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: April 28th 2015
Subject: Correcting for ACIS EEPROM corruption (v 1.6)

1. Introduction

With Chandra well into its 16th year of operations, it is prudent to consider the longevity of the EEPROM devices that contain the boot code for the ACIS digital processors, since they cannot be reprogrammed or bypassed in orbit.

While designing the digital flight processor for the TESS spacecraft, our attention was drawn to the shelf life of particular EEPROMs. The data sheet on the parts used in ACIS – the Hitachi HN58C1001 – specifies a data retention period of 10 years. We assumed when building ACIS that this would not be a serious limitation, but when we recently contacted the chief parts engineer for Space Electronics, Inc., the company that sold us the parts for ACIS, he assured us that the data retention limit was real, but that the decay mechanisms are still poorly understood. While a literature search has uncovered no actual failure reports, we read that "hotter temperatures make the problem worse," although this statement also appears largely faith-based.

We conclude that the ACIS EEPROMs will degrade. Although it is 18 years since they were last written, and they currently show no sign of failing, there is no guarantee that they will continue to function indefinitely. Although we cannot predict when it will happen, a time will come when both EEPROMs will have degraded into a non-useful state; the issue in this report is to determine an operational strategy that will carry us from the start of degradation to the end of their useful life.

In this document, the names of files, uplink commands, and telemetry packets are *italicized*; the names of software functions, variables, constants, and command and telemetry fields are written in `typewriter` font.

2. Monitoring the active flight EEPROM

The contents of the 1 Mbyte EEPROM from BEP-A in the flight unit were last downlinked on March 10 2014. The first 111,128 words – the rest are filled with `0xff` bytes – contain the bootstrap loader and the code that initializes I_CACHE and D_CACHE, and these were found to match the file that was used to write the flight EEPROMs. The CVS-controlled pathname of that file is "*models/cur_bep/acisBepRom.bin.gz*". Since there is no indication that reading an EEPROM will cause it to degrade faster, its contents can and should be monitored at frequent intervals so that errors can be identified and remedial action taken, as described in Section 7 of this Report.

The `eeprom_cksum` program (see Fig. A.1 in the Appendix) computes the 32-bit cyclic redundancy checksum of the EEPROM. It is loaded into BEP I_CACHE with a single *writeBep* command, and is executed by an *execBep* (see Fig. A.2). The checksum is compared with the 'expected' value; if they match, the program waits for 10 seconds before returning; otherwise it returns immediately, in both cases reporting the actual checksum in the `returnedValue` field of a *bepExecuteReply* packet. 4 seconds after receiving the *execBep*, the BEP is sent a *readBep* command to dump all of EEPROM to telemetry. If the checksums match, `eeprom_cksum` will still be executing and the *readBep* will be rejected since the BEP's command manager will not allow the two commands to execute simultaneously; if, on the other hand, the checksums differ, `eeprom_cksum` will already have ended and *readBep* will be executed, and the contents of EEPROM will be dumped.

This rather convoluted logic is necessary since the BEP's memory dumping routine, `rdBep`, cannot be called directly from a program invoked from an *execBep* command because program and command would contend for the same telemetry buffers. These buffers, located in bulk memory, are written to the DPA's serial digital output port by the Downlink Telemetry Controller (DTC), a dedicated hardware device. Dumping EEPROM with a genuine *readBep* also allows the BEP's telemetry manager to manage the resulting *bepReadReply* packets in the normal manner, interleaving them with packets from housekeeping and active science tasks.

3. Monitoring the spare flight EEPROM

The steps needed to monitor the EEPROM belonging to the flight unit's backup BEP are shown in Table. 1. Since this necessitates halting the active BEP, there may be no need to do this while BEP-A is still useable, but recall that a BEP can be halted and, provided it remains powered up, warm-booted again without having to reload patches, so the 9 steps will proceed rapidly. Steps 4 through 7 are shown in Fig. A.2.

