r e p o r t

MIT Kavli Institute
One Kendall Square, Room 300-545
Cambridge MA 02139–4307
Tel: 617-253-7277
Fax: 617-253-8084

To: ACIS Science Operations Team
From: Peter Ford, NE83-545 <pgf@space.mit.edu>
Date: June 6th 2016
Subject: Creating and Testing ACIS Flight Software Patches (v 1.0)

1. Preliminaries

Before starting out, check that the `ACISFS`, `FS`, and `CVSROOT` environment variables are set, and that the MIPS cross-compilers are at the head of your `PATH` and `LD_LIBRARY_PATH` variables. These definitions are essential to ensure that the procedures described in this document execute correctly.

```
setenv ACISFS /nfs/acis/h3/acisfs
setenv FS $ACISFS/flightbld/flight1.5
setenv CVSROOT $ACISFS/configcntl
setenv PATH $ACISFS/$ARCH/bin:$PATH
setenv LD_LIBRARY_PATH $ACISFS/$ARCH/lib:$LD_LIBRARY_PATH
```

The unpatched flight software must be accessible at "*$FS*" and it is assumed that you have created a private copy of the patch build tree using *cvs*, *i.e.*,

```
cvs checkout patches
```

Before starting on a new patch, it is good practice to check that you have no outstanding updates to existing patches. To verify this, change your working directory to "*patches*" and type

```
cvs diff
```

Before testing your patches, check that the *expect* and *wish* (*tcl* interpreter) commands are in your execution path and that they access their dynamic *tcl* and *tk* libraries.

2. Updating an Existing Patch

This is much easier than creating a new patch. If you add new procedures or class methods to an existing patch, *e.g.*, to "*mypatch*", you'll need to update `PKGFUNCS` in "*mypatch/standalone.mak*" (if that file exists) and add new "`func`" command lines in "*mypatch/mypatch.pkg*". Several other lines in that file may need to be updated, especially the "`test`" commands that are to be run automatically during regression testing, which refer to test scripts, usually written in the *expect* dialect of *tcl*, in subdirectories of "*mypatch/testsuite*". See §4,5 for details.

3. Adding a New Patch

The first step is to think up a name for you patch, *e.g.,* "*mypatch*", and create a new subdirectory of that name in your local "*patches*" directory. Within that directory, you should create a series of files. The precise choice will depend on the type of patch. For instance, inline patches will need only an assembler source, "*mypatch.S*".

Makefile	to create a stand-alone "*mypatch.bcmd*" (sometimes named *standalone.mak*)
mypatch.C	to replace one or more C++ methods in the BEP
mypatch.H	to define variables and C++ classes and structures for the BEP
mypatch.c	to replace one or more C routines in the FEPs
mypatch.h	to define variables and C structures for the FEPs
mypatch.S	to create inline assembler patches in the BEP or FEPs
mypatch.mak	to compile "*mypatch*" as part of a patch release
mypatch.pkg	to control all phases of compilation, linking and testing
testsuite	directory used for regression tests

The "*mypatch.pkg*" and "*mypatch.mak*" files are always required. Consult the following sections for examples of how to set them up. Once the various files are present, tell *cvs* about them by executing the following command in your "*patches*" directory:

```
cvs add mypatch
```

This merely alerts *cvs* to the locations of these files and directories. Once you have compiled a stand-alone version of "*mypatch*" and tested it a few times, you'll want to save the files you added. Do this by typing

```
cvs commit mypatch
```

You'll be prompted to edit one or more lines of text which will act as descriptions of the files. If you want to describe the files individually, execute a separate "`cvs commit` *filename*" for each file. If you need to change a file after committing it, just edit and save it and then execute "`cvs commit` *filename*". This time, the text that you enter will be logged against a release number and will serve to document your changes.

4. Contents of *.pkg Files

Much information about the patch is to be contained in the ASCII file "*mypatch/mypatch.pkg*", which contains a mixture of comments and commands. Comments begin with "`#`" and continue to the end of the line. Commands begin with one of the following keywords:

`approval`	5 fields: <*release*> <*patch-load-eco-number*> <*username*> <*date*> <*action*>
`bcmd`	file to contain *addpatch* commands for standalone patch
`docref`	MKI document reference <`36-*****.**`> (optional)
`eco`	number of the ECO describing this patch
`environment`	either "`flight`" or "`engineering`"
`fepinline`	2 fields: <*pseudo-assembler-listing of patch*> <*pseudo-assembler-listing of FEP module*>
`fepobject`	"*mypatch.o*" to link with other FEP objects
`func`	2 fields: <*original BEP method*> <*replacement BEP method*>
`ident`	`$id$` field for revision control
`object`	"*mypatch.o*" to link with other BEP objects
`partnumber`	ACIS configuration database ID for the patch item
`reason`	a brief description of the patch
`sco`	number of software change order relating to this patch
`ser`	number of software enhancement request relating to this patch
`source`	name of *pkg*, *mak*, or source file to build patch
`spr`	number of software problem report relating to this patch
`test`	3 fields: <*test-type*> <*test-directory*> <*test-command*>
`tool`	set to "`PENDING`" if no *bcmd* file is to be generated
`version`	the revision number or letter of this patch

Most of these commands should only appear once in the file, but others can be repeated, *e.g.*, "`approval`" should define each successive version of the patch; "`func`" should name each function that is to be replaced; "`source`" should identify each source file that is to be compiled or assembled; multiple "`spr`" reports may be applicable; and there can be as many "`test`" lines as necessary to run each of the regression tests.

