CSUN CUBESAT

April 2015

J. Fowler's Guide To:

csTelemetry, csReFlashProgramImage, csMemTest, csProgramImageCheck, csFlashOP, csJournal,

& selections from csCommandParser

(Questions: yrolg4tseuq@sbcglobal.net)

csTelemetry

When ground station sends the command for getTelemetry, this is where most of the action takes place. It's necessary to note that the behavior of "getTelemetyString" is greatly affected by the outside influence of the getTelemtry function handlers in commandParser, namely on_GET_TELEMETRY and more_GET_TELEMETRY. As of not too long ago, getTelemetry was upgraded to function in a streaming state. This change, along with bit-packing, made for a very complex function.

Because the streaming state will require multiple calls to getTelemety there is the need to carry over some important information with every call. The structure is composed of the following fields:

Global.h :: Global->csLink->csStreamState->csGetTelemetryStream

`UINT32 epochStart`	Ground's original start request time
UINT32 epochEnd	Ground's original end request time
UINT32 epochStartCur	Current call to getTelemetry start time
UINT32 epoechEndCur	Current call to getTelemetry end time
UINT8 offset	Backup of the requested sensor to pull data for.
UINT32 fileOffset	Stored after the first call to getTelemetry. Copies the last position read from file, so that next time we don't have to re-search for the appropriate start location.
UINT8 useFileOffset	Flag which indicates, if set true, that we have already been through once and want to use the fileOffset value. It will also be used as a general indicator for other checks.
UINT8 overFlow	Indicates if the number of bits read could not fit into the write buffer.
UINT8 num_4Bits	In relation to the overFlow variable, this value will either be 1 or 2, because we will only ever be 4 bits or 8 bits over the buffer limit.
UINT8 bufferBits	Stores the last 8 or 4 bits of read telemetry. This is essentially the overflow buffer. If there was an overflow, these bits will be tacked onto the beginning of the next clean buffer we get when getTelemetry gets called again in the streaming state.

CScommandParser.c::on_GET_TELEMETRY

Very straight forward. Start and end epoch times, and sensor offset are parsed out and saved. Values in the csGetTelemetryStream are reset. (Probably even a better idea considering it's part of a union of structures!) The call is made to getTelemetryString, error codes handled, remaining bytes updated, and return. The buffer which will be written to is the commandParserbuffer, now located in radio code and referred to as the radio buffer. The buffer size that radio gives us varies and is complicated by itself, but for the first call to getTelemetry we are guaranteed to have a buffer size of 1999 bytes to fill. This works out nicely because we have 4 bytes for the returned epoch time, and 1995 bytes is a multiple of 3. The packed nature of the bits is 12 bits for every telemetry reading, which is 1.5 bytes, hence, with a multiple of 3 we have no overflow!

CScommandParser.c::more_GET_TELEMETRY

This function will be called the second and every single time after the first initial call which is handled by on_GET_TELMETRY. This function is essentially the same as the last, except it as to deal with buffer overflows, buffer offset starts, etc. If the csGetTelemetryStream overFLow flag is set, it determines if there are 4 or 8 bits left over from the last read. These bits are written to the radio buffer and the appropriate flags are set which well influence getTelemetryStrings input arguments. The number of telemetry readings is carefully calculated, taking into account the number of bits, if any, already written to the buffer. The call is made to getTelemetryString, and procedure afterwards is not complicated.

CStelemetry::getTelemetryString

The general idea is to open the correct .TEL file on the SD card, find where to start, read the values, bit-pack them, and store them into an output buffer. The .TEL files are composed of the following format: { epochTime (4 bytes) | sensorTelemetryValue (2 bytes each for every sensor) }. A single telemetry value on the SD card is stored in 2 Bytes, but only 12bits of the 16bits are significant. A new .TEL file is created every day, and named by the day-epoch-time, that is, number of days since January 1, 2014. This isn't handled in getTelemetry, but I think it's good for the reader to know.

The return value of getTelemetryString needs its own explanation. The INT32 represents both error/notification codes, and the number of bytes read. (Looking back, it probably would have been easier to make it into a struct, but it works just fine as it is.) Bytes read are placed in the most significant 16 bits, and error status bits are set in the lower 16. If a critical error occurs, the return value will only have that critical error value, which is a negative number.

If you take a look at the .h file, you'll notice a series of error codes. The negative valued ones are critical erros which will cause getTelemetryString to exit immediately. The other errors, may cause the function to end, but may also indicate that there was some data read and written successfully. Some of them, such as NOTICE_SECOND_FILE, are only there to give the user an indication of what the function is doing and if it's behaving properly.