Table 1. Monitoring the EEPROM of BEP-B

1. Ensure that no science run is active and that no exposure or event packets remain to be written.
2. Execute "`WSPOW00000`" command to power down all FEPs and video boards.
3. Execute the "`SOP_ACIS_SWAP_BEPA_B`" procedure to switch from BEP-A to BEP-B.
4. Execute a "*writeBep*" command to copy `eeprom_cksum` to I_CACHE.
5. Execute an "*execBep*" to run the program and return the CRC-32 checksum in `returnedValue`.
6. Wait 4 seconds, after which `eeprom_cksum` will still be running if the checksums matched.
7. Execute "*readBep*" to dump EEPROM, but only if the checksum didn't match.
8. Execute "`SELECT BEP A`" (`1BSELICL` with `1BSELICL1=0`).
9. Execute the "`SOP_ACIS_WARMBOOT_HKP`" procedure to warm-boot BEP-A and restart DEA H/K.

If we find that either BEP's bootstrap loader still functions, but that the rest of its EEPROM is so damaged that the operating system won't run (or won't accept *writeBep* or *addPatch* commands), we must first dump the contents of its initialized EEPROM via "boot-via-uplink" (see Section 5), locate the damaged words, and then boot-via-uplink a second time while correcting for the EEPROM corruption. Procedures to perform these functions are described in Sections 6 and 7, below.

4. Uplink Booting

When the ACIS Back-End Processor (BEP) starts, it first executes the `__boot` procedure in EEPROM. Unless the hardware `BOOT_VIA_UPLINK` flag (a.k.a. `STAT_BOOT_MOD`) is set, `__boot` copies the remainder of initialized EEPROM into instruction memory (I_CACHE) and data memory (D_CACHE), and then jumps to `__start` in I_CACHE, to copy the default data tables from EEPROM to I_CACHE and to start the Nucleus/RTX multi-tasking operating system.

If the `BOOT_VIA_UPLINK` flag is set (by a command sent to the DPA's hardware serial interface), `__boot` bypasses the rest of EEPROM and instead copies data from the BEP's input FIFO, which buffers commands sent to the DPA's software serial interface. The first command must be *startUpload*, followed by zero or more *continueUpload* commands. Subfields in *startUpload* define the starting `loadAddress`, `totalCount` (the total length in words, including those in subsequent *continueUpload* commands), and `executeAddress`. If the bootstrap loader expects *continueUpload* but receives *startUpload* instead, it terminates the previous load – leaving the data in place – and begins the new load at the new `loadAddress`. In this way, a series of 'partial' commands can initialize separate memory segments, *i.e.*, I_CACHE, D_CACHE, and bulk memory. As soon as `totalCount` is satisfied, the loader jumps to the most recent `executeAddress`.

The portion of the `__boot` procedure that is essential for uplink booting consists of 174 instruction words, about a third of which are `nop`s containing all zeroes. The remainder represent 0.1% of the initialized EEPROM, and the degradation, which turns ones into zeroes, is most likely to start in the other 99.9%. While it is still possible to boot via uplink, we shall be able to dump the EEPROM contents and reboot the BEP from the damaged EEPROM, patching the corruption as we go, as described in Sections 6 and 7.

5. Dumping EEPROM via uplink

Dumping the EEPROM from a procedure executing alone in bulk memory is quite straightforward because there are no conflicting processes trying to use the DTC and we can therefore run with DTC interrupts

turned off. The `eeprom_dump` procedure, listed in Fig. A.4 of the appendix, establishes a single telemetry buffer within bulk memory and uses it repeatedly. Once the last packet is written, the procedure returns control to the bootstrap loader. If `BOOT_VIA_UPLINK` is still asserted, it waits for another *startUpload* command; otherwise, it reboots the BEP in 'cold' or 'warm' mode, according to the state of the `STAT_WARM_MOD` flag.

6. Booting a damaged EEPROM via uplink

Uplinking the entire contents of initialized EEPROM in a series of *startUpload* and *continueUpload* commands would take a very long time. Instead, assuming that the contents of the EEPROM are already known, either from either `eeprom_cksum` or `eeprom_dump`, and the damage is confined to a limited number of words, all that needs to be uploaded is a routine, `eeprom_patch` (see Fig. A.5), that copies the initialized EEPROM contents to I_CACHE and D_CACHE, followed by a series of updates to what was copied, followed by a jump to *_start* to initialize the BEP's operating system.