Finally, "*mypatch.pkg*" should end with four sections containing ASCII text to describe the patch, its usage and its impact on ACIS execution. These sections must be headed with the four capitalized lines, displayed in blue in the following:

```
NOTES:
The reason for the patch, symptoms and impact of the anomaly, description of the fix, etc.

COMMAND IMPACT:
Commanding changes required when this patch is active.
```

```
TELEMETRY IMPACT:
Telemetry changes to expect when this patch is active.

SCIENCE IMPACT:
Changes that may appear in ACIS science data when this patch is active.
```

5. Patches that replace BEP methods

These are usually the simplest patches to write. To replace one or more BEP methods, *e.g.*, `MyProg()` in `MyClass with a method of the same name`, create a "*mypatch.C*" file containing the following C++ code:

```
// Include C++ headers that define the existing classes you're using
#include "ipcl/interface"
#include "filesscience/MyClass.H"

// Define a new class Test_MyClass and define its replacement method(s)
class Test_MyClass : public MyClass
{
public:
     void MyProg();
};

// Define the replacement method(s)
void Test_MyClass::MyProg()
{
     // insert code
}
```

Since your replacement method will probably want to access `private` or `protected` variables in `MyClass`, most ACIS classes already define themselves as friends of a class of the same name but prefixed with "`Test_`", *i.e.*,

```
class MyClass {
  friend class Test_MyClass;
  . . .
}
```

but this may not always be the case, so you may have to override the `private` and/or `protected` statements in the "`*.H`" include files by adding the following lines before the `#include` statements:

```
// Include this redefinition if you have trouble accessing private variables
#ifndef private
#define private public
#endif

// Include this redefinition if you have trouble accessing protected variables
#ifndef protected
#define protected public
#endif
```

If *mypatch* is a standard (required) patch, copy "*BUILD-template/MakeStandard*" to "*mypatch.mak*". Then add the following line at the start of "*mypatch.mak*".

```
PKG     = mypatch
```

Alternatively, if *mypatch* is an optional patch, copy "*mypatch.mak*" from "*BUILD-template/MakeOptions*" and add the following line at the beginning:

```
OPTIONS = mypatch.o
```

Standard patches can replace FEP subroutines in the same way as BEP methods: when building the patch load, each object file is automatically relinked into a single FEP object which the *addPatch* commands append to the FEP load array in BEP memory. However, there is no easy way to create an optional patch that replaces FEP subroutines, since there is no optional FEP object to link against, so the patch has to include BEP code to load its FEP component. Look at the *ctireport1* and *ctireport2* patches for examples.

To process our simple patch example, "*mypatch.pkg*" must include "`source`" commands to name the files required to compile it, a "`func`" command to name each replaced method, and an "`object`" command to indicate that the patch will create an object file that is to be linked with others into a common patch load. Finally, there should be one or more "`test`" commands to replicate the problem that necessitated the patch, and one or more "`test`" commands that show that the problem has gone away when the patch is applied.

```
func    MyClass::MyProg Test_MyClass::MyProg
object  mypatch.o
source  mypatch.pkg
source  mypatch.mak
source  mypatch.C
test    bug1 testsuite/bug make SCRIPT=runtest1.tcl ACISSERVER=$(ACISSERVER) \
        TOOLS=$(TOOLS) PATCHDIR=$(PATCHDIR)
test    bug2 testsuite/bug make SCRIPT=runtest2.tcl ACISSERVER=$(ACISSERVER) \
        TOOLS=$(TOOLS) PATCHDIR=$(PATCHDIR)
test    fix testsuite/fix make SCRIPT=runtest.tcl ACISSERVER=$(ACISSERVER) \
        TOOLS=$(TOOLS) PATCHDIR=$(PATCHDIR)
```

In this example, two tests demonstrate the problem, run from *expect* scripts "*runtest1.tcl*" and "*runtest2.tcl*" in the "*mypatch/testsuite/bug*" directory, and one test verifies the patch: "*runtest.tcl*" in "*mypatch/testsuite/fix*".

You will also need to create a "*Makefile*" to compile *mypatch* into a stand-alone "*mypatch.bcmd*". You can start by executing the following in your "*mypatch*" directory:

```
../tools/bin/standalone.sh mypatch
```

which will create "*standalone.mak*" in that directory and then use it to attempt to create "*mypatch.bcmd*". For complicated patches, you may need to edit "*standalone.mak*", and you might choose to rename it "*Makefile*" for later convenience.

6. Inline BEP Patches

Before attempting an inline patch, some familiarity with the MIPS CPU architecture and its assembler language is a necessity. You also need to know how the GNU *g++* compiler uses the CPU registers. To help design inline patches, all ACIS flight software has been compiled with the '`-Wa,"-alh"`' option which writes a commented pseudo-assembly listing to *stdout*. The current build in "*/nfs/acis/h3/acisfs/flightbld/flight1.5*" contains a "**.lst*" listing for every compiled object "**.o*", and if the rules of the previous sections are adopted, the patch objects will do so too.

The first step after identifying the code to be patched is to identify its "**.lst*" file. Since the sources were compiled with maximum debugging, the listing will contain many extraneous lines. If they get in the way, filter the file through *lst2txt* to remove them: only the assembler instructions and commented C and C++ code will remain. The assembler code is output in columns, as follows:

```
line-number  hex-offset   hex-value      mnemonic   arguments   # comment
```

Note that some instructions will generate two or more "`hex-value`" words: the GNU assembler recognizes more instruction mnemonics than are implemented in the MIPS 3000 architecture, so these will be translated into groups of two or more MIPS instructions to perform the task. In most cases the inline patch will not need to reference addresses outside the patch itself, but if this is necessary, the registers must be loaded by hand, as described below.

