SPLICE: A DSL for Software-Defined Satellite Applications

Exodus Orbitals
contact@exodusorbitals.com
https://www.exodusorbitals.com

***Abstract**:* In this paper, we summarize the Exodus Orbitals experiment concepts on OPS-SAT satellite, European Space Agency software lab in space, that can be also defined as a first "software-defined satellite" – a satellite that is able to support multiple customers with different software applications simultaneously. Our mission application was a prototype of domain-specific programming language, including interpreter and task scheduler, allowing to create and test short scripts ("proto-apps") to identify the necessary concepts for our own software-defined satellite research and development project. Both high-level and low-level specifications are described in the whitepaper, but only low-level implementation is currently implemented and ready for deployment on OPS-SAT. The goal for such a DSL is writing portable software applications that can be space qualified once and later reused on different satellite platform, provided the runtime implementations remains compatible. A number of possible application examples are given as well.

1. Purpose: Identifying Concepts for Software Defined Satellite Architecture

1.1 Introduction

Flight software is major part of any satellite or spacecraft mission. For most of the space exploration history, the software and hardware components of the spacecraft were tightly coupled together. However, this is not the only possible approach. As demonstrated by OPS-SAT nanosatellite mission from European Space Agency, mission software can become a separable part of the satellite mission and achieve a degree of independency from underlying hardware platform [1]. This is actually a very common mode of software interoperation in the world of terrestrial computers, as many software solutions are packaged into an "application" suitable for distribution and installation across a wide variety of supported hosts. In such a way, interested software developers can create a number of useful applications for space industry without the need of ownership of expensive satellite hardware and going through the numerous regulatory and engineering challenges.

1.2 Design goal: "virtualized" satellite architecture

The nuances of space environment demand a number of precautions before allowing users to access and control the satellite instruments and subsystems.

- First a necessary level safety must be ensured through the mission envelope protection done by supervisory software. This system will prevent inadvertent or deliberate failure of the satellite subsystems or entire mission. A resource isolation and separation of user and supervisor segment is highly desirable for the satellite OBDH subsystem. Users should only see a subset of satellite features necessary for their software application and not beyond. Furthermore, the survival of satellite takes priority over user task and software application execution is expected to be controlled by supervisory mechanisms and less so by the application developers.
- Second, as the different host satellites may have different capabilities and payload instruments, it would be convenient to define a hardware abstraction layer, similar to one used in modern mobile and desktop operating systems, taking away the burden of understanding low-level hardware details from users and allowing the operation of satellite instruments in a relatively convenient manner.
- Last and the most important, the programming language for application development in space environment has to account for the all the capabilities and limitations of the satellite platform. While the local computation capabilities are not much below the terrestrial analogs, the uplink and downlink communication channels are a lot more constrained in both bandwidth and availability (typically, only a few short communication windows are available daily for a satellite in low-Earth orbit). Therefore, a DSL (Domain-Specific Language) with appropriate runtime and software libraries is the best approach for the application development process.
- For the OPS-SAT mission, the supervisory software and hardware abstraction layer is taken care of by primary onboard computer and NMF framework[2], leaving the domain-specific language runtime parts as our contribution to the mission experiment.

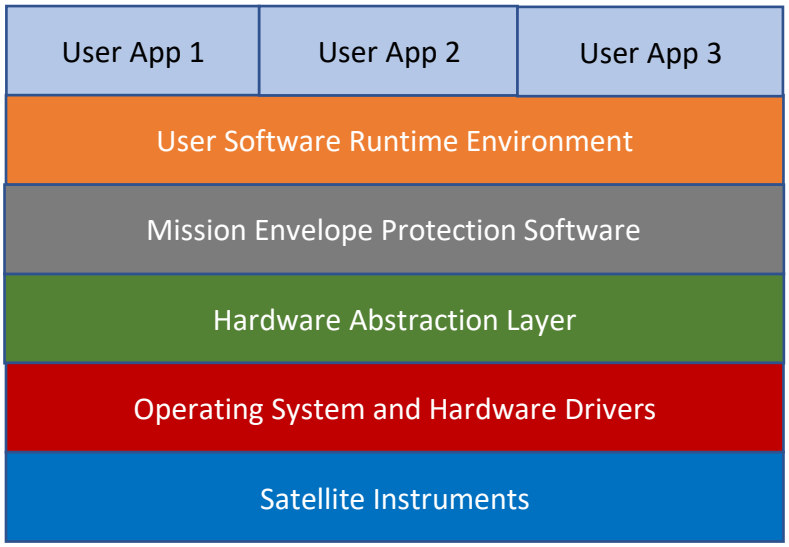


Image 1 – An example of proposed architecture: single satellite, multiple user apps

