

FSW is TWO systems, not one

- ◆ The FSW that we know is indispensable to spacecraft operations is a real-time system. It drives the hardware of a robotic vehicle and its instruments.
- ◆ As missions become more capable, a second system emerges. It's a non-real-time system that manages data.

The second system

- ◆ The software we usually think of when we say “flight software” is the real-time, safety-critical, interrupt-driven *foreground* system.
- ◆ The other part of “flight software” – which we typically don’t recognize as being different in nature – is the non-real-time, discretionary, time-sharing *background* system.

Where it lives

- ◆ Scheduling theory tells us that the real-time tasks should use no more than about 60% of CPU time in order to ensure they don't miss deadlines.
- ◆ The remaining 40% of CPU – “idle time” – isn't really idle: it's time available for interruptible tasks, the second system.
- ◆ That second system should include everything that is not truly real-time.



Second system overview

- ◆ All tasks run at the same priority, the lowest priority supported by the O/S.
 - Possibly multiple background subsystems.
 - To the real-time FSW they all look like “idle”.
- ◆ Tasks have no deadlines.
 - Locking one another out for prolonged periods is okay, so long as everybody gets a chance to run eventually.
 - Each task must release CPU on finishing a unit of work – usually blocks on something.



Through the looking glass

- ◆ The foreground and background systems of a spacecraft's FSW share many qualities that distinguish them – both – from workstation or PC software.
- ◆ But in some ways the character of the background system is the exact inverse of the character of the real-time FSW.
- ◆ And the two can coexist in perfect compatibility. It's been demonstrated.



What they have in common...

- ◆ ...is most of the flight project cultural values:
 - Project schedule can't be jeopardized.
 - Module coupling must be minimized.
 - No dynamic system memory allocation.
 - Make maximum use of available resources.
 - Contain and, as possible, tolerate faults.
 - Test as you'll fly, fly as you tested.
 - Formal, controlled development process.

Where they differ

- ◆ What is optimized
 - Determinism
 - Portability
- ◆ Mutual exclusion
- ◆ Data sharing
- ◆ Message passing
- ◆ Memory management

Context

- ◆ The MSL architecture is clearly an excellent approach to the design of an FSW real-time system (but other good approaches are possible).
- ◆ Postulate: the ION architecture is a good approach to the design of an FSW non-real-time system (but again other good approaches are possible).

What is optimized

- ◆ The foreground system must be reliable, else you lose the spacecraft.
 - Fixed scope enables minimal, mission-specific design which enables comprehensive testing.
- ◆ The background system must be efficient, else not enough work gets done.
 - Minimize wasted space and cycles.
 - This adds complexity, so support portability: reliability comes from extensive multi-mission testing history.

Mutual exclusion

- ◆ Real-time FSW can't tolerate lengthy mutual exclusions: tasks miss deadlines.
- ◆ In the background system, no problem. You can serialize an entire subsystem on a single mutex, because there are no deadlines.
 - Only one task runs at a time anyway.
 - Loops, function calls in critical section: okay.
 - Minimize cycles spent on task switching.

Data sharing

- ◆ Data sharing requires mutual exclusion, so it's no good in the foreground system.
- ◆ In the background system, lengthy mutual exclusion is okay – so shared access to data is okay.
- ◆ Which is good, because shared access is also the fastest way multiple tasks can operate on the same data.

Message passing

- ◆ Because shared access is excluded, the real-time system uses message passing to enable multiple tasks to operate on common data.
- ◆ But because shared access is okay in the background system, message passing is not needed.
 - No cycles wasted in copying anything.
 - Signal “data ready” by giving a semaphore.

Memory management

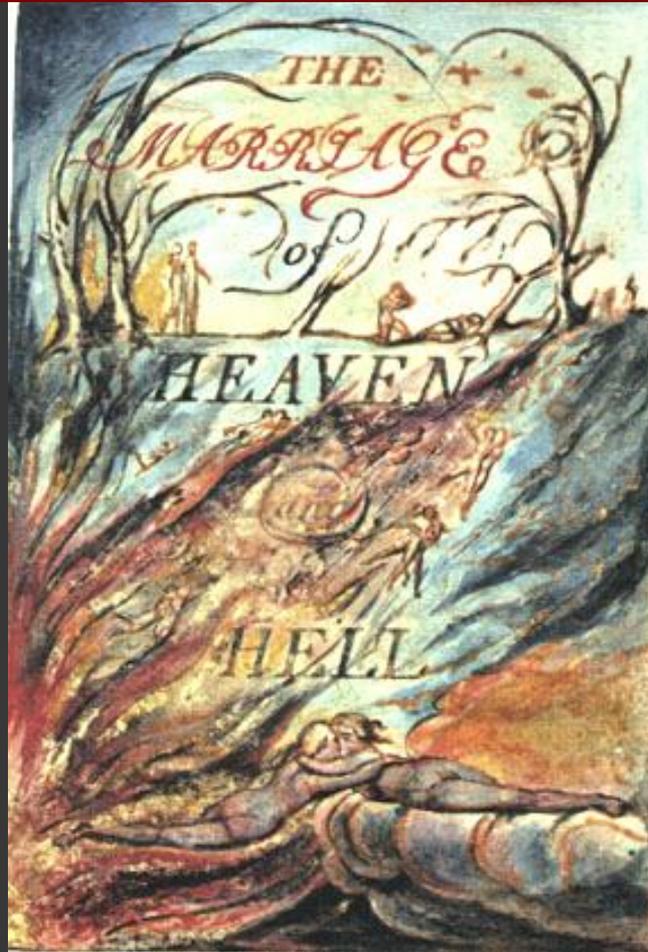
- ◆ Because mission scope is fixed, all foreground FSW memory can be in fixed-length arrays, each with margin.
- ◆ But the background system gains reliability from being multi-mission, hence portable and evolvable.
 - Management of private common heap: pooled resource, pooled margin, efficient use of space.
 - Automatically adapts to mission scope change.

How they work together

- ◆ The foreground system never calls the background system's library functions – never blocks, ignores background tasks.
- ◆ Background tasks are interrupted whenever foreground tasks need to run.
- ◆ For communication between the two:
VxWorks message queues.
 - Non-blocking at the foreground end.
 - Blocking at the background end.

Happy coexistence

FSW Architecture and Design Principles



http://en.wikipedia.org/wiki/The_Marriage_of_Heaven_and_Hell

