Dynamic Memory Allocation Optimizes Integration of Blackfin® Processor Software

By Lidwine Martinot [lidwine.martinot@analog.com]

Typical DSPs usually have a small amount of fast on-chip memory. Microcontrollers usually have access to larger external memories. The Blackfin processor has a hierarchical memory architecture that combines the best of both approaches, providing several levels of memory with differing performance levels. For applications that require the most determinism, it can access on-chip SRAM in a single core clock cycle. For systems that have larger code sizes, larger on-chip and off-chip memory is available—with increased latency.

By itself, this hierarchy is only moderately useful; today's high-speed processors would typically run at much slower speeds, because larger applications would only fit in slower external memory. To improve performance, programmers have the option of manually moving key code in and out of internal SRAM. Also, the addition of data and instruction caches into the architecture makes external memory much more manageable. The cache reduces the manual movement of instructions and data into the processor core. This greatly simplifies the programming model by eliminating the need to worry about managing the flow of data and instructions into the core.

While Blackfin's memory is versatile and easy to use in many applications, there are some applications, such as embedded cell phone systems, in which memory allocation can be difficult for *any* embedded processor. In this kind of application, the instruction cache does not provide the same level of code management as manual movement of data in and out of SRAM. This article suggests a dynamic memory allocation tool to deal with the challenge.

An essential element in the development of protocol stack and application software for mobile phone platforms is the efficient handling of memory resources in the system. In the past, memory resources were distributed "by hand" to each piece of code within the system; but the growing number of modules such as video and voice recognition makes solutions using this approach more challenging to optimize. A *dynamic memory allocator* can be used to allocate and free memory in a large application, removing the need to manage this task manually. This article describes some of the principles of dynamic memory allocation and demonstrates a specific implementation that takes into account the overall system considerations and the division of Blackfin's memory into different spaces with various properties (price, speed, dual-access possibility).

Memory management solutions

In a large embedded application, there are several memory-management approaches that can be realized. The major approaches are described below.

Stack. All variables and buffers can be simply declared on top of a function. They are stored in the Stack space, and that space is released only when exiting the function. The main disadvantage of this solution is Stack growth, e.g., the Stack keeps growing during the function's lifetime. Its lifetime can sometimes be very long, since the function may be recursive and/or interruptible.

Manual overlap. Another popular solution consists of hard-coding the buffer's address using sections defined at the *link* stage. This is a bit more flexible than allocating in the stack, because it allows memory overlap. If two modules are never going to interrupt each other, their temporary memory could share the same memory section. Yet, as the number of modules grows, this solution really becomes difficult to manage for an integrated system. In addition, other memory problems—such as inappropriate overlap, or insufficient buffer sizes for a given section—can be very hard to track. To make matters worse, it is even more difficult when a new feature is needed that requires two functions that have never previously overlapped in time to run concurrently. Figure 1 shows an example of a manual overlap-based implementation.

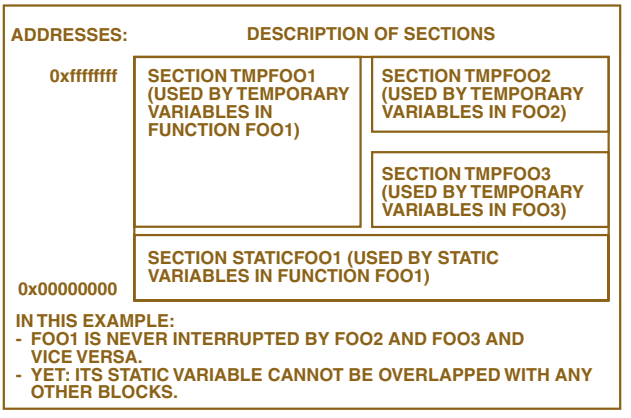

Figure 1. Manual overlap of memory.

Dynamic allocation. Dynamic allocation enables *memory overlap*: once a memory space is not needed, it is freed and can be reused. Unlike the stack allocation method, dynamic allocation does not result in an increase of uncontrolled memory space. In fact, the memory used by a function is released as soon as it is not required, rather than waiting for the end of the function.

What are the features to consider when developing a dynamic memory allocator?