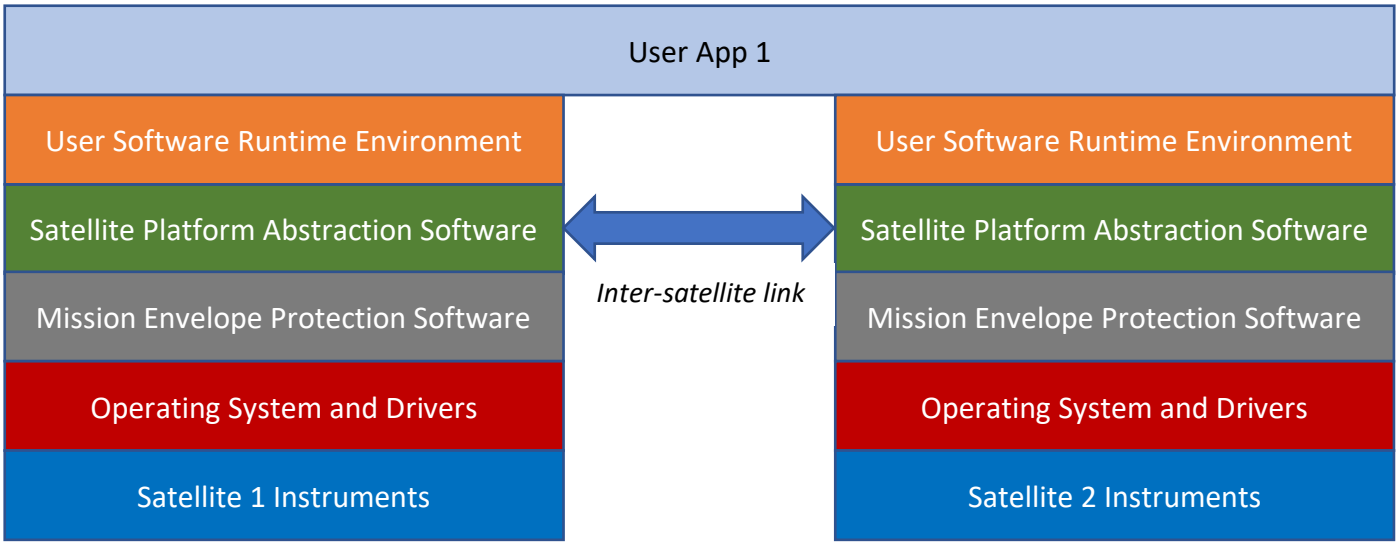


Image 2 – Multiple satellites can form a single "virtual" one, to run a distribute user application

1.3 Application architecture and execution model notes

As illustrated above, there are multiple possible architectures of user applications using the "virtual satellite" approach. A single physical satellite can host multiple user apps, each only using a subset of satellite resources. Alternatively, a single user application can potentially span across multiple physical satellite using instruments and hardware capabilities beyond the scope of any single satellite. As many of satellite components, such imaging camera or mechanical actuators do not offer instant response time and ground and inter-satellite links introduce significant delay into execution flow, a large degree of asynchronous operations is expected of application code. To protect the satellite assets from inadvertent or deliberate malfunctions from user software operations, the nature of proposed DSL is more of declarative (specifying *what* should happen) and less of imperative (directly commanding *how* the hardware should operate) kind.

Part 2. SPLICE on ASTRA – our OPS-SAT experiment

2.1 ASTRA

ASTRA or Asynchronous Space Task Runtime App is a first half of our OPS-SAT experiment software, written using NMF Java application framework. It does pretty exactly what is says in the name as its components include command interpreter, virtual machine and task scheduler that allows uploading and executing command sequences of commands, implemented in Splice programming language, as well as interaction with underlying satellite hardware via NMF functionality. Current implementation of ASTRA understands only VM binary code obtained from compiling Splice scripts, as specified below. The idea is that binary machine code specification will stay the same, while the users will get access to the progressively more advanced high-level programming language features. Hardware components available for user control include GPS receiver, imaging camera and fine-guidance ADCS. Access to software-defined radio transceiver is one of the features scheduled for the future versions.

