STANFORD ENGINEERING

SSDL
Space and Systems Development Laboratory

AA 236C Flight Software Final Report

Andres Nötzli, Ashe Magalhaes, Grant McLaughlin, Osagie Igbeare, Thomas Teisberg

Table of Contents

Introduction

In this report, we discuss the progress on the QB50 project that we made in the Spring 2015 quarter. In addition, we describe the next steps needed to complete the project successfully.

Background Information

We discuss helpful background information.

Accounts and repositories

Primary Google Drive *Contains resources from AA236B and AA236C 2014-15*
https://drive.google.com/a/stanford.edu/folderview?id=0B6zwRToF1deLYXRaNjNEa21rNm8&usp=sharing
(Editable by anyone with an @stanford.edu account.)

AA 236C Spring Flight Software Drive
From folder above, navigate to FSW > Spring

SSDL GitHub Organization
https://github.com/SSDL
Contact anyone on the team to add members

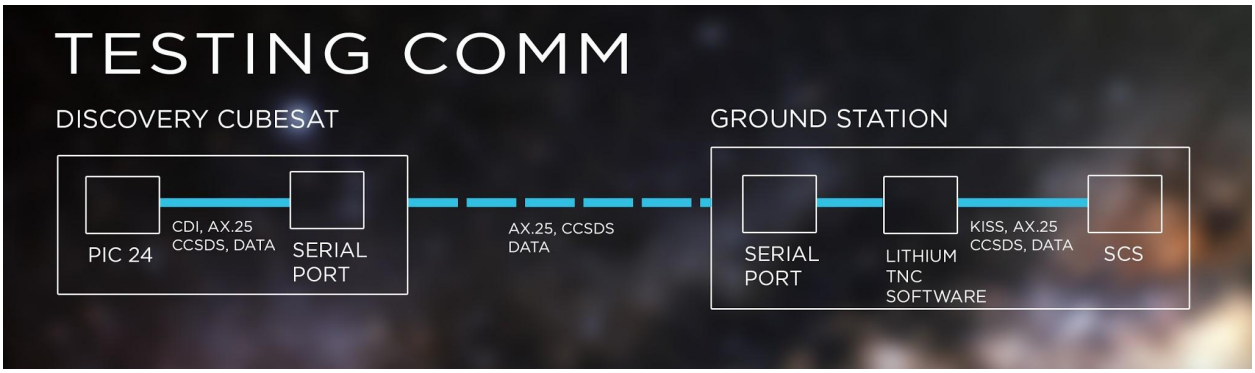
QB50 Flight Software Repository
https://github.com/SSDL/qb50_fsw_x

SCS configuration files
https://github.com/SSDL/scs-config/

lithiumTNC (software used for testing)
https://github.com/4tXJ7f/lithiumTNC

Protocols

We spent a major part of the quarter figuring out the protocols to use with SCS. Because the existing documentation is rather poor, we give a brief overview here, pointing to the relevant documents for further details.
The following diagrams give an overview of the protocols used at different points in the communication chain.



Telemetry

At a high level, a telemetry packet is structured as follows:

AX.25		CCSDS			CCSDS	AX.25	
AX.25 Transfer Frame Header	Telemetry Transfer Frame Secondary Header	CCSDS Packet Header	Telemetry Data Field Header	Source Data	Packet Error Control	Telemetry Transfer Frame Trailer	Frame-Check Sequence
128 bits	32 bits	48 bits	64 bits	Up to 2048 bits	16 bits	8 bits	16 bits

The parts in blue are described in QB50-EPFL-SSC-SCS-ICD-AX.25-TFF, the parts in green in QB50-EPFL-SSC-SCS-ICD-TMTC_PD. AX.25 is a standard protocol designed for use by amateur radio operators. The use of AX.25 is a requirement of the QB50 project (QB50-SYS-1.5.13). We only use a subset of the protocol: Only UI frames are being used and supported. It is important to note that the primary AX.25 header and some of the AX.25 specific functionalities like bit stuffing are implemented by the radio and the TNC. Theoretically, CCSDS packets do not need to be “aligned” with AX.25 frames; a AX.25 frame can contain an arbitrary number of CCSDS packets (even partial packets, if the CCSDS packet is very large). Another way to think about it is that AX.25 transports a *stream* of information, so the segmentation at the AX.25 level has no meaning nor impact on the protocols that it transports. This notion is somewhat imprecise because the AX.25 header contains a First Header Pointer in the Secondary Header

which specifies the start of the first CCSDS packet inside the AX.25 frame. For now, we decided to always send one CCSDS packet in one AX.25 frame in order to keep the code simple, with the option of implementing unaligned CCSDS packets if deemed necessary later in the development process. We currently do not think that this will be necessary. As a result of this decision, the current flight software always sends 0x00 as the First Header Pointer.
The flight software generates telemetry packets and sends them to the radio using the radio's Command and Data Interface (CDI) over UART. The CDI format is described in the Radio Interface Manual provided by Astronautical Development. To send a message via the CDI, the flight software has to add an 8 byte header containing the OpCode 0x1003 and a 2 byte checksum footer.

Storing Telemetry

We will only be able to communicate with the satellite a few times per day for a couple of minutes. The satellite has no mechanism to detect whether the ground station receives the telemetry that it sends. Thus, it stores every piece of telemetry on one of two SD cards for later retrieval. We reuse LMRST-Sat's TAP structure to store packets on the SD card. A TAP is a telemetry packet. The project defines a collection of TAP *types* with an associated TAP *ID*. Each TAP type contains a fixed list of data (e.g. position data, temperature data, etc.). We made this decision for two major reasons:

- We can reuse most of LMRST-Sat's code for storing and retrieving telemetry to/from the SD card
- The TAP sequence number is a nice way of referring to past telemetry and organizing it on the SD card (the system keeps track of a sequence number for each TAP ID and uses parts of the sequence number to determine the folder of the SD card to store the TAP in)

When storing telemetry on the SD card, we store the CCSDS and TAP headers/footers along with the actual data. The AX.25 frame is added only when sending the data over the radio. All the AX.25 fields are generated on-the-fly. We added the option `send_metadata` to the TAP structure `TAPStruct_t`, which is defined in `TAP.c` and stores metadata for each TAP ID. The new option specifies whether the TAP header is sent over the radio for a given TAP ID or not. The next section contains some additional information about the fields in the TAP header/footer.

Telemetry types

The type of telemetry is described by the Application Process ID (APID), Service Type and Service Subtype. The CCSDS Packet Header contains the APID, while the Telemetry Data Field Header contains the Service Type and the Service Subtype. The following telemetry types are currently used:

- Telecommand verification (Service Type = 1)
- Housekeeping (APID = 10, Service Type = 3, Service Subtype = 25)
- INMS Data Management (APID = 14, Service Type = 128, Service Subtype = 1)

In the following, we provide some more details about the telemetry types.

