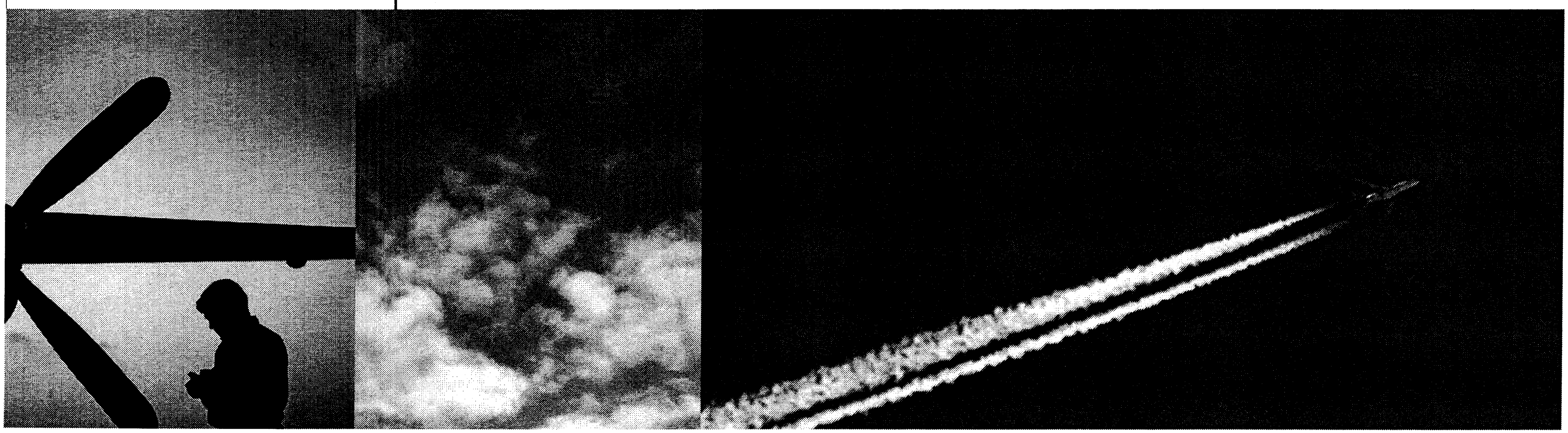
Flight Software Design and On-Orbit Maintenance

Alexander C. Calder
Flight Software Sustaining Engineering Group
November 5, 2007



CSC
EXPERIENCE. RESULTS.

Introduction

- Why modify FSW in flight?
  - Work around hardware problems on orbit
  - Correct software problems missed pre-launch
  - Enhance software capabilities during mission
- Aspects of FSW design affecting maintainability
  - Resource margins
  - Linking: static v. dynamic
  - On-board file systems
  - Parameter setting: table-driven v. command driven
- Methods for in-flight FSW modification
- Case studies

FSW Design: Resource Margins

- Memory
  - Modified code = bigger code (usually)
  - GSFC Code 582 recommends at least 20% margin on memory that can be written in-flight
- Telemetry
  - Modified code may add new telemetry
  - Need:
    - Spare space within packets
    - Capacity to add new packets
    - Bandwidth margin

FSW Design: Linking

- Static Linking
  - Code built as absolute executable image
  - FSW component locations fixed by linker
  - No symbol table needed
  - Patch capability requires spare space in image
  - Flight memory image exactly duplicable on ETU
- Dynamic linking
  - Application code built as relocatable module(s)
  - Module locations assigned by OS at runtime
  - Symbol table maps modules to locations
  - Flight memory image cannot be exactly duplicated on ETU
  - Tradeoff:
    - Requires different maintenance approach
    - Offers potentially greater flexibility

FSW Design: File Systems

- To be really useful, file system (if there is one) should support:
  - File creation & deletion
  - File copy & move
  - Directory structure
  - Directory creation & deletion

FSW Design: Parameters

- Parameter setting via command is convenient
- Parameter setting via table load supports configuration control
- Tradeoff: operational convenience v. configuration control

Methods for FSW Modification In-flight

- Statically Linked Code
  - Inline Patch
  - Jump-Logic-Return Patch
  - Task Replacement
- Dynamically Linked Code
  - Task Replacement
  - Adding a New Module
  - Function Replacement
- Load New Image & Reboot

Inline Patch

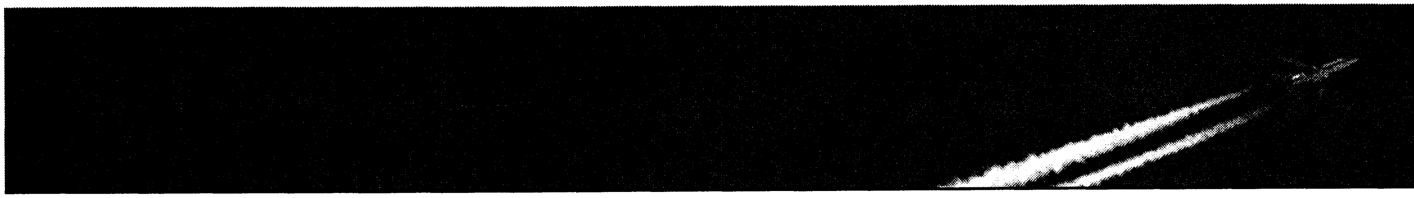
- Overwrite one or a few words in executing RAM image
- Can't add new code
  - Change must fit within existing code
- Feasible for:
  - Change to a hardcoded constant
  - Change to one or a few machine instructions
- Caveat: Is the target instruction cached?