2.1.1 Architectural notes

- The core of the software is a command interpreter implemented as abstract virtual machine with dedicated memory storage, an opcode dictionary and a set of integer and floating-point registers. The primary data structure of this VM is an execution list of the user task groups. ASTRA runtime maintains this list in memory during power-on status and load/store that as necessary during powered-off state. The execution of the user task is a conditional operation, subject to satellite supervisory restrictions.
- The scheduler is the second major component of the ASTRA runtime and provides the necessary features for task execution depending either on fixed repeat frequency or a number of external or internal conditions, matching the specifications of SPLICE programming language.
- The last but not the least is the high-level abstract satellite model allowing access to the actual satellite instrument. This part is provided mostly by NMF framework, with necessary pass-through conversions provided by the ASTRA code.

2.1.2 Execution notes

- After start-up:
    - Task list and saved context states are loaded from persistent storage.
    - Scheduler resets the task execution state to the initial value
- During runtime:
    - Scheduler continuously checks each task for preconditions and last run timestamp and executes them as required.
    - Any additional scripts uploaded via uplink are processed and added to the task list ready to be executed by the scheduler code
    - Tasks perform operations such as instrument commands and data processing.
- During shutdown
    - The task list and anything else of value is saved to persistent storage. The VM context state is ***not*** saved anywhere, so next VM run starts from blank slate.

2.2 SPLICE: The Programming Language for Extraterrestrial Applications

2.2.1 Introduction

Space Programming Language Interpreter & Command Environment or SPLICE is a Domain Specific Language that enables development of software applications tailored for the conditions of space environment. It is very explicit at *what* should be done but does not specify *how* execution of the program should be done, leaving that to ASTRA runtime. This approach allows to create a space-based software development platform for a much wider audience than ever before, combining both ease of use with sufficient robustness for mission-critical applications in a demanding environment. Its major planned high-level features are described below.

2.2.2 Requirements

- **Suitable for real-time applications**. A typical mission for the satellite in space may consist of the performing a set of imaging observations of the ground below, with specific set for exposure values and wavelength filters selection. It needs to precisely timed or otherwise made conditional on other mission parameters (satellite orbital position and attitude orientation) to be performed successfully.
- **Robustness**. Reality of space environment often interferes with the above plans: a satellite may be in dangerously low power state, satellite camera may face into the wrong direction or satellite onboard computer can tripped into reboot by a stray particle. Satellite hardware resources can be limited and not every downlink session will allow mission data to be downloaded. Therefore, its runtime environment has to be smart when it comes of task execution and storage of results and data.
- **Data-centric**. When editing and transferring instrument data and telemetry parameters to and from different satellite subsystems and uplink/downlink channels, one need good tools for aggregation, abstraction and marshaling of data, from individual bits to multi-megabyte imagery data. Any satellite instrument or component is expected to have some parameters or variables that can be either read and written to (like imaging camera exposure) or only read from (temperature sensors, very commonly). It would be useful to represent this difference in their declaration syntax and also check the validity of operations done upon them before the execution
- **Portability**. The cost of software development for satellite missions comes from the need to qualify the software in the space, that is to run under realistic scenarios onboard the real satellite in space environment conditions. Therefore, it makes sense to compile source code not to target platform directly, but to some intermediate representation that be reused on multiple satellite hardware platforms, following the original Java “write once, run everywhere” promise.
- **Simplicity**. Finally, the last aspect of proposed specification of language is about not what to include into its feature set, but what to exclude from it. Available development tools should allow the developers to interact with the most interesting parts of the satellite – its instruments and instrument-generated data while offloading all the work necessary for satellite survival, such as power management as well as specifics of real-time systems outside of user scope as much as possible.

2.2.3 High-level design specifications

SPLICE major programming constructs are *tasks* and *queues*, inspired by the real-time operating systems like VxWorks or FreeRTOS that are often used as execution platforms for satellite mission software. Note that other elements of real-time systems, like interrupts or priorities are not exposed directly to users and are managed implicitly, within the language runtime, to reduce the complexity of software development for the users. Other language constructs, such as variables, statements, expressions, functions and custom-defined types follow the syntax taken from other programming languages such as C, Python and Rust.

Tasks are lists of actions, that are specified to be executed either at given intervals or on certain conditions, specified by the following qualities:

- How often they are run, either once only or within intervals between repeated execution defined in time range (1 second, 1 minute or 1 hour), only once or as fast as the task scheduler allows.
- When they are triggered to run – either unconditionally after fixed time intervals or conditionally bound by external (measured environment parameter value) or internal state (a successful execution of prerequisite task).
- What exactly are they doing – command the instruments or perform necessary calculations with the data values received.