Housekeeping

When the satellite sends housekeeping telemetry, the first byte in the Source Data determines the type of telemetry (SID). We implement two types of housekeeping telemetry:

- TAPs (0 < SID < 255)
- Whole Orbit Data (SID = 255)

The TAPs have the same structure as in LMRST-Sat:

Sequence number (4 bytes)	Time (6 bytes)	TAP Data	Checksum (2 bytes)

Note: LMRST-Sat stores a TAP ID at the beginning of a TAP. We use the SID byte to store the TAP ID (fortunately, they are at the same position, so using them interchangeably is easy).
The Whole Orbit Data is a packet required by the QB50 project (requirement QB50-SYS-1.4.1) which provides basic health information about all satellites to the project.

SCS configuration

SCS uses an XML configuration file (`Server/CoreDistribution/MIB/qb50-base.xml`) to describe the format of telemetry messages. In order to support a new telemetry packet, multiple sections of the XML file need to be changed as described in the following sections.

Parameters (Mib/Housekeeping/Parameters)

Each <Parameter> tag corresponds to a value of a telemetry packet. The important attributes are:

Number	Unique ID (used later as a reference to this parameter)
Name	The name that will be shown in the MissionData Client
Ptc (Parameter Type Code) and Pfc (Parameter Format Code)	Stores the type and format of the parameter as described in ECSS--E--70--41A pp. 189
IsSpare	If this is set to true, then the MissionData Client does not show the value
Description	The MissionData Client shows the description when the user hovers the mouse cursor over the value
Unit	The MissionData Client shows the unit next to the value
Calibration	ID of the calibration that turns the raw value into the actual value

Calibrations (Mib/Housekeeping/Calibrations)

There are multiple types of calibrations:

TextualCalibration	Maps an integer range to a text. The Check attribute of the RangeText tags describes the type of the value (Nominal, Warning or Danger)
PolynomialCalibration	Turns an integer value into a floating point value using a polynomial. Optionally, the NumericCheck attribute contains the ID of the check for the resulting values

CheckOnlyCalibration	Only does the check, no conversion

Numeric Check (Mib/Housekeeping/NumericCheck)

Every <NumericCheck> tag describes one check with the following attributes:

Id	Unique ID used as a reference
Name	Name of the check
DangerLow/DangerHigh	Values below/above this threshold are considered to be dangerous
WarningLow/WarningHigh	Values below/above this threshold are considered to be warnings

SIDs (Mib/Housekeeping/Sids)

Each <Sid> tag describes a telemetry packet with the following attributes:

Number	The SID of the telemetry type
Apid	The APID of the telemetry
IsRepeated	Can there be multiple samples in a single packet? E.g. WOD requires this
TimeInterval	Time interval between the samples
Name	Name of the telemetry type
Description	Description of the telemetry type
Parameters	Space separated list of parameter IDs that describes the sequence of parameters in the telemetry packet

Telecommand

Telecommands have a similar structure as telemetry but there are different fields. At a high level, a telecommand is structured as follows:

AX.25		CCSDS			CCSDS	AX.25
AX.25 Transfer Frame Header	Telecommand Transfer Frame Secondary Header	CCSDS Packet Header	Telecommand Data Field Header	Source Data	Packet Error Control	Frame-Check Sequence
128 bits	8 bits	48 bits	24 bits		16 bits	16 bits

Like the telemetry packets, the parts in blue are described in QB50-EPFL-SSC-SCS-ICD-AX.25-TFF and the parts in green in QB50-EPFL-SSC-SCS-ICD-TMTC_PD.

In order to support large telecommands (e.g. for uploading INMS scripts to the satellite), the Telecommand Transfer Frame Secondary Header contains a Sequence Flag (2 bits) which denotes the start, the continuation and the end of a segmented command. Note: This implies that long commands are segmented at the AX.25 level.

Ground Station Hardware

The ground station sends and receives data using a Terminal Node Controller (TNC). The TNC is responsible for converting an AX.25 message to an analog signal and vise versa. The ground station uses the KISS protocol to communicate with the TNC over serial. KISS is a simple standard protocol (http://en.wikipedia.org/wiki/KISS_%28TNC%29). A TNC typically supports multiple operation modes, so it needs to be set to KISS mode.

Completed Work

We describe the work that we completed over the course of this quarter.

Communication

A lot of the work throughout the quarter was related to communication between the satellite and the ground station in an effort to make the flight software compatible with SCS. This work had two major components with different challenges: sending data from the satellite to the ground station (telemetry) and sending data from the ground station to the satellite (telecommands). We break down the work done for both directions.

Telemetry

Flight Software

LMRST-Sat was not using SCS for the ground station, so we had to change the flight software to make it compatible with SCS. To do this, we implemented the AX.25.h/.c module which implements functions to deal with the secondary headers of the AX.25 protocol. The AX.25 protocol is a requirement of the QB50 mission (QB50-SYS-1.5.13) and encapsulates CCSDS and application data as outlined previously. More details on the protocol can be found in the document “SCS Description and Interface Control”, Ref.: QB50-EPFL-SSC-SCS-ICD-D2501. CCSDS communication is managed by the CCSDS.h/.c module, which handles packaging telemetry packets with a CCSDS header and footer and parsing the CCSDS level of telecommands. The packet includes fields for the application ID, service type, service subtype, time, and data; this is detailed in QB50’s “Recommendation for Flight Software Implementation” with a reference number of QB50-EPFL-SSC-SCS-ICD-FSW-1-0. We further configured the FSW to QB50s standard by implementing a module (task_WOD.h/.c) for the whole orbit data. Whole Orbit Data (WOD) is a set of housekeeping data collected over the whole orbit once a minute. The parameters include satellite mode, battery voltage, battery current, bus currents, temperature of COMM systems, EPS, and batteries and serve to identify the health status of the CubeSat over the period of the mission. For those parameters that our satellite is not tracking (temperature of COMM systems and EPS), we pass in a 0 as is outlined by QB50 in the document Whole Orbit Data Packet Format (von Karman Institute for Fluid Dynamics

Aeronautics / Aerospace Department). The class sends WOD packets that contain 32 minutes of data from the satellite to the ground station once every 30 seconds.

Grant's edits

- Mention structural changes to the FSW to work around RAP
- Map telemetry through functions and modules

SCS configuration

We needed to change SCS to support new types of telemetry. Unfortunately, documentation was rather sparse. The background information section contains the information that we pieced together. Because the complete list of TAPs is currently not known, we limited ourselves to configure the Bus TAP (TAP ID/SID = 3). Unfortunately, SCS spreads its configuration files across multiple folders which makes upgrading to a new version and keeping the configuration files under version control a pain. To improve the situation, we created the scs-config repository that contains the SCS configuration files and we replaced the configuration files in the SCS folders with hard links (https://msdn.microsoft.com/en-us/library/windows/desktop/aa365006%28v=vs.85%29.aspx) to the files in the repository.

Telecommand

// @Grant: Describe your changes to support telecommands here

- Work around RAP
- Splice together commands or scripts and route them
- Removal of AX25 Secondary header
- Removal of CCSDS frame
- Existing echo functionality needs to be replaced with required echo protocol
- Map telecommands through functions and modules

Testing Infrastructure

In order to test the communication between the satellite and the ground station, we implemented a tool called lithiumTNC that replaces the link between the ground station and the satellite. An important design goal of lithiumTNC was to replace the link as seamlessly as possible so that we can test the full software stack without any changes just for testing. To achieve this, lithiumTNC simulates the satellite radio on one end and the terminal node controller (TNC) of the ground station on the other end. The software communicates with both parties through a serial port.

