User Involvement
Techniques for Handling Memory Constraints in the UI

Abstract:

This paper describes some UI design patterns to use when creating software to run in limited memory.

It is a draft version of a chapter to add to the authors' book Small Memory Software, and follows the structure of other chapters in that book.

Major Technique: User Involvement

How can you manage memory in an unpredictable interactive system?

- Memory requirements can depend on the way users interact with the system.
- If you allocate memory conservatively, the systems functionality may be constrained.
- If you allocate memory aggressively, the system may run out of memory.
- The system needs to be able to support different users, who will use the system in quite different ways.
- Users using the system need to perform a number of different tasks, and each task has different memory requirements.
- The system may have to run efficiently on hardware with greatly varying physical memory resources.
- The system's functionality is more important that its simplicity.

In many cases, especially in interactive systems, memory requirements cannot really be predicted in advance. For example, the memory requirements for the Strap-It-On PC's word-processing application Word-O-Matic will vary greatly, depending the features users choose to exercise — once user may want voice output, while another a large font for file editing.

The memory demands of interactive systems are unpredictable because they depend critically on what users choose to do with the system. If you try to produce a generic memory budget, you will over-allocate the memory requirements for some parts of the program, and consequently have to under-allocate memory for others.

For many interactive systems, providing the necessary functionality is more important than making the functionality easy to learn or to use. Being able to use a system to do a variety of jobs without running out of memory is sufficiently important that you can risk making other aspects of the interface design more complicated if it makes this possible. This is especially important because a system's users presumably know how they will use the system when they are actually using it, even through the system's designer may not no know this ahead of time.

Therefore: *Make the system's memory model explicit in its user interface, so that the user makes their own decisions about memory.*

Design a conceptual model of the way the system will use memory. Ideally, this model should be based on the conceptual model of the system and the domain model, but providing extra information about the system's memory use. This model should be expressed in terms of the objects users manipulate, and the operations they can perform on those objects, rather than the objects used directly in the implementation of the system.

Expose this memory model in your program's user interface, and let users manage memory allocation directly — either in the way that they create and store user interface level objects, or more coarsely, balancing memory use between their objects and the program's internal memory requirements.

For example, Word-O-Matic allows the user to choose how much memory should be allocated to store clip-art, and uses all the otherwise unallocated space to store the document. Word-O-Matic also displays the available memory to the user.

Consequences

The system can deliver more behaviour to the user than if it had to make pessimistic assumptions about its use of memory. The user can adjust their use of the system to make the most of the available memory, reducing the memory requirements for performing any particular task. Although the way user memory will be allocated at runtime is unpredictable, it is quarantined within the Memory Budget, so the memory use of the system as a whole is more predictable. Some user interfaces can even User Involvement about *memory fragmentation*.

However: Users now have to worry about memory whether they want to or not, so the system is less usable. Worrying about memory complicates the design of the system and its interface, making it more confusing to users, and distracting them from their primary task. Given a choice, users will choose systems where they do not have to worry about memory. You have to spend programmer effort designing the conceptual model, and making the memory model visible to the user.

Implementation

There are number of techniques which can expose a system's memory model to its users:

- Constantly display the amount of free memory in the system.
- Provide tools that allow users to query the contents of their memory, and the amount of memory remaining.
- Generate warning messages or dialogue boxes as the system runs out of memory, or as the user allocates lots of data.
- Make the user choose what data to overwrite or delete when they need more memory.
- Show the memory usage of different components in the system.
- Tell the user how their actions and choices affect the system's memory requirements.

Conceptual Models and Interface Design

A conceptual model is not the same as an interface design — rather, it is an abstraction of an interface design (see Constantine & Lockwood 1999, Software for Use). Where an interface design describes the way an interface looks and behaves in detail, a conceptual model describes the objects that should be present on a given part of an interface, the information those objects need to convey to users, and the operations users should be able to carry out upon those objects. To design an interface that presents memory constraints to the user, you should first determine what information about memory interface needs to present and how that information is related to the existing information managed by the user interface, and only then consider how to design the appearance and behaviour of the interface to present the information about memory use.

Granularity of Modelling

Conceptual models of memory use can be built with varying degrees of sophistication and differing amounts of detail. A very coarse model — perhaps modelling only users' and the system's memory consumption — will lead to a very simple interface that is easy to operate but may not provide enough information to make good decisions about memory use. A more fine grained model, perhaps associating memory use figures with every domain object in the interface, will give more control to users, but be more complex and more difficult to operate. Very detailed models that reify internal system components as concepts in the interface (so that the "font renderer" or "file cache" are themselves objects that users can manipulate) can provide more control, at a cost of further increase the complexity of the application.

Static and Dynamic Modelling

Conceptual models of memory use can be made statically or dynamically. Static models do not depends on details of a particular program run, so they can embody choices mad as the system was designed, or configuration parameters applied as the system begins running. In contrast, dynamic decisions are made as the program is running. A conceptual model of a system's memory use may describe static parameters, dynamic parameters, or a mixture of both. Generally static models makes it easier to give guarantees about a system's behaviour, because they cannot depend on the details of a particular run of a system. In contrast, precisely because they may depend on differences between runs, dynamic models can be more flexible, changing during the lifetime of the system to suit the way it is used, but also run the risk of running out of memory.

General Purpose Systems

The more general a system's purpose, the more difficult memory allocation becomes. A system may have to support several radically different types of users – say from novices to experts, or from those working on small jobs to those working on big jobs. Even the work of a single user can have different memory requirements depending upon the details of the task performed: formatting and rasterising text for laser printing may have completely different memory requirements to entering the text in the first place. Also, systems may need to run on hardware with varying memory requirements. Often the memory supplied between different models or configurations of the same hardware can vary by a several orders of magnitude and the same program may need to run on systems with 128Kb of memory to systems with 128M or more.

Supporting Different User Roles

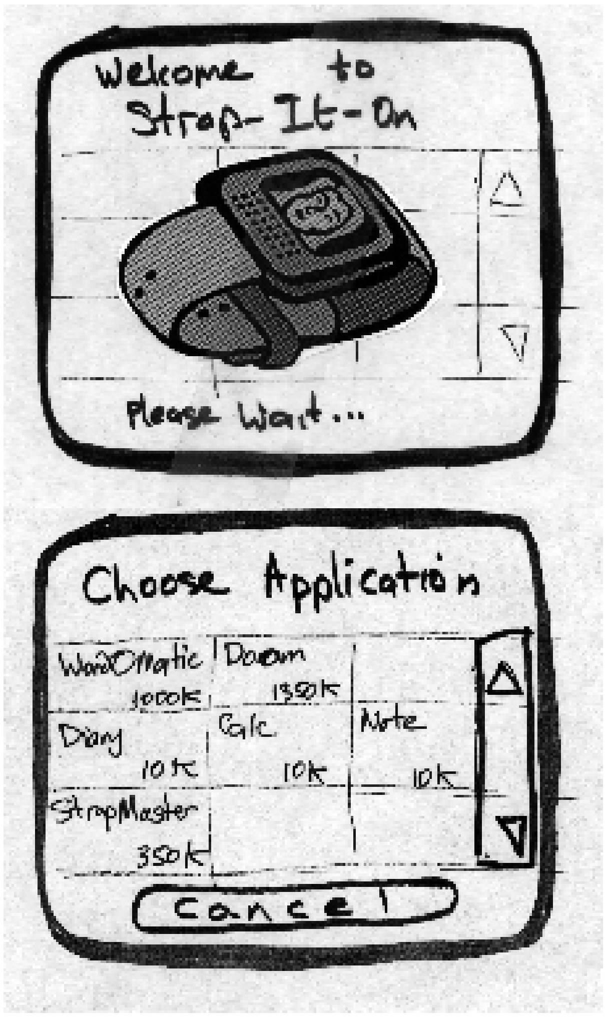
Different users can differ widely in the roles they play with respect to a given system, and often their memory use (and interest in or capability to manage the system's memory use) depends upon the role they play. For example, the users of a web-based information kiosk system would play two main roles with respect to the system — a casual inquirer trying to obtain information from the kiosk, and the kiosk administrator configuring the kiosk, choosing networking protocol addresses, font sizes, image resolutions and so on. The casual inquirer would have no interest in the system's model of memory use, and no background or training to understand or manipulate it, while the kiosk administrator could be vitally concerned with memory issues.

The techniques and processes of User Role Modelling from Usage-Centered Design (Constantine & Lockwood, 1999) can be used to identify the different kinds of users a system needs to support, and to characterise the support a system needs to provide to each kind of user.

Examples

For example, after it has displayed its startup screen, the Strap-It-On wrist mounted PC asks its user to select an application to run. The select an application screen also displays the amount of memory each application will need if it is chosen.

[note: all scanned pictures to be redrawn].



Specialised Patterns

The rest of this chapter contains five patterns that present a range of techniques for making the user worry about the systems memory use. It describes ways that a user interface can be structured, how users can be placed directly in control of a system's memory allocation, anddescribes how the quality of a user interface can be traded off against its memory use.

FIXED SIZED USER MEMORY describes how user interfaces can be designed with a small number of user memories. Controls to access these memories can be designed directly into the interface of the program, making them quick and easy to access. Fixed sized user memories have the disadvantages that they do not deal well with user data objects of varying sizes, or more than about twenty memory locations.

VARIABLE SIZED USER MEMORY allows the user to store variable numbers of varying sized data objects, overcoming the major disadvantages of designs based on fixed sized user memory. The resulting interface designs are more complex than those based on fixed sized memory spaces, because users need ways of navigating through the contents of the memories and must be careful not to exhaust the capacity.

MEMORY FEEDBACK, in turn, addresses some of the problems of variable sized user memory: by presenting users with feedback describing the state of a system's memory use, they can make better use of the available memory. Providing memory feedback has a wider applicability than just managing user memory, as the feedback can also describe the system's use of memory — the amount of memory occupied by application software and system services.

USER MEMORY CONFIGURATION extends Memory Feedback by allowing users to configure the way systems use memory. Often, information or advice about how a system will be used, or what aspects of a system's performance are most important to its users, can help a system make the best use of the available memory.

Finally, **LOW QUALITY MULTIMEDIA** describes how multimedia resources — a particularly memory-hungry component of many systems — can be reduced in quality or even eliminated altogether, thus releasing the memory they would otherwise have occupied for more important uses in the system.

See Also

The memory model exposed to the user may be implemented by **FIXED ALLOCATION or VARIABLE ALLOCATION — FIXED SIZE USER MEMORY** is usually implemented by **FIXED ALLOCATION** and **VARIABLE SIZE USER MEMORY** is usually implemented by **VARIABLE ALLOCATION.**

FUNCTIONALITY A LA CARTE [Adams 95] can present the costs and benefits of memory allocations to the user.

A static **MEMORY BUDGET** can provide an alternative to **USER MEMORY CONFIGURATION** that does not require users to manage memory explicitly, but that will have higher memory requirements to provide a given amount of functionality.

The patterns in this chapter describe techniques for designing user interfaces for systems that have limited memory capacity. We have not attempted to address the must wider question of user interface design generally — as this is a topic which deserves a book of its own. Schneiderman's *User Interface Design* is a general introduction to the field of interface design, and Constantine and Lockwoods' *Software for Use* presents a comprehensive methodology for incorporating user interface design into development processes. Both these texts discuss interface design for embedded systems and small portable devices as well as for desktop applications.

Fixed Size User Memory

Also known As: Fixed Number of User Memories

How can you present a small amount of memory to the user?

- You have a small amount of user-visible memory.
- Users need to store a small number of discrete items in the memory
- Every item users need to store is roughly the same size
- Users need to be able to retrieve data from the memory particularly easily.
- Users cannot tolerate much extra complexity in the interface.