Queues represent the major building I/O block for instrument command queues, uplink or downlink channels or persistent/temporary data storage, specified by the following properties:

- Interface description of the source or the destination of the data flow, specifying property type, command or telecommunication data packet format.
- Read or write availability, as some queues, for example connected to telemetry sensors can only provide read-only functionality, and some such as ones of communication subsystem can offer both read and write functions.
- Queues can be either persistent, saving their intermediate state to the persistent storage (such underlying file location) or ephemeral, discarding it between the execution runs.
- As both ephemeral and persistent memory is a finite resource, queues also need to define maximum number of data elements stored internally and *shedding* behaviour – what to do when amount of data being written into the queue exceeds its internal storage capabilities. Shedding behaviour can be defined as “newest element gets discarded first” or “oldest element gets discarded first”, depending on the operational needs.

Multiple tasks can be grouped into groups and queues into instruments, for better program structure and user convenience. To ensure deterministic behaviour, loops and control flow constructs are provided only at the task-level, as final authority of execution flow is determined by ASTRA runtime and not exclusively so by the language syntax alone.

*2.2.4 SPLICE syntax **as planned***

Most of the syntax examples below should be self-explanatory for any software developer exposed to the commonly used programming languages, such as Python, C/C++ and Rust.

```
/* Comments are the same as in C-derived languages, both for
single and multiline types */

/* Types declaration use Rust-like syntax */

//32-bit signed integer
var an_integer:i32;

//8-bit unsigned integer
var unsigned_int:u8;

//single-precision float
var a_float:f32;

/* variables can be initialized at declaration, for cardinal
types initialization is optional though */
var initialized_integer:i32 = 0;

/* strings are always allocated statically and has to be always
initialized at declaration */
var a_string:string=”a string value”;

/* constants declaration examples */

//floating point example
const PI:f32 = 3.14;

//integer constant
const TASK_OK:i32 = 0;

//and a string one
const INFO:string = “a string constant”;

/* Custom type declarations */

//complex types can be assembled from built-in ones
type TPacket {
      header: u32;
      data_a: f32;
      data_b: f32;
      data_c: f32;
}
```

```
//type declarations can use nested complex types
type TCommand {
      uid: i32;
      command: string;
      data: TPacket;
}

//variables of complex types can be declared as well
var packet:TPacket;

// ..and initialized, when necessary or desired so
// but for some of the field's initialization can be optional
var packet:TCommand = {
      uid:0,
      command: "Command string",
      data: {
            header:0,
            data_a:3.4,
            data_b:-0.5,
      }
};

// constants can be of custom type as well
const TEST_PACKET:TPacket = {
      header:0,
      data_a:0.0,
      data_b: 0.0,
      data_c:0.0
}

/* Instrument declarations - semantically, they are like
"interfaces" in .NET languages */

//a simple instrument example - GPS receiver
inst gps
{
      //"prop" keyword indicates read-only value
      prop lat:f32;
      prop lon:f32;
      prop alt:f32;

      //can be turned on/off safely from user programs
      var powered:bool;
}
```

```
/* UHF transceiver instrument example. Note the use of queues,
as being in a ground station view is required to send or receive
any data */
inst uhf_comms
{
	//NOT safe to turn it off from user programs!
	prop powered:bool;
	queue downlink {
		size: 100;
		type: TPacket;
		// shedding modifer, as explained above
		shed: "last";
		// access modifier: "push" is write-only
		accs: push;
	}
	queue uplink {
		size: 100;
		type: TPacket;
		//access modifier: "pull" is read-only
		accs: "pull";
	}
}

/* another example - taken from actual satellite mission */
inst camera
{
	var exposure:f32;
	var gainR:f32;
	var gainB:f32;
	var gainG:f32;
	var powered:bool;
	prop imageCount:u32 //read-only property
	queue commands
	{
		size: 10;
		type: TCommand;
		accs: "push"; //"push" is write-only access modifier
		shed: "last";
	}
	queue images
	{
		size: 10;
		type: BMP; //can be RAW, JPG or PNG, for example
		accs: "pull";
		shed: "first";
	}
}
```

```
/* task and groups declarations examples: in the most basic
case, a list of actions to be executed under certain conditions
and with specific execution frequency. groups are collections of
tasks, where all tasks has visibility of each other execution
results and local data in read-only mode*/

group task_group_1
{
        task task_1
        {
                //tasks can have their own variables and constants
                data {
                        var lat:f32=32.21;
                        var lon:f32=120.1;
                        const ADCS_MODE_NADIR = 5;
                        var target_mode: TADCSMode = {
                                command:"set_mode",
                                mode_id: ADCS_MODE_NADIR,
                                duration: 300s
                        }
                }

                /* frequency of execution, can be "once",
                "always" or time value in seconds */
                freq: "once";

                /* prerequisite section, statements below
                has to be true to allow execution */
                preq {
                        gps.lat == lat;
                        gps.long == lon;
                }

                /* executive section - the actual actions
                and logic of the task. Task execution result is
                saved after task execution and updated after each
                run, so it can be used for other tasks as
                prerequisite */
                exec {
                      //pushing a command to instrument queue
                      adcs.command.push(target_mode);

                      //returns task execution result
                      return TASK_OK;
                }
        }
```

```
        /* one task can read another task data (from the
        same group), but not write to. Variables, constants
        or queues can be used for data transfer between tasks */
        task task_2 {
               data {
                 const NX:i32=task_1.ADCS_MODE_NADIR;
                 var gainR:f32=1.0;
                 var gainB:f32=0.5;
                 var gainG:f32=0.5;
                 var exp:f32=0.1;
                 var cam_commd:TCommand;
               }

               /* time literals have prefix as "s","m","h" and so
               on, depending on mission needs */
               freq: 60s;

               /* task execution result can be a prerequisite too
               in addition to conditional expressions */
               preq {
                      adcs.mode == NX;
                      task_1.result == TASK_OK;
               }
               /* There can be no loops or conditional
               expressions in the execution section - just a list
               of statements */
               exec {
                      cam_commd.uid =1;
                      cam_commd.command = "RAW";
                      cam_commd.data_a = gainR;
                      cam_commd.data_b = gainB;
                      cam_commd.data_c = gainG;
                      /* instruments can be controlled through
                      command queues or by directly editing them
                      parameters */
                      camera.exposure =0.2;
                      camera.command.push(cam_commd);
                      /* task context, including local data is
                      saved between runs, allowing complex
                      execution logic */
                      gainR = gainR + 1.0;
                      gainB = gainB + 0.5;
                      gainG = gainG + 0.1;
                      return TASK_OK;
                  }
          }
}
```