Implementation

In order to simulate the TNC and the radio, lithiumTNC implements two protocols: KISS for the TNC and the CDI of the radio. Our software only supports data frames. The official manual of the lithium radio[1] specifies the CDI protocol. The CDI protocol allows the microcontroller to talk to the radio. Our software supports the Transmit (0x1003) op code for sending data from the satellite to the ground station.

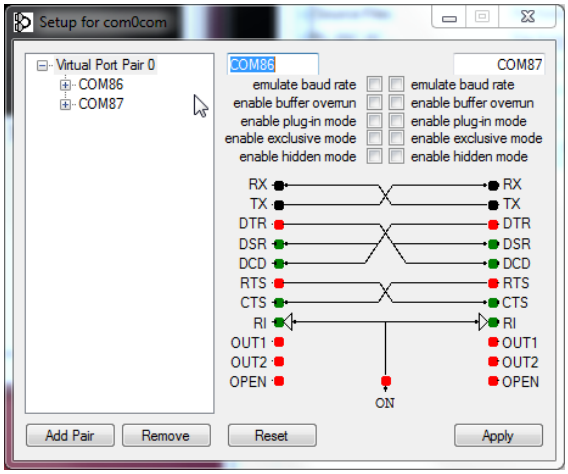
[1] http://www.astrodev.com/public_html2/downloads/firmware/Li1_Programming_Pack_R3pt10.zip

We implemented lithiumTNC in C and used the POSIX interface for serial communication. We use CMake for the building the project. The serial interfaces are configured with hardware and software flow control disabled and without parity bits. During the development, we accidentally had software flow control enabled which lead to a hard-to-debug issue when a packet contained one of the characters which were reserved for software flow control.

Usage

Because we use the POSIX interface for serial communication, the software requires a POSIX compliant operating system. Our current setup uses Ubuntu running in a virtual machine on the ground station using VMWare Player. We use com0com to create a pair of virtual serial ports for the communication between SCS and lithiumTNC. We configured VMWare player to pass control over one of the virtual serial ports and the serial port that connects to the satellite to Ubuntu. While this solution proved to be somewhat finicky to set up, we found it to be stable once running.

The following screenshot shows our current com0com configuration:



Implications for the flight software

Because lithiumTNC simulates the radio, the flight software does not need to implement separate code paths for sending data over USB or the radio, improving test coverage. This is in contrast to the LMRST-Sat code which had separate UART1_packetize/He1_packetize functions. This allowed us to simplify and clean up the existing code.

INMS

Previous Work

Work on the INMS integration was started during AA 236B (Spring 14-15) by the CONOPS team (Jan Kolmas and Thomas Teisberg). The final report from this team is available in the Google Drive (as well on the SSDL servers). Last quarter's work focused on building preliminary understanding of the software requirements needed to upload and run scripts. Skeleton code for executing scripts on Discovery was written last quarter. This code has been expanded upon and implemented within the satellite codebase this quarter.

INMS Scripting Overview

The following section will outline the process for storing and running scripts on the satellite. This process is documented in the INMS ICD and User Manual, however the documentation does not provide a straight-forward step by step explanation of the control flow. We hope that this report will help next year's team get up to speed with less difficulty.

What are scripts?

The INMS, Discovery's scientific payload, is commanded to turn on, turn off, collect data, and do a few other housekeeping tasks through "scripts" that are written by the QB50 team, uploaded to each satellite, and run at periodic intervals throughout the day. Each script is a binary file containing one or more sequences of instructions and information about when to run each instruction. It is the responsibility of the flight software to correctly manage the scheduling of each command. Some of the commands must also be acted upon by the flight software while others are simply passed to the INMS.

Script Upload and Multiple Scripts

Scripts will be provided to each QB50 team periodically throughout the mission to be uploaded to the satellite. Teams will also be provided with a set of scripts prior to launch to pre-load. The satellite must store up to 7 scripts. Each script is stored in a "slot." When a team uploads a script to their satellite, they must specify the "slot" in which to store the script. If a new script is uploaded to a slot that is already full, the old script is deleted. This serves as the way to replace and terminate scripts.

While there are 7 script slots (and thus 7 scripts may be stored at the same time), only one script may be running at any given time. The script that will run is chosen based on "eligibility." A script becomes eligible when a UTC timestamp in the script headed is reached. When a script becomes eligible, it starts running, stopping whatever script was previously running. Thus the currently running script is always the script whose start timestamp was most recently reached. In the event that the satellite restarts and no script is running, a script starts running only when the *next* start timestamp is reached (i.e. the previously running script is not resumed).

Implementation: Storing Scripts

In the code written this quarter, when a script is received, the timestamp is extracted and the file is saved to the SD card. The script slot and timestamp are then stored in an array in memory. Because script files can be quite large (see discussion of script size below), it is important to not store all 7 scripts in memory.

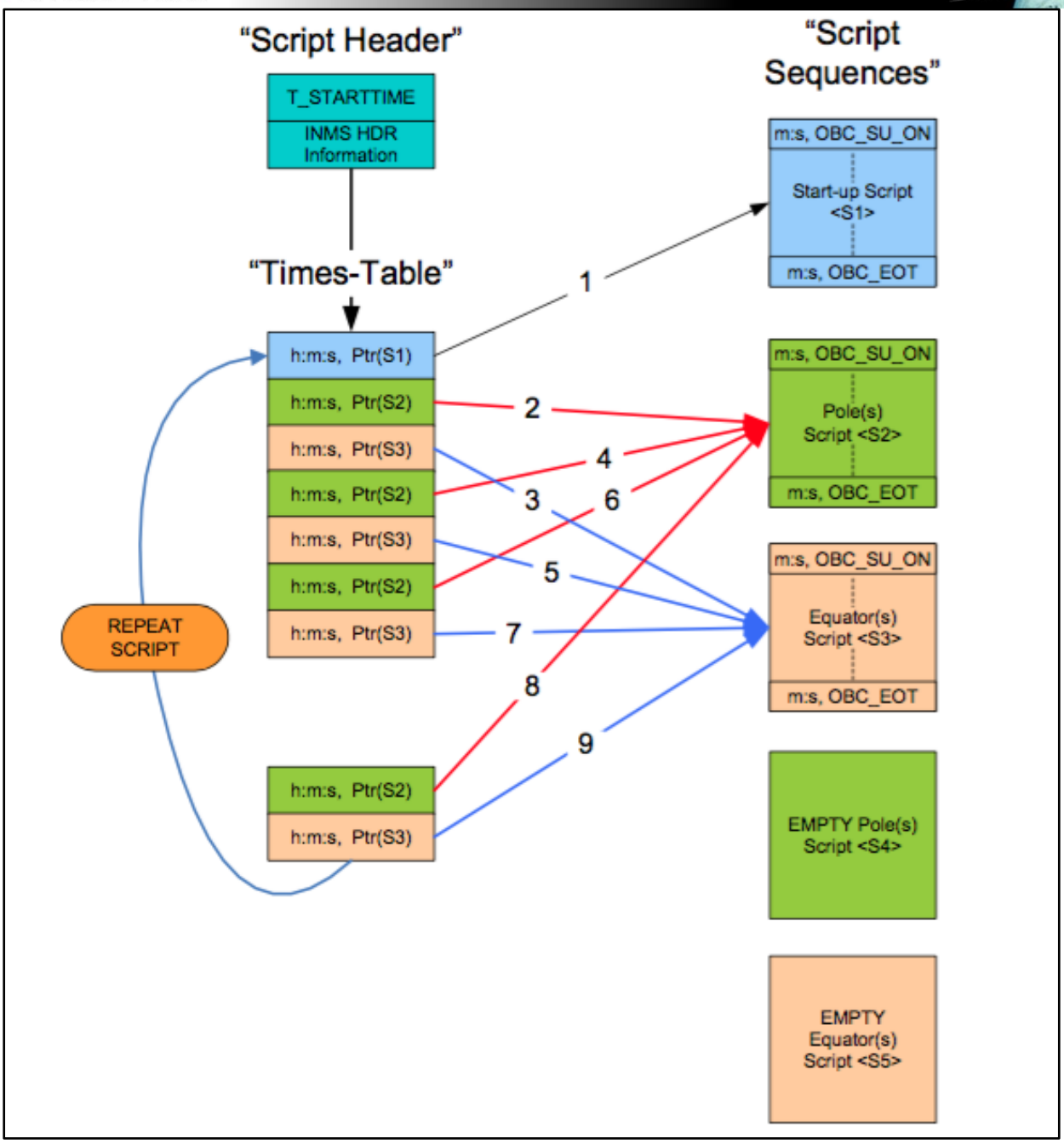
Within the INMS module of the code, there are two Salvo tasks that run simultaneously. task_ScriptTimer is responsible to selecting the appropriate script to run. task_ScriptHandler manages the execution of the selected script.