Some systems have only a small amount of memory available for storing the users' data (and presumably only a small amount of data that users can store). This user data is often a series of discrete items — such as telephone numbers or configuration settings where each item is the same size. For example, the Strap-It-On needs to definitions for its voice input feature. Each macro requires enough memory to recognise a three second spoken phrase of the user's choice, and the commands that are to be executed when the voice macro facility recognises that phrase.

Users need to be able to retrieve data from memory quickly and easily — after all, that's why the system is going to the trouble to store such a small amount of data. For example, the point of the Strap-it-On's voice input macros are to make data entry more efficient, streamlined, and "fun to do all day" (to quote the marketing brochure). Similarly music synthesisers store multiple 'patches' so that they can be quickly recalled during a performance, and phones store numbers because people want to dial them quickly.

One approach to this problem is to let the user choose things to store, until the device is out of memory, when it stops accepting things. This is quite easy to implement but gets pretty unsatisfactory when there's only a small amount of memory. Users will tend to get the idea that the device has infinitude of memory, and consequently will be surprised when the system refuses their requests. Also, you'll need some kind of interface to retrieve things from the memory, to delete things that have already been stored, and so on — all of which will just take up more precious memory space.

Therefore: *Provide a small, fixed number of memory spaces, and let the user manage them individually.*

Design a fixed number of "user memories" as explicit parts of the user interface conceptual model. Each user memory should be represented visually as part of the interface, and the design should make clear that there are only a fixed number of user memory spaces — typically by allocating a single interface element (often a physical or virtual button) for each memory. Ideally each memory space should be accessed directly via its button (or via a sequence number) to reinforce the idea that there are only a fixed number of user memories.

The user can store and retrieve items from a memory space by interacting with the interface element(s) that represent that user memory. Ideally the simplest interaction, such as pressing the button that represents a user memory, retrieves the contents of the memory — restoring the device to the configuration stored in the memory, running the macro, or dialling the number held in that memory. Storing into a user memory can be a more complex interaction, because storing is performed much less frequently than retrieval. For example, pressing a "store" button and then pressing the memory button might store the current configuration into that memory. Any other action that uses the memory should access it in the same way.

Finally, an interface with a fixed number of user memories does not need to support an explicit delete action from the memories: the user simply chooses which memory to overwrite.

For example, the Strap-It-On allocates enough storage for nine voice macros. This storage is always available (it is allocated permanently in the memory budget; the setup screen is quickly accessible via the Strap-It-On's operating system, and is designed to show only the nine memory spaces available.

Consequences

The idea that the system has a number of memory spaces into which users can store data is obvious from the its design, making the design *easy to learn*. The fixed number of user memory spaces becomes part of users' conceptual model of the system, and the amount of memory available is always clear to users. Because the number of memories is fixed, the interface can be designed so that users *can easily and quickly* choose which memory to retrieve. The *graphical layout* of the interface is made easier, because there are always the same number of memories to display.

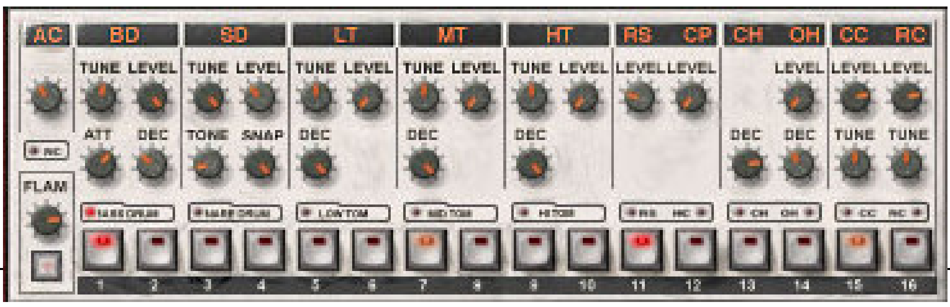
However: The user interface architecture is strongly governed by the memory architecture. This technique works well for a small number of memory spaces but *does not scale* well to allocating more than twenty or thirty memory spaces, or storing objects of more than two or three different sizes. User interfaces based on a fixed number of user memories are generally less *scalable* than interfaces based on some kind of variable allocation. Increasing the size of a variable memory may simply require increasing the capacity of a browser or list view, but increasing the number of fixed user memories can require a redesign of the interface, especially if memories are accessed directly.

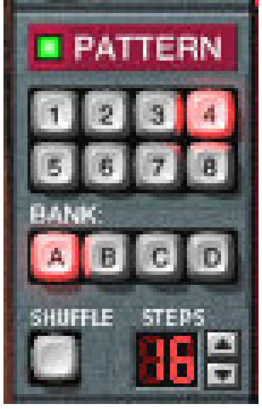
Implementation

There are three main interface design techniques that can be used to access fixed size user memories — *direct access, banked access, and sequential access*. This illustrates the difference between a conceptual model and in interface design — the same conceptual model for fixed sized user memories can be realised in several different ways in an actual user interface design.

Direct Access. For a small number of user memories, allocate a single interface element to each memory. With a single button for each memory, pressing the button can recall the memory directly — a fast and easy operation. Unfortunately, this technique is limited to sixteen or so memories because few interfaces can afford to dedicate too many elements to memory access.

For example, the drum machines in the ReBirth-338 software synthesiser provide a fixed size user memory to store drum patterns. The sixteen buttons across the bottom of the picture below correspond to sixteen memory locations storing sixteen drum beats making up a single





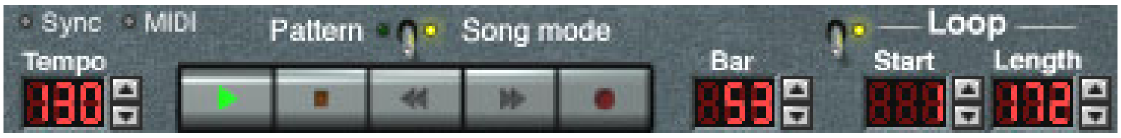
pattern — we can see that the bass drum will play on the first, seventh, eleventh and fifteenth beat of the pattern.

Banked Access. For between ten and one hundred elements, you can use a two dimensional scheme. Two sets of buttons are used to access each memory location — the first set to select a memory bank, and the second set to select an individual memory within the selected bank. Banked access requires many fewer interface elements than direct access given the same number of memory spaces, but is still quite quick to operate.

Again following hardware design, the ReBirth synthesiser can store thirty-two patterns for each instrument in emulates. The patterns are divided up into four banks (A, B, C, D) of eight patterns each, and these patterns are selected using banked access.

Sequential Access. For even less hardware cost (or screen real estate) you can get by with just a couple of buttons to scroll sequentially through all the available memory spaces. This approach is quite common on cheap or small devices because it has a very low hardware cost and can provide access to an unlimited number of memories. However, sequential scrolling is more difficult to use than direct access, because more user gestures are required to access a given memory location, and because the memory model is not as explicit in the interface.

A ReBirth song is made up of a list of patterns. The control below plays through a stored song. To store a pattern into a song, you select the position in the song using the small arrow



controls to the left of the "Bar" window, and then choose a pattern using the pattern selector shown above.

Single Memories

Special cases of fixed sized user memory are systems that provide just one memory space. For example, many telephones have a "last number redial" feature — effectively a single user memory set after every number is dialled. Similarly, many laptop computers have a single memory space for storing backup software configuration parameters, so that the machine can be rebooted if it is misconfigured. A single memory space is usually quick and easy to access, but obviously cannot store much information.

Memory Exhaustion

One advantage of fixed size user memory designs is that users should never experience running out of memory — rather, they will just have to decide which user memory to overwrite. Certainly, if all memories (either full or empty) are accessed in the same way, the system never needs to produce any error messages explaining that the system has run out of memory.

Storing Variable Sized Objects

A fixed number of fixed sized memory locations does not cope well when storing objects of varying sizes. If an item to be stored is too small for the memory space, the extra space is wasted. If an item is too large, either it must be truncated, or it must be stored in two or more spaces (presumably wasting memory in the last overflow space), or the user must be prevented from creating such an item in the first place.

Alternatively, if an interface needs to store just two or three different kinds of user data objects (where each kind of object has a different size) the interface design can have a separate set of user memories for each kind of object that needs to be stored. This doesn't avoid the problem completely, since the size and number of the memory spaces must be determined in advance, and it is unlikely that it will match the number of objects of the appropriate kind that each user wishes to store.

Initialising Memories

An important distinction in the design of conceptual models for fixed size user interfaces is whether the system supports a fixed sized number of *memory spaces* or a fixed sized number of *objects*. The differences is that if the system as a number of memory spaces, some of the spaces can be empty, but it doesn't make sense to have a empty object stored in a memory space. In general, designing in terms of objects is preferable to designing in terms of memory spaces. For example, there is no need to support retrieve operations on empty spaces if there can be no empty spaces, For this to work, you need to find good initial contents for the objects to be stored. The memory spaces of synthesisers and drum machines, for example, are typically initialised with useful sounds or drum patterns than can be used immediately, and later overwritten.

One compensating advantage of having the idea of empty memories in a conceptual model is that you can support an *implicit store* operation that stores an object into some empty memory space, without the user having to chose the space explicitly. This certainly makes the store operation easier, but (unlike a store operation that explicitly selects a memory space to overwrite), and implicit store operation can fail due to lack of memory — effectively treating the fixed size memories as if they were variable sized. The Nokia 2210e mobile phone supports implicit stores into its internal phone book, but, if there are no empty memories, users can choose which memory to overwrite.

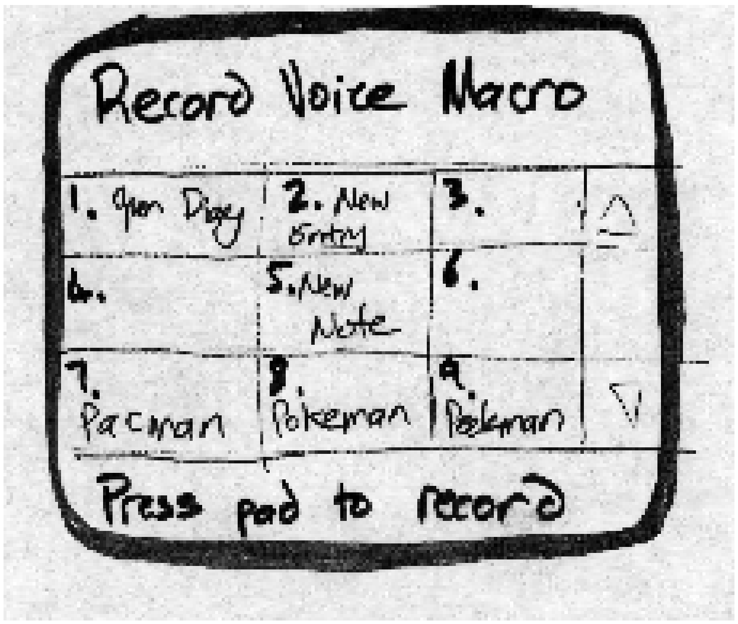
Naming Memories

Where there are more than four or five memories — and certainly where there are more than sixteen or so — you can consider allowing the user to name each memory to make it easier for users to remember what is store in each memory space. A memory name can be quite short — say eight uppercase characters — and so can be stored in a small amount of memory using simple **STRING COMPRESSION** techniques.

Of course, there is still a trade-off between the *memory* requirements for storing the name and the *usability* of a larger number memories, but there is no user providing a system with large numbers of memories if users can't actually find the things they have store in them. If the system includes a larger amount of preset data in stored in **READ-ONLY STORAGE** you can supply names for these memories, while avoiding naming user memories stored in scarce RAM or writeable persistent storage.