Part 3. SPLICE on ASTRA: Implementation Details

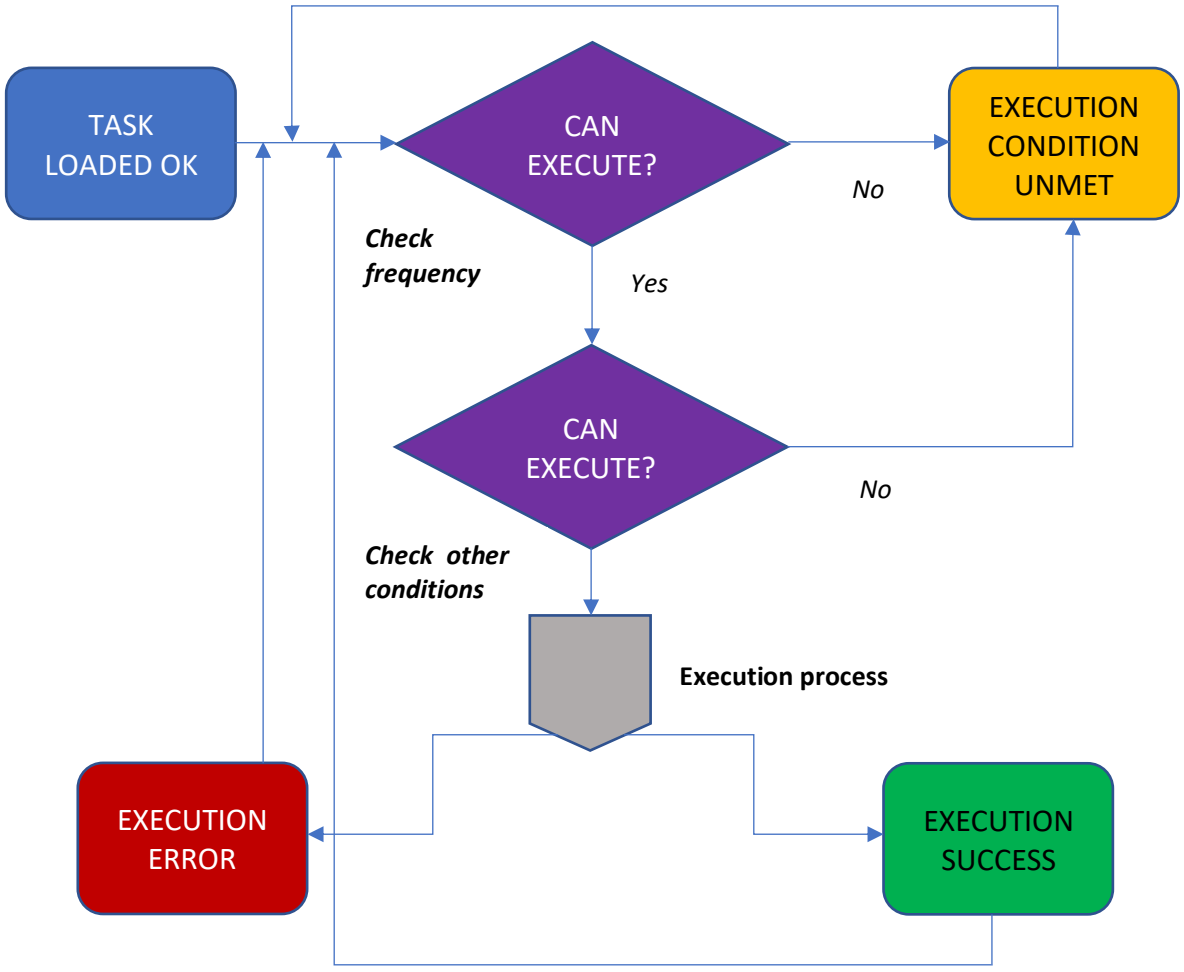
3.1 ASTRA architecture

The design goal of ASTRA runtime is to allow execution of small user programs that can be uploaded via OPS-SAT ground infrastructure provided and command it via NMF framework features provided, without the need to build and install the full-scale NMF application on the satellite already deployed in space. Therefore, ASTRA provides a simplified virtual machine and task scheduler that allows access to some of the NMF functionality in a more abbreviated scripting language as compared to Java. The architecture of ASTRA runtime closely match the features of SPLICE programming language.

- *VM.* Virtual Machine is a simplified model of a real computer hardware that allows execution of user software while allowing to interact available hardware functionality and capabilities of NMF framework.
    - *Registers.* 16 integer (ALU) and 16 floating point (FPU) registers are user accessible from assembly language. Additional floating-point registers are used to provide current ADCS information, such as attitude quaternion, Sun vector and magnetometer readings. The users can read the values from those registers' values, but not write into them.
    - *Memory buffer.* The program memory is split into 16 groups, each capable of holding of 16 tasks up to 256 4-byte words long, for a total of 256 kB user-accessible memory.
    - *Runtime clocks*. There is simple user-accessible software timer, with a 1 second resolution available for users and 1ms internal resolution, stored as 64-bit value. It counts internal VM time and gets reset each time VM application is restarted.