Alright, now for the meat of all this.

A series of preliminary checks are made, such as if the time interval is positive or negative. A negative time interval is a critical error. The appropriate file is opened and a check is made to see if the file size mod the number of bytes written in a telemetry SD write is equal to 0. If it isn't, this indicates corruption in the file and possibly bad data. This is a critical error. If we are returning in the streaming state and know where to start, we seek to that location, else, we attempt to find the right starting location in the file. Because the file may contain large gaps, doubles, etc., it isn't possible to seek to the right location based off a simple math operation, nor is it very effective to search linearly. Instead, a binary search O(lg2(n)) is made until the closest starting position is found. Another file integrity test takes place in this algorithm, in that if an epoch time of less value than an epoch time located earlier in the file is found, a critical error is returned.

The next step is critical and sets the mood for the rest of the function. The first group of telemetry values are read from the SD card. If this is our first call to getTelemetry, we simply place the epoch time and bit-pack the telemetry values and place them into the buffer. If it isn't the first call and we have something like 4 bits already written to the buffer, we take this into consideration and modify a special variable called "index" which is what controls the number of reads from the SD card. We ONLY write the epoch time to the output buffer on the first call.

The special variable "index" not only keeps the read loop bounded, but also serves as an indicator to which bytes need to be bit-shifted. If index is even, the next right location is flush, and if odd, the next write location has already been populated with 4 bits. You can now get an idea to what modifications were made earlier if the flag was set for 4 bits already having been written to the buffer in more_GET_TELEMETRY.

If we fail to read from the SD card, we assume that we have hit the end of a file and attempt to open the next day's file. This works in such a way that we literally could, if ground requested it, read several days-worth of telemetry at a time.

Gaps and doubles in epoch times are handled nicely. If a gap occurs, we write a filler value to the output buffer. 0xFFF. (To avoid mistaking this with values that really are 0xFFF, we changed those real values to 0xFFE.) On a double, the

value is simply ignored, we back up the index, and go back to the top of the iteration loop. If we encounter an epoch time that is less than a previous epoch time, that is a critical error and we exit immediately.

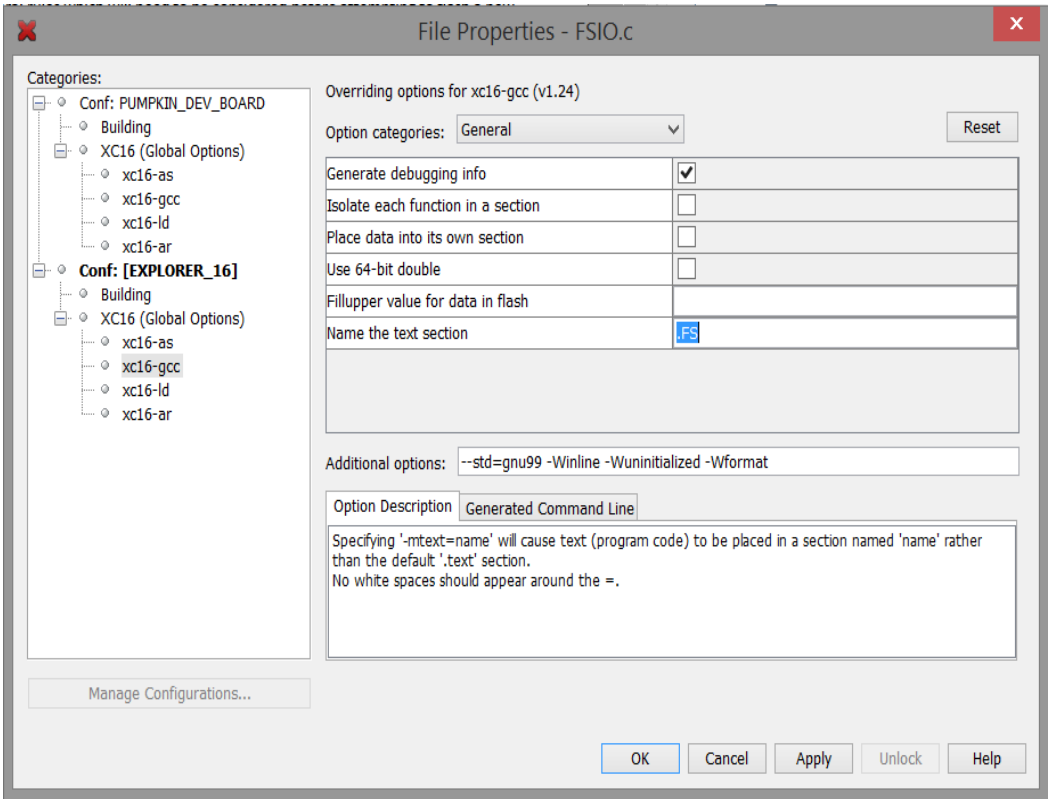
On the last iteration we determine if we will have a buffer overflow. We write what we can to fill the output buffer, and store the remaining bits into the bufferBits variable, setting the appropriate flags as well.