A dynamic memory allocator is made up of two functions: one allocates memory space; the other frees memory. The allocation reserves some space to serve memory requests. When the *free* function has been called, the reserved space is freed and can be used to fulfill further requests. For example, let's build a very basic dynamic memory allocator to understand all the trade-offs such a piece of code has to deal with. We will start with some basic definitions and then describe the allocator.

Chunk. Let's assume the allocator can give the required memory a *chunk* of a big memory space. It is easy to understand that the whole space cannot be taken away to serve the first request. Instead, the initial memory space can be split into different chunks of different sizes.

Header. When a memory request is made, how do we know that a given piece is big enough? The *size* has to be kept in memory somewhere. One solution among others is to keep it in a *header* inside the chunk. This is an element of *memory overhead.* Also, at least one bit in the header needs to be dedicated to indicate whether the chunk is free or is in use.

Wandering through the chunks. If the first chunk is too small, how do we jump to the next chunk? If all chunks are consecutive in memory, it is enough to know the size of the chunk to jump to the next. Another solution consists of keeping a *pointer* to the next chunk in the header—this is the principle of *linked lists.*

Finding a fit. How do we select which free block is going to serve the request? A necessary condition is to find a free chunk whose size is at least the required size. The first chunk that meets this requirement can then be used. This policy is called the *first-fit*. Another policy, the *best-fit* policy, consists of looking for the *smallest free chunk* that can accommodate the request. This is the most challenging dilemma a dynamic memory allocator has to deal with: *speed versus memory size*. The first-fit is fast but might lead to huge memory losses, while the alternative of finding the best fit requires time. A compromise can be reached with the use of several linked lists of chunks (*bins*), in which each list has its chunks of a similar size. The *best-fit* policy selects the bin, while the *first-fit* selects the chunk within the bin.

Fragmentation. Another solution consists of using the first-fit policy—and releasing the end of the chunk that is bigger than the request. One downside of this solution is that soon the memory is made up of several scattered blocks (different in size, usually small) of unused memory. Future allocation is difficult due to the small free spaces that result. This situation is called *memory fragmentation*.

To speed a request, some allocators are based on linked lists of free chunks. This saves some time, since the search can avoid considering all in-use chunks. This method does have a disadvantage, however. If only the free chunks are kept in lists, it is hard to have all of them placed consecutively in memory; this problem prevents the allocator from being able to take two adjacent medium chunks and put them together (or *coalesce* them) to build a bigger one.

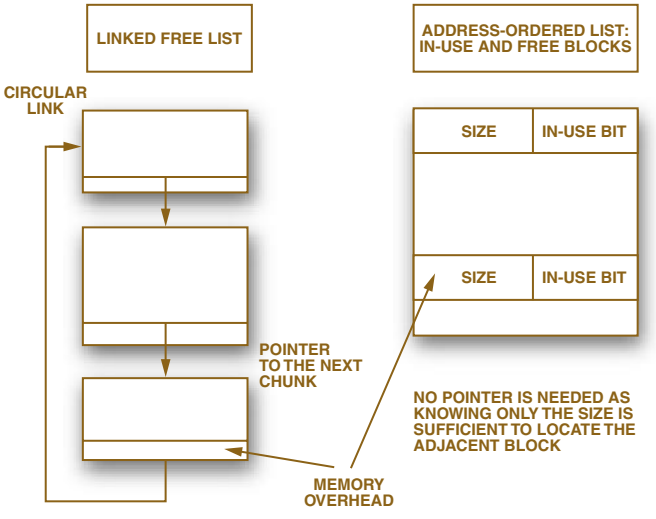

Figure 2. Examples of dynamic allocators.

We have now introduced all the concepts and compromises for understanding the allocator designed for the Blackfin mobile phone system: ADIalloc.

The current implementation: ADIalloc

The constant addition of signal-processing features (new video and audio standards, for instance) has motivated the development of an allocator referred to as ADIalloc for cell phone applications. It is intended to help reduce both *time-to-market* of the product using the processor—by avoiding undesired memory overlap—and *cost*, by reducing the peak memory usage.