Examples

The StrapItOn PC has nine memories that can be used to store voice input macros. Each memory is represented by one of nine large touch-sensitive areas on the StrapItOn's screen. To record a macro, the user touches the are representing the memory where they want to store the macro — if this memory is already in use, the existing macro is overwritten.



The Korg WaveStation synthesiser contains several examples of fixed sized user memories. The most obviously is that its user interface includes five soft keys that can be programmed by the user to move to a particular page in the WaveStation's menu hierarchy. The soft keys are accessed by pressing a dedicated "jump" key on the front panel, and then one of the five physical keys under the display.

The main memory architecture of the WaveStation is also based firmly on fixed sized user memories. Each WaveStation can store thirty-two 'performances', that can refer to up to four 'patches', each of which can play one of thirty-one 'wave sequences'. Patches and performances have different sizes, and each are stored in their own fixed-sized user memories, so if you run out of patch storage, you cannot utilise empty performance memories. Patch and performance memories are addressed explicitly using memory location numbers entered by either cursor keys, turning a data-entry knob, or typing the number directly using a numeric keypad. Patches and performances can also be named --- performances with up to sixteen characters, patches with only ten.

Each wave sequence may refer to up to two hundred waves, however the total number of waves referred to by all wave sequences cannot exceed five hundred. (Approximately five hundred waves are stored in ROM, and a wave sequence describes an order in which the ROM waves should be played). Waves are added implicitly to wave sequences using the "enter" key --- if either of the limits on individual or total wave sequence lengths is exceeded, the WaveStation displays an :"Out of Memory" error message (see the **MEMORY NOTIFIER** pattern.)

[Picture to be drawn from manual -- currently en route from Sydney to Wellington. Numbers to be checked against the manual!]

Known Uses

Music others synthesisers often provide a small fixed number of memory locations for programs, and users think of these systems as having just that number of memories. For example, the Yamaha TX-81Z synthesiser provided 128 preset patches (synthesiser sound programs) organised as four banks of 16 patches each, plus one bank of 16 user-programmable patches. The TX-81Z also included 16 memories for “performances” including references to patches but also information about the global state of the synthesiser (such as MIDI channel and global controller assignments) — performances were user data objects of different sizes to the patches. The TX-81Z also included some other user memories for other things, such as microtonal scales and delay effects.

GSM mobile phone SIM cards are smart cards that store a fixed number of phone memories containing name and number (the number of memories depends on the SIM variant). Users access the memories by number. Many telephones have something similar, though simpler systems don't store names. The same SIM cards can also store a fixed number of received SMS (short message service) text messages — the user is told if a message could not be stored because the store overflowed. SIM cards can also store a fixed number of already read messages in a numbered store. This store is visible to the user and accessed by message number.

The FORTH programming environment stored source code in 1024 character blocks. This allowed the system to allocated a fixed sized buffer for the text editor, made screen layout simple (16 lines of 64 characters that could be displayed on a domestic TV screen) and to store each block in a single 1024 byte sector on a floppy disc. Each block on disc was directly referenced by its sequence number.

An early Australian laser printer required the user to uncompress typefaces into one of a fixed number of memory locations. For example, making Times Roman, Times Roman Italic, and Times Roman Bold typefaces available for printing would require three memory locations into which ROM Times Roman bitmaps could be uncompressed, with some bitmap manipulations to get italic and bold effects. Documents selected typefaces using escapes codes referring to memory locations. Larger fonts had to be stored into two or more contiguous locations, making the user worry about memory fragmentation as well as memory requirements, and giving very interesting results if an escape code tried to print from the second half of a font stored in two locations.

Many video games store just the top 10 scores (and the three initials of the players who scored them). This has a number of advantages: it requires very little memory, allows a constant graphical layout for the high-score screen, and adds automatically overwrites the 11th best score when they have been beaten, increasing player's motivation.

The Ensoniq Mirage sound sampler was the ultimate example of User Involvement. The poor user — presumably a musician with little computer experience — must allocate memory for sound sample storage by entering two digit hexadecimal numbers using only increment and decrement buttons. Each 64K memory bank could hold up to eight samples, provided each sample was stored in a single contiguous memory block. In spite of the arcane and frustrating user interface (or perhaps because of the high functionality the interface supported with limited and cheap hardware) the Mirage was used very widely in the mid-1980s popular music, and maintains a loyal if eccentric following ten years later.

See also

VARIABLE-SIZED USER MEMORY offers an alternative to this pattern that explicitly models a reservoir of memory in the system, and allows users to store varying numbers of variable sized objects.

MEMORY FEEDBACK can be used to show users which memories are empty and which are full, or provide statistics on the overall use of memory (such as the percentage of memories used or free).

Although systems' requirements do not generally specify **FIXED SIZED USER MEMORIES**, by negotiating with your clients you may be able to arrange a **FEATURECTOMY**.

A **MEMORY BUDGET** can help you to design the number and sizes of **FIXED SIZED USER MEMORIES** **your system will support.**

You can use **FIXED ALLOCATION** to implement fixed sized user memories.

Variable-Sized User Memory

How can you present a medium amount of memory to the user?

- You have a medium to large amount of user-visible memory
- Users need to store a varying number of items in the memory
- The items users can store vary in size
- The memory requirements for what the user will need to store are unpredictable.

Some programs have medium or large amounts of memory available for storing user data. For example, the Strap-It-On wrist-portable PC provides a file system to allow users to store application data. Files can vary in size from a few words to several pages, and within the bounds of the systems memory, some users store a few large files while other users store many small files. The behaviour of some users changes over time — one week storing many small files, the next one large file, the next a mixture.

One approach to organising the memory would be to provide **FIXED SIZED USER MEMORY**. The system could allocate a fixed number of fixed-sized spaces to hold memos the user wishes to store. Of course, this suffers from all the problems of fixed allocation: memory spaces holding small memos will waste the rest of the space, and long memos must somehow be split over a number of different spaces. Another alternative would be to require users to pre-allocate space to store memos, but this requires users to be able to accurately estimate the size of a new memo before it is created, and the pre-allocation step will greatly complicate the user interface.

Therefore: *Randomly allocate users' objects from a reservoir of free memory.*

Allow the user to store and retrieve items flexibly from the systems' memory reservoir. The reservoir does not have to be made explicit in the interface design — although it may be. Each item stored in the memory should be treated as an individual object in the user interface, so that users can manipulate it directly. You also need to provide an interface to allow the user to find particular items they want to use, and to explicitly delete objects from the reservoir making the memory space available for the storage of new objects.

For example, the Strap-It-On uses **VARIABLE SIZED USER MEMORY** for its file system. A reservoir large enough to hold ten thousand words (about a hundred thousand characters of storage) is allocated to hold all the users' files. When users create new files they are stored within the reservoir until they are explicitly deleted. The file tool displays the memory used by every file in a browser view, the percentage of free memory left in the reservoir pool in its status line, and also uses error messages to warn the user when memory use exceeds certain thresholds (90%, 95% 99%).

Consequences

Users can store new objects in memory quite *easily,* provided there is enough space for them. Users can make *flexible* use of the main memory space, storing varying numbers and sizes of items to make the most of the systems capacity — effectively reducing the system's *memory requirements*. Users don't need to choose particular locations in which to store items or to worry about accidentally overwriting items or to do pre-allocation, and this increases the system's *usability*.

Variable sized memory allocation is generally quite *scalable*, as it is easier to increase the size of a reservoir (or add multiple separate reservoirs) than to increase the number of fixed size memories.

However: The program's *memory model is exposed*. The user needs to be aware of the reservoir, even though the reservoir may not be explicitly presented in the interface. The user interface will be *more complex* as a result, and the graphical design will be more difficult. Users will not always be aware of how much memory is left in the system, so they are more likely to *run out of memory*.

Any kind of variable allocation decreases the *predictability* of the program's memory use, and increases the possibility of *memory fragmentation*. Variable sized allocation also has a higher *testing cost* than fixed sized allocation.

Implementation

Although a variable sized user memory can give users an illusion of infinite memory, memory management issues must still lurk under this façade: somewhere the system needs to record that the objects are all occupying space from the same memory reservoir, and that the reservoir is finite. Even if the reservoir is not explicit in the interface design, try to produce a conceptual model for the use of memory which is integrated with the rest of the system and the domain model, so that the user can understand how the memory management works. Consider using **MEMORY FEEDBACK** to keep the user informed about the amount of memory used (and more importantly, available) in the reservoir — typically by listing the amount of memory occupied by objects when displaying the objects themselves.

A conceptual model based on variable sized user memory is more sophisticated than a similar model built on fixed sized user memory. A model of variable sized user memory must include not only empty or full memory locations, but also a more abstract concept of "memory space" that can be allocated between newly created objects and existing objects if their size increases. The objects that can be stored in user memory can also be more sophisticated, with varying sizes and types.

Multiple Reservoirs

You can implement multiple reservoirs to model multiple hardware resources, such as disks, flash ram cards, and so on. Each separate physical store should be treated as an individual reservoir. You need to present information about each reservoir individually, as the amount and percentages of free and used memory. You can also provide operations that work on whole reservoirs, such as backing up all the objects in once reservoir into another or deleting all the objects stored in a reservoir.

You will also need to ensure that the user interface associates each object with the reservoir where it is stored — typically by using reservoirs to structure the way information about the objects is presented, by grouping all the objects in a reservoir together. For example, most desktop GUI filing systems show all the files in a single disk or directory together in one window, so that the association between objects and the physical device on which they are stored is always clear.

Fragmentation

As with any kind of **VARIABLE SIZED DATA STRUCTURE**, the reservoir of user object memory may be subject to fragmentation. This can cause problems as the amount of memory that is reported as being available may be less than the amount of memory that can be used in practice. So for example, while the Strap-It-On's file memory may have 50K free characters, the largest single block might be only 10K – not enough to create a 15K email message.

One way to avoid this problem is to show users information about the largest free block of memory in each reservoir, rather than simply the amount of free memory. Another approach is to implement **MEMORY COMPACTION** — say with a user initiated compaction operation, in the same way that PC operating systems include explicit defragmentation operations.

Unfortunately, both these approaches complicate the users' conceptual model of the interface to include fragmentation. An alternative approach is to choose a data structure that does not require explicit compaction, either by compacting the structure automatically on every operation, or using a linked structure that splits objects across separately allocated memory blocks and so does not need compaction to use all the available space.

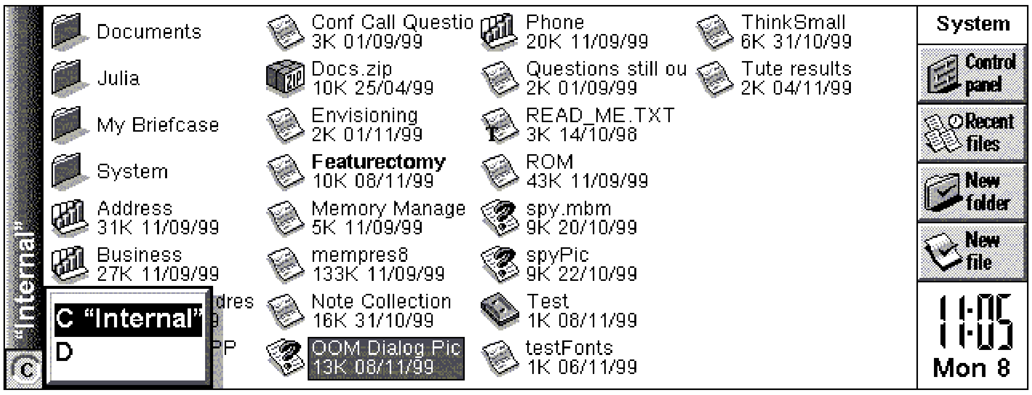
Caching and Temporary Storage

Caching can play havoc with the user's model of the memory space if applications trespass on that memory for caches or temporary storage. For example, many web browsers (including the Strap-It-On's Weblivion) cache pages in user storage, and browsers on palmtop and desktop machines similarly maintain temporary caches in user file storage. The minor problem here is that naïve measurements of the amount of free space will be too low, as some of the space is allocated to caches; the major problems is that unless the memory is somehow released from the caches it cannot be used for application data storage.