csReFlashProgramImage

We can literally re-flash the satellite while it is running with the same program image it's currently running, or with a completely new and different one! There are limitations, however, which I will discuss in this document. Images are stored and read from the SD card. The image is read 1 flash page at a time, which comes out to 2048 bytes, if each 24 bit instruction is stored in a UINT32. Juan was nice enough to let me increase radio buffer length to this size to accommodate. Upon a successful re-flash, the function will cause the satellite to reset (Ground should not expect to hear an "ok" response). The entire process seems to only take around 10-20 seconds, plus satellite startup initialization time, if successful.

The function "illBeBack" was carefully designed to have limited contact with the rest of flash memory, for obvious reasons. It cannot very well call an external function to do something if that function is in a section of flash memory that has just been erased. The function has been given its own page in flash memory starting at address 0x2A400, as to not erase itself in the process. It's not that simple, however, because we need to read from the SD card. This means that we require external calls to the file system. Not only do these functions need to exist, but they need to be at a constant calling address known to IllBeBack, hence, the entire fileSystem library needed to be placed in its own section of flash memory and cannot be re-flashed. But wait! The file system also calls external functions, such as those functions in the c library, so those need their own page too.

Placing functions into their own pages in flash memory was done via inline linker messages (__attribute__ ((address(…)))), but most of it was done by modifying the linker script itself. Different sections were declared (program, library, fileSys), and filtering rules were placed which place different files into different sections of flash memory based upon "text" assignment. Text assignment is done by right clicking on a .c file, select properties. You may have to override build options. You should get something like what is shown in this screen shot. I was able to name every file under the ".CubeSat" name by going to the project properties. I then changed individual files. There is a separate filter in the linker script which filters in libraries, which are given their own space in flash memory.



With a call to re-flash the program image, only addresses 0x00000 through 0x1FFFE, inclusive, will be erased and written to.

General-Flash Memory Partitions:

Start Address	End Address	Partition Type
0x00000	0x1FFFE	General Code. Only this gets Re-Flashed, including interrupt vector table!
0x20000	0x22FFE	Libraries
0x23000	0x2A3FE	File System
0x2A400	0x2A7FE	Re-Flash Program Image & output ROM Image
0x2A800	0x2ABFE	Journal Summary

RULES FOR NEW PROGRAM IMAGES:

1. Thou shalt not modify the flash location of the File System Library
2. Thou shalt not modify the flash location of the Re-Flash Program Image Function
3. Thou shalt take extreme precaution upon adding new libraries to the code base. (Possible, but dangerous)
4. Thou shalt not modify library functions used by the File System, nor their content, nor their flash position.
5. Thou shalt take extreme caution. You most likely will not have a second chance.

Better understanding of the assembly can be found in my section on FlashOP.

csMemTest

This is the volatile memory check. The ram is diagnosed via a series of bit-flip tests to determine if the bits can be set properly. Various error codes are available which indicate if an error occurred in the data section or the stack. The address bounds of these locations should be updated before the satellite launches, and should be checked with every new future image to be flashed to the satellite.

A sanity check is performed first. This checks the direct values which the function will use to store and compare values taken from the ram. If these fail the bit-flip tests, the function returns an error code.

If the to be used variables are found sane, we simply start at the start of the data section and make our way though. Each iteration of the loop we test if the current address is the same as one of the section of variables we are working with (those that were already tested for sanity at the beginning), and if it is the same, we skip over the length of them. The bit-flip tests are done, and the original value is restored.