Running a Script

As mentioned above, only one script runs at a time. The anatomy of a script is as follows:

Script Header	• Size of script file • UTC Start Time • Additional script metadata
Times Table	• 4 byte entries containing an HH:MM:SS daily start time and a reference to the script sequence to run • Last entry has a script sequence reference of EOT (0x55, end of table)
Script Sequences	• Up to 5 script sequences (S1-S5), each containing commands and delays after running each command • S1 is reserved for once-a-day setup tasks • Scripts sequences can be of variable length
Checksum	• Fletcher-16 checksum of the entire script

Once a script has been selected to run, the times table must be read to determine which script sequence to execute. The times table is a daily table of when to run each set of commands (script sequence). A times table might say, for example, “run S1 at 8 AM, run S2 at 3 PM, and run S3 at 9 PM.” When the bottom of a times table is reached (in this case, after running S3 at 9 PM), the script execution continues from the top again (thus S1 would be run the next day at 8 AM). This continues until the script is no longer running.



Within each script sequence, there are a set of time command pairs. For each pair, the command is run and then there is a delay[2] for the length of the specified time. For example, a pair might indicate to run the OBC_SU_ON command and then wait 2 minutes. This delay is intended to allow the execution of the command to finish before proceeding to the next step. There is no checksum or other means of verifying that individual commands sent to the INMS

[2] There is a small inconsistency in the INMS ICD regarding the order of this delay (before or after the command). Our interpretation is that the delay comes after the command. See the Appendix for more details.

have been received properly. It is simply assumed that all commands are transmitted properly. If the INMS does not provide the appropriate data in response, then an error sequence is initiated.

Executing a Command

There are two types of commands: on board computer commands (OBC_) and science unit commands (SU_). OBC_ commands require some direct action on the part of the flight software. OBC_ commands are simply passed along (over serial) to the INMS. Each command type is detailed in the table below from the INMS ICD:

Command	CMD_ID	RSP_ID	LEN	Parameters
OBC_SU_ON	0xF1	-	2	seq_cnt , **SAFETY_ON**
OBC_SU_OFF	0xF2	-	1	seq_cnt
SU_RESET	0x02	-	1	seq_cnt
SU _STIM	0x04	0x04	2	seq_cnt, t_STIM_run,
SU _LDP	0x05	-	LEN	seq_cnt, MODE , addr, B0, B1, ...B_LEN-3
SU _HC	0x06	0x06	4	seq_cnt, [STIM+V_start]U16, sw_HC_anaU8
SU _CAL	0x07	0x07	4	seq_cnt, [STIM+V_start]U16, sw_CAL_anaU8
SU _SCI	0x08	0x08	6	seq_cnt, [offset +STIM+V_start]U16, t_dwellU16, rpt
SU HK	-	0x09	-	Fixed 174 bytes packet – see SU_HK for details
SU _STM	-	0x0A	-	Fixed 174 bytes packet – see SU_STM for details
SU_DUMP	0x0B	0x0B	1	seq_cnt
SU _HVARM	0x53	-	1	seq_cnt
SU _HVON	0xC9	-	1	seq_cnt
SU_ERR	-	0xBB	-	Fixed 174 bytes – see SU_ERR for details
OBC_SU_ERR	-	0xFA	-	Fixed 174 bytes – see OBC_SU_ERR for details
~~**OBC_SU_HK**~~	–		–	~~command deprecated~~
OBC_EOT	0xFE	-	1	seq_cnt

After the command is executed, no other commands should be executed until the delay time is elapsed. After each command execution, the software must also check if the current script has been preempted (i.e. if another script is supposed to start executing now). If this is the case, the script execution is immediately stopped, and the satellite begins executing the new script.

Outstanding Documentation Clarifications

Two aspects of the INMS documentation required clarification. We posted questions in the INMS section of the QB50 forums under the username “thomasteisberg”. As of the time we submitted this report, there have been no replies. See the Appendix for a full listing of the forum posts and the References section for information on joining the forum.

Implementation Notes

All of the functionality described above is preliminarily implemented in the flight software. See the Code Development below for more information.

Script Size

One major issue identified is that there is no reasonable maximum script size defined to the best of our knowledge (see Appendix for a forum post clarifying this). The script header reserves two bytes to store the size of the script (in bytes). This implies an absolute maximum script size of 64 kb.

$$\frac{2^{16}}{1024} = 64\ kbytes$$

Unfortunately, this is too large to be stored in memory on the satellite. We assume, however, that there is no intention to actually produce script files that large. The largest example script we are aware of is 206 bytes.

In addition, it would be preferable to avoid dynamically allocating any memory in flight to prevent memory-related bugs which are difficult to test for and can easily cause a system crash. Because there is no limit to the number of commands in each script sequence or the number of times table entries, it is difficult to determine how to allocate memory to each of these parts, even if an overall maximum was known (although there would, of course, be ways to work around this by statically allocating a chunk of memory and manually managing it).

The size of a script (in binary form) can be determined as follows:

$$script\ size = 12 + 4t + 1 + \sum_{j=1}^{S} \sum_{i=1}^{C_j} 4 + l_{j,i}$$

t is the number of entries in the times table (not including the EOT entry)
S is the number of script sequences (maximum of 5)
C_j is the number of commands in script sequence j
$l_{j,i}$ is the length in bytes of payload of command i in script sequence j (0 to 255 bytes)

Suggested Approach

It is not feasible to support scripts as large as 64 kb. As such, we suggest choosing a maximum script size and simply ignoring scripts that are too large. We are still hoping to get some clarification from the QB50 organizers (see Appendix), however, if we cannot get a more restrictive maximum size, the following could be a reasonable approach for choosing a maximum size and dimensions.