There are two MIPS peculiarities that you must remember at all times: (1) when the CPU executes a 'branch' or a 'jump' instruction (hint: the mnemonic starts with 'b' or 'j'), it always executes the next instruction as well, whether or not it takes the branch or jump, and (2) when an instruction calls for a word to be loaded from memory into a register, or stored into memory from a register, the word will not arrive at its destination until at least two machine cycles later, *e.g.*, when loading into a register, its contents cannot be used by the next instruction. This feature will often force the GNU compiler to add a null instruction, "`nop`", between 'load' and 'use'. These `nop`'s are very useful for inline patches since they can be replaced by any single-instruction command not associated with the registers that are currently loading or storing their contents.

As an example, the *rquad* patch fixes a problem in the BEP's `Pixel3x3::attachData()` method which was compiled from the following C++ code:

```
if (col > 0) {
    quad = exposure->getQuadrant(col - 1);
    doclk[0] = exposure->getOverclockDelta(quad);
    split[0] = exposure->getSplitThreshold(quad);
}

quad = exposure->getQuadrant(col);
doclk[1] = exposure->getOverclockDelta(quad);
split[1] = exposure->getSplitThreshold(quad);

quad = exposure->getQuadrant(col);
doclk[2] = exposure->getOverclockDelta(quad);
split[2] = exposure->getSplitThreshold(quad);
```

The third "`quad =`" instruction is passing the wrong value to `getQuadrant()` – it should be `col+1`. Here's that part of "*/nfs/acis/h3/acisfs/flightbld/flight1.5/filesscience/pixel3x3.lst*" that includes the bad instruction:

```
209:              **** split[1] = exposure->getSplitThreshold (quad);
1066 05b0 8e020054         lw      $2,84($16)
1067 05b4 00000000         nop
1068 05b8 8c42000c         lw      $2,12($2)
1068      00000000         nop
1069 05c0 02002021         move    $4,$16
1072 05c4 0040f809         jal     $31,$2
1073 05c8 02202821         move    $5,$17
1077 05cc afa2002c         sw      $2,44($sp)
210:              ****
211:              **** quad = exposure->getQuadrant (col); # <== should be (col+1)
1080 05d0 8e020054         lw      $2,84($16)
1081 05d4 00000000         nop
1082 05d8 8c420008         lw      $2,8($2)
1082      00000000         nop
1083 05e0 02002021         move    $4,$16
1086 05e4 0040f809         jal     $31,$2
1087 05e8 02402821         move    $5,$18
1091 05ec 00408821         move    $17,$2
212:              **** doclk[2] = exposure->getOverclockDelta (quad);
1094 05f0 8e020054         lw      $2,84($16)
1095 05f4 00000000         nop
1096 05f8 8c420004         lw      $2,4($2)
1096      00000000         nop
1097 0600 02002021         move    $4,$16
1100 0604 0040f809         jal     $31,$2
1101 0608 02202821         move    $5,$17
1105 060c afa20020         sw      $2,32($sp)
```

The calls to the functions `getSplitThreshold`(), `getQuadrant`() and `getOverclockDelta`() are made with '`jal`' instructions. By convention, register $4 points to the class object associated with the method, *i.e.*, `exposure`, and the function arguments are in registers $5, $6, and $7. (If more arguments are needed, they are passed on the execution stack.) Note that in these examples, $5 is loaded while the branch is being taken. All functions return their results in register $2 and (if needed) register $3, and they return to the address in $31. Starting at line 1080 (offset `0x05d0`), our example performs the following operations:

```
1080 05d0 lw   $2,84($16)  # load address of 'exposure' into register 2
1081 05d4 nop              # wait while register 2 loads
1082 05d8 lw   $2,8($2)    # load address of getQuadrant() into register 2
1082 05dc nop              # do nothing
1083 05e0 move $4,$16      # copy contents of register 16 (which contains the
                           #   address of the pix3x3 object) into register 4
1086 05e4 jal  $31,$2      # save address of next instruction in register 31
                           #   and branch to the address in register 2
1087 05e8 move $5,$18      # copy the contents of register 18 (which contains
                           #   the value of 'col') into register 5
1091 05ec move $17,$2      # copy the value returned by getQuadrant() into reg 17
```

Since the '`move`' instruction at line 1087 (offset `0x05e8`) is executed while the jump instruction is being processed, `getQuadrant`() is called with register 4 pointing to the `pix3x3` object and register 5 contains the value of `col`. A careful inspection of "*pixel3x3.lst*" shows that register 18 isn't used after line 1087, so its contents can easily be incremented by replacing an available '`nop`' instruction. The *rquad* patch replaces the last '`nop`' before the call, at line 1082 (offset `0x05dc`). The code to insert a single unsigned add instruction is written to "*rquad.S*" and contains the following assembler instructions:

```
        .text
        .globl   pixel3x3_lst_05d4_05d4
        .ent     pixel3x3_lst_05d4_05d4
pixel3x3_lst_05d4_05d4:
        addu     $18,$18,1          # $18 contains the "col" value
        .end     pixel3x3_lst_05d4_05d4
```

where the "`addu`" command replaces offsets `0x05d4` through `0x05d4` of "*pixel3x3.lst*", *i.e.*, a single instruction The starting location of the patch must also appear in a "`.globl`" instruction so that it can be recognized by the loader. The remaining directives are required by the MIPS cross-assembler, which should be invoked with the following options:

```
acis-g++ -fvtable-thunks -c -g -mno-gpopt -G 0 -mlong-calls -Wa,"-alh"
```