If a single error occurs, a bit map will be returned to ground which indicates where the errors have occurred. Because of the length of volatile memory, each bit returned represents 8 memory addresses in memory. So, say you have 64 memory addresses and errors occurred in addresses 3 and 42, then the map would look something like this: 10000100

I've found this function to run in a matter of seconds, perhaps 1-5 seconds.

csProgramImageCheck

This function calculates the bit-sum of the Flash Memory, excluding the journal portion. (0x00000 -> 0x2A7FE) The only challenge associated with this code was the needed assembly language, as has been the common theme for most of the functions I've written. Further information on the assembly language aspect can be found in my explanation of FlashOP.

csFlashOP

Any work we do with the flash memory will require the use of assembly language. I essentially wrote a flash operation API which can be easily used by other functions to read, erase, and write to flash memory.

First, a little bit about flash memory. The flash is partitioned out into flash pages. Each page in this particular flash memory, is composed of 512 instructions, where each instruction is 24 bits in length. The addresses increment by 2, so every new flash page will be an increment of 0x400. (Example: Page 0 = 0x00000 -> 0x003FE, and Page 1 = 0x00400 -> 0x007FE) Each page in flash memory is composed of several row, in this case, each row consists of 64 instructions making for a total of 8 rows in a page. To erase flash memory, you must erase an entire page at a time. When erased, the bits are all set to 1's. You can write to flash without first erasing it, but you can only set 1's to 0 and not vice-versa, hence, it's normally always a good idea to erase a page before writing to it.

Special variables are used throughout the flash operation code. They hold memory addresses which indicate in the assembly portion what table page and address to locate. (More about that later.) To access them in the assembly portion, they need to be in Global Space. I currently have them in the .h file as static variables. (This is subject to change.) When accessed in assembly, they are proceeded by an underscore. So, flashHiAddr would be named _flashHiAddr in the assembly portion.

In each of the operations we first need to locate where we want to read/write/erase from. We first need to load the right table page. The table page "TBLPAG" is another partition group which partitions flash pages into tables. The table page is identified by the most significant 8 bits of a flash address. (The addresses are 24 bits in length.) Since we only work in addresses 0 to 0x02AC00, the table pages will only range from 0 – 2.

The instructions TBLRDL and TBLRDH read the low 16bit word and the high 8bit byte of an instruction respectively. There are two variations of read functions. One just reads the lower 16bits (used for journal operations), and the other will read the entire 24bit instruction.

Both write and erase require a special sequence of instructions to be carried out before access is given to fulfill the instruction. This is a kind of safeguard I guess. This operation takes place on the NVMKEY register. The NVCOM register contains bit fields indicating what action to perform on the flash memory. So, we would in general set the NVCOM register up to do what we want it to do, unlock the "Start operation" bit in NVCOM by doing the sequence upon NVMKEY, and then setting the "start operation" bit.

Erasing a page of flash with the above procedure is easy enough. Writing to flash is more complicated. Writes are done before hand to a "latch". Then the operations, as stated above, take place and the stored values are written to flash.

If you want more detailed information than I suggest looking in the microchip manual. Suffice to say, these operations have been unit tested and tested on the satellite. They have so far withstood the test of time.

csJournal

There are two types of journals. One is located on the last page of flash memory. The other is a collection of files which will be located on the SD card, which are updated with notifications of specific events. I will be explaining the details of the first.

The journal is stored on flash memory so that the written values will be retained after a satellite reset. It's currently only about 70 bytes in length, but is subject to change as the journal content is finalized. It is given its own page in flash memory because of the need to erase an entire page of flash before re-writing to it.

The best way to work with the journal is via the Journal_SetStruct and Journal_GetStruct functions. Both of these functions rely heavily upon flash operations as found in csFlashOP, in particular,

readProgramMemoryLoWord and writeProgramMemroyLoWord. As stated by these names, we are only reading/writing to the lower 16 bits of the 24 bit instruction. If you wish to update a journal value, you should call the Journal_GetStruct function, modify the returned journal structure, and then write it via the Journal_SetStruct. This function will take care of erasing the flash page, writing the journal, and calculating and writing the checksum. The checksum is located at the last 2 bytes of flash memory.

The size of the journal structure should be divisible by 16bits. Most of the functions have assumed this to be the case. This was done because, at the time, the total length of the journal was unknown. It was also convenient to do so because we are using 16bits of each instruction in flash memory.

csCommandParser – Selections

The commands located in this file are for the most part the outer shell of functionality. They normally call a separate function located somewhere else and handle the return values of said function. Each command is passed values from radio-land, including a link_command_t which contains a subfield called 'msg' which is the arguments from ground. The other passed in structure is called link_response_t, which provided a pointer to a buffer location inside the commandParserBuffer. I should note that the base start of 'response_buffer' is on an odd address. This can cause strange anomalies if you attempt to convert it to say a UINT16* and then try to write values to it.