The maximum payload length of any specified command is 6, leaving a maximum command length of $4 + 6 = 10$ bytes. Assume that every script has 5 script sequences (the maximum number) and that each has the same number of commands. This simplifies the script size to:

$$script\ size = 12 + 1 + 4t + 5 \cdot 10 \cdot C = 13 + 4t + 1295C$$

From here, reasonable values for t (the number of times table entries) and C (the number of commands per script sequence) can be selected. For example, if 2kb of space is available, $t = 20$times table entries and $C = 195$commands per sequence could be chosen. Based on all available information, this should accommodate all reasonable scripts.

In summary: If the QB50 team does not reply, we suggest making the assumptions outlined above and choosing a maximum number of times table entries and commands per sequence. 20 times table entries and 195 commands/sequence is an entirely arbitrary but reasonable choice.

Anatomy of a Script File

The following table shows an example script in both human-readable form and in hex. This is intended to help future teams get up to speed with the format of the scripts.

Script Header
This section is fully described by Table 15-5 in the INMS ICD.

`Script_LEN:`
`007D`
`Script_HDR:`
`AD5C07CF`
`AD5C07CF`
`2312`
	
	`7D 00`
	
	`AD 5C 07 CF`
	`AD 5C 07 CF`
	`23 12`

Times Table
This section defines when each script sequence begins to run. The end of the times table is defined by an EOT (end of table) symbol (0x55).

`TimeTable:`		
`00 05 00`	`S1`	`00 05 00 41`
`00 06 00`	`S2`	`00 06 00 42`
`00 07 00`	`S3`	`00 07 00 43`
	`EOT`	`55`

Script Sequences
This section has each of the sets of commands that are referenced in the times table. The first set of commands is S1. They increase in order (up to a maximum of S5). The end of a script sequence is marked by an `OBC_EOT` command.

`ScriptSequences:`		
`S1:`		
`00 02 OBC_SU_ON`	`01 01`	`02 00 F1 01 01`
`00 02 SU_STIM`	`02 02 40`	`02 00 04 02 02 40`
`00 02 SU_DUMP`	`01 03`	`02 00 0B 01 03`
`00 02 OBC_SU_OFF`	`01 04`	`02 00 F2 01 04`
`00 04 OBC_EOT`	`01 05`	`04 00 FE 01 05`
`S2:`		
`00 02 OBC_SU_ON`	`01 06`	`02 00 F1 01 06`

00 02 SU_STIM 02 07 15 00 02 SU_HVARM 01 08 00 20 SU_HVON 01 09 00 20 SU_SCI 06 10 C88E020005 00 02 SU_DUMP 01 11 00 02 OBC_SU_OFF 01 12 00 04 OBC_EOT 01 13 S3: 00 02 OBC_SU_ON 01 14 00 02 SU_STIM 02 15 04 00 02 SU_DUMP 01 16 00 02 OBC_SU_OFF 01 17 00 04 OBC_EOT 01 18	02 00 04 02 07 15 02 00 53 01 08 14 00 C9 01 09 14 00 08 06 0A C8 8E 02 00 05 02 00 0B 01 0B 02 00 F2 01 0C 04 00 FE 01 0D 02 00 F1 01 0E 02 00 04 02 0F 04 02 00 0B 01 10 02 00 F2 01 11 04 00 FE 01 12
Checksum Fletcher-16 checksum as documented in the INMS ICD.	
XSUM: 93 FF	 93 FF

Bugs Solved

The following highlight specific bugs that were remedied at the onset of testing INMS functionality.

SD Card Open Error

```
sprintf(buffer, "script%d.bin", script_id);

//sprintf(str_tmp, STR_TASK_INMS_TIMER ": %s is the buffer", buffer);
//user_debug_msg(DBG_MSG, str_tmp);

file = fopen(buffer, "rb");
```

The function to open files on the SD card was improperly implemented. The first code snippet above illustrates what existed in code before debugging. Strangely, this code still compiled although the syntax of the fopen was missing an underscore. This implementation caused a system wide reboot whenever we tried to read from the SD Card. In addition, the “rb” parameter was unrecognized by the SD library, contrary to online documentation. The code snippet below illustrates the changes to f_open function. These changes allowed the function to run properly.

```
sprintf(buffer, "script%d.bin", script_id);

//sprintf(str_tmp, STR_TASK_INMS_TIMER ": %s is the buffer", buffer);
//user_debug_msg(DBG_MSG, str_tmp);

file = f_open(buffer, "r+");
```

Script Start Time Error

```
uint32_t scriptUTC(unsigned char* buffer){
    return ((int)buffer[2]) | ((int)buffer[3] << 8) | ((int)buffer[4] << 16) | ((int)buffer[5] << 24);
}
```

The scriptUTC function returns the overall start time of the script in UTC (the number of seconds since the beginning of the respective epoch). Above, the function casts the 4 bytes as ints before combining them into a 32 bit value.

However, because the PIC24 is a 16-bit mcu, the max number of bits it can handle in a single operation is 16, unless the “long” cast is invoked. Because this operation involves 32 bits, each byte is cast as a long and then combined into a 32 bit value, as seen below. The image below the code snippet illustrates the start time of the 4 example scripts loaded onto the SD card.

```
420
421  unsigned long int scriptUTC(unsigned char* buffer){
422      return (long)buffer[0] | ((long) buffer[1]<<8) | ((long)buffer[2]<<16) | ((long)buffer[3]<<24);
423
424  }
425
```

```
00:00:11:21.56 msg: SeqNum[01] = 143, Len = 81
00:00:11:21.62 msg: task_WDT: Ran ...
00:00:11:21.66 msg: task_scriptTimer: current time is 3843671603
00:00:11:21.73 msg: task_scriptTimer: Script 1 start time is 3488792148
00:00:11:21.81 msg: task_scriptTimer: Script 2 start time is 3489060878
00:00:11:21.89 msg: task_scriptTimer: Script 3 start time is 3488914870
00:00:11:21.97 msg: task_scriptTimer: Script 4 start time is 3488796818
00:00:11:22.04 msg: task_ADC: V:179
00:00:11:22.82 msg: task_ADC: V:181
```

Code Development

There are two main task functions in INMS.c, task_ScriptTimer and task_ScriptHandler. task_ScriptTimer periodically checks to see if there is an upcoming script that needs to be run (based on UTC time) and calculates a “wait time” before the next script can be run. Once the wait time = 0, i.e. the script start time = the UTC time of the onboard mcu, task_ScriptTimer triggers a flag causing task_Script Handler to properly handle the script.

task_ScriptTimer()

```
419
420         // Find the next script that will become eligible
421         script_id = -1;
422         min_wait = 1;
423         for(i=0;i<=MAX_SCRIPT_ID;i++) {
424             if(time_get_time() > script_utc_start[i]) continue;
425             wait = script_utc_start[i] - time_get_time(); // Time to wait before script eligible
426             if(script_id < 0 || wait < min_wait){
427                 script_id = i;
428                 min_wait = wait;
429             }
430         }
431         // script_id is now the slot id of the next script that needs to run
432         // min_wait is the time difference (in seconds) before that script is eligable
433
```