The **CAPTAIN OATES** pattern describes how you can design a system so that applications release their temporary storage when it is required for more important or permanent uses. The key to this pattern is that when memory is low, the system should signal applications that may be holding cached data. In response to receiving the signal, any applications holding cached data should release the caches.

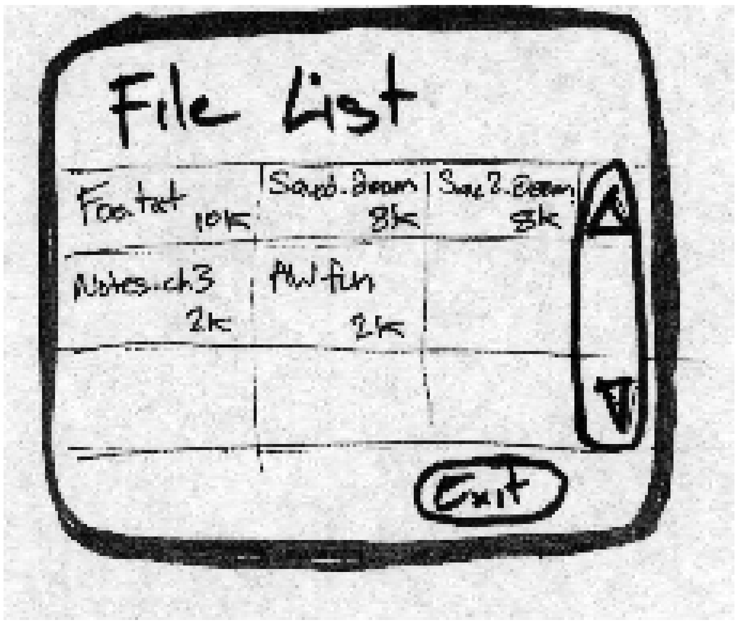
Examples

A Psion Series 5 allows users to store text files and spreadsheet files in persistent storage reservoir (call a "drive" but implemented by battery backed up RAM). The browser interface illustrated in the figure below shows the files stored in a given drive, and the size of each file. Clicking on the "drive letter" in the bottom left hand corner of the screen produces a menu allowing users to view a different reservoir.



```
Documents            Conf Call Questio    Phone                ThinkSmall
                     3K 01/09/99          20K 11/09/99         6K 31/10/99
Julia                Docs.zip             Questions still ou   Tute results
                     10K 25/04/99         2K 01/09/99          2K 04/11/99
My Briefcase         Envisioning          READ_ME.TXT
                     2K 01/11/99          3K 14/10/98
System               Featurectomy         ROM
                     10K 08/11/99         43K 11/09/99
Address              Memory Manage        spy.mbm
31K 11/09/99         5K 11/09/99          9K 20/10/99
Business             mempres8             spyPic
27K 11/09/99         133K 11/09/99        9K 22/10/99
C "Internal"    dres Note Collection      Test
                9    16K 31/10/99         1K 08/11/99
D               P    OOM Dialog Pic       testFonts
                     13K 08/11/99         1K 06/11/99
```

The StrapItOn PC can also list all the files in its internal memory, and also always lists their sizes.



[KJX hotmail account smallersoftware@hotmail.com]

Known Uses

Most general purpose computers provide some kind of variable sized user memory for storing users' data — either in a special region of (persistent) primary storage, such as the PalmPilot, Newton, and Psion Series 5, or on secondary storage, if the form factor permits. Similarly, multitasking computers effectively use variable sized user memory — users can run applications of varying sizes until they run out of memory.

[More examples to do: Akai series of audio samplers; sequences; MP3 players?]

See also

You probably need to use **VARIABLE ALLOCATION** to implement variable sized user memory. **MEMORY FEEDBACK** can help the use avoid running out of memory. The size of the reservoir may be able to be set by **USER MEMORY CONFIGURATION.** **FIXED SIZED USER MEMORY** can be an alternative to this pattern if only a few, fixed sized objects need to be stored.

Memory feedback

How can users make good decisions about the use of memory?

- You have a **VARIABLE SIZED USER MEMORY**
- There is a medium to large amount of memory available
- Memory is proving a constraint on program functionality
- Users are managing some form or memory allocation
- Users memory allocation choices affect the performance or functionality of the system.

Some systems have reasonably large amounts of memory available, but memory allocation is difficult — typically because the memory allocation needs to match users' priorities or tasks and these are unpredictable, changing over time and between different users. For example, the StrapItOn has a fair amount of main memory, but this is quickly used up if users open too many applications simultaneously.

One way to deal with these problems is for users to accept some of the burden of managing the systems memory, perhaps by using **VARIABLE SIZED USER MEMORIES** — indeed, this is the solution adopted by the Strap-It-On's designers. Unfortunately, this solution raises another problem: how can the users get enough information to manage the memory effectively? Presumably memory allocation matters, and is too difficult to leave to the system — this is why users have been given the responsibility. But users need to make good memory allocation choices, or else the system won't work.

Other solutions are to provide information in printed documentation or in the help system about how the users' choices affect the memory of the system. But, in an interactive system where memory use changes dynamically at every run, this isn't really enough, because theoretical knowledge doesn't help work out what is wrong with the system's memory allocation right now, or how any changes to the allocation will affect the functioning of the system[1].

Therefore*: Provide feedback to users about the current memory use, and the consequences of their actions.*

As part of your interface design, you should have produced a conceptual model of the way the system uses memory (see **USER INVOLVEMENT**). Design ways to present the information in this model to users, as and when they need it. Include widgets in the display that tell users how much memory they have left, and lets them know the consequences of their decisions on the use of memory. These controls should be able to be accessed at any time, and integrated with the rest of the interface.

For example, the Strap-It-On user interface includes a dialog showing the user all the applications that are running, and how much memory is allocated to each application. This dialog is easy to reach from any application running on the StrapItOn. Furthermore, when the system runs low on memory, a variant of this dialog is displayed that not only lists all the applications and their memory usage, but also invites the user to terminate one of them to free up some memory space.

[1] Unless you are the kind of person who can intuit memory problems from the feel of the systems command line. There are people like this.

Consequences

Users are in better contact with the memory use of their systems. Their mental models about the way the system uses memory may be more likely to reflect the way the system uses memory. The *usability* of the system as a whole is increased (although still less than a system where users don't have to worry about memory).

Because users know how much memory is available, they are less likely to *exhaust* the system's memory.

However: The user needs a more sophisticated mental model of the system, and its implementation, that is, the way the system uses memory. This isn't really anything to do with the user's work, as it's an artefact of the implementation. The information about memory can confuse or distract the user from their primary task. Programmer *effort* will be required to implement the actual feedback interface elements, and programmer *discipline* may be required so that other parts of the system provide the necessary information to the feedback interface.

Memory feedback has to be tested independently, increasing the *testing cost* of the program.

Implementation

There are four main ways to provide feedback

- Memory report — an explicit report on the systems memory utilisation
- Memory meter— a continuous (unintrusive) meter of the systems memory use
- Memory notifier — a proactive warning when memory exhaustion is near
- Massive feedback — the system partially fails under memory exhaustion

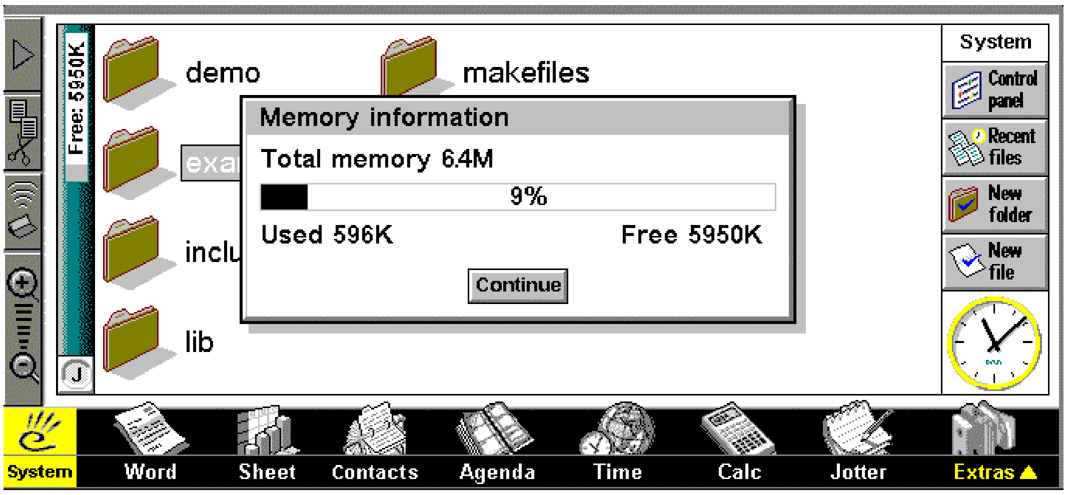
Memory Report

A memory report simply reports the memory use of the system, describing the various components consuming memory, and the amount of memory consumed by each component. Of course, the information presented in the report should be in terms of the system's conceptual model for memory use. Similarly, the units used to describe the memory should be comprehensible by most users, such as percentages or some other measure with typical values in double figures. A memory report can often be combined into the interface used to allocate, delete and browse **FIXED SIZE USER MEMORY OR VARIABLE SIZED USER MEMORY** because the information needed for all these tasks are so similar.

Because there are typically quite a few components in the system consuming memory, a memory report will need to provide a fair amount of detail to the user and take up quite a bit of screen space. So memory reports usually have to be designed as a part of the interface for users to request explicitly. One of the main purposes for the memory report is for the user to find out where the memory is being used in the system, so the report should be sorted in order of the memory consumption of each component, with the largest component first.

In an interactive system is it better if the display can be updated to reflect the instantaneous state of the memory in the system, although this will take code and effort. Alternatively, a memory report may be a passive display that is calculated each time it is requested.

For example, users of the Psion Series 5 can request a dialog box that provides a memory report, giving the total amount of user memory available, the amounts used and free, and the percentage of memory used.



Memory information

Total memory 6.4M

9%

Used 596K Free 5950K

Continue

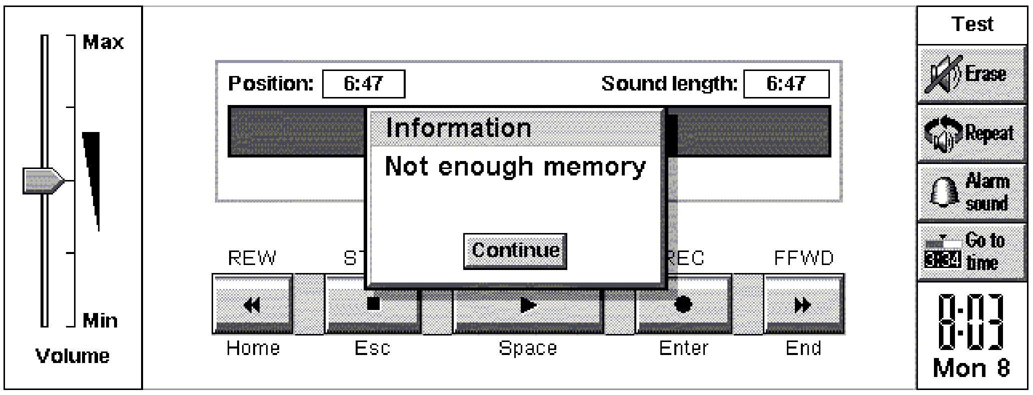
Memory Meter

A memory meter is a continuous display of the amount of free memory in the system. Because a memory meter is displayed continuously, it needs to be small an unintrusive part of the interface, so it cannot present the same amount of information as a memory report. To provide more information, make a memory report easily accessible via the memory meter.

