired block size to an SBA index, i.e. binno = size/8 – 3.

small chunk A small chunk is one that fits into an SBA bin.

TC See **top chunk.**

tight heap A heap that has very little margin because of insufficient RAM and thus is prone to failure due to fragmentation.

top bin Top heap bin in bin[] for heap hn. It handles all chunk sizes from its minimum size up.

top chunk is the last chunk before the end chunk in heap hn. Initially, it and the donor chunk contain all free heap space. Allocations which cannot be satisfied by the SBA, donor chunk, nor larger bins come from TC.

upper bin array **UBA** is that portion of bin[] array that is above the SBA.

use_dc mode If the dcsz parameter in eh_Init() <= 24 or if there is no SBA, **mode.fl.use_dc** is turned OFF and DC will not be used. When ON, an SBA-size chunk is taken from the donor chunk if the SBA bin for it is empty. When OFF, allocation skips DC and goes to the next larger occupied bin. mode.fl.use_dc can be turned ON and OFF by eh_Set(), but this is error prone.

Index

Index