As shown above, we set a wait variable equal to the number of minutes before next recent script should be run.

```
    OS_WaitSem(SEM_INMS_NEWSCRIPT, min_wait * TICKS_PER_SEC);
    if(OSTimedOut()){
        // Semaphore timed out - run the selected script
        index_script = script_id;
        OSSignalSem(SEM_INMS); // signal task_ScriptHandler to start
        // TODO: Re-init SEM_INMS_NEWSCRIPT semaphore
    } else if (script_utc_start[script_id] - time_get_time() < 2 * script_timer_delay) {
        // This is an edge case were we are very close to being ready to
        // run the next script but a new script just arrived.
        if(script_utc_start[script_last_updated] < script_utc_start[script_id]){
            // New script starting soon, wait for it to be ready and then start it
            OS_DelayTS(script_utc_start[script_last_updated]-time_get_time());
            index_script = script_last_updated;
            OSSignalSem(SEM_INMS);
        } else {
            // New script not starting soon, wait for the previously selected
            // script to be ready and then start it
            OS_DelayTS(script_utc_start[script_id]-time_get_time());
            index_script = script_id;
            OSSignalSem(SEM_INMS);
        }
    }
```

That wait variable is then passed to this WaitSemaphore that will update the current script number and signal task_ScriptHandler to start.

task_ScriptHandler()

```
114         // loop:
115         while(1)
116         {
117             // Wait for a valid script to run
118             if(index_script==-1)
119             {
120                 // no script to run yet, wait for flag from task_ScriptTimer
121                 // OI: above is true - WaitSem does this...
122
123                 OS_WaitSem(SEM_INMS, OSNO_TIMEOUT);
124                 //OS_DelayTS(script_noscript_delay);
125                 //continue;
126             }
127
```

scriptHandler_image1

This is where task_ScriptHandler begins, index_script at this point has been updated from task_ScriptTimer and the WaitSem is the signaler for task_ScriptHandler to begin.

```
130             // New script loaded. Read, parse and reset indices.
131             if(index_script_loaded != index_script){
132                 index_script_loaded = index_script;
133                 loadScript(&binary, index_script); //add function to read from memory
134                 OS_Yield(); // Script load might take a while - let other things run
135                 script = parseScript(binary);
136                 index_tt = 0;
137                 index_sequence = 0;
138                 index_command = 0;
139                 next_tt_time = 0;
140             }
141
```

scriptHandler_image2

Here, we load and parse the script.

```
142             // Loop of timetable entries
143             while(index_tt<script.tt_length)
144             {
145                 // reinitialize break_from_tt flag
146                 break_from_tt = false;
147
148                 // TODO: wait for timetable entry
149
150                 next_tt_time = script.tt_entries[index_tt].TIME_sec | ((long)script.tt_entries[index_tt].TIME_min << 8) | ((long)scr
151
152                 if(time_get_time() < next_tt_time) continue;
153
154                 index_sequence = script.tt_entries[index_tt].Script_INDEX & 0b00001111;
155
```

scriptHandler_image3

We then loop through all the time tables. Each time table has it's own start time, we wait until the current time "time_get_time()" is equal to the start time of the next recent timetable "next_tt_time".

```
156                 // Loop of commands
157                 while(index_command<script.sequences[index_sequence].num_commands){
158                     // CMD_ID is 0xF* for OBC commands. All other CMD_ID's are INMS
159                     // commands (or invalid.)
160                     // Ref: INMS ICD 15.10.1, REQ INMS-I-200
161
162                     // OBC Commands are:
163                     // OBC_SU_ON (0xF1), OBC_SU_OFF (0xF2), OBC_SU_ERR, OBC_EOT (0xFE)
164                     // (OBC_SU_ERR is a response packet only)
165                     // Ref: INMS ICD 15.10, REQ INMS-I-190
166                     switch(script.sequences[index_sequence].commands[index_command].CMD_ID){
167                         case OBC_SU_ON:
168                             // Turn on INMS
169                             // Ref: INMS ICD 15.11.1
170                             // TODO: Turn on power to INMS
171
172                             //turns on 3.3 V line (pin RP21 - RG6 - IO.11) and 5V line
173                             //(pin RP26 - RG7 - IO.10)
174
175                             PORTG &= ~(BIT6 | BIT7);
176
177                             break;
178                         case OBC_SU_OFF:
179                             // TODO: Turn off INMS
180                             // Ref: INMS ICD 15.11.2
181                             PORTG |= (BIT6 | BIT7);
182
183                             break;
184                         case OBC_EOT:
185                             // End of sequence
186                             // Ref: INMS ICD 15.11.5
187                             index_command = script.sequences[index_sequence].num_commands;
188                             break;
```

scriptHandler_image4

We loop through the commands of the timestable.
OBC_SU_ON - we turn on the 3.3V and 5V line of the INMS. In this case we pull the pins low as they are connected the the gates of pMOS's.

OBC_SU_OFF - we turn of both lines to the INMS.

OBC_SU_EOT - command sequence has ended.

```
189                     default:
190                         // TODO: Command is probably for the INMS (or invalid)
191
192                         // send CMD_ID to INMS
193                         csk_uart2_putchar(script.sequences[index_sequence].commands[index_command].CMD_ID);
194
195                         // Debug print out:
196                         #ifdef PRINT_CMD_ID
197
198                         unsigned char cmd_id = script.sequences[index_sequence].commands[index_command].CMD_ID;
199
200                         sprintf(str_tmp, STR_TASK_INMS_OUT ": CMD_ID:%X",cmd_id);
201                         user_debug_msg(DBG_MSG, str_tmp);
202
203                         #endif
204
205
206                         // loop thru parameters sending each one to INMS
207                         for (i=0; i<sizeof(script.sequences[index_sequence].commands[index_command].parameters)-1; i++)
208                         {
209                             csk_uart2_putchar(script.sequences[index_sequence].commands[index_command].parameters[i]);
210
211                             #ifdef PRINT_PARAMETERS
212
213                             unsigned char paramtr = script.sequences[index_sequence].commands[index_command].parameters[i];
214
215                             sprintf(str_tmp, STR_TASK_INMS_OUT ": PARAM:%X",paramtr);
216                             user_debug_msg(DBG_MSG, str_tmp);
217
218                             #endif
219
220                         }
221
```