Basic principles

The current implementation is more focused on speed performance than memory overhead. The memory is partitioned into *bins*. Each bin holds *chunks* of memory of equal size. The chunks in a bin have consecutive addresses, allowing a fast jump from one chunk to the next. The policy to find the chunk that suits the request is *best-fit* for the bin and *first-fit* within the bin—meaning the first free chunk, since all chunks have the same size. Moreover, the size of chunks in bins is chosen to facilitate finding the best bin: they are all related by powers of 2. Chunks in bin (N+1) are double the size of chunks in bin N (it is also possible for bin N to contain 0 chunks...)

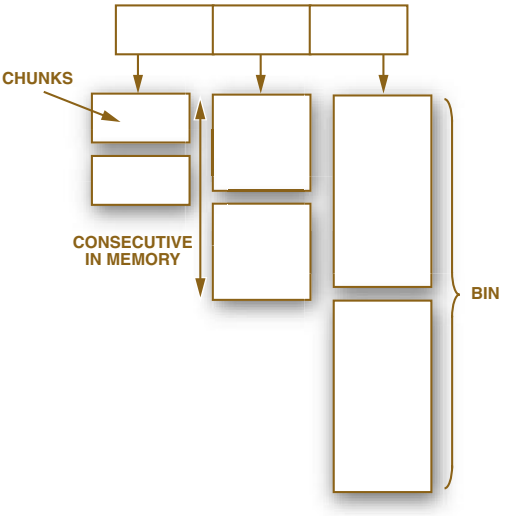

Figure 3. Bins/Chunks configuration of ADIalloc.

Some software modules may occasionally need one "big" chunk. However, if big chunks are allowed, the memory is going to be partitioned into very few chunks. Instead of one big chunk, it is better to have two smaller chunks that would be coalesced together to form a big chunk in the few cases where it is needed. Consequently, *coalescing* two chunks together is allowed.

To guarantee speed, each chunk has a header that indicates if it is available and coalesced. In the case of coalesced chunks, the size of the coalesced companion, or "buddy," is kept in the header. This is used to quickly restore the header of the buddy when the couple is freed.

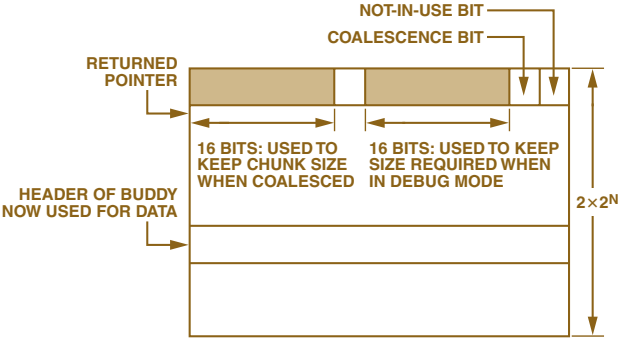

Figure 4. Chunks in ADIalloc.

What is specific to Blackfin

Blackfin adds yet another dimension to the memory allocator: its data memory space is partitioned into several memory levels. The memory levels have different characteristics in terms of price, speed, and dual-access possibility:

- The *external* memory, Lext, is big and less expensive to use—but is accessed with higher latency.
- The *on-chip* memory, L1, has fast access. It is itself split into different banks and sub-banks, allowing two items of data to be accessed at the same time (*dual access*) from separate sub-banks.

- L2 is in between, in terms of price and speed. However its speed can be improved by caching it into L1. *Caching* is an additional dimension.

Stack. Although (as seen earlier) allocating all variables in a Stack is not a good solution, a Stack is still needed. For small buffers, loop counters, and indexes there is no point to losing cycles because of allocation. Yet there might be some uncertainty about the allocation—stack or dynamic—of some buffers until the system-integration stage. This is why the Stack is seen as an additional memory level.

Cache. As mentioned above, Blackfin can cache L2 memory into L1—or parts of L1. In that case, it is advantageous not to have to readapt the allocator's code to the new memory. During initialization, the allocator is able to read the cache configuration from some dedicated Blackfin registers, and then decide about its bins and chunks. Yet since the allocator has to be tested on any platform, it must remain minimally Blackfin-specific. Only reading the data-cache configuration is Blackfin-specific. Apart from that, the allocator can be fully tested on a PC with a compiler other than Blackfin's. The only difference there is that choice of memory resource is not related to the platform's speed or dual-access features.

With all the above features ADIalloc becomes an important piece of software. Therefore it should be made as "flexible" as possible, as long as this does not overly impact the number of cycles.