The Psion Series 5 can also constantly display a memory meter listing the amount of free user memory (shown towards the left side of the screen in the above illustration).

Memory Notifier

A memory notifier is a proactive interface element (such as a popup warning window or audio signal) which is used to warn the user that the system's memory use is reaching a critical stage — such as no memory left to allocate! More usefully, memory notifiers can be generated as the systems memory gets low, but before it is totally exhausted, to give the user time to recover, or even as a warning in advance of any operation that is likely to allocate a large amount of memory. Of course, since notifier messages are modal, they will interrupt the work users are trying to do with the system and have to be explicitly dismissed. Because of this, they should be saved for situations where user intervention of some kind really is required, and memory meters used to give warnings in less critical situations. For example the following Figure is displayed by the Psion Series 5 voice recorded application, to indicate that it has run out of memory and consequently has had to stop recording.



Information

Not enough memory

Continue

As with all notifier messages and dialog boxes, a memory notifier should provide information about thereason for the notification and the actions the user can take to resolve the problem — either through an informative message in the notifier, or through a help message associated closely with it. Typically, a memory notifier should suggest something users can do to reduce the system's demands for memory.

The major technical problem with implementing notifier messages (and then contingent help) is that memory notifiers by definition appear when the system is low on memory, so there is often little memory available that can be used to display the notifier. Windows CE, for example, provides a special system call to display an out of memory dialog box in the system shell. The shell preallocates the resources required by this dialog box so that it can always be displayed.

```
if (!(addr = (VirtualAlloc(stuff))) {
   SHShowOutOfMemory(hwndowner, 0);
   return E_OUT_OF_MEMORY;
}
```

Passive Feedback

If temporary memory exhaustion causes computations to suffer Partial Failure or multimedia quality to drop, the results of the computation or multimedia can simply be omitted — output windows remain blank, the frame rate or resolution of generated images is reduced, and so on. Users notice this drop in quality, and free memory by shutting down other tasks so that the more important tasks can proceed.

Passive notification makes memory exhaustion produce the same symptoms as many other forms of temporary resource exhaustion (such as CPU overload, network congestion, or thrashing). This has the advantage that the same remedy (reducing the load on the system) can relieve all of these problems, but the disadvantage that the precise cause of the overload is not immediately clear to users.

Passive Feedback should never be used to report exhaustion of long-term memory — use a memory notifier for this case. Precisely because passive feedback is passive, it may not be noticed by users, which, in the case of long term memory, comes to silently throwing away users' data.

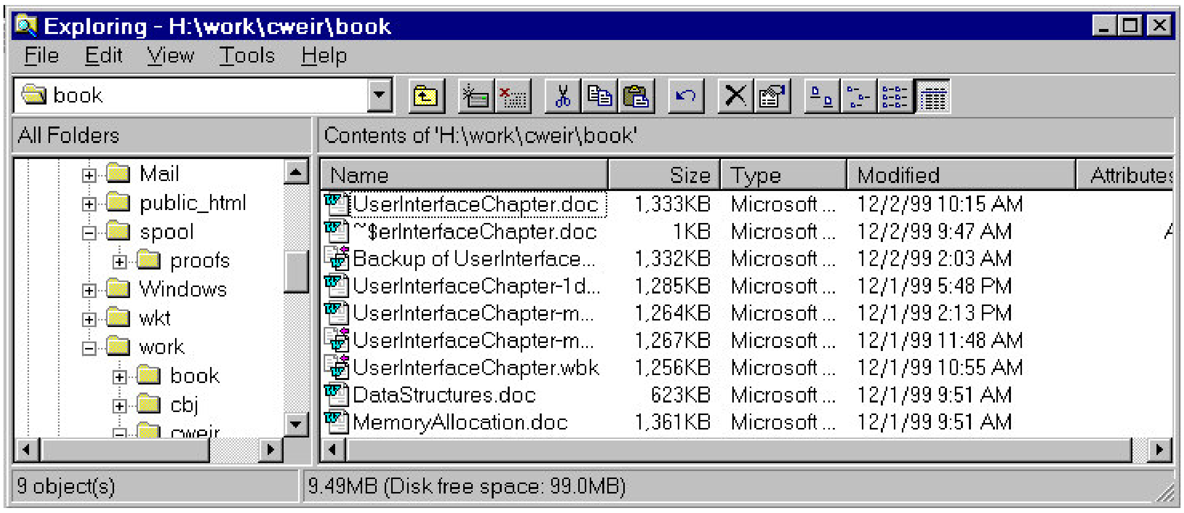
Ward Cunningham's Checks pattern language for information integrity describes a general technique for implementing Passive Feedback. [Cunningham PLOPD2]

Standard interfaces or Special-purpose interfaces

There are generally two alternatives for incorporating memory information into a user interface — either the application's general purpose displays can be extended to include memory information, or special purpose displays can be designed that focus on memory information.

Extending an application's standard, general purpose displays has several advantages: the memory feedback can be tightly integrated with the domain information present by the application; because of this integration, the feedback can be easily incorporated into each user's model of the system; and users don't need to learn a new part of the interface to manage memory.

A good example of this kind of integration can be found in many file system browers (including those of Windows, the Macintosh, and the PalmPilot), that seamless include file size information along with other information about the files. Indeed, size information has listed along with file names for so long that most habitual users of computer systems expect it to be listed, and do not realise that the size of each file is really an implementation concern to help them manage the system's memory. The illustration shows how Windows Explorer lists file sizes as the second most important piece of information about a file, other than the file name. On the other hand, including memory feedback in an integrated view does mean users are present with it whether or not they really need to know about it, and this also takes up screen real estate that could be used for to display more useful information, such as files' full names and types.


Exploring - H:\work\cweir\book

File Edit View Tools Help

book

All Folders

Mail
public_html
spool
proofs
Windows
wkt
work
book
cbj
cweir

Contents of 'H:\work\cweir\book'

Name	Size	Type	Modified	Attributes
UserInterfaceChapter.doc	1,333KB	Microsoft ...	12/2/99 10:15 AM	
~$erInterfaceChapter.doc	1KB	Microsoft ...	12/2/99 9:47 AM	A
Backup of UserInterface...	1,332KB	Microsoft ...	12/2/99 2:03 AM	
UserInterfaceChapter-1d...	1,285KB	Microsoft ...	12/1/99 5:48 PM	
UserInterfaceChapter-m...	1,264KB	Microsoft ...	12/1/99 2:13 PM	
UserInterfaceChapter-m...	1,267KB	Microsoft ...	12/1/99 11:48 AM	
UserInterfaceChapter.wbk	1,256KB	Microsoft ...	12/1/99 10:55 AM	
DataStructures.doc	623KB	Microsoft ...	12/1/99 9:51 AM	
MemoryAllocation.doc	1,361KB	Microsoft ...	12/1/99 9:51 AM	

9 object(s)

9.49MB (Disk free space: 99.0MB)

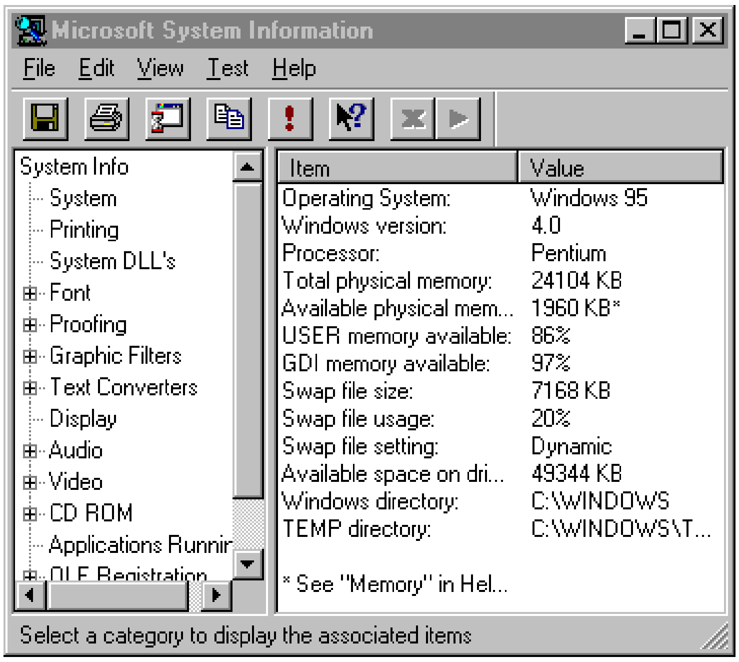
.

Alternatively, you can design special purpose displays to present your memory feedback, rather than integrating then directly into your application's main interface. This avoids cluttering up the display to present users with memory feedback that they do not need, but of course makes it more difficult for users to use, and to learn about, the memory feedback that you do provide. Separate interfaces also make it easier to provide special purpose operations (such as formatting or backing up a storage device) that are not really part of the operations in the system's domain.

As with any user interface, it is important that an interface for displaying memory information is as simple and unintrusive as possible. Obviously, it is not possible for an urgent low memory warning notifier to be unintrusive, but most memory reports and memory meters do not need to be designed to draw users' attention away from the actual work they are doing with the system.

Finally, if you do choose to design one or more separate interface contexts for managing memory, it is important that the information displayed in those interfaces is mutually consistent, and also consistent with the information displayed by the application's main user interface. Microsoft Windows unfortunately provides several counter examples to this principle — with a number of tools that provide information about memory use (from special monitor tools to application about boxes) but where every tool displays quite different, often contradictory information about the system's memory use.

For example, the illustration below shows two of Windows information tools displaying completely unrelated information about memory use. Furthermore, displaying an application "About" box also provides information about memory use — in this case, saying that "Available Physical Memory " is 24104 KB".

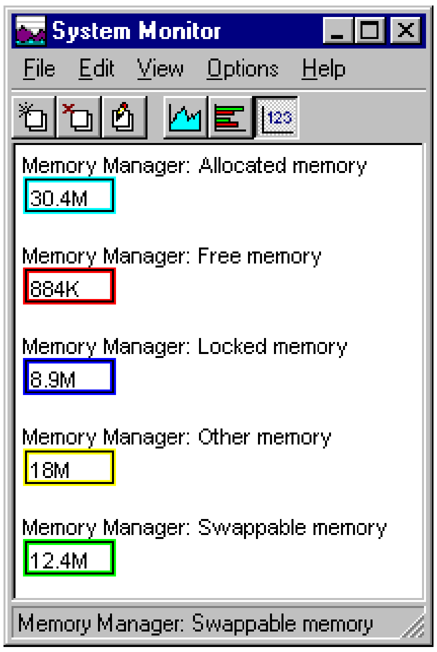


System Info
- System
- Printing
- System DLL's
- Font
- Proofing
- Graphic Filters
- Text Converters
- Display
- Audio
- Video
- CD ROM
- Applications Runnir
- OLE Registration

Item	Value
Operating System:	Windows 95
Windows version:	4.0
Processor:	Pentium
Total physical memory:	24104 KB
Available physical mem...	1960 KB*
USER memory available:	86%
GDI memory available:	97%
Swap file size:	7168 KB
Swap file usage:	20%
Swap file setting:	Dynamic
Available space on dri...	49344 KB
Windows directory:	C:\WINDOWS
TEMP directory:	C:\WINDOWS\T...