Compiling "*mypatch.S*" with these options will create "*mypatch.o*", which can be ignored, and will write a listing to its standard output, which you should redirect into "*mypatch.lst*" and then process as follows:

```
../tools/bin/digestInline.pl patchId mypatch.lst bepmap > mypatch.bcmd
```

where *patchId* is the 16-bit integer value for the *patchId* field of the first *addPatch* command (subsequent values will be incremented), and *bepmap* is a valid BEP load map used to resolve global references in "*mypatch.lst*". The *patchId* values will be reassigned when flight versions are created: all values in a given load must be unique. The above inline example makes no references to BEP addresses outside the patch itself. If this is necessary, the address should be determined from the BEP load map and loaded into a register explicitly. For instance, if the patch is to call the "`BiasThief::biasReady(void)`" method, the assembler code would be:

```
        lui      $2,0x800a         # $2 = 0x800a0a10
        or       $2,$2,0x0a10      #
        jalr     $2,$31            # call BiasThief::biasReady(void)
```

The address of `biasReady()` must be split into upper and lower half-words and loaded into register 2 in two steps. Note that the MIPS assembler can do this for you in a single "`la`" instruction, but this will not be correctly interpreted by the "*tools/bin/digestInline.pl*" script run by "*mypatch.mak*".

7. Patching the FEPs

A purely inline patch to apply identically to all FEPs is prepared in the same way as an inline BEP patch described above, but the "*digestInline.pl*" script must be called with two additional arguments: a valid FEP load map (the unpatched version is "*$FS/fep/acisFepSci.map*") and the file that contains the default FEP software load array, "*$FS/fep/acisFep.c*".

If a FEP patch adds a new subroutine or replaces an existing one, the process is quite different. Any inline code should be collected in "*mypatch.S*" and the subroutine(s) in "*mypatchX.c*". The latter must be compiled, linked, and converted to FEP loader format:

```
acis-gcc -fvtable-thunks -c -g -nostartfiles -nostdlib \
         -Wl,"-Tfeppatch.x" -Wl,"-RacisFepSci.ab" mypatchX.o -o mypatchX.ab -lc
acis-nm –numeric-sort –demangle mypatchX.ab > mypatchX.map
cnvrtFepObj.pl cmbox ringbuf mypatchX.ab > mypatchX_patch.c
```

The resulting "*mypatchX_patch.c*" defines one or more structures to be added to the FEP load array in BEP memory. The next step is to compile the inline code "*mypatch.S*" and resolve its external references. In this example, it is assumed that these references are in a FEP source module "*$FS/fep/fepSource.c*" which has been compiled into a pseudo-assembler file "*fepSource.lst*":

```
acis-gcc -fvtable-thunks -c -g -mno-gpopt -G 0 -mlong-calls -Wa,"-alh" \
         mypatch.S > mypatch.lst
resolveFep.pl mypatchX.map mypatchX.lst fepSource.lst > mypatchX_tmp.lst
```

Finally, the inline code and the subroutines are merged into the patch load, "*mypatch.bcmd*":

```
digestInline.pl patchId mypatchX_tmp.lst bepmap mypatchX.map acisFep.c > mypatch.bcmd
mergefep.pl patchId bepmap mypatchX.map acisFep.c mypatchX_patch.c >> mypatch.bcmd
```

The above prescription will allow you to build stand-alone FEP patches. If more than one is to be run in the same patch load, the load maps and absolute FEP image must be updated as each patch is created. The code in "*release/Makefile*" will do this automatically provided "*mypatch.pkg*" contains the following entries:

```
fepobject mypatchX.o
fepinline mypatch.lst fepCtl.lst
```

For this to work, "*mypatch.mak*" must correctly refer to the various maps and images, *i.e.*, "*feppatch.x*", "*acisFepSci.map*", "*acisFepSci.ab*", "*fepSource.lst*", and "*acisFep.c*". Consult the existing patches, *e.g.*, *condoclk*, for examples.

8. Adding New FEP Modes

If the FEP patches are not intended to be permanent, but only to be applied for the duration of a single science run, the build procedure is rather different because the BEP must itself load the patches into the FEP and then ensure that the usual FEP software is reloaded for the next science run. The patch source will consist of one or more C++ source files (*e.g.*, "*mypatch.C*") containing new and/or replacement BEP methods, a "*mypatchFep.S*" file containing inline FEP instructions, and perhaps a "*mypatchFepX.c*" file containing new FEP subroutines. The first step is to compile "*mypatchFepX.c*" into "*mypatchFepX.ab*", "*mypatchFepX.map*" and "*mypatchFepX.c*", as in the previous section. Then "*mypatchFep.S*" should be compiled, linked against "*mypatchFepX.map*", and appended to "*mypatchFepX.c*":

```
acis-gcc -fvtable-thunks -c -g -nostartfiles -nostdlib \
        -Wl,"-Tfeppatch.x" -Wl,"-RacisFepSci.ab" mypatchFepX.o -o mypatchFepX.ab -lc
acis-nm –numeric-sort –demangle mypatchFepX.ab > mypatchFepX.map
cnvrtFepObj.pl cmbox ringbuf mypatchFepX.ab > mypatchFepX_patch.c
acis-gcc -fvtable-thunks -c -g -mno-gpopt -G 0 -mlong-calls -Wa,"-alh" \
        mypatchFep.S > mypatchFep.lst
digestFep.pl mypatchFepX acisFepSci.map mypatchFep.lst >> mypatchFepX_patch.c
acis-gcc -fvtable-thunks -c -g -mno-gpopt -G 0 -mlong-calls -Wa,"-alh" \
        mypatchFepX_patch.c -o mypatchFepX.o > mypatchFepX.lst
```

The "*mypatch.pkg*" file should contain two "`object`" definitions, "*mypatch.o*" and "*mypatchFepX.o*", which will be linked together and converted to "*mypatch.bcmd*" by running "*release/Makefile*".