Flexibility of the allocator

Macro. C-macros are extensively used in the ADIalloc implementation. Indeed *ADIalloc* is itself a macro. The first benefit is to be able to replace quickly one allocator by another without having to rewrite all pieces of software that invoke ADIalloc. For instance, this can be used to investigate the performance of different dynamic allocators.

Alloca. Another advantage of the macro is to be able to use Stack as a memory level without having to invoke the allocator in a more complex manner than would be done with a malloc. Indeed, allocating in Stack cannot be achieved through a function call. Instead, when ADIalloc is invoked with Stack as memory level, 'alloca' is executed. (Alloca is available with most compilers. It reserves space on the Stack only when the alloca instruction is executed—unlike the declaration on the Stack on top of a function which reserves the space for the function lifetime.) The macro ADIalloc tests the memory level required and redirects it to an alloca or to a function call to the allocator, ADI_alloc.

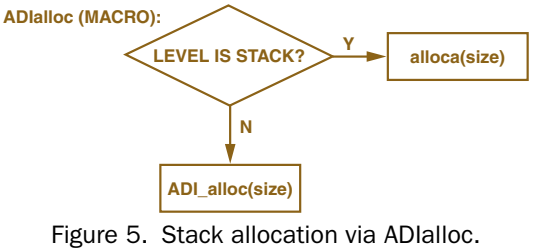

Figure 5. Stack allocation via ADIalloc.

Storage of the desired memory level. It is a really great advantage to be able to deal with the different memory levels on the Blackfin. To make the best use of this feature the memory levels are not fixed at compile time. Hence, for each allocation the allocator allows testing of different memory levels without having to rewrite or recompile the software module's C code. A software module is accompanied by a table that contains the memory level required for such or such allocation. The table's content can be changed at run time as simply as writing a new desired memory level at a specific address. Nevertheless it should be noted that if the memory level required cannot be provided, the allocator picks up another level—the closest one in terms of memory access speed.

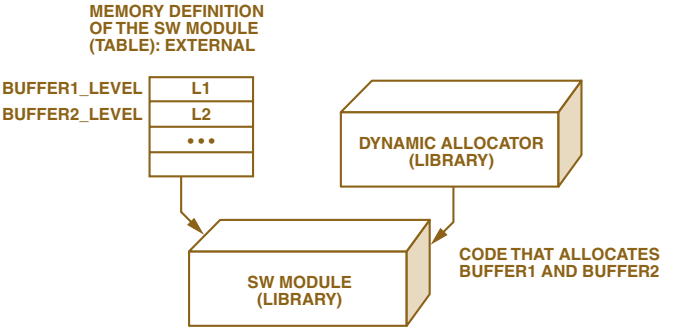

Figure 6. Input table: desired memory level.

Change Bins/Chunks Configuration. Another flexible feature of ADIalloc is the ability to change the bins and chunks configuration without having to recompile the allocator's code. Indeed all variables defining this configuration are saved into tables. The tables are read during the initialization. At any time the tables' content can be changed—which will modify the bins/chunks configuration the next time the initialization is called. Not having to fix the bins/chunks split at compile time leads, as a next feature, to having a smart wrapper around the allocator that dynamically resizes the memory. We can also think of a system running two consecutive tasks that require two different memory configurations. When a task finishes, the allocator initialization is called with the configuration that best suits the second task.

Finally, ADIalloc is derived in two flavors: the first is used for development and integration, the second one is used in the final product. During development *debug* features are mandatory. The next section provides further details of the current implementation and how to make the best use of debug features.

How debug features improve implementation

Common issues when using memory allocator are inefficiencies attributable to the allocator and the risk of not allocating and freeing the memory properly—resulting mainly in memory leakage.

The allocator knows the memory partition. It also knows the amount of memory requested and which memory addresses are free. This allows debug features to be developed to take steps to avoid memory leakage.

Track a free that has been forgotten. The first reason for a memory leak occurs when a memory is allocated but never freed. This can be easily prevented. In debug mode (not in normal mode, since this test takes many cycles) the allocator builds statistics of the memory usage. If the last report shows that some memory space is still in-use, it means a free has been forgotten. To track the problem more deeply, one can use another report which contains buffer names, their addresses, and if they are being freed or allocated (the report is built each time the allocator or the free function is called).

