

## Stella Functional Description

All existing versions of Stella are designed to emulate one or more other operating systems. As a result, at present, Stella does not have a native operating systems interface. This description of the Stella functions, therefore, includes only those functions which have been found necessary to emulate these other operating systems. It does not describe all that could be implemented, nor does it describe the access mechanisms which are operating system interface dependent.

All current implementations of Stella are for MC680x0 processors. There is, however, nothing in the Stella architecture to exclude its use on other processors. Its rational approach to the provision operating systems facilities is particularly appropriate to more specialised devices such as digital signal processors and graphics processors.

### Stella Owners, Users and Things

Many multitasking operating systems encapsulate the task executing a program, that program's code, its data and its context into a single closed unit, creating what is sometimes termed a "virtual machine". The UNIX process is probably the most thoroughly studied form of "total encapsulation". To overcome some of the problems inherent in total encapsulation, two other forms of encapsulation are being promoted. The first is the "multi-threaded process" which allows more than one task (thread) to share a context, data and, possibly, code. The second is the "object" which encapsulates code and data, but which has no context or task of its own.<sup>1</sup>

Both of these "improvements" are a small step towards the Stella approach. In Stella, a task has a context. Indeed, it is the context which defines a task. These are not encapsulated, because tasks and contexts are merely two different ways of looking at the same thing. On the other hand, a task needs neither code nor data of its own. Code and data can exist independently of each other and of any task. This is true in the real world, and Stella is a "real world" operating system.

In Stella anything (device drivers, protocols, algorithms, control blocks etc.) in any form (code, data or both), that might be conceivably be of use to a task, can be linked into the operating system. As they can be anything, they are called "Things".

As a result of this approach, any tasks can share data or code as efficiently as the threads of a multithreaded process: there is no need for them to share a context. Unlike the threads of a multithreaded process, Stella tasks share data and code in an orderly fashion. This is based on two concepts: ownership and usership.

A task **owns** some resources (blocks of memory) such as the memory used to store its context, its stack and data areas, and other tasks. It can grant, to other tasks, access to some or all of the resources it owns. A task **uses** common resources such as shared code, shared data or shared objects. Looking from this point of view, a task uses operating system facilities in the same way as any other common resource. There is nothing special about the operating system.

<sup>1</sup> With the increasing use of table driven applications and threaded code interpreters for user interfaces, large scale data manipulations etc. the distinction between code and data is itself becoming blurred.

## Stella Functional Description

The ownership and usership concepts are used to maintain the system in a tidy state. Tasks themselves do not need to include code to remove themselves. Tasks can be removed at any time, without notice.

The ownership concept is used to ensure when a task is removed, that any resources owned by the task are also removed. Likewise, if a shared resource is removed from the system, any users of that resource are also removed. This can be used to ensure, for example, that when shared memory is released by the removal of a task, all tasks using the shared memory are also removed. This is similar to the removal of a multithread process, but is far more general.

Shared resources can be used by a task for the whole of its existence (in the same way as a dynamic link library on other systems), or a task can use and free resources as required.

### Stella Memory Model

Stella uses a "real world" memory model. References to memory use real addresses. Program code should be position independent, and programs neither need to make, nor should make, any assumption about the location of any item in memory, whether it is shared or private<sup>2</sup>. There is, however, provision for transient memory allocation which can be used to provide virtual workspace. This virtual workspace can be used for any transient memory purposes: scratch pads, filing system buffers, window save areas or even (if the hardware is available) virtual memory pages.

With the exception of tasks which are encapsulated in a virtual memory scheme, all memory has a common address space. This does not mean that it is necessarily shared, as, hardware permitting, individual tasks can be restricted to accessing their own memory pages and, possibly, some shared pages.

### Stella Communications

Stella makes use of intrinsically safe communications structures. These can provide facilities for passing messages or data, synchronising tasks or signaling.

There is a range of standard structures which can handle fixed (1 bit upwards) or variable (1 byte upwards) sized items. There are standard structures for queued and others for one item at a time communications. New structures can be created for new requirements.

The elimination of semaphores from these structures not only removes the possibility of deadly embraces (neither interrupt priority induced single semaphore embraces or multiple semaphore embraces are possible in Stella), but also prevents high priority tasks having to wait for a low priority task to set a semaphore.

As no additional protection is required for these communication mechanisms, they are frequently incorporated as in-line code rather than being called as subroutines.

---

<sup>2</sup> This view of memory is closer to reality and is, therefore, both more efficient and easier to maintain than a virtual memory scheme. The only problem that might be encountered is with the "C" programming language, which was designed for a virtual memory environment. The assumptions of a fixed program location are easily taken care of, and since any substantial data areas used by a "C" program are likely to be allocated by malloc(), "C" programs which make fixed locations of data are very rare.