For the FEP patches to be applied, "*mypatch.C*" must patch two BEP methods from "*filesscience*": `setupFepBlock()` to call `fepManager.loadRunProgram`(*fepId*, *mypatchFepXHanger*) to load the patch into *fepId*, and `terminate()` to call `fepManager.resetFeps()` to ensure that the BEP reloads the usual FEP software for the next science run. Consult the "*cc3x3*" patch for an example of how this is done and how to set up "*mypatch.mak*".

9. Stand-Alone Testing

All stand-alone tests are performed in sub-directories of a patch's "*testsuite*" directory. These tests are of two types, depending on the function of the patch. If it is to correct a known deficiency, two sets of tests will be performed, a "*bug-hw*" or "*bug-sw*" to reproduce the problem without the patch, and "*fix-hw*" or "*fix-sw*" to verify that the patch eliminates the problem. If the patch adds a new capability to the instrument, a single class of "*smoke*" tests will be developed to verify that the patch executes successfully without catching fire! The contents of a "*testsuite*" directory for a bug-correction patch will look something like the following:

testsuite/bug-hw	run tests here to replicate a deficiency
testsuite/bug-hw/Makefile	run an *expect* script
testsuite/bug-hw/runtest.tcl	*expect* script to run a test to replicate a deficiency
testsuite/fix-hw	run tests here to demonstrate a patched deficiency
testsuite/fix-hw/Makefile	run an *expect* script
testsuite/fix-hw/mypatch.bcmd	copy of stand-alone patch
testsuite/fix-hw/runtest.tcl	*expect* script to run a test to patch a deficiency

Scripts and data files common to all tests should be stored in "*testsuite*" itself. The tests are written in the *expect* extension of the *tcl* language. The ACIS convention is that each test script is passed three arguments:

basedir	the directory containing the "*mypatch*" files
toolsdir	the location of patching tools relative to "*basedir*"
patchdir	the location of the current test directory relative to "*basedir*"

Within the *expect* scripts, it is important to refer to auxiliary files through these arguments because their values will change when the same scripts are used to test a patch build, and again when certifying the build. A typical script begins as follows:

```
#! /usr/bin/env expect

# —— Save the three command arguments ——
lassign $argv basedir toolsdir patchdir

# —— Embed the procedure library ——
source $basedir/$toolsdir/lib/lib-exp/runtest_support.tcl

# —— Start a command pipe ——
spawn $basedir/$toolsdir/bin/cmdclient $env(ACISSERVER)
```

```
set cmd_id $spawn_id
# —— Start a telemetry pipe ——
spawn $basedir/$toolsdir/bin/tlmclient $env(ACISSERVER)
```

This establishes the test environment, allowing a "`send -i $cmd_id`" command to issue commands to the BEP and "`expect`" commands to monitor the BEP's telemetry packets, after filtering through "*psci -m -u*". The next part of the *expect* script loads the patches and warm-boots the BEP:

```
# —— Reload patches ——
cold_boot
load_patch_list "$basedir/$toolsdir/share/opt_tlmio.bcmd\
                 $basedir/$toolsdir/share/opt_printswhouse.bcmd\
                 $basedir/$toolsdir/share/opt_dearepl.bcmd\
                 mypatch.bcmd"
# —— Reboot and apply patches ——
warm_boot
# —— Set the pixel switch for image loader input ——
system make loaderselect
```

This example loads the optional "`tlmio`" and "`printswhouse`" patches that cause software housekeeping entries to be reported as user pseudo-packets, from which they can be recognized in "`expect`" commands. It also loads the "`dearepl`" patch that ignores the DEA and expects input from the image loader peripheral. Finally, it assumes that stand-alone "`mypatch`" is present in the test directory as "*mypatch,bcmd*" so it loads the patch and tells "*Makefile*" to send a command to the EU's pixel switch to select input from the image loader.

The next step is typically to power up one or more FEPs and video boards. This is done by defining a list of 6 CCD numbers to assign to the 6 FEPs (using `ccdId=10` to indicate that no CCD is to be assigned).

```
# —— Assign CCDs to FEPs ——
set ccd_list {10 7 5 0 1 3}
set last_fep {5}
# —— Power up the boards ——
power_on_boards $ccd_list
# —— Wait for the boards to power up ——
expect {
   -re ".*SWSTAT_FEPMAN_ENDLOAD: $last_fep\[\r\n]*" { }
    timeout { fail "Power-up Failure" }
}
```

The `last_fep` variable is set to the index of the last FEP to be powered up, and is used in the subsequent `expect` statement to recognize when this FEP has been loaded, so the script can continue. We are now ready to start a science run. We assume that "*Makefile*" contains targets "`bias`" and "`image`" to load the image loader with a test bias map and a test x-ray candidate image, respectively.

```
# —— Load parameter block and wait for acknowledgment ——
send -i $cmd_id "load q te 4
    parameterBockId      = 0x00000001
    fepCcdSelect         = $ccd_list
    fepMode              = 2 # FEP_TE_MODE_EV3x3
    bepPackingMode       = 2 # BEP_TE_MODE_GRADED
    . . .
    fepLoadOverride      = 0
}
"
command_echo 1 9 {load te}
```

```
# —— Send a bias map to the image loader ——
system make bias

# —— Start the run and wait for acknowledgment ——
send -i $cmd_id "start 2 te 4\n"
command_echo 1 14 {start science run}
```

When commanding the BEP, use "send -i $cmd_id *cmd*" where *cmd* is a valid *bcmd* command. *cmd* must end with a newline character. Then call "command_echo" to wait for the BEP to acknowledge the command. "command_echo" takes 3 parameters: the value of the "result" field of the expected "*command_echo*" packet denoting success, the value in the "commandOpcode" field, and a string to include in the resulting error message should "command_echo" encounter a failing command or a reply timeout.