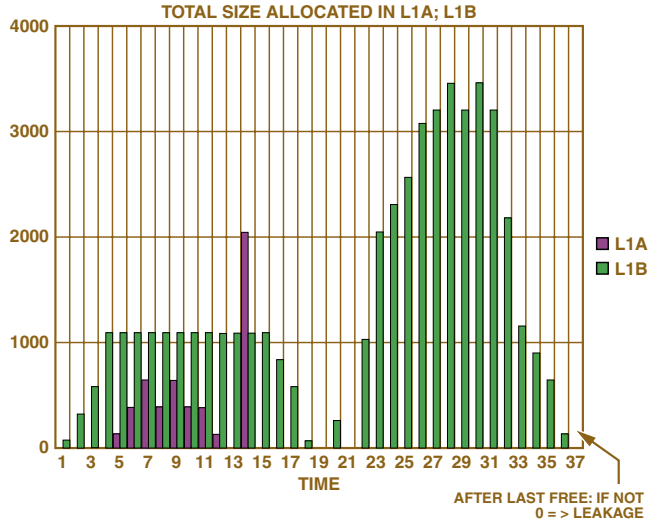

Figure 7. How to track a free that has been forgotten.

Track that more space than reserved is used. The other type of leakage occurs when a buffer allocates less space than what it needs, and starts using the space outside what has been allocated to it. In debug mode the allocator "marks" all free memory spaces with a special code (a code which has a very low probability of being a "real" datum). It not only marks free chunks, but also includes all the addresses inside a chunk not required by the allocation. In each allocated chunk the required size is also kept as part of the allocated chunk. Hence each time the allocator is entered (for a new allocation or a free) it verifies that:

- The free chunks only contain the special code
- The allocated chunks contain the special code between the required size and the end of the chunk

The function that does this check can also be called at any time outside the allocator. When leakage is noticed, a message is built and passed to another module which outputs it in one form or another (screen, special visualization tool, high-speed logger for real-time analysis, etc.)

```
ADIalloc:ptrB adi_free_log_when_leak_t xx;
ADIfree: ptrB xx.address_freed       = (void*)   0x00429374;
              xx.error_warning       = ERROR_PROBABLY_LEAKAGE;
              xx.address_of_leakage = (void*)   0x004291B4;
```

Figure 8. Example of a viewer to track the allocator messages (case of leakage).

Help select bins/chunks configuration. The allocator debug features can also partly resolve the concerns regarding the allocator inefficiencies. In debug mode the allocator saves such data as the memory required versus the memory allocated, the number of chunks used per bin, etc. This provides an easy way to avoid big inefficiencies—such as having some bin sizes that are never used.