    - *Store and load features*. Upon the start of VM, a list of previously running task is loaded from the task list file using the predetermined location and file name. Location can be changed via NMF functionality during runtime, but not the filename. Upon the termination of VM execution, the currently running tasks are saved into the same file and location.
    - *Output logging.* Output of the program, error messages and additional debug messages are written to a separate file, with a new file created each time VM is restarted, with a timestamped name (using *"yyyyMMdd_HHmmss"* format). The level of details going into the output file is controlled by an exposed variable controlled via NMF functionality.
- *Task Scheduler*. The execution of user tasks is the primary purpose of our OPSSAT experiment. The execution of the tasks depends on both specified frequency and other conditions from either instrument measured values or internal parameters.
    - *Task frequency*. The tasks are executed on variable interval basis, with a smallest possible interval between subsequent executions of 1 second. The fixed time interval is not the only possible value, as tasks that are executed only once or as fast as the scheduler allows are also possible. The time interval range spans from 1 second to 8 hours, with progressively coarser increments: intervals from 1 to 59 seconds are specified at 1 second increments, 1 to 59 minutes up to 1

minute, and 1 to 8 hours at 1-hour increments. This limitation come from a need to fit the frequency value into an 8-bit field of the current task header format.

- *Task lifecycle*. Each task within VM memory can be in one of the possible states during its lifetime under control from Astra scheduler, as illustrated below. All tasks start in "Loaded OK" state and then the scheduler periodically checks if it is the right time to execute the task. Upon completion of the task run, it is marked as having either successful or unsuccessful execution result and the cycle repeats.