Jump-Logic-Return

- Jump from existing code to new code, then return to existing code
- Can be done at source code level (e.g. in C) by replacing a function
  - New function loaded to unused memory
  - Calls to old function patched inline to call new function
- Requires free space built in to FSW image

Task Replacement (Static)

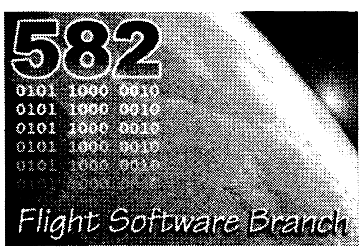
- Halt task, load new version, restart task
  - Task must be designed to stop & restart cleanly (no memory leaks, broken pipes, etc.)
  - Task/fault management must permit task to stop & restart
  - Overwrite task in place
    - Requires each task have its own spare memory
  - Leave old task in place
    - Requires spare memory somewhere big enough to accommodate new task
  - If task can't be restarted independently, FSW has to be rebooted



CSC
EXPERIENCE. RESULTS.

Task Replacement (Dynamic)

- Halt task, load new version, restart task
  - Uses OS task management & file system
  - Task must be designed to stop & restart cleanly (no memory leaks, broken pipes, etc.)
  - Task/fault management must permit task to stop & restart
  - Each task image should reside in its own file
  - File system should be flexible
    - Allow multiple versions of task files
    - Allow creation/deletion/concatenation of files



CSC
EXPERIENCE. RESULTS.

Adding a Module

- Some OS (e.g., VxWorks) can load a separate executable module
- Module may not execute automatically when loaded
  - Can be spawned as a task from OS shell
  - Functions in module can be invoked from OS shell
- Module's global symbols (function names, global variables) added to OS symbol table when loaded
- Module can access FSW globals via symbol table

582
0101 1000 0010
0101 1000 0010
0101 1000 0010
0101 1000 0010
0101 1000 0010
Flight Software Branch

Function Replacement

- Achieve the effect of a jump-logic-return in a dynamically linked system
- Function addresses not known a priori
- Addresses must be obtained from symbol table

Load New Image & Reboot

- May be necessary if FSW architecture precludes other methods
- Uplink may take multiple passes over several days
- Requires reboot of processor for changes to take effect
- Reboot probably puts spacecraft in safehold
  - Disrupts normal mission ops
  - May pose risk to sensitive instruments

Case Study 1: Jump-Logic-Return
Adding an In-Flight Timing Diagnostic on Space Technology 5

- Digital Sun Sensors (DSS) on ST5 reported multiple spurious sun pulses
- Accurate diagnosis required precise timing data on DSS ISR task execution
- All FSW tasks had calls to timing diagnostic output functions used during development
- Patch added a new function to accumulate timing data for subsequent dump to ground
- Existing diagnostic output functions patched to call new function
- Code is written in C and is statically linked

Case Study 1: Jump-Logic-Return Memory Map of ST5 FSW Image

Start Address	End Address	Description	Size
0x80020000	0x801749CF	OS & FSW Code	1.3 MB
0x801749D0	0x801FFFFF	Spare Space	557 kB
0x80200000	0x802D3AEB	Initialized & Uninitialized FSW Data	846 kB
0x802D3AEC	0x806FFFFF	Spare Space	4.2 MB
0x80700000	0x807439D3	OS Kernel Code & Data	270 kB
0x807439D4	0x807FFFFF	Free Memory Pool	750 kB

Case Study 1: Jump-Logic-Return
As-Launched Source Code for Timing Diagnostics

```
void OSPerfLog_entry(u_dword id)
{
    OS_write_io_word(id, 1);
}

void OSPerfLog_exit(u_dword id)
{
    OS_write_io_word(id, 0);
}
```

Case Study 1: Jump-Logic-Return
As-Launched Disassembly for Timing Diagnostics

```
ffffffff8016fc88:     24050001     li      $a1,1
ffffffff8016fc8c:     0c05c3f4     jal     80170fd0
<ClockRate+0x7f5ff4d0>
ffffffff8016fc90:     00000000     nop

ffffffff8016fcc4:     00002821     move    $a1,$zero
ffffffff8016fcc8:     0c05c3f4     jal     80170fd0
<ClockRate+0x7f5ff4d0>
ffffffff8016fccc:     00000000     nop
```

Case Study 1: Jump-Logic-Return
Modified Source Code for Timing Diagnostics

```
void OSPerfLog_entry(u_dword id)
 {
    OSPerfLog_add(id, 1);
 }

 void OSPerfLog_exit(u_dword id)
 {
    OSPerfLog_add(id, 0);
 }
```

Case Study 1: Jump-Logic-Return Modified Disassembly for Timing Diagnostics

```
ffffffff8016fc88:     24050001     li      $a1,1
ffffffff8016fc8c:     0c0c0000     jal     80300000
<ClockRate+0x7f78e500>
ffffffff8016fc90:     00000000     nop

ffffffff8016fcc4:     00002821     move    $a1,$zero
ffffffff8016fcc8:     0c0c0000     jal     80300000
<ClockRate+0x7f78e500>
ffffffff8016fccc:     00000000     nop
```

CSC
EXPERIENCE. RESULTS.