* See "Memory" in Hel...

Select a category to display the associated items



Memory Manager: Allocated memory 30.4M

Memory Manager: Free memory 884K

Memory Manager: Locked memory 8.9M

Memory Manager: Other memory 18M

Memory Manager: Swappable memory 12.4M

Memory Manager: Swappable memory

Display Styles

Memory reports do not have to be textual. Graphical displays can be quite effective, especially for presenting an overview of the state of memory use, such as the pie charts used by Windows to display information about disk space, although this particular example is quite intrusive — the pie does not need to be as large as it is! Other graphical alternatives include bar charts, line charts, and even old fashioned meter dials, used by various tools in Unix and Windows NT.

[kjx pictures of resource meters]

Graphs of resource usage against time can also provide much useful information quite plainly, by showing not only the current state of the system, but also the trends as the system runs. For this reason, many resource meters, including Unix's xload and Windows System Monitor use simple time series charts, either as alternatives or in addition to displays showing the current state.

Displaying information about different kinds of memory

Often a system can have several different constraints that apply to different kinds of memory. In this situation, allocating or freeing space in one kind of memory does not affect the memory consumption in another kind of memory space. For example, any of the following memory spaces may be constrained:

- Global heap space shared by the whole system
- Heap space for each individual process
- Stack space for each individual process
- Reservoirs used to store objects in VARIABLE SIZED USER MEMORY.
- Physical secondary storage — in disk, flash cards, or battery backed-up RAM
- Memory buffers or caches used by file or network subsystems, paging, or for loading packages
- Garbage collected heaps (and different regions within the heaps for sophisticated algorithms)

If a system has several different kinds of memory with different characteristics, each of these needs to be treated individually. This will complicate the users' conceptual model of the system, because to understand and operate the system users will have to understand what each different kind of memory is for, how it is used, and how their use of the system affects its use of each different kind memory. Memory reports or memory meters can display information about the different kinds of memory in the system, so that users can manage each kind as necessary.

The TeX batch-mode document formatting system uses pooled allocation, so when it is configured different amounts of memory must be set apart to represent strings, characters within strings, words, control sequences, fonts and font information, hyphenation exceptions, and five different kinds of stack. Every time TeX completes running it produces a memory report in a log file that itemises its use of each different kind of memory. Typically, users only read these log files when TeX runs out of memory, and the information in the log can help determine precisely why TeX ran out of memory. For example, if font memory is exhausted then the document has used too many fonts — this can only be fixed by configuring a version of TeX with more font space, or by using **DATA CHAINING**, and subdividing the document so each part uses fewer fonts. Alternatively, a stack overflow generally means the user has accidentally written a recursive macro call, and needs to find and fix the macro definition.

```
Here is how much of TeX's memory you used:
 523 strings out of 10976
 5451 string characters out of 73130
 56812 words of memory out of 263001
 3434 multiletter control sequences out of 10000+0
 16468 words of font info for 45 fonts, out of 200000 for 1000
 14 hyphenation exceptions out of 1000
 23i,10n,21p,311b,305s stack positions out of 300i,100n,500p,30000b,4000s
```

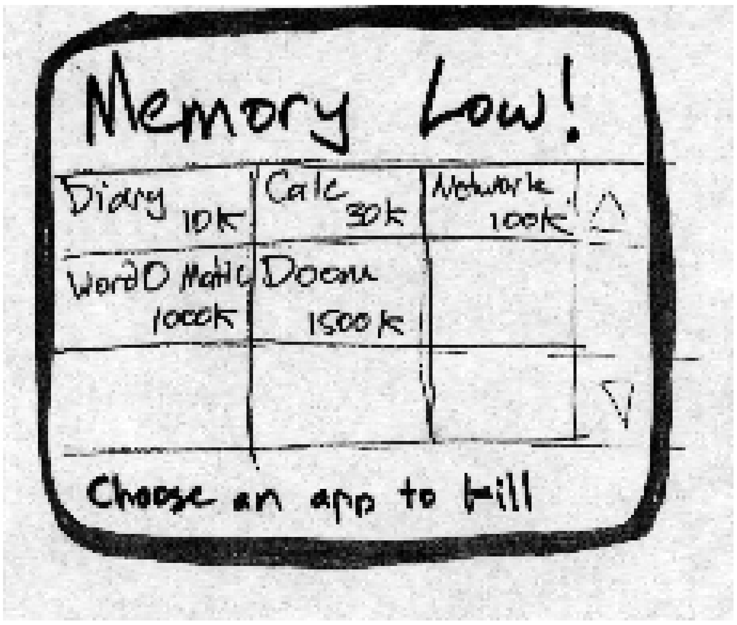
The Self programming language (Ungar and Smith, 1999) includes a sophisticated Spy tool that visualises Self's use of many different sorts of memory.

[kjx pictures of self spy]

Examples

The users' conceptual model for the Strap-It-On PC includes explicit objects that represent running instances of applications, and includes the amount of memory used by the application as one of the attributes of that object. Every time users display applications, the amount of memory they occupy is displayed with them, to ensure users understand the relationships between applications and memory consumption.

The figure below illustrates how this information is included into the Strap-It-On's "Please Kill an Application" dialog. This dialog is presented to the user whenever the system is running low on memory, and lists all the running application in order of their memory use (with the most profligate listed first). Using the wrist-mounted touch screen, uses can choose which application they wish to kill, thereby releasing memory for tasks they presumably consider more important.



Known Uses

The Macintosh and MS-Windows systems display the amount of free disk space in every desktop disk window. As secondary storage has become less of a constraint, the prominence of the disk space meter has declined (from the title bar on the top of a Macintosh window to a footnote in a status line in Windows95).

The virtual machine for the Self programming language incorporated a sophisticated memory meter. The Spy window graphically displayed the memory usage of a number of different heap regions, plus details of paging and garbage collector performance (for example, memory that is swapped out is displayed greyed out).

The Macintosh "About Box" shows the memory usage of each application and the operating system. MS-Windows applications typically also provides some memory use information in their about boxes: their top of the line applications have an over-the-top system information utility in their about boxes – no doubt to assist telephone support engineers working with the likes of Dilbert™'s boss! Windows also provides a series of memory monitors and system tools that are mostly useless (because the statistics they display don't make sense to most users, and don't seen to agree with each other) – but some applications do have their own 'about boxes' that display useful memory information.

See also

USER MEMORY CONFIGURATION describes how memory feedback can be combined with an interface used for adjusting the memory configuration.

If you have a **VARIABLE SIZED USER MEMORY** the feedback can be combined with the interface used for accessing the user memory.

If you are doing **LOW QUALITY MULTIMEDIA** at runtime you can provide feedback about the trade-offs users make between quality and memory.

User Memory Configuration

Sometimes the system needs to adjust its memory requirements to suit user tasks

- You have a medium to large amount of system memory
- The system needs to allocate that memory for a number of different purposes related to the internal functions of the system.
- The allocation depends upon the tasks the user will perform.
- The allocation affects the relative performance or quality of the systems support for some user tasks.

Some programs have medium or large amounts of memory available, but this memory needs to be divided up to service a number of competing demands. For example, the Strap-It-On needs to allocate memory to store temporary copies of users documents as they are being edited, font caches to speed up rendering of those documents when they are displayed, image caches to store images rendered at screen resolution, and so on. Making the memory allocation between these competing demands will alter the performance, quality of user experience, or even the functionality of the system. If for example, you allocate no memory to the font cache perhaps the system won't display proportional fonts; if you allocate no memory to the sounds buffers, it won't play any sounds, and so on.

A simple approach to handling this is to just allocate a fixed amount of memory to each system component — either an absolute amount of memory, or perhaps some proportion of the system's overall available memory. While this is at least a memory allocation it doesn't really solve the problem: memory will be allocated to services in which users are uninterested, and the services the user considers more important will be starved of memory. Even if the system tries to dynamically juggle memory between various different uses, it still effectively has to guess what users' preferences are going to be.

So these approaches really don't solve the problem. The allocation of memory needs to depend upon the current user's current tasks: if editing a document, perhaps fast font rendering is most important; if playing a video game, perhaps stereo sound is important. The system cannot make these choices in advance because it cannot know the priorities of the particular user.

Therefore: *Let users choose the system's priorities for memory.*

Design a conceptual model for the system's memory use. Ideally the model should be related to (or created from) the conceptual model of the interface and the user domain model. For if users have to manage memory themselves, you'll want the interface for it to be as closely integrated with the rest of the system as possible. A Memory Budget is a good basis for such a model.

Design an interface through which users can indicate how they would like memory to be allocated based on the conceptual model underlying the whole interface. Using this interface, users can then juggle memory themselves, allocating memory to those parts of the system they consider important.

User memory configuration can also manage trade-offs between memory demands and other requirements (typically time performance), as well as between competing demands for memory.

For example, the Strap-It-On interface includes a “system properties” tool that displays details of the attached hardware, software, and system services. This tool also allows users to adjust the amount of memory allocated to system services (including the font renderer, disk cache, sound player, and so on), and even to turn unwanted services off, completely removing them from memory.

Consequences

The system can allocate memory just where users wants it, making more *efficient* use of the available memory, because the system doesn’t have to guess the users preferred allocation of memory space. This effectively reduces the program’s *memory requirements* to support any given task.

Users can tailor memory configurations to suit hardware that was not envisaged by a program’s original designers. This increases the *scalability* of the system, because if more memory is available, it can be put to use where it will provide the most benefit.

However: The user’s conceptual model of the system now has to incorporate quite a sophisticated model of systems use of memory. Interactions with the system are more *complicated* because users have to adjust the memory configuration from time to time. If users don’t make a good memory allocation then the system will be worse off than if it had tried to do the allocation itself. These problems can easily reduce the system’s *usability*.

The implementation must be reliable enough to cope when the memory allocation is changed, costing *programmer effort* to implement the flexibility, and *programmer discipline* and *testing costs* to make sure every component is well behaved. By definition, allowing the user control over a system’s memory use must decrease the *predictability* of the system’s memory use.

Supporting centralised user memory configuration encourages *global* rather than local control of memory use.

Implementation

A system’s user memory configuration needs to be tightly coupled to its **MEMORY FEEDBACK**. Before users can decide how they would like the system to allocate its memory, they need information about the system’s current use of memory and the current configuration. Therefore the interface for memory configuration must be as close to the interface for memory feedback as possible — ideally, both patterns should be implemented in the same part of the interface; at a minimum, both interfaces should share similar terms and an underlying conceptual model. **MEMORY FEEDBACK** can also give direct instructions on configuration. For example, MS Windows out of memory message includes the instructions to "close down some applications, then expand your page file".

Alternatively, if the user memory configuration may be a task normally performed by a specialised user role, such as a system administrator, rather than a more normal user. If this is the case, then it makes sense for memory configuration to be supported in the same way as any other configuration task required by the application, typically in a special part of the interface used by the administrator to accomplish all configuration tasks.