- *Execution conditions*. Beyond the frequency check there are a number of task-specific conditions that are checked during the execution, such as parameter value of the given instrument or some other task execution result. If any of those conditions is not true, the task execution stops, and the task returns into "execution condition not met" state.
- *Inter-task communication*. Tasks within the same group have the following permissions – they can read and write to their own properties/queues and read other tasks' data but not write to them. Tasks from the different groups do not have any visibility about each other. In this way, both a degree of cooperation and user access protection is provided, using "groups" as access control feature.

- *NMF pass-through functionality*. Access to the underlying satellite hardware is provided mostly by existing NMF framework, where the NMF functions can be called using VM opcodes and NMF parameters read and written through VM registers. Not all of the underlying hardware features are immediately available, but those can be added to ASTRA capabilities in the subsequent releases.
    - *Accessing satellite telemetry.* As an example, attitude quaternions, sun vector and magnetometer reading are fed into read-only VM register bank, where the user scripts can read the values and use for any purpose. The updates on the telemetry values are done outside the main thread and are independent from user script execution.
    - *Executing satellite commands.* In a similar manner, VM executes an NMF function call from opcode logic to perform an update on satellite parameter or execute an instrument command using the values taken from user-accessible registers for function call.

*3.2 SPLICE syntax **as implemented***

***None of the high-level described syntax described above is available yet**!*

It was originally planned to implement all high-level features of the programming language, but due to the time constraints only a simple assembler translator is available for converting mnemonic commands into Astra VM machine language. However, most of its high-level features can be made available at a later date after mission launch without modifications to the ASTRA runtime deployed on the satellite as the fully-featured compiler can be developed and tested on the ground independently from software deployed in space as long as its output remains compatible with machine language specifications. Therefore, current implementation of SPLICE only offers enough features to allow verification of the core concepts and a simple assembly language to ease the programming efforts.

SPLICE assembly language syntax

Before describing the assembly language, it makes sense to briefly describe the structure of ASTRA machine language used in execution of user scripts and the target of SPLICE assembler translator. It uses fixed-length, 32-bit words that can encode either machine instruction or data. Machine instruction is split into individual byte parts for the following purposes:

Byte 1	**Byte 2**	**Byte 3**	**Byte 4**
OPCODE	PREFIX or MODE	OPERAND 1	OPERAND 2

Most of the assembly commands follow the binary machine word format closely, as described above: *[opcode] [operand 1] [operand 2] [operand 3].* When less than three operands are required, empty placeholder value are used using NOP opcode value (0x00). Basic arithmetic, conditionals and satellite subsystem commands are included in the language definitions. Trigonometric and other auxiliary functionality is provided as well. Control flow features are included as well, those being HLT and CMP opcode. There are no jump or loop opcodes as control of the script execution part is expected to be handled less so by the user and more by the ASTRA command. The list of opcodes is given in the table below:

Opcode	Operand 1	Operand 2	Operand 3	Notes
OP_NOP	N/A	N/A	N/A	No action
OP_MOV	Mode prefix	Source	Destination	Copies value from source to destination
OP_LEA	Register id	Task id	Address	Loads a value from given location
OP_CMP	Operator	Register A	Register B	Compares register A to register B
	Operator	Task id	Register id	Checks task result against register value
OP_SET	Instrument id	Parameter	Register id	Sets instrument property from register
OP_GET	Instrument id	Parameter	Register id	Gets instrument property value
OP_ACT	Instrument id	Action id	Register id	Sends a command to instrument
OP_HLT	N/A	N/A	N/A	Halts program execution
OP_STR	Mode prefix	N/A	Register id	Writes a value to output log
OP_FMA	Register A	Register B	Register C	Fused multiply-add, C = C*B + A
OP_FSD	Register A	Register B	Register C	Fused divide-subtract, C= C/B - A
OP_SIN	Mode prefix	Register A	Register B	Sine and arcsine, B = sin(A)
OP_COS	Mode prefix	Register A	Register B	Cosine and arccos, e.g. B = cos(A)
OP_TAN	Mode prefix	Register A	Register B	Tan or atan, e.g. B = tan(A)
OP_POW	Mode prefix	Register A	Register B	Power, root and logarithm functions
OP_NOR	Register A	Register B	Register	NOR logical operation, C = A NOR B

Executable file format

As ASTRA understands only its own machine language, each SPLICE task must be converted into a machine-readable executable file format before being uploaded to the satellite. Typical ASTRA program is a sequence of machine words, split into header, data and code segments stored. Header is a single machine word, containing the necessary metadata about the program, described in detail below. The code segment is mandatory and should always end with OP_HLT opcode, while data segment is optional. The program is stored as a comma-separated list of the machine words in hexadecimal format. Compiled scripts use *.splx* extension by convention.