scriptHandler_image5

If none of the above applies, the command is for the INMS. Here we pass the command header to the INMS over csk uart 2 (which is uart 3 on the PIC 24) and then pass each of the command parameters to the INMS.

```
225                     // Ref: REQ INMS-I-170, INMS ICD 15.4
226                     OS_WaitSem(SEM_INMS_RX, INMS_TIMEOUT*TICKS_PER_SEC);
227                     if(OSTimedOut())
228                     {
229
230                         // ******* Semaphore Timed Out - ERROR HANDLING *******
231
232                         // turn off INMS
233                         PORTG |= (BIT6 | BIT7);
234
235                         // generate OBC_SU_ERR packet
236
237                         static char  ERROR_PACKET[174] = {0};
238
239                         ERROR_CODE = TIME_OUT_ERROR;
240
241                         fillErrorPacket(script, *ERROR_PACKET, ERROR_CODE);
242
243                         // TODO: push ErrorPacket out to TAP
244
245                         // wait 60 sec.
246                         OS_Delay(ERROR_WAIT*TICKS_PER_SEC);
247
248                         // turn on INMS
249                         PORTG &= ~(BIT6 | BIT7);
250
251                         // reinit UART - clears TX/RX buffer of any bad data
252                         csk_uart2_open(CSK_UART_9600_N81);
253
254                         // reinit semaphore
255                         OSCreateSem(SEM_INMS_RX, 0);
256
257                         // flag to break and go to next timetable entry
258                         break_from_tt = true;
259
```

scriptHandler_image6

If the INMS does not respond in 400 sec, there was an error. As per the requirements of INMS_ICD pg. 50, we turn off the INMS, generate a 174 byte Error packet, wait 60 sec, then turn the INMS back on. We also initialize the uart and the wait semaphores that we use. We then set the “break_from_tt” flag to true - to be discussed below.

Break Flags

```
288                 // Check for another script that is ready to pre-empt this one.
289                 if(index_script_loaded != index_script){
290                     // TODO: Do we need to save the state of the current script?
291                     break_from_script = true;
292                     break;
293                 }
294
```

scriptHandler_image7

If task_scriptTimer finds a script that has is ready to run while we are in the midst of working through a script - we break out of the current script, save our position in the current script, and then process the new script that interrupted us. We process the new script from the start.

This functionality is achieved via the code snippets immediately above and below. When task_scriptTimer finds a new script “index_script_loaded” (global variable) will be updated. When update it will no longer equal “index_script” which will cause “break_from_script” to be true. Since break from script is true, we break out of the script level while loop shown in scriptHandler_image1.

```
301             // Break out of this script entirely
302             if(break_from_script) break;
303
```

scriptHandler_image8

```
269             // INMS Error Handling - need to break out of current timetable - script sequence and go to next
270             // timetable entry. Ref: INMS ICD, pg. 50
271             if(break_from_tt) break;
272
```

scriptHandler_image9

If the INMS doesn’t respond in 400 sec. we break out of the entire time table and move to the next most recent time table. This functionality is achieved via the break_from_tt flag, when set to true, we break out of the loop (in scriptHandler_image3) and wait for the next most recent timestable to begin.

INMS Simulator

QB50 provides both a script generator and an INMS emulator - these can be found in a folder called INMS_Sim which is both on the “Legit IAR Workstation” and the ponse network. More information on the generator and emulator can be found in the documentation titled “QB50-INMS-MSSL-UM-13001_INMS_Simulator_User_Manual_Issue_2”. This document can be found

by entering the phrase between quotes into Google or by navigating to the INMS folder in the Spring FSW Google Drive. File path is FSW - Spring - INMS.

INMS Script Size

Section 15.6 of the INMS ICD Section defines the first two bytes of the INMS script header as containing the length (in bytes) of the script. Based on this, we assume there is a 64K absolute maximum size of each script. Given the limited memory available on the PIC 24, it is strongly suggested that future FSW teams implement a cap, less than 64K, on the max size of script we can accept. See Suggested Approach under Outstanding Documentation Clarifications.

Misc

SCS Update to Version v2.1

During the quarter, the team behind SCS released a new version of SCS, which fixed a couple of bugs. We followed QB50-EPFL-SSC-SCS-UM-D2503 pp. 20 to perform the upgrade.

Cleanup/Style Guide

In an effort to make the code of the flight software more accessible, we removed unused files and defined a style guide to make the code more consistent. Unfortunately, not the whole code base conforms to the style guide just yet but a large portion has been cleaned up. We published the style guide on https://github.com/SSDL/qb50_fsw_x/wiki/Style-Guide.

Further Development

We describe the remaining tasks that we need to finish in order to fulfil the requirements of the QB50 project.

Telemetry

While the infrastructure for telemetry is in place and seems to work quite well, we still need to decide on the final list of TAPs, implement the final list of TAPs in the flight software as well as configure SCS correspondingly and store telemetry to the SD card (the code for storing TAPs on the SD card is currently not being used due to the other changes in mailman).

Telecommands

The final list of telecommands needs to be assembled and implemented. We have to make sure that we have commands that cover the requirements (commands for the INMS, commands to disable/reenable the radio).

INMS

TODOs in INMS.c

- Push ErrorPacket out to TAP
- Handle response from INMS (if necessary)
- Save the current state of the script
- Re-init the SEM_INMS_NEWSCRIPT semaphore

- In processAndStoreScript(): Send a TAP that slot number was bad
- In fillErrorPacket(): Write a function that takes a byte and puts it into Little Endian format

Decide on dynamic vs. static allocation for active script

- If using malloc, as we are now, then a stack monitor needs to be implemented
- If not using malloc, current code, needs to be adjusted such that scripts are saved in static arrays
- Suggested option: Use only static allocation and create strict bounds on the maximum size of scripts. See the "Script Size" section for details on this approach.

Test task_ScriptTimer and task_ScriptHandler

Neither of the functions have been tested. Currently, we can read the UTC start times from script but the intended functionality of both tasks has yet to be tested and validated.

We focused more on development and pushed off testing. We strongly suggest that the next team be mindful of regularly compiling and testing on the PIC24 hardware.

Watchdog

The requirement QB50-SYS-1.4.6 states that:

> The OBSW shall protect itself against unintentional infinite loops, computational errors and possible lock ups.

To fulfil this requirement, we have to port, adopt and test the code of LMRST-Sat's watchdog. To do this, we should adapt the code to SCS's requirements and remove any functionality for Vizion II. When testing, we must connect our code to the hardware in order to visually confirm through the flashing of the LED that our software restart is accompanied with a hardware response.

Hardware support

We need to integrate the driver code from the hardware team. We also need to add support for the second SD card reader.

Deployment sequence

The deployment sequence needs to be implemented. Note: Before deploying deployables such as the antenna and the solar panels, we have to wait 30min after ejection to be compliant with the requirements (CDS-2.4.2, CDS-2.4.3).

Startup task

Similar to the LMRST-Sat flight software, we need a startup task that takes care of all the initializations if the system gets rebooted. This task needs to be well tested and stable.

Callsign

The communications with Discovery will occur over amateur radio bands. At present, we are planning to use the callsign of one of the team members (Thomas Teisberg, KK6TFS). This

would, however, require that he be present for every uplink to the radio. He is, however, graduating in 2017 before the end of the mission lifetime. A preferable option would be to get permission to use the W6YX club callsign, allowing any member of the club with permission to communicate with the satellite. Thomas has sent an email to the club president and faculty advisor.

Contributions of Team Members

We briefly describe who contributed to which parts of the project.