With the command under way, the *expect* script will typically wait until the bias map has been created, after which it will send dummy events into the image loader and wait until the jobs starts producing exposure records. The script commands will look something like the following:

```
# —— Monitor job execution ——
expect {
    -re "bepStartupMessage.*\[\r\n]*" {
      fail {Bus crash reproduced}
    }
    -re "SWSTAT_FEP_STARTBIAS.*\[\r\n]*" {
      system make image
      exp_continue
    }
    -re "exposure\[^\r\n]*\[\r\n]*" {
      send -i $cmd_id "stop 3 science\n"
      command_echo 1 19 {Stop science run}
      exp_continue
    }
    -re "scienceReport.*\[\r\n]*" {
      pass {Science run ends successfully}
    }
    timeout {
      fail {Science run timed out}
    }
}
```

Note that each clause of the "expect" command defines a test which, if successful, causes one or more statements to be executed. If "pass" or "fail", the script ends immediately with an appropriate message. Otherwise, the clause must end with "exp_continue", otherwise the "expect" command will terminate. Each test should result in either a "pass" or a "fail" in order to report success or failure in the test log.

Once a test script has been debugged, it should be added to the "test" directives in its "*mypatch/mypatch.pkg*" file so that it will be used as a recursion test for subsequent patch loads.

Before running a stand-alone test, the "shim" interface to the engineering unit must be established. Each "*Makefile*" in the "*testsuite*" directories must contain targets "shim" to set up the interface and "unshim" to release it again when the tests are completed. Stand-alone tests are best run in 'report' mode, in which a copy of the *expect* script is prefixed to its output. If you have several expect scripts, *e.g.,* "*runtest1.tcl*", "*runtest2.tcl*", etc., specify the name of the particular script in the command line, as follows:

```
make report SCRIPT=runtest2
```

If "SCRIPT=*name*" is omitted, "*runtest.tcl*" will be run. Copies of the script's standard output and error streams will be written to "*mypatch.target.YY.MM.DD.HH:MM:SS*.log" where "*target*" is defined in the "*Makefile*".

10. Building a Patch Load

All patch loads are compiled and tested in the "*release*" subdirectory of "*patches*" under the direction of the "*PatchRelease.spec*" file in that directory. That file contains a series of directives, i.e., a command followed by one or more arguments. The most important commands are:

`releasetag` *tag*	the *cvs* tag associated with all required (standard) patches and all build utilities
`require` *name*	the name of a required patch
`optiontag` *tag*	the *cvs* tag associated with all optional patches and all build utilities
`option` *name*	the name of an optional patch
`ipcltag` *tag*	the *cvs* tag associated with the IP&CL definitions in this build
`depends` *p1 p2*	if optional patch *p1* is included in the load, patch *p2* must also be present
`conflict` *p1 p2*	optional patches *p1* and *p2* cannot appear in the same optional load.

The order of "`require`" and "`option`" commands is critical. They will be built in that order. The remaining commands only affect the contents of the logs that are generated while the patch load is built and tested.

The current ACIS convention is that "`releasetag`" values are named "review-release-X" while under development and "release-X" after the ECO is released, and "`optiontag`" values are named "review-release-*X*-opt-*Y*" and "release-*X*-opt-*Y*", respectively. After updating "*PatchRelease.spec*", change to the "*patches*" directory and type

```
cvs diff
```

to check that there are no outstanding updates not yet committed to the *cvs* repository. Then type

```
make tags
```

to tag the standard and optional files with the appropriate "`releasetag`" and "`optiontag`" values from "*PatchRelease.spec*".

You are now ready to build the patch load by changing to the "*release*" directory and typing

```
make distrun
```

which will compile everything in sub-directories named "*standard*" and "*options*" and create a "*dist*" subdirectory for the *bcmd* command files and load maps. If the compilation succeeded, save the patches in the "*tools/share*" directory and copy the results to "*/nfs/acis/h3/acisfs/patchbld*" by typing

```
make share
cd ../tools/share
cvs commit
```

Before testing the patch load, there is one last task to perform. To ensure that the "*TXinit*" parameter block of the "`txings`" patch retains its original location in the D-cache heap, it may be necessary to adjust the value of "`TY_DUMMY`" in "*txings/txings.C*". So inspect the value if "`TXinit`" in the "*release/options/BUILD/opt_txings.map*" file that you have just created and, if it isn't `0x8003dc30`, add or subtract the appropriate number of words (each of 4 bytes) from "`TY_DUMMY`" to make it so. If you have made changes in "*txings/txings.C*", use "`cvs commit`" to copy them to the repository, then execute "`make tags`" in the "*patches*" directory and "`make distrun`" and "`make share`" in "*patches/release*".

Once a patch has been successfully tested, it should be documented in an Engineering Change Order (ECO) and reviewed by the ACIS team. When a patch load includes several new or updated patches, they are usually reviewed at the same time, but each patch is described by its own ECO. Find the ECO of a similar existing patch and modify it—past examples have used various versions of Adobe Frame, Microsoft Word and Apple Pages. Present the review board with a hardcopy or PDF of the ECO, of the source code in the "*mypatch*"

directory, and of the log files from the "*testsuite*" directories. The ECO itself should be saved in the "*mypatch*" directory and checked into *cvs*.