Here is a list of the commands I worked on in this file. I will be extremely brief for most of them because most of the functionality exists within the functions which they call.

on_GET_TELEMETRY- GetTlemetry is a streaming command. Thus, this function is called on the first pass only. The global structure which handles streaming commands is updated with information passed from ground. Basic checks are made on the validity of the passed in values. getTelemtryString is called, and the results are evaluated. The retuned value is parsed out because it is the combination of both bytes written and error response. The remaining number of bytes to be read are calculated and sent back to readio, along with the number of bytes read this pass. We are expected to fill a total of 1999 bytes this pass, with the 1 extra byte being the error response byte, thus making an even 2000. (See csGetTelemetry for more information.)

more_GET_TELEMETRY- This function behaves just like on_GET_TELEMETRY, except that it also deals with buffer overflows and left-over bits. After the first pass of getTelemetry, this function will be called for the remaining duration of the getTelemetry command. If bits from the last call are left over, they are written to the return buffer and flags are set which will indicate to getTelemetryString the condition of the passed in write buffer. After that point, it just follows suit with on_GET_TELMETRY. There is to be no assumed amount of bytes to be read for each call. Radio passes us a value which indicates the number of bytes it wants us to write to the buffer. This is done because radio may be attempting to send a packet in the middle of the buffer and doesn't want us to overwrite it.

on_START_SEQUENCE- Part of the pending command group. This is the first pending command that should be loaded in a sequence. It indicates when a sequence of pending commands should start executing based upon its wait conditions and values. This command also includes exit values which indicate the conditions upon when the sequence should be aborted. If this command is received while another sequence exists, the old one is deleted and this one will begin a new one. As with the other pending commands, it is en-queued to the cmd_queue and handled later by CSpendingProces, which does most of the heavy lifting.

on_END_SEQUENCE- Part of the pending command group. This is the last pending command that should be loaded in a sequence. In fact, ground passes as one of its arguments the number of expected commands on the pending command sequence list. If the value does not match with the number of commands, plus this one, on the pending command queue, the sequence is aborted and deleted, with an error message sent in response. Upon en-queue-ing the command, the link is set to active and immediate commands are allowed to run once again. The actual command is handled later by CSpendingProcess.

on_RELOAD_RADIO_CONFIG- Part of the pending command group. Only sets up the conditions to when the radio configurations should be re-loaded. The configurations themselves will exist in the flash journal summary. The actual grabbing of these values and giving them to radio take place in CSpendingProcess, where the radio configs are grabbed from flash memory and passed into a function provided by the radio team.

on_LOAD_RADIO_CONFIG- Follows the same template as on_RELOAD_RADIO_CONFIG except that a new set of radio configuration values are passed up from ground along with the other arguments. These are all stored away in their appropriate structures, and the actual handling of this command takes place in CSpendingProcess.

on_VOLATILE_MEM_CHECK- Shell for calling memTest_bitFlip, a function contained within csMemTest. The value "OK" is retuned in the case that no errors occur, or that there was an error in self-diagnostic, and thereby the test was not completed. If errors are found in volatile memory (data and stack sections), a value representing the number of 'groups' corrupted is returned, along with a series of bits which represent those groups. (See csMemTest for more information.)

on_RUN_PGRM_IMG_CHECK- Bit-sum of entire flash (minus the flash journal) is found and returned as a UINT32 to ground.

on_GET_JRNL_SUMMARY- Gets the current journal structure in flash memory and returns it to ground. Originally I just passed in the response_buffer type converted into Journal_GetStruct, but this would generate an address error. This is because the pointer starts on an odd address and could not handle UINT16's being written to it.

on_REFLASH_PGM_IMG- A call to this command will only return a value if the operation failed to open the file, which name is passed in via argument from ground. If the reFlash is successful, the satellite will have reset itself. If it's not successful, the satellite is probably dead.

on_COPY_PGM_IMG- This was a nice debugging tool I developed to help with testing reFlash Program Image. This is just the shell for the function call, but the function will copy the current contents of flash memory from addresses 0x000 to 0x20000 into the SD card in the format which can then be read by reFlashProgramImage. This can allow us to make changes and save the current image without having to remove the SD card and manually placing an IMG file onto it.

Summary

Logs can be found in the individual files along with inline documentation. For further questions you may email me at the email provided at the top of this document.

Good Luck!