Andres Nötzli

- Telemetry (FSW + SCS configuration)
- lithiumTNC

Ashe Magalhaes

- Telemetry (FSW + SCS configuration)

Grant McLaughlin

Add contributions here

Osagie Igbeare

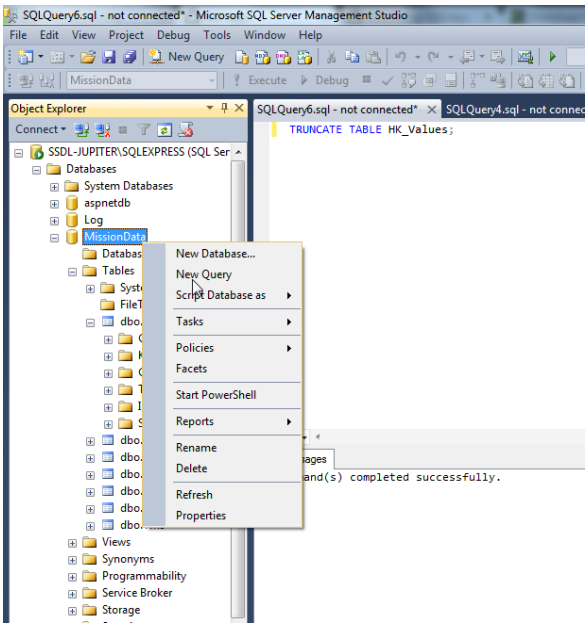
- INMS
- Microchip Stack Size Monitor
- Microchip Stack and Heap Size Limitations

Thomas Teisberg

- INMS Software
- QB50 Documentation Clarifications
- (Ongoing) Contacting W6YX about using their callsign

Tips/Suggestions

Clearing the ground station data



While testing, it can be helpful to clear the data stored on the ground station. The easiest way is to execute the following command using the SQL Server Management Studio on the MissionData database:

```
TRUNCATE TABLE HK_VALUES;
```

Consider making a backup of the database before executing this command as it is deleting data.

If SCS does not receive any data from lithiumTNC

Try restarting the virtual machine and make sure to start SCS *before* lithiumTNC. If nothing else helps, try to remove and add the pair of virtual COM ports in com0com.

References

Paride Testani
paride.testani@vki.ac.be
Paride can provide access to the QB50 forums. Create an account on the QB50 forums and then email Paride. The process may take a week or so.

Appendix

QB50 Forum Post: INMS Script Maximum Size

Posted in Sensor Units -> INMS by thomasteisberg on 5/20/15.

The script header allocates two bytes for the script size, which presumably sets an INMS script size limit of 64KB. I couldn't find a maximum size defined anywhere else.

Is there a defined maximum script size? 64KB is a big chunk of memory to deal with.

Thanks,
Thomas

No response as of June 1, 2015.

QB50 Forum Post: Inconsistency in INMS flowchart (ICD Figure 15.6)

Posted in Sensor Units -> INMS by thomasteisberg on 5/27/15.

This is just a small thing that we wanted clarification on. In two places in the INMS ICD, it specifies clearly that the delay time for each command in a script sequence should occur AFTER the command is executed:

From the INMS ICD Section 15.8:
```
2-byte delay time field: deltaTIME: the length of time (in seconds)
to wait AFTER executing the CMD.
```

From the REQ INMS-I-128:
```
The sequences consist of a COMMAND, and a DELTA-TIME field. The
COMMAND is read and executed FIRST, then, a delay of DELTA-TIME
seconds is elapsed.
```

The flowchart in Figure 15.6 seems to indicate that the delay occurs BEFORE the command is executed.

Is it safe to assume that the correct behavior is always to run a command and then wait? Obviously this would only really affect the first and last commands, but we just want to make sure we're implementing everything correctly.

Thanks!

No response as of June 1, 2015.

Microchip Technical Support | Stack Size Monitor

```
Hello Osagie Igbeare,
```

Microchip Engineering Support has added comments to support Ticket 292034.

Comments:
<<Do you know how we might be able to find the size of our stack at runtime? >>

There are several way of doing this.

One way is to set up a timer interrupt and periodically examine the value of the stack pointer.
Or if you know a routine that is likely to be one of the routines called at the deepest level where you would want to know the Stack usage/free stack, you can examine the stack pointer.
Or you could put this in every routine, including interrupt service to examine the stack usage in each routine.

The way you can examine the stack is by use of inline assembly to get the stack value, and then compare with a running maximum, updating the maximum as necessary:

```
int16_t sp;
asm ("mov w15,%0\n\t" : "=r" (sp));
```

Also refer to the section "6.4 STACK" which talks about the Stack, SP_init, SPLIM and the WREG15 (Stack Pointer)

The 16-bit devices dedicate register W15 for use as a software Stack Pointer.
The linker will allocate an appropriately sized section and initialize __SP_init and__SPLIM_init so that the run-time startup code can properly initialize the stack.
The run time start up code is available in the "src\libpic30" folder of compiler installation directory.

Here is the stack initialization method from the runtime start up code:

```
mov      #__SP_init,w15    ; initialize w15
        mov      #__SPLIM_init,w14
;;  __SP_init         = initial value of stack pointer
;;  __SPLIM_init      = initial value of stack limit register
```

You may also refer to the below thread which talks about the stack monitoring:
http://stackoverflow.com/questions/19760432/runtime-stack-monitoring-on-microchip-pic32-and-dspic33e

Regards,

Anima

Microchip Technical Support | Stack and Heap Size Limitations

Hello Osagie Igbeare,

Microchip Engineering Support has added comments to support Ticket 292034.

Comments:

Hi Osagie,

We did get your voicemail about still being unable to set the heap/stack to more than 21,876 bytes. I did do a quick test to confirm that that is an issue and I was able to reproduce the same problem, although the limit was different for me.

If I set the heap to 70565 then there was not enough room for the stack (compilation failed) even though the requested stack size was only 32 bytes. There should be enough room for that in this device.

When I set the heap to 10,000, dynamic memory (Heap + Stack) is limited to 30720 bytes. Any other values below 30720 for the heap will result in the stack occupying the remainder of the 30720 bytes (For example: 30,000 for heap will result in 720 for stack). This is using an example with no global variables. I assume we are hitting the same limit except your project has globals that reduce the available dynamic memory.

I do not know where this limit is coming from as the device clearly has more available memory. I will have to defer to Anima for an answer as she is the expert in this area.

If you would like to provide more details for Anima please provide them here in the ticket.

Anima, can you please look into the dynamic memory limit that Osagie and I have discribe above?

Thanks,
Tad

Equipment

Machines

- Station 5 in Durand 012-A
- Ground Station in Durand 012-A
- Private laptops of the group members

Software

- IDE: MPLAB X v2.30
- Compiler: XC16 v1.23
- Git client: Git, GitHub for Windows, GitHub for Mac
- Ground station: SCS
- VM software: VMWare Player 7
- COM I/O: HyperTerminal, Realterm
- Virtual COM ports: com0com

Contact Information

Andres Nötzli
noetzli@stanford.edu
(650) 228-3861

Ashe Magalhaes
amagalhaes@stanford.edu
(201) 566-8845
Undergrad ‘17

Grant McLaughlin
Add your favorite email and phone number here

Osagie Igbeare
intrig87@stanford.edu
(631) 335-7207
Coterm ‘15

Thomas Teisberg
teisberg@stanford.edu
(434) 322-0360
Undergrad ‘17