The file format is very simple and consists of the of the following sections:

1. Header – contains necessary information for ASTRA runtime using the following 8-bit fields:
    - Group id
    - Task id
    - Frequency
    - Data segment offset
2. Code – contains the instructions in ASTRA machine language, generated from the assembly source.
3. Data – has a list of numeric variables used in program calculations. Data types supported include only signed integers and single precision floating values.

All the values from header, code and data parts are packed into fixed-width 32-bit long words, in a comma-separated value file, suitable to be uploaded through NMF tools available for OPSSAT mission.

Assembly by examples
The structure of assembly source files closely matches the executable program structure, with header, code and data segment parts. Note that the single-line comments are possible in the assembly source files.

Example 1: Orbital period calculation

```
//header - group id = 3, task id = 4, frequency set to every 20s
3,4,20,14
//code segment
OP_LEA, FREG_A, 4, 1
OP_LEA, FREG_B, 4, 2
OP_LEA, FREG_C, 4, 3
OP_LEA, FREG_D, 4, 4
OP_LEA, FREG_E, 4, 5
OP_LEA, FREG_F, 4, 6
OP_LEA, FREG_G, 4, 7
//Perform calculations using the stored values
OP_POW, PRE_NORMAL, FREG_D, FREG_A
OP_FSD, FREG_E, FREG_B, FREG_A
OP_POW, PRE_NORMAL, FREG_F, FREG_A
OP_FMA, FREG_E, FREG_C, FREG_A
OP_FMA, FREG_E, FREG_G, FREG_A
OP_MOV, PRE_MOV_RAM, FREG_A, 8
OP_HLT
//data segment with necessary values
6928.0f
3.986E5f
3.1415926f
3.0f
0.0f
0.5f
2.0f
```

Notes: the formula used is $T = 2\pi\sqrt{\frac{a^3}{\mu}}$, with necessary numerical values stored in data segment. Fused multiply-add and subtract-divide opcodes are used to perform calculations, with a single opcode used for both root and power calculations. The output is written to a data segment location and can be used by another script. The data values are written after the OP_HLT instruction, using “f” suffix for floating point values or “i” for integer ones. The task is executed without added conditions, with only frequency requirement being in place.

Example 2: Satellite camera operations

```
//header: group id, task id, frequency, data offset
2, 4, 60, 13
OP_LEA, FREG_B, 4, 2
OP_SET, INST_IMG, P_IMG_EXPOSE, FREG_B
OP_GET, INST_IMG, P_IMG_EXPOSE, FREG_A
OP_LEA, IREG_B, 4, 1
//check the ADCS mode - has to be nadir-pointing
OP_GET, INST_ADC, P_ADC_MODE, IREG_A
OP_CMP, FPU_EQ, FREG_A, FREG_B
OP_CMP, ALU_EQ, IREG_B, IREG_A
OP_ACT, INST_IMG, A_IMG_DO_JPG, IREG_A
OP_ACT, INST_IMG, A_IMG_DO_PNG, IREG_A
OP_ACT, INST_IMG, A_IMG_DO_BMP, IREG_A
OP_GET, INST_IMG, P_IMG_NUMBER, IREG_A
OP_STR, PRE_STR_ALU, IREG_A
OP_HLT,
//data segment
5i
0.1f
```

Notes: This is an instrument operations example, making the satellite camera take photos in three different formats, and writing the value of total image taken into output log file. The task is triggered every minute and is also conditional on the current satellite attitude state, which value is copied into one of the special-purpose VM registers by iADCS subsystem monitor.

Part 4. User Application Examples

Given below is a list of simple demos, possible to write with current implementation of SPLICE.

- Perform an imaging of a celestial object, given its coordinates . There is a need to consider satellite orbit, tracking of celestial object specifics, camera parameters, Sun avoidance envelope and so on – with some parts has to be handled by users and some are handled by underlying language runtime.
- Perform an imaging of other space satellites that happen to be in proximity to the user satellites – this task requires identifying targets of opportunity, correct timing of flyby windows and selecting correct camera parameters.
- Get most recent satellite map of the area on the ground, given bounding box coordinates (top-left and bottom-right latitude and longitude). This example requires awareness of satellite position and changing satellite attitude when overflying the target area.

References

1. https://www.esa.int/Enabling_Support/Operations/OPS-SAT_your_flying_laboratory
2. https://github.com/esa/nmf-mission-ops-sat