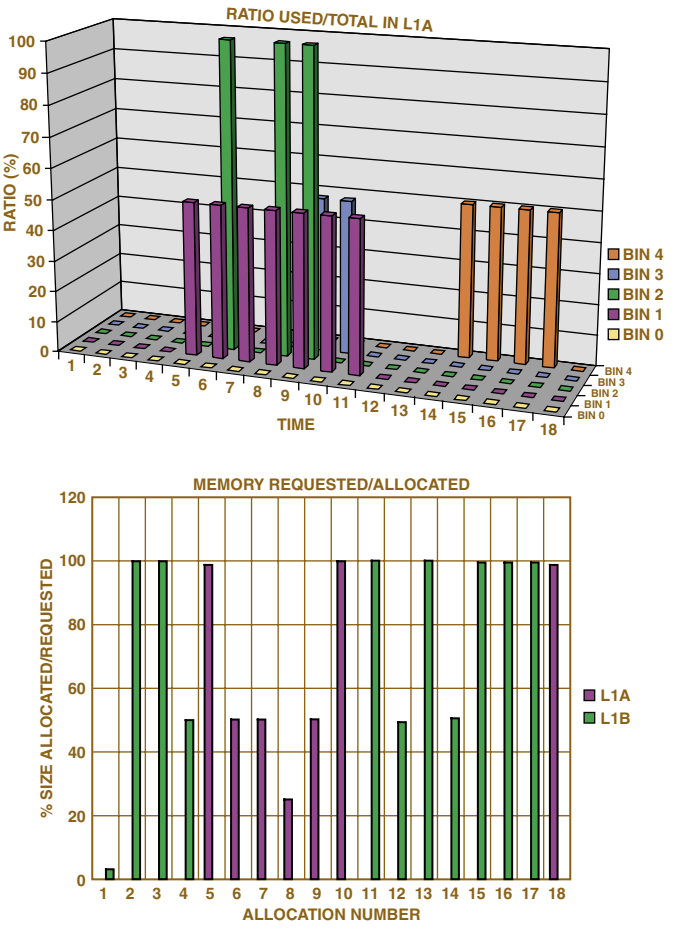

Figure 9. Data captured to help select the best Bins/Chunks configuration.

Memory repartition between memory levels. A big concern is then how to apportion the memory levels among the different pieces of software. Obviously, the fast-access memory suits best every single piece of code. Yet a choice has to be made since this memory is limited. This choice can be made only once whole software modules are built into a *system.* Usually the time-critical tasks will need the fastest memory. The allocator can assist in making such choices.

The allocator is all the more helpful, as it can be delivered with a wrapper that takes care of running all possible memory configurations for a specific module while conserving the number of cycles required. This helps one to know the impact on cycles of not being able to get the fastest memory for a specific buffer.

Table I. Performance Matrix

Index In Table	L1_B	PASS/FAIL	L2	PASS/FAIL	Lext	PASS/FAIL
pChannelInstance	–82	PASS	–71	PASS	–119	PASS
pSharedMemStruct	–73	PASS	–66	PASS	–109	PASS
pShared_BurstDec_CCDec_Interleave	94	PASS	56	PASS	–48	PASS
pShared_EQ_CCDec_Mod_Info	5	PASS	–81	PASS	–67	PASS
CC_Dec_IO_EDGE_PDTCH	130*	PASS	–74	PASS	324	PASS
pDeInterleave	–232	PASS	–57	PASS	18115	PASS
pOutHeader	15	PASS	–116	PASS	506	PASS
pScratch_Header_Decoder	–281	PASS	–83	PASS	3719	PASS
Metric	–82	PASS	10440	PASS	123346	PASS
pPathMetric	–417	PASS	–84	PASS	77394	PASS
pOutRLC_Data	–199	PASS	–83	PASS	1832	PASS
pScratch_Data_Decoder	–75	PASS	450	PASS	23624	PASS

*Means: +130 cycles if the buffer is in L1_B compared to the reference configuration.

The numbers shown in the table represent the difference in the number of cycles needed to run the unit test in the new configuration as compared to the reference configuration.
The Reference Configuration is what is provided as default by the module's writer.
PASS indicates that the result of running the unit test on the new configuration is the same as that of running the reference configuration.
The Reference Number of cycles is: 128078.

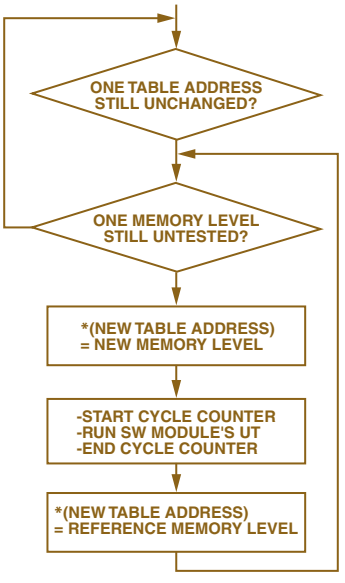

Figure 10. Unit test flowchart.

The wrapper runs a software module *unit test* (UT). The first time it runs it, the allocator is asked to return the pointer's name and the address of the table where it looks for the memory level. After collecting all addresses where it needs to look for memory levels, the wrapper re-runs the UT for all possible memory configurations.

CONCLUSION

The current ADIalloc implementation is one possible implementation of a dynamic memory allocator. Its use has shown that the most useful features of the current implementation are the *debug* features. They reduce the risks linked to dynamic allocation (especially the risks of leakage). At the same time they help better manage complex memory structures. It has now become much easier in cell phone applications to add new software modules inside Blackfin without having to rework the division of memory between modules.

ACKNOWLEDGEMENT

The author acknowledges the invaluable contributions of Rick Gentile, of the Blackfin Applications Group, and Zoran Zvonar, DSP/Systems group leader.