7. Source Code

The following components are saved in the ACIS CVS repository under "*patches/eeprom_patch*".

Name	Rev	Description
acis-eeprom-utils.pages	1.5	Apple Pages™ source for the current document
eeprom_cksum.bcmd	1.1	Load *eeprom_cksum* into iCache, execute it, wait 4 seconds, then dump EEPROM
eeprom_cksum.c	1.3	Compute EEPROM checksum and wait 10 seconds if it matches the argument
eeprom_dump.bcmd	1.1	Load and execute *eeprom_dump* from uplink
eeprom_dump.c	1.3	Run in bulk memory and dump EEPROM contents
eeprom_patch.bcmd	1.1	Load and execute *eeprom_patch* from uplink
eeprom_patch.c	1.4	Run in bulk memory, copy and patch EEPROM, then reboot the BEP
eeprom.h	1.3	C header file for load-from-uplink programs
make-bcmd.pl	1.3	Perl script to convert assembler listing into load-from-uplink command packets
Makefile	1.8	*make* script to compile the C programs and convert them to *bcmd* format

These programs are described in detail in the Appendix. Before running the *make* command to recreate the *bcmd* files, be sure to include the directory containing the MIPS cross-compiler in `$PATH` and the directory containing its runtime libraries in `$LD_LIBRARY_PATH`, *i.e.*, "`$CVSROOT`/../`$ARCH`/*bin*" and "`$CVSROOT`/../`$ARCH`/*lib*", respectively.

8. References

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

"Microcircuit, CMOS, 1 Megabit, electrically erasable Programmable Read-Only Memory (EEPROM)," MIT 36-02306.

"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/SDM03) February 3, 2000.

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

Section 22.4, "Cyclic Redundancy and Other Checksums", W. Press *et al.*, in *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 3rd edition, 2007.

9. Glossary

BCMD ..... ACIS command format and the script that converts them to binary packets
BEP ..... ACIS Back End Processor — the digital unit that controls ACIS.
CRC ..... Cyclic Redundancy Check(sum)
CVS ..... Concurrent Versions System — version control for ACIS flight s/w.
D_CACHE ..... The radiation-hard data cache memory used in ACIS BEPs.
DEA ..... ACIS Detector Electronics Assembly — CCD sequencers and digital converters.
DPA ..... ACIS Digital Processor Assembly — containing 6 FEPs and 2 BEPs.
DTC ..... Downlink Telemetry Controller (external DMA for serial BEP output)
EEPROM ..... Electrically-Erasable Programmable Read-Only Memory.
FIFO ..... The hardware interface between DPA software serial commands and the BEP.
I_CACHE ..... The radiation-hard instruction cache memory used in ACIS BEPs.
MIPS ..... Microprocessor without Interlocked Pipeline Stages — a.k.a. *Mongoose* CPU.
Nucleus/RTX ..... Commercial multitasking operating system running in the BEP.

Appendix – Source code

A.1 The eeprom_cksum program

The following C function returns the 32-bit cyclic redundancy checksum of each 32-bit word between its `from` and `to` arguments. Note the convention used throughout this appendix that the '`to`' address is that of the first word *beyond* the end of the block to be summed. The third argument is the expected value of the checksum. The remaining arguments cause the program to wait for `0.1*ticks` seconds if the checksums match. Since `eeprom_cksum` executes within the `MemoryServer` task, the appropriate C++ code to execute a timed wait would be `"memoryServer.sleep(ticks)"` which is implemented in C code by passing two arguments to the BEP's `Task::sleep()` routine: the address of the `memoryServer` object and the `ticks` value.