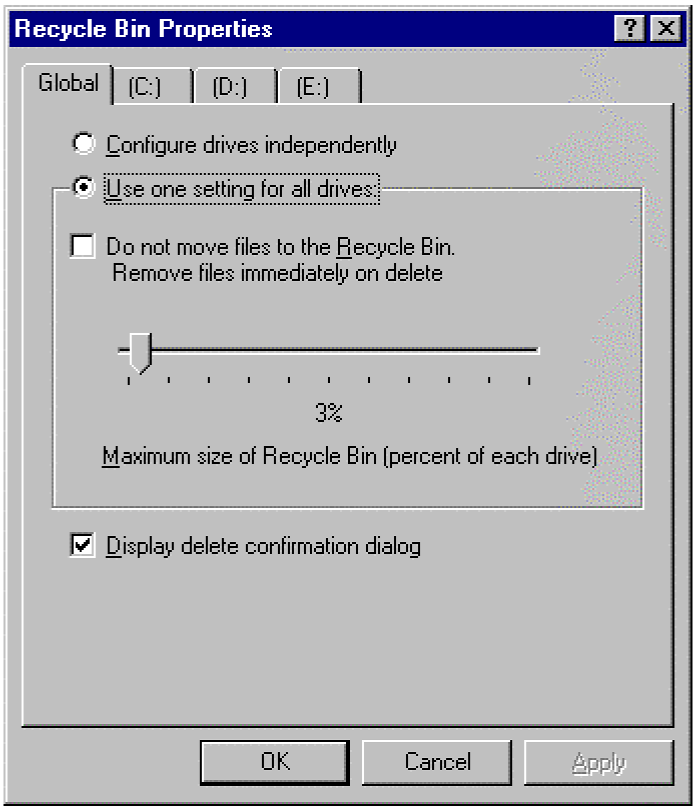
More generally, there are four main techniques you can use to design interfaces for user memory configuration:

- Editable graphical displays
- Dialog boxes with text fields
- Binary choices rather than continuous values.
- Textual configuration files.

Graphical Displays

Where the device hardware supports bitmapped displays and graphical user interfaces, and where configuration is part of the standard use of the system, then the interface for memory configuration should match that of the rest of the program. That is, users should interact with appropriate graphical widgets to supply the information. Generally, some kind of slider control should be chosen to represent the kind of continuous numeric data usually required by configuration parameters.

Microsoft windows and the Palm Pilot all make great use of sliders to configure their memory use. For example, the Windows dialog box below shows a slider use to allocate the secondary storage space for caching deleted files in the 'Recycle Bin'.


Recycle Bin Properties

Global | (C:) | (D:) | (E:)

( ) Configure drives independently
(●) Use one setting for all drives:

☐ Do not move files to the Recycle Bin. Remove files immediately on delete

3%

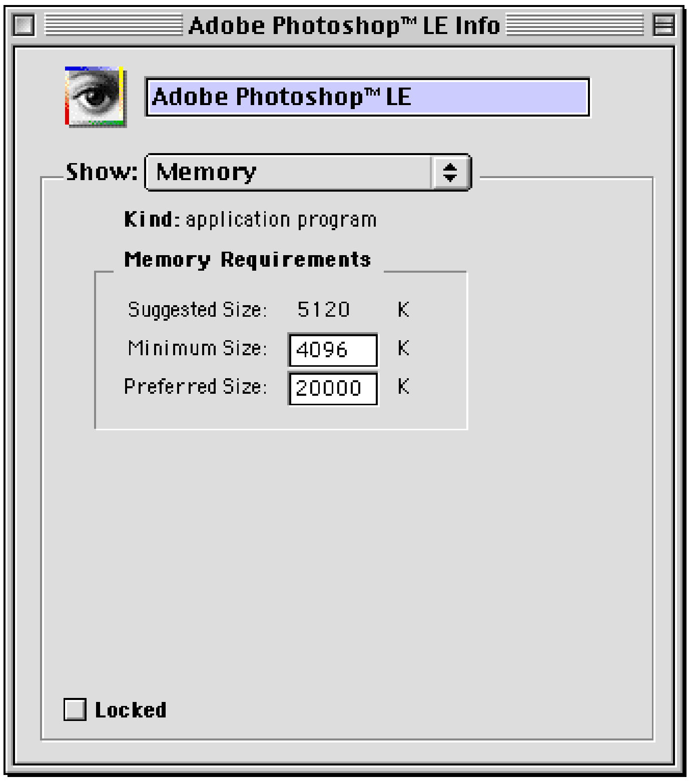
Maximum size of Recycle Bin (percent of each drive)

☑ Display delete confirmation dialog

Text Fields

Technically, graphical sliders and other widgets can have some problems: they can make it difficult for users to enter precise values for configuration parameters, and can be hard to implement correctly. Simple text files can be a common alternative to sliders, especially if a large range of values many need to be entered (from 100K to 500M is not unreasonable) or if precision is more important than graphical style.

For example, Apple Macintosh computers allow users to configure the amount of memory that should be allocated to each program when it is executing, via text entry fields in the application's information box.


Adobe Photoshop™ LE Info

Adobe Photoshop™ LE

Show: Memory

Kind: application program

Memory Requirements

Suggested Size:	5120	K
Minimum Size:	4096	K
Preferred Size:	20000	K

Locked

Binary Choices

Another alternative is to present binary choices about memory use, rather than continuous scalar choices. Binary questions are often easier for users to understand because they can be phrased in terms of a simple conceptual model of the application. Consider the difference between a scalar choice (to allocate between zero and ten megabytes to an antialiased font cache) and a binary choice (to turn font antialiasing on or off). For users to understand the continuous choice, they must not only understand what font antialiasing is, and why they might want it, but also what a byte or a kilobyte is, how many they have at their disposal, and how many they wish to spend on antialiasing. Once they have chosen or altered the configuration value, it may not be easy to see the difference the parameter makes (say between a one hundred kilobyte cache and a one hundred and three kilobyte cache).

Alternatively, a binary choice is simpler than a continuous choice: users need only have a rough idea about what font antialiasing is, and the effects of either value of the parameter will be quickly apparent.

For example, Windows NT allows you to turn various system services on or off, but not to have a service half-enabled. PKZIP offers a choice between optimise for space and optimise for speed.

[kjx example, installer for something? NT SERVER manager]

Textural Configuration Files

The simplest way to implement user memory configuration, especially at configuration time, is to read in a configuration file defining the values of the configuration parameters. If your environment has an operating system, you may be able to use environment variables or registry parameters as alternatives to reading in a file. These options have the advantages that the are trivial to implement, and for users who are software engineers or system administrators may be at least as easy to use as more extensive graphical facilities. On the other hand, they are difficult to use for the less technically savvy, and are typically only suitable for parameters to be set at configuration time.

[kjx example, config.sys or similar?]

Static versus Dynamic Configuration

As with many decisions about memory use, user memory configuration can be either static supported at configuration time before the system begins running, or dynamically adjustable while the system is running. Generally, static configuration is easiest to implement (allowing dynamic configuration of

Handling memory configuration at installation or configuration time also has the advantage that that is when the software's resource consumption will be foremost in users minds, and so they are most likely to be interested in making configuration decisions. The disadvantage is that without having used the software, they will not necessarily be in a good position to make those decisions.

Automatic and Default Configuration

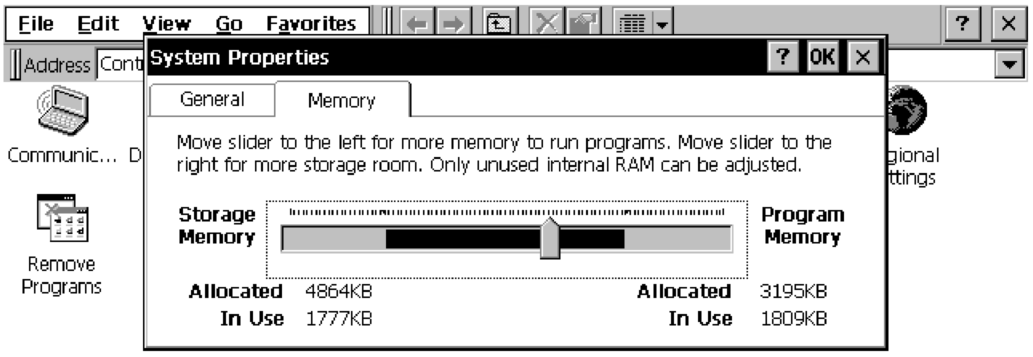
One way to mitigate the negative effects of this pattern is to combine it with some level of automatic allocation, or, at the very least provide some sensible default allocations. This way, unsophisticated users who don't need or want to care about memory allocation will at least receive some level of performance from the system, while those users who are willing to tailor the memory allocation strategy can receive better performance. Also, when users take control of memory allocation, it can be useful to provide some sanity checking, so that, for example, you cannot deallocate all the memory currently used by the running program.

For example most MS Windows installations dialogs allow the user to optimise the use of secondary storage by choose between several pre-set installation configurations, or by designing their own custom configuration. Most users simply use the "workstation" configuration that is selected by default

[kjx wizard?]

Examples

Windows CE uses a slider to configure its most important memory parameter — balancing the amount of memory allocated to storing users data ("Storage Memory") versus the amount of memory allocated to running applications ("Program Memory"). Users can adjust the mix by dragging the slider, within the bounds of the memory currently occupied by storage and programs.

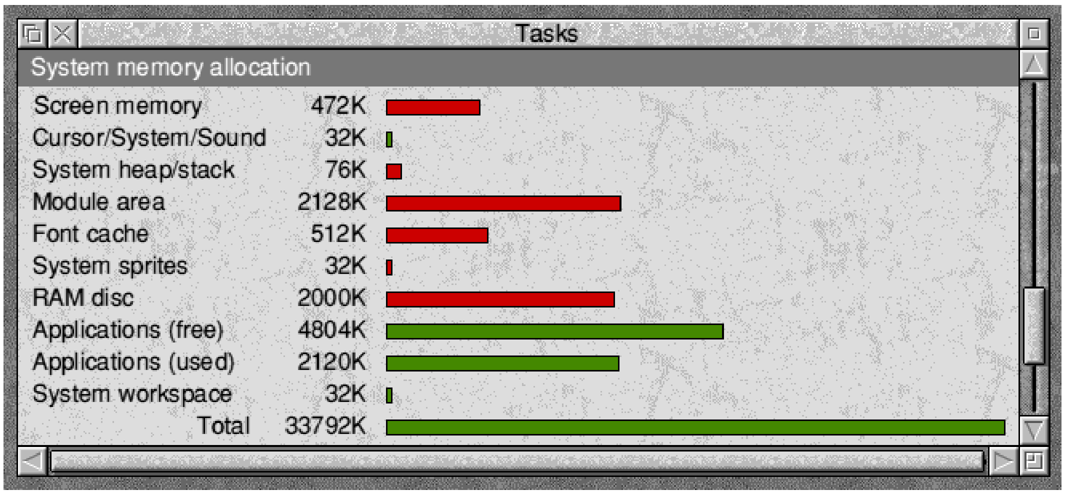


Move slider to the left for more memory to run programs. Move slider to the right for more storage room. Only unused internal RAM can be adjusted.

Storage Memory		Program Memory	
Allocated	4864KB	Allocated	3195KB
In Use	1777KB	In Use	1809KB

The Acorn Archimedes operating system has a paradigmatic example of combined memory feedback and memory configuration. The system can display a bar graph showing the system's current memory configuration and memory use for a whole range of parameters —

for example, 1000K might be configured for a font cache, but only 60K of the cache is actually occupied. The user can then drag the bars representing tuneable parameters to change the system's memory configuration. System services are automatically disabled if the amount of memory allocated to them is zero.