Case Study 2: Task Replacement Figure of Merit (FOM) task on Swift/Burst Alert Telescope (BAT)

- Dynamically linked code modules with flexible file system
- Uplink steps
  - Break new FOM task object into multiple files
  - Uplink files over several passes
  - Concatenate files into one task object file
  - Stop old FOM task
  - Load new FOM task to RAM from object file
  - Start new FOM task
- On-board startup script modified to start new FOM task in event of reboot
- Old FOM task still present as object file
  - Can be used if new FOM task fails

582
Flight Software Branch

Case Study 3: Patching Dynamically Linked Code

- A proof-of-concept experiment using Swift/BAT FSW lab
- Uses symbol table features of VxWorks
- Test function:

```
#include "dummyfunc.h"
void rundummy()
{
  dummyfunc();
}
```

Case Study 3: Patching Dynamically Linked Code

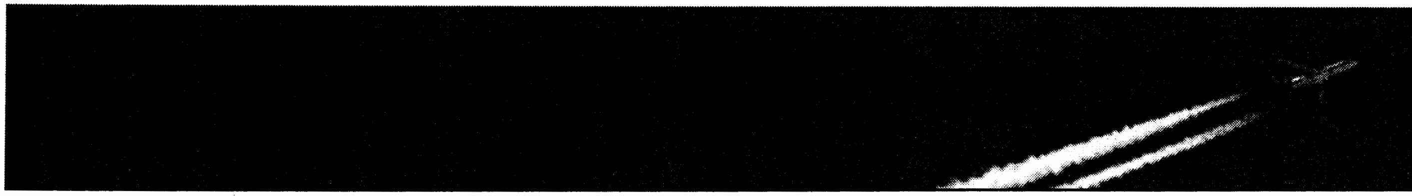
```
#include <stdio.h>
#include "dummyfunc.h"
int myvalue=1984;

void dummyfunc() {
        sy_shellStream->write("I am a C function dummyfunc1\n");
        sy_shellStream->write("myvalue is: %d\n", myvalue);
}
```

```
#include <stdio.h>
#include "dummyfunc.h"
void dummyfunc2() {
        sy_shellStream->write("I am another C function dummyfunc2\n");
        sy_shellStream->write("myvalue=%d\n", myvalue);
}
```

Case Study 3: Patching Dynamically Linked Code

- Fragments from function patchit1():
- Declarations

```
#define BL_MASK 0x48000001
#define SX_MASK 0x03FFFFFF
char *dummyfunc_name = "dummyfunc__Fv";
char *dummyfunc2_name = "dummyfunc2__Fv";
char *call_function_name = "rundummy__Fv";
char *whereis_dummyfunc = NULL;
char *whereis_dummyfunc2 = NULL;
char *whereis_target = NULL;
char *whereis_call_function = NULL;
unsigned long int branch_to_old;
unsigned long int branch_to_new;
unsigned long int target_offset = 0x000C;
```



CSC
EXPERIENCE. RESULTS.

Case Study 3: Patching Dynamically Linked Code

- Get function addresses

  symFindByName(sysSymTbl, dummyfunc_name, &whereis_dummyfunc, &dummyfunc_symtype);

  symFindByName(sysSymTbl, dummyfunc2_name, &whereis_dummyfunc2, &dummyfunc2_symtype);

  symFindByName(sysSymTbl, call_function_name, &whereis_call_function, &call_function_symtype);

582 Flight Software Branch

CSC
EXPERIENCE. RESULTS.

Case Study 3: Patching Dynamically Linked Code

- Test Results:
- Load all functions

  ld <targ/nram/temp/dummyf1.o
  value = 7326896 = 0x6fccb0 = myvalue + 0x3cc
  ld <targ/nram/temp/dummyf2.o
  value = 7337136 = 0x6ff4b0 = dummyfunc2(void) + 0x4ac
  ld <targ/nram/temp/rundummy.o
  value = 7336480 = 0x6ff220 = rundummy(void) + 0xfc
  ld <targ/nram/temp/patchit1.o
  value = 7332528 = 0x6fe2b0 = patchit1(void) + 0x8bc

582
Flight Software Branch

Case Study 3: Patching Dynamically Linked Code

- First execution of rundummy

  rundummy()

  I am a C function dummyfunc1
  myvalue is: 1984

Summary

- Maintainability should be considered in FSW design
  - Margin on system resources
  - Spare memory in a static FSW image
  - Task modularity in a dynamic FSW system
  - Flexibility of file system