11. Regression Testing

The first step in testing the many patches that go into a patch load is to establish a reliable connection to the ACIS Engineering Unit. To start the interface, either to test a stand-alone program in a sub-directory of a particular "*testsuite*" directory, or to test a patch release in the "*patches/release*" directory, type

```
make ACISSERVER=host shim
```

where "*host*" is the name of the computer that is attached to the Image Loader and the L-RCTU interface. Currently, this is the host named "*cypress.mit.edu*", running Linux CentOS 6.4. It is usually more reliable to run the test in the same computer as the interface. It takes one or two minutes to start the interface, at which time a message "Port CRTRTS turned on" will be written to "*host*.log" in the current working directory. Most tests, including those that test patch loads, will use the Image Loader, which can be selected by typing

```
make ACISSERVER=host loaderselect
```

The regression tests may now be run by typing

```
make ACISSERVER=host distcheck < /dev/null >& patch-FGH-2.log &
```

In this example, the standard output and error streams are redirected to a file and the tests are run in the background. If the command is to be run from a remote window, the standard input should be redirected to "*/dev/null*". The status of the tests can be monitored by passing the file to the *./show-status* command, *viz.*

```
# ./show-status patch-FGH-2.log
Module           Test       Line  Lines   Mins     Result
fepbiasparity2   bugCc      1741   1139   0:11  *** PASS ***
fepbiasparity2   bugTe      2880    371   0:05  *** PASS ***
buscrash2        fixCc      3252    233   0:04  *** PASS ***
fepbiasparity2   fixCc      3486   1166   0:11  *** PASS ***
buscrash2        fixTba     4653    275   0:04  *** PASS ***
buscrash2        fixTe      4929    414   0:05  *** PASS ***
. . .
txings           smoke     27667  10373   1:26  *** PASS ***
teignore         smoke     38041    451   0:06  *** PASS ***
ccignore         smoke     38493    438   0:05  *** PASS ***
All              All       38727          6:45  *** PASS ***
```

Once a patch load has been successfully tested, *i.e.*, all tests have reported "`*** PASS ***`", you should copy them one final time to the distribution directory,

```
make share
```

and create an ECO to describe the patch load. This is typically very short: a title page listing the individual patch updates, a table showing the contents of the standard and optional patches, and a diagram of the BEP and FEP storage areas showing how much space is still available in D- and I-cache.

12. Patch Certification

Once the individual patch changes and additions have been reviewed and the patch load compiled and tested, individual combinations of optional patches must be separately validated. This is done in the "*certsrc*" directory. Before making any changes, type

```
make clean
```

to flush out the remains of previous tests. Each combination of optional patches is given its own sub-directory in "*certsrc*", *e.g.*, at patch level FGH, four optional patches were desired: "*cc3x3*", "*eventhist*", "*compressall*" and "*txings*", but only in three combinations, so the directories were named

```
cc3x3+eventhist
cc3x3+eventhist+compressall
cc3x3+eventhist+compressall+txings
```

Within each of these directories are subdirectories for testing each of the optional patches used in the compination. For instance, the "*cc3x3+eventhist+compressall+txings*" directory contains

cc3x3	test of *cc3x3* patch in combination with other patches
cc3x3+eventhist+compressall+txings.pkg	script to control certification tests and
compressall	test of *compressall* patch in combination with other patches
eventhist	test of *eventhist* patch in combination with other patches
smtimedlookup	test of *smtimedlookup* patch in combination with other patches
txings	test of *txings* patch in combination with other patches

The contents of the test directories themselves, *e.g.*, "*cc3x3+eventhist/cc3x3*", are usually copied directly from the corresponding "*smoke*" or "*fix-**" directories of the individual patches, with their `load_patch_list` arguments augmented by "*standard.bcmd*" and the particular combination of "*opt_*.bcmd*" patches. New files and directories should be added to the *cvs* repository via the "*cvs add*" command. Once the contents of the "*certsrc*" directories are fixed, tag the files and run the tests, *e.g.*,

```
cd certsrc
cvs tag release-F-opt-G-cert-H
make all < /dev/null >& cert-FGH-1.log &
```

The test will apply, independently, to each combination of optional patch specified by the `CERTIFICATES` macro in "*certsrc/Makefile*". For each combination, all the standard "*fix**" and "*smoke*" tests will be run from their respective "*release/standard/*/testsuite*" directories, but with the full set of standard and optional patches required for this combination. This is achieved by assigning the space-delimited list of "**.bcmd*" patch files to the `$CERT_PATCHES` environment variable. When an *expect* script invokes a "`load_patch_list`" procedure, the latter checks whether this variable is defined. If it is, it ignores the list of files it is commanded to load, and loads the files named in `$CERT_PATCHES` instead. After testing the standard patches, "*certsrc/Makefile*" runs the "*smoke*" tests for the optional patches in the combination. As in regression testing, the progress of the certification test can be determined by filtering the log file through "*./show-status*" in the "*certsrc*" directory.

13. Patch Release

To create a PDF describing the patch release, first create a directory in "*patches/archive*" with a 3-letter name denoting the levels of standard and optional patches, and of certification, *e.g.*, "*FGH*". In this directory, install a "*Makefile*" to put everything together. The most important variables in this file are

`LEVEL=`	*`N-N-N`*	Patch and certification levels
`MOD=`	*`mypatch`*	Name of the updated patch
`ECO=`	*`nnn`*	Number of ECO defining *mypatch*
`ECO1=`	*`nnn`*	Number of ECO defining the patch release
`ECO2=`	*`nnn`*	Number of ECO defining the certification
`SPR=`	*`name`*`.ps` *`name`*`.ps`	Name of one or more software problem report files
`DIST=`	*`dir`*	Directory containing notes from regression tests
`CERT=`	*`dir`*	Directory containing notes from certification

The sources to be listed in the PDF must be specified individually in the "`SRC=`" list. Otherwise, the "*Makefile*" puts everything together automatically. It only remains to distribute it to the review board and place a copy in "*acis.mit.edu:~ftp/pub*" for general access.