Figure A.1. eeprom_cksum.c – return checksum of initialized EEPROM

```
#define  CRC32_POLY  0xedb88320  /* CRC-32 polynomial generator */

unsigned eeprom_cksum(
   unsigned *from,               /* Address of first word in the block */
   unsigned *to                  /* Address of first word after end of block */
   unsigned cksum,               /* Expected 32-bit CRC checksum of block */
   unsigned *task,               /* Address of the memoryServer object */
   unsigned (*sleep)(),          /* Address of the Task::sleep routine */
   unsigned ticks                /* Number of 0.1 second intervals to wait */
) {
   unsigned table[256];          /* table[] allocated on stack */
   unsigned crc = ~0;            /* CRC-32 checksum */
   unsigned *dp, ii, jj;         /* scratch */

   /* Construct the CRC look-up table */
   for (dp = table; dp < table+256; *dp++ = jj) {
      for (ii = 0, jj = dp-table; ii < 8; ii++) {
         jj = (jj & 1) ? ((jj >> 1) ^ CRC32_POLY) : (jj >> 1);
      }
   }
   /* Calculate the CRC-32 checksum of the data array */
   for (dp = addr(from); dp < to; dp++) {
      for (ii = 0, jj = *dp; ii < 4; ii++, jj >>= 8) {
         crc = (crc >> 8) ^ table[(crc ^ jj) & 0xff];
      }
   }
   crc = ~crc;

   /* Wait 0.1*ticks seconds if the CRC matches cksum in the argument list */
   if (crc == cksum) {
      sleep(task, ticks);
   }

   /* Return the computed checksum value */
   return crc;
}
```

This C program is compiled by *acis-gcc*, the MIPS cross-compiler, with the `-Wa,-alh` flags, creating a pseudo-assembler listing. The *make-bcmd.pl* script (see Section A.7) then converts the listing into one or more *writeBep* commands which are fed to *bcmd* to create the command packets that load the program into an unused block of I_CACHE. Currently (at patch level FGH), addresses above `0x800c85d0` are available. The expected value for the flight EEPROM checksum is `0x8e9fdcc0` and the addresses of the `memoryServer` object in D_CACHE and the `Task::sleep()` routine in I_CACHE are read from the BEP load map. The full command sequence is shown in Fig. A.2.

The CRC values for the EEPROMs in BEP-A and BEP-B of the engineering unit are `0x08602ea3` and `0x56374f8d`, respectively. They differ from the flight EEPROMs because they contain test data recorded at addresses `0xbfc6c860` and above, which is filled with `0xffffffff` in the flight EEPROMs. Note however that the microboot blocks from `0xbfcffff0` through `0xbfcfffff` are the same in all the EEPROMs.

A.2 Executing eeprom_cksum

The command sequence is shown below. After a *writeBep* command to copy the *eeprom_cksum* code into I_CACHE, the program is started with an *execBep* command and is passed 6 parameters (see Fig. A.2.) Then, after a 4 second wait, a *readBep* command tells the BEP to dump its EEPROM contents, but this will be rejected if `eeprom_cksum` is still executing, *i.e.*, if the checksums match.

Figure A.2. bcmd commands to load and execute the eeprom_cksum program

```
# Load the program into an unused part of I_CACHE
write 1001 0x800c85d0 {
    0x27bdfbe8 0xafb00410 0x2410ffff 0x27aa0010 0x27a30410 0x8fae0428 0x00000000
    ...
}
wait 4
# Execute the eeprom_cksum program
exec 1002 0x800c85d0 {
    0xbfc00000      # address of first word of EEPROM
    0xbfd00000      # address of first word beyond end of EEPROM
    0x8e9fdcc0      # expected CRC checksum value
    0x80004c04      # address of memoryServer object in D_CACHE
    0x800872d4      # address of Task::sleep() method in I_CACHE
    100             # sleep time in units of 0.1 seconds
}
wait 4
# Try to dump EEPROM, but fail if the previous command is still executing
read 1003 0xbfc00000 262144
```

A.3 Boot-via-uplink programs

The remaining programs interact with the BEP hardware in several ways. The *eeprom.h* file (see Fig. A.3) defines these hardware dependencies, *e.g.*, the addresses of memory-mapped hardware registers and their various sub-fields and masks, and other addresses extracted from the load map of version 11 of the flight software, *i.e.*, the version burned into the EEPROMs. *eeprom.h* also specifies the structure of *bepReadReply* telemetry packets and defines the following macros:

`val(addr)`	Load from, or store, into `addr`, an unsigned integer representing the address of a memory-mapped hardware register. The result is given the `volatile` attribute to prevent the compiler from trying to optimize access to that location as if it were a local variable.
`addr(value)`	Convert `value`, typically an unsigned integer, into a 32-bit address.
`write_icache(addr,value)`	Load `value` into I_CACHE at address `addr`. It is permitted for both `value` and `addr` to have side effects, *e.g.,* `write_icache`(`to`++, *`from`++) increments the `to` and `from` pointers once each.
`wait(n)`	Reset the watchdog counter; then loop for `n` iterations, where `n` should be chosen so that the execution time is ~0.4 milliseconds, thereby guaranteeing that the end of a DTC transfer will be intercepted before the hardware inserts any `0xb7` fill bytes into the output stream.

Figure A.3. eeprom.h – common values and macros for the EEPROM programs in this appendix

```
#include "filesboot/mips.h"
#include "filesboot/bep.h"
#include "filesboot/mongoose.h"
/* Hardware addresses */
#define MMAP_DTCSTART   0xa0180018    /* DTC start register address */
#define MMAP_DTCEND     0xa018001c    /* DTC end register address */
#define EEPROM_START    0xbfc00000    /* Address of first word in EEPROM */
#define EEPROM_END      0xbfd00000    /* 1st word after end of EEPROM */
#define _loadRom        0xbfc0b780    /* Start of EEPROM load */
#define _ftext          0x80080400    /* Start of initialized I_CACHE */
#define _etext          0x800c0970    /* 1st word after initialized I_CACHE */
#define _fdata          0x80000000    /* Start of initialized D_CACHE */
#define _edata          0x80020b70    /* 1st word after initialized D_CACHE */
#define __start         0x80080400    /* System starting address */
#define PACKET_ADDR     0xa0004000    /* Packet buffer address */
#define PATCH_TABLE     0xa0003000    /* Patch table address */
#define TIMER_ADDR      0x800bc870    /* Nucleus RTX timer routine */
/* Interrupt masks and register bits */
#define INTR_DISABLE    0x10000015    /* Disable DTC interrupts */
#define CNTL_DNLKENB (   1 << 1)      /* DTC enable in control register */
#define STAT_DNLKINTR (  1 << 5)      /* DTC interrupt in status register */
#define PULS_DNLKCLR (   1 << 1)      /* DTC interrupt clear in pulse register */
#define val(a)          *(volatile unsigned *)(a)
#define addr(a) (        unsigned *)(a)
#define write_icache(v,a) {\
                           val(ICACHE_DATA_REG) = (unsigned)(v);\
                           asm volatile ("" : :);\
                           val(ICACHE_ADDR_REG) = ((unsigned)(a)\
                              & ADDR_BOUND_MASK) | ICACHEADDR_W;\
                }
#define wait(n)         {\
                           volatile int ii=(n);\
                           val(WATCHDOG)=0xffffffff;\
                           while (--ii >= 0);\
                }
/* Structure of a readBepReply telemetry packet */
typedef struct {
        unsigned p_sync;              /* Packet synch word */
        unsigned p_hdr;               /* Type, length and id */
        unsigned p_cmdid;             /* ID of execBep command */
        unsigned p_ticks;             /* BEP interrupt counter */
        unsigned *p_origin;           /* Block starting address */
        unsigned p_count;             /* Block length in words */
        unsigned *p_addr;             /* Packet starting address */
        unsigned p_data[1016];        /* Packet data content */
} PKT;
```

Note that writing to I_CACHE via the `write_icache` macro is a two-step procedure: first write the value to the hardware register mapped into BEP memory at `ICACHE_DATA_REG`, and then write the I_CACHE address to another hardware register at `ICACHE_ADDR_REG`. Both write operations take three machine cycles, so we must prevent the C compiler from optimizing the program, putting the write instructions too close together. This is done by sandwiching them either side of an '`asm volatile ("", : :)`' directive, which causes the GNU compiler to add an extra `nop` instruction. Similarly, the loop index in the `wait` macro is given the `volatile` attribute to prevent the compiler from 'optimizing' the `while` loop out of existence.