System memory allocation

Screen memory	472K	🟥
Cursor/System/Sound	32K	🟩
System heap/stack	76K	🟥
Module area	2128K	🟥
Font cache	512K	🟥
System sprites	32K	🟥
RAM disc	2000K	🟥
Applications (free)	4804K	🟩
Applications (used)	2120K	🟩
System workspace	32K	🟩
Total	33792K	🟩

Known Uses

Web browsers, such as Netscape and Internet Explorer, allow users to configure the size of page caches in primary or secondary storage.

Venerable operating systems such as MS-DOS, VMS, and Unix have traditionally provided configuration files or command line parameters that can be used to tweak memory use. DOS's CONFIG.SYS is a classic example. Users (for some special meaning of the word) can increasing the number of buffers, open files supported at the cost of decreasing the memory available to applications. More cuddly operating systems, such as Windows NT, sometimes provide configuration dialogue boxes that offer similar features. For example, NT's service manager window allows users to turn operating system services on and off, and turning a service off releases that service's memory. The operating system BeOS takes this one step further and allows users to turn CPUs on and off — mostly off in practice, unfortunately, since this option instantly crashes the machine!

See also.

MEMORY FEEDBACK can provide users with the information they need to make sensible configuration choices when configuring a system's memory use.

Users can be allowed to reduce the amount of memory allocated to multimedia in a program if the program supports dynamically applying **LOW QUALITY MULTIMEDIA.**

Allowing users to alter memory allocation to other parts of the program can be implemented with help from the **PARTIAL FAILURE, CAPTAIN OATS** and **MEMORY LIMIT** patterns.

USER MEMORY CONFIGURATION is the run-time complement to a **MEMORY BUDGET**.

You probably need to use some form of **VARIABLE ALLOCATION** and **MEMORY LIMITS** to implement user memory configuration successfully.

Low Quality Multimedia

Also Know As: Space-time Trade-off; Never Mind the Quality, Feel the Width.

How can you reduce the memory requirements for multimedia?

- You need to support graphics, sound, or other multimedia, to enhance the user experience.
- The memory requirements for the multimedia are too big for your memory budget.
- The broad details of the multimedia are more important than the fine details.
- The multimedia is peripheral to the focus of the program.
- The program needs to provide real-time or near real-time response.

Some programs need to include multimedia presentation or interaction. For example, the original specifications for the StrapItOn required it to play a video of an imploding supernova being consumed by a black whole (complete with 3D graphics and stereo-surround-sound effects by a chart-topping grunge band) every time a memo was deleted from its database. Arguably this should be a candidate for a **FEATURECTOMY** but since competing products support a similar feature, management ruled it had to be included. This pleased Gerald the Geek, who had spent the last three months tweaking the photo-relativistic star implosion simulation after Our Founder's video was cancelled.

Unfortunately, graphics (particularly video animations), sound, three-dimensional models and other multimedia resources occupy large amounts of memory — secondary storage, perhaps, when they are not in use, but also large amounts of primary memory when then are being displayed, played, rendered, or otherwise presented to users. The memory requirements for multimedia can often make a large contribution to the memory requirements of the program as a whole.

Many multimedia resources are, however, only peripheral to the users' tasks the program is supposed to support — that is, they may help ensure user's enjoy using the program, but they don't actually help users get their work done. Microsoft Windows contains a number of examples of such peripheral uses of multimedia — animations in dialog boxes for file move or delete, the irritating help "wizards", and musical phrases played as the system starts up and shuts down. A typical installation of Windows 95 requires six megabytes for sound and music files alone, but will operate quite successfully with sound output muted and wizards deactivated.

So, how can you support complex multimedia presentations while remaining within your memory budget?

Therefore*: Reduce the quality — bits per pixel, sample length, size, complexity, or detail — of the user experience multimedia.*

The memory requirements of a multimedia resource depend upon the size and quality of the resource. If you increase the size or the quality, you will increase the memory requirements. More to the point, if you decrease the quality, you can decrease the memory requirements for a given resource size. So, to fit a given amount of multimedia into a fixed amount of memory — never mind the quality, feel the width — that is, decrease the quality of the multimedia to fit the memory available.

Often, multimedia resources are constructed when the program is being built, and so you can process them in advance — often simply by changing them to use a lower quality format. If you are using dynamic memory allocation, or cannot process all the multimedia resources in advance, consider using **PARTIAL FAILURE** to let quality degrade gradually as resources are consumed, by determining the amount of memory available and reducing quality until they fit.

For example, by reducing the Strap-It-On imploding supernova animation to a 32 pixel square using only 256 colours and showing only 10 frames, the whole animation could be fitted into 10K. Gerald was still upset, until someone else pointed out it meant a) they could include lots more animations if the quality was kept low, and, b) they would need lots of work in advance to compress and tweak the animations.

Consequences

By reducing the quality of the multimedia used in a program, the *memory requirements* for the multimedia, and thus the program as a whole, are reduced. Using lower-quality multimedia can also increase the *time performance* and especially the *real-time response* of the program.

If several pieces of multimedia can be treated in the same way (such as requiring all images to be black and white, and a maximum 200 pixels square) can increase the *predictability* of the program's memory use. Removing truly unnecessary multimedia can increase the program's *usability.*

However: The *quality* of the presentation is reduced, and this may reduce the program's *usability*. Reducing multimedia quality can make it harder to scale the program up to take advantage of environments where the higher quality could be supported, reducing the program's *scalability*.**Programmer *effort* is required to process the multimedia to reduce its quality. Reducing quality dynamically will take even more *effort*, and increase *testing cost*.** Some techniques for reducing multimedia's quality of storage requirements may suffer from *legal* problems, such as software patents.

Implementation

By carefully tailoring your multimedia quality to match the output device, you can keep perceived quality high, while lowering memory requirements. The simplest case is that it is never worth storing data at a higher quality than the output device can reproduce. For example, if you only have a 75dpi screen, why store images at 300dpi? 75dpi images will look exactly the same, but take up much less space. Similarly, if you only have an 8-bit laptop speaker, why store 16-bit sound?

Many sound and image manipulation programs explicitly allow multimedia to be stored in different formats, so that you can make the trade-off between quantity and quality — these formats often use **COMPRESSION** to save memory. Specialist tools are also available for automatically adjusting the quality of image files to be served on the web, to reduce their memory size and thus download time.

Unfortunately, tailoring multimedia to fit a particular device is a one-way trip: if a better device becomes available then low-resolution multimedia will not be able to take advantage of it. Similarly, if you need to enlarge an image, for example, once the resolution has been reduced it cannot be increased.

Other Space Trade-offs

This pattern can also be used to trade off other qualities against memory use and multimedia quality. In particular, multimedia presentations can often require large amounts of CPU time or regular (soft-real time) amounts of it. Meeting these requirements can also reduce a program's absolute time performance or real-time response. Downloading large multimedia presentations requires a large amount of network bandwidth. Reducing the quality of the multimedia in your system can simultaneously increase a program's responsiveness while reducing its requirements for bandwidth, secondary storage, and main memory.

Quality vs. cost trade-offs can be made statically while the program is being written, when the program is installed or configured, or dynamically as the program runs. For example, an animated user interface may ideally need to produce 16-20 frames per second. If resources are not available, it could dynamically either produce fewer frames and display each frame for longer, reducing the frame rate, or it could maintain the frame rate but reduce the amount of detail in each frame. Making trade-offs dynamically means that the multimedia quality can be adjusted precisely to suit the system that is displaying it, but has the disadvantage that the full quality presentation needs to be stored somewhere, typically on secondary storage, or may need to be downloaded over a slow network link. On the other hand, reducing quality at design time means that only the lower quality multimedia needs to be stored or transmitted, and any computation necessary to reduce quality (such as **ADAPTIVE FILE COMPRESSION** can be carried out before hand, without tight constraints on CPU time or memory use.

User memory configuration

With some care, you can design your software so that users can choose what multimedia features will be included as the program runs, and which will be left as options. For example, a web browser could let users choose:

- Whether or not to download images automatically
- If not, when they would like images to be downloaded
- Whether or not to play sounds found on any web pages
- Whether or not to download or show any animated images found on web pages.

As discussed in the **USER MEMORY CONFIGURATION** pattern, these choices could be binary or continuous. While binary choices are easier to understand than a continuous choice, continuous choices could also be used — for example, an option to download only images below a certainly size would enable small images to be downloaded quickly without user intervention, while requiting an explicit user request to download large images.

The Virtue of Prudence

Once you have chosen to reduce the quality of the multimedia, or even the presentation quality of your user interface, you do not have to settle for an interface that feels low-quality overall. In fact, with clever interface design it is possible to make a low-quality interface a virtue, rather than a liability. The early Macintosh user interface design is a classic example of this. Early Macintoshes had physically smaller screens than competing IBM PCs, did not support colour graphics, did not have nearly as many keys on the keyboard or expansion slots inside the chassis. With clever design (and marketing!) all of these were turned in the Macintoshes favour, so that various PC software manufacturers eventually copied its interface to remain competitive. More recently, the PalmPilot is similarly making a virtue of graphics display that has at least ten times less overall resolution than a PC screen. The Playstation is another example — although it cannot match the resolution of a high-quality PC monitor because it must display its output on domestic TV receivers, its games are carefully designed to suit the resulting "grungy" low-resolution feel (and to take advantage of 3D hardware most PCs do not support). In the music industry, recent synthesisers include features to reduce the quality of a sound sample to provide a lower-fidelity distorted sound highly sought after by "alternative" musicians.

Examples

Different kinds of multimedia formats can require vastly different amounts of memory to store, but when it comes to getting an idea across (rather than impressing users with production values) the benefits of higher resolution formats may not be worth the effort.

For example, the ASCII text to describe a small rodent occupies five bytes

```
mouse
```

and a logically expanded version occupies only three bytes

```
rat
```

A simple graphical representation occupies about forty-six bytes:

```
    () ()______
    /**        )   _
   O_\\-m--m-/_____)
```

and a bitmapped version about 108 bytes in GIF format.

A line drawing in postscript occupies rather more space:

[Addison-Wesley to provide an line-drawn icon for a mouse]

and a photorealistic picture, even more.

[Addison-Wesley to provide a library photograph of a mouse]

On a larger scale, Microsoft PowerPoint can save presentations in a number of formats, for different screen types, for printing onto overhead transparencies, or for displaying on the Worldwide web. The space occupied by a presentation can change by a factor of ten depending upon how it is stored.

[need actual examples here]

Known Uses

The Non-Designers Web Book [ndwb] describes how images to be presented on web page can be tailored to reduce their memory consumption (and thus download time) while maintaining good

The Windows95 sound recorder tool makes trade-offs between quality and memory requirements explicit, showing how much memory a given sound occupies, and allowing users to choose a different format that will change the sound file's memory use.

Many desktop computers, from the Apple][series to most modern PC clones, allow users to choose the resolution and colour depth used by the display screen. In many of these machines, including the including the BBC Micro and Acorn Archimedes, choosing a lower resolution or fewer colours (or black-and-white only) left more memory free for running programs.

Web browsers, including Netscape and Internet Explorer, allow users to tailor the display quality to match their network bandwidth, by choosing not to download images automatically or caching frequently accessed pages on local secondary storage.

See also

COMPRESSION can provide an alternative to lowering multimedia quality. Rather than reducing memory requirements by reducing quality, reduce the requirements by using a more efficient data representation. Unfortunately, while many types of multimedia resources can be compressed efficiently for storage, then need to be uncompressed before they can be used, and can cost more *processor time* and *temporary memory* than using the uncompressed resource directly.

A good **FEATURECTOMY** may let you remove unnecessary multimedia from the program completely, while **USER MEMORY CONFIGURATION** will let your users choose how much memory to spend on multimedia, as against getting useful work done.