14. Environment Variables used while Building and Testing Patches

`ACISSERVER`	The name of the computer connected to the image loader and L-RCTU
`ACISTOOLSDIR`	The pathname to ACIS EGSE tools
`ARCH`	The type of operating system in use, *e.g.*, "linux", "solaris", etc.
`CERT_PATCHES`	Pathnames of **.bcmd* patches used in certification regression tests
`CVSROOT`	The pathname to the ACIS *cvs* repository
`LD_LIBRARY_PATH`	A list of dynamic libraries to use
`PATH`	A list of directories containing executable files

15. Existing Patches

The following table lists all patches current at this time. "Usage" refers to how the patch is tested and combined in a load: a required patch is always included in the "standard" part of a load; one or more optional patches are added to the standard part and the combination is then certified; EGSE patches are only run on the engineering unit, either as part of regression testing and certification, or to test some other ACIS function. "Type" distinguishes in-line patches from those resulting in an object file ("**.o*") which may be linked with other objects for inclusion in a patch load.

patch name	usage	loc	type	comment
badpix	required	BEP	inline	fix error in bad pixel location
biastiming	obsolete	BEP	object	fix hand-over between science and bias thief threads
buscrash	required	BEP	object	fix bus crash when FEP powered down while creating bias maps
buscrash2	required	BEP	object	fix bus crash when FEP powered down while trickling bias maps
cc3x3	optional	BEP	object	add CC 3x3 continuous clocking mode
		FEP	inline	
ccignore	optional	FEP	inline	prevent FEP from ignoring initial science frames in CC mode
compressall	optional	BEP	object	fix bug when compressing "incompressible" raw frames
condoclk	required	FEP	object inline	fix sudden changes in overclocks by conditioning the running averages
cornermean	required	BEP	inline	fix reporting of negative corner mean pixel values
corruptblock	required	BEP	inline	fix response to corrupt parameter blocks
ctireport1	optional	BEP	object	add option to report precursor charge in the outlying pixels of 5x5 timed exposure mode
		FEP	object inline	
ctireport2	optional	BEP	object	add option to report precursor charge in the low-order bits of three corner pixels in 3x3 timed exposure mode
		FEP	object inline	
deaeng	EGSE	BEP	object	run flight-like and engineering-like video boards in the engineering unit
dearepl	EGSE	BEP	object	replace video board access with dummies in the engineering unit
digestbiaserror	required	BEP	inline	fix bug in reporting bias parity plane errors
eventhist	optional	BEP	object	add event histogram mode
fepbiasparity1	obsolete	FEP	inline	save diagnostics of massive numbers of FEP bias parity errors

patch name	usage	loc	type	comment
fepbiasparity2	required	FEP	object inline	save diagnostics of massive numbers of FEP bias parity errors
feppoweroff	EGSE	BEP	object	test of BEP behavior when powering down a FEP
fepthrottle	EGSE	FEP	object inline	report 1% of event candidates found by a FEP
forcebiastrickle	EGSE	BEP	object	force bias trickling after RADMON_ENABLE
histogrammean	required	FEP	inline	fix bug in register overflow during histogram calculation
histogramvar	required	FEP	inline	fix bug in unsigned division during histogram calculation
hybrid	obsolete	BEP	object	test of a possible "hybrid" clocking mode
printswhouse	EGSE	BEP	object	print software housekeeping to user pseudopackets
reportgrade1	optional	BEP	object inline	report gradecode statistics in software housekeeping
rquad	required	BEP	inline	fix bug in corner pixel mean computation
slowpram	EGSE	BEP	object	add delays to PRAM generation steps
smtimedlookup	optional	BEP	object	replace conditional code to select FEP and BEP processing modes with lookup tables to facilitate development of new modes
squeegy	optional	BEP	object inline	add new mode to read out small areas of CCDs repeatedly within apparently normal raw frames
teignore	optional	FEP	inline	prevents FEP from ignoring initial science frames in TE mode
tlmbusy	required	BEP	object	fix bug in telemetry output when switching between output packets
tlmio	EGSE	BEP	object	write user pseudopackets into telemetry, bypassing the output queue
txings	optional	BEP	object	trigger bilevel alarm on rising threshold crossing averages
untricklebias	obsolete	BEP	object inline	absorb bias thief thread within the science task
zap1expo	required	FEP	inline	prevent overclock averages from including the first exposure frame

16. References

"ACIS Software User's Guide," MIT 36-54003, Rev. A, (NAS8-37716/DR/SDM05) July 21, 1999.

"ACIS Software IP&CL Structure Definition Notes", MIT 36-53204.0204, Rev. N, March 15, 2001.

"ACIS Software Detailed Design Specification (As-Built)," MIT 36-53200, Rev. A, (NAS8-37716/DR/SD-M03) February 3, 2000.

"DPA Hardware Specification and System Description," MIT 36-02104, Rev. C, April 15, 1997.

"Expect", D. Libes in "Tcl/Tk Extensions", ed. Mark Harrison, O'Reilly & Associates, Inc., 1997.

Gerry Kane, *MIPS RISC Architecture*, Prentice Hall, NJ, 1989.

17. Glossary

BCMD ..... ACIS command format and the script that converts it to binary packets
BEP ..... ACIS Back End Processor — the digital unit that controls ACIS.
CVS ..... Concurrent Versions System — version control for ACIS flight s/w.
D-CACHE ..... The radiation-hard data cache memory used in ACIS BEPs.
ECO ..... Engineering Change Order.
EU ..... ACIS Engineering Unit — a hardware simulator.
FEP ..... ACIS Front End Processor — one of 6 digital ACIS x-ray event filters.
I-CACHE ..... The radiation-hard instruction cache memory used in ACIS BEPs.
MIPS ..... Microprocessor without Interlocked Pipeline Stages — a.k.a. *Mongoose* CPU.
PDF ..... Portable Document Format — © Adobe, Inc.
LRCTU ..... Littlefield Remote Command and Telemetry Unit — EU interface.
SPR ..... Software Problem Report.