A.4 The eeprom_dump program

This program (see Fig. A.4) reports the contents of EEPROM as *bepReadReply* telemetry packets. It is copied into bulk memory and executed by the bootstrap loader in response to a boot-via-uplink command. It cannot be passed any arguments, and its execution stack is small (1024 words), but by way of compensation, we can safely assume that the DTC isn't being used by any other code threads. Once the last packet has been written, `eeprom_dump` returns control to the bootstrap loader. If the `BOOT_VIA_UPLINK` flag had been cleared by a DPA hardware command while the packets were being written, the loader will attempt to reboot the BEP from the EEPROM image; otherwise, it will expect to read more boot-via-uplink commands.

The `wait` macro value (`77`) in `eeprom_dump` was chosen so as to cause a wait of 0.4 millisecond, which is sufficiently short to guarantee that the DTC doesn't insert fill bytes (`0xb7`) between packets. The `bepTick-Counter` field in each *bepReadReply* packet is the number of waits divided by 256, which approximates the 0.1 second ticks of the BEP interrupt timer, which is unavailable while booting via uplink.

Figure A.4. eeprom_dump.c – boot via uplink program to dump EEPROM to telemetry

```
#include "eeprom.h"
void eeprom_dump(void)
{
     const unsigned *from = addr(EEPROM_START);     /* Start of region to dump */
     const unsigned *to = addr(EEPROM_END);         /* After region to dump */
     unsigned mask = INTR_DISABLE;                  /* Disable DTC interrupts */
     unsigned ticks, npkt, *dp;                     /* Counters and data pointer */
     /* Initialize the packet header */
     PKT *pkt = (PKT *)PACKET_ADDR;
     pkt->p_sync = 0x736f4166;              /* Store packet synch word */
     pkt->p_cmdid = 1;                      /* Store ID of startUpload command */
     pkt->p_origin = from;                  /* Store address of start of block */
     pkt->p_count = to-from;                /* Store length of block in words */
     /* Turn off interrupts and clear the DTC */
     asm volatile ("mtc0 %0,$12" : : "d" (mask));
     val(BEP_CTRL_REG) &= ~CNTL_DNLKENB;
     /* Write the packets */
     for (npkt = ticks = 0; from < to; npkt++) {
          int len = (to-from) > 1016 ? 1016 : (to-from);
          /* Fill the packet header */
          pkt->p_hdr = (npkt << 16) | (len + 0x407);
          pkt->p_ticks = ticks >> 8;
          pkt->p_addr = from;
          /* Copy packet data */
          for (dp = pkt->p_data-1; --len >= 0; *++dp = *from++) {
               ;
          }
          /* Start the DTC transfer */
          val(MMAP_DTCSTART) = (unsigned)pkt;
          val(MMAP_DTCEND) = (unsigned)dp;
          val(BEP_CTRL_REG) |= CNTL_DNLKENB;
          /* Wait until DTC becomes idle */
          while ((val(BEP_CTRL_REG) & CNTL_DNLKENB) && ++ticks) {
               wait(77);
          }
          /* Clear the DTC interrupt */
          val(BEP_PULS_REG) = PULS_DNLKCLR;
     }
}
```

A.5 The eeprom_patch program

This program (Fig. A.5) is also run in boot-via-uplink mode. It initializes I_CACHE and D_CACHE from the EEPROM in the same manner as the bootstrap loader itself, then applies a series of patches derived by comparing previous dumps with the correct EEPROM contents, and finally jumps to the `__start` routine in I_CACHE to continue BEP initialization.

The patches are defined by a table that must be loaded into the `PATCH_TABLE` address in bulk memory (defined in *eeprom.h*) before `eeprom_patch` starts. The first word in the patch table specifies the number of pairs of words that follows. The first word in each pair defines the address in I_CACHE or D_CACHE of a word that is to be replaced, and the second word defines its replacement value. Note that the addresses, which must begin on a word boundary, are *not* those of the corrupted locations in EEPROM; they are the addresses to which the corrupted words have been copied.

If `eeprom_patch` is successful, the BEP will attempt to reboot and, if successful, write a *bepStartupMessage* packet. If the `WARMBOOT` flag was set and I_CACHE had retained a set of valid patches from an earlier time, those patches will be applied. The `BOOT_VIA_UPLINK` flag will still be set.

Figure A.5. eeprom_patch.c – boot via uplink program to load from EEPROM & patch corrupted words

```
#include "eeprom.h"
void eeprom_patch(void)
{
   const unsigned *from = addr(_loadRom);
   const unsigned *dat = addr(PATCH_TABLE);
   /* Copy to I_CACHE */
   unsigned *to = addr(_ftext);
   while (to < addr(_etext)) {
      write_icache(to++, *from++);
   }
   /* Copy to D_CACHE */
   to = addr(_fdata);
   while (to < addr(_edata)) {
      *to++ = *from++;
   }
   /* Execute one or more patch() macros, e.g.,... */
   for (npatch = *dat++; npatch--; dat += 2) {
      if (*dat < I_CACHE_LO || *dat > I_CACHE_HI) {
         val(*dat) = dat[1];
      } else {
         write_icache(*dat, dat[1]);
     }
   }
   /* Jump to __start to initialize RTX */
   to = addr(__start);
   asm volatile ("jr %0" : : "d" (to));
}
```

A.6 An example of patching via uplink

Fig. A.6 shows a typical command sequence using `eeprom_patch`. The first *start* command loads the patch count and the (`address,value`) pairs into `PATCH_TABLE`. Note the large `totalCount` field (`0x1000000`), guaranteeing that the loader won't start execution until the second *start* command has loaded the program.

Figure A.6. bcmd commands to load and execute eeprom_patch via uplink boot

```
# Restart the BEP in boot-via-uplink mode
halt bep
set bootmodifier on
run bep
wait 1
# Load the patch table into bulk memory; then read more boot-via-uplink commands
start 0 uplink 0xa0003000 0x1000000 0 {
     2                          /* Number of (addr,value) pairs to follow */
     0x800e30f0 0x0000ffff      /* Replace a word in I_CACHE */
     0x8000130c 0x00000005      /* Replace a word in D_CACHE */
}
wait 1
# Load and execute the eeprom_patch program
start 1 uplink 0xa0000000 84 0xa0000000 {
     0x3c04bfc0 0x3484b780 0x3c06a000 0x34c63000 0x3c038008 0x34630400
     ...
}
wait 1
# Turn off the boot-via-uplink flag (optional)
set bootmodifier off
```

A.7 The make-bcmd.pl script

Since the programs described in this document are (intentionally) written as single routines without separate data segments or long jumps (*i.e.,* instructions with 26-bit address fields that must be updated by the MIPS linker) the assembler listings can be converted into *writeBep* commands or, in the case of boot-via-uplink programs, into *startUpload* and zero or more *continueUpload* commands. Fig. A.7 shows a Perl script that can translate a listing file into *bcmd* commands.

Figure A.7. make-bcmd.pl – convert compiler listing to writeBep or startUpload/continueUpload commands

```
#! /usr/bin/env perl
#
# Usage: make-bcmd.pl [-u] <load-address> <assembler-listing-file>
$up = shift(@ARGV) if $ARGV[0] eq '-u';   # boot via uplink if -u specified
$load = shift(@ARGV);                     # load and execute address
$addr = eval $load;                       # convert $load to decimal
$nmax = $up ? 122 : 125;                  # maximum data words in first command
@fmt = ("continue %d uplink", "write %d 0x%08x");

while (<>) {
   next unless /^\s*\d+ [0-9a-fA-F ]{4} ([0-9a-fA-F]{8}) /;
   $txt .= sprintf("0x%08x\n", unpack("V", pack("N", hex($1))));
   next if ++$words && --$nmax;
   $nmax = 125;                           # max data words in remaining packets
   $txt .= sprintf ("}\nwait 1\n$fmt[! $up] {\n", ++$ncmd, $addr += 500);
}
# write command packet(s)
print $up ? "start 0 uplink $load $words" : "write 0";
print " $load {\n$txt}\n";
exit 0;
```