MicroPython Virtual Machine for On Board Control Procedures

Thomas Laroche[(1)], Pierre Denis[(1)], Paul Parisis[(2)], Damien George[(3)], David Sanchez de la Llana[(4)], Thanassis Tsiodras[(4)]

(1) Spacebel, Hoeilaart Office Park, Ildefonse Vandammestraat 7, B-1560 Hoeilaart, Belgium, name.surname@spacebel.be
(2) Spacebel, Liège Science Park, Rue des Chasseurs Ardennais 6, B-4031 Angleur, Belgium, name.surname@spacebel.be
(3) George Robotics Limited, c/o PEM, Salisbury House, Station Road Cambridge, CB1 2LA, United Kingdom dpgeorge@georgerobotics.co.uk
(4) ESA – ESTEC - Keplerlaan 1, P.O Box 299, 2200 AG Noordwijk, The Netherlands, name.surname@esa.int

Abstract

This paper presents the qualification of the MicroPython Virtual Machine and its integration in spacecraft On Board Software. It explains the improvements made to the Virtual Machine. It describes the surrounding On Board Software services required in view of its operational use on board as part of the On Board Control Procedures solution.

Keywords: *On Board Software, On Board Control Procedures, On Board Control Procedure Engine, Virtual Machine, Python, MicroPython, Improvements, Integration, Qualification,*

1 INTRODUCTION

On Board Control Procedures (OBCPs) are flight procedures that can be dynamically uploaded, even after launch, and executed on board. They provide a flexible way to operate the spacecraft, to extend the On Board Software (OBSW) functionality or to modify the behaviour of on board applications. They are increasingly envisaged for implementation of core Payload Software functionality. They are most of the time written in scripting language, compiled as bytecode and executed in a Virtual Machine (VM).

To allow for the use cases sketched above, the VM that executes the OBCPs must be tightly integrated in the OBSW so that they can be controlled by the OBSW and they can access OBSW services. This integration can however not be at the detriment of safety. Possible faults in the OBCP or in the VM cannot propagate and jeopardise the OBSW. Appropriate Fault Containment must be implemented. This does not exempt from qualifying the VM to the required level and from defining an adequate development and test approach for OBCPs.

Spacebel has selected Python as Scripting Language and MicroPython[1] as Virtual Machine for their new generation OBCP solution[2].

The VM port for LEON[3] has been submitted to extensive testing, both on LEON emulator and on hardware target. It has been integrated in the EUCLID OBSW, together with the ESA qualified RTEMS and Math Library.

2 DISCUSSION

2.1 Typical Usage

As defined in the ECSS-E-ST-70-01C [1] standard, OBCPs may come in two ways: On Board Operation Procedures (OBOP) and On Board Application Procedures (OBAP).

OBOP are kinds of macro-command typically provided by the ground operations team and uploaded on board to operate the spacecraft. OBOP can also take part to the Fault Detection Isolation and Recovery (FDIR), where they detect complex failures or implement recovery procedures. They participate to on board autonomy when a rapid reaction is needed in spite of reduced spacecraft visibility or long propagation delay.

[1] The MicroPython VM is developed by George Robotics. It is IPR of George Robotics LTD. It is made available under the MIT Open Source license.

[2] The OBCP Engine is developed by Spacebel. It is IPR of Spacebel S.A.

[3] The porting on the LEON and the RTEMS has been funded by ESA. It is IPR of George Robotics LTD. The code and documentation are distributed under ESA Community License type 3, permissive.

OBAP implement part of the basic functionality of the spacecraft. OBAP may be considered as part of or as extensions to the OBSW itself. They are increasingly being considered, in particular for payloads.

The product requirements depend on the intended use. In this respect, OBAP are more demanding than OBOP, both in terms of the constructs of the language and of access to the on board services. While it defines the language features, the ECSS-E-ST-70-01C standard [1] does not fully specify the functionality and it does not define the actual interface to the underlying OBSW services. Additional requirements may be specific to the mission. The main requirements however vary little from a mission to another or from a domain to another so that the core solution can constitute a reusable building block that can be configured to the specific needs of the missions.

2.2 Alternative Implementations

As mentioned above, OBCPs are most of the time written in scripting language. Similar functionality can however also be provided in different ways:

a. The **Dynamic Linking** of compiled library, provided it is supported by the RTOS, also allows extending the OBSW functionality. It however suffers from several drawbacks stemming mainly from the fact that the additional procedures execute at the same level as the OBSW. The additional module may potentially access any OBSW item and impacts the OBSW scheduling. As a result, possible misbehaviours clearly jeopardise the whole OBSW.

b. In this respect, **Time and Space Partitioning** provides additional security through segregation while also allowing dynamically loading dedicated partitions with compiled libraries. This however assumes that the whole OBSW is based on TSP and that adequate hardware and software interfaces have been foreseen for the corresponding partitions. Though a potentially elegant solution this is not the case for most OBSW.

c. This clearly leaves room for **Interpreted Procedures**. Interpreted OBCPs are pieces of software that can be uploaded, interpreted and executed on board, on demand, at any time and that may interact, to a given extent, with the rest of the data handling system (DHS). They are written in a high level language that is first compiled on ground to yield an intermediate byte code. The bytecode is then uploaded to be executed on board in a virtual machine that interprets the instructions and interacts with the rest of the On Board Software – while also providing some kind of isolation for fault containment. They differ from native applicative components in that their invocation and execution may be controlled. This concerns in particular the ability to suspend or abort their execution.

2.3 Selection of the Technology

The various alternatives to implement interpreted procedures differ in the user language that is used to write the OBCPs on ground and in the Virtual Machine that interprets and executes them on board.

User languages can be proprietary and specific or they can be open. Open languages offer the advantage of being well standardized and used by a wide community. This potentially allows reusing user-friendly development environment and reducing the learning curve of writing OBCPs.

Virtual machines can be proprietary, open source or commercial. A key characteristic of VMs is their complexity that can make them heavy to embark and difficult to qualify for on board applications.

With respect to the above discussions, Python, Java, Ruby and Lua were identified as the most popular languages. Amongst these languages, the standard Python and Java VMs were considered too heavy and the MicroPython VM (see [4]) was regarded as not mature. Lua was therefore initially selected and prototyping activities were successfully carried out.

In the meantime MicroPython however gained in maturity. Also, though it was initially not destined to space applications, a first project funded by ESA allowed porting the MicroPython VM on LEON and RTEMS (see [3]). The decision was then taken not to disperse in different solutions but to join the ESA efforts on a common target, in order to qualify the MicroPython VM, with the intent to reuse it as building block on future ESA missions.

Python is a widely used powerful scripting language. MicroPython implements Python 3.4, with however some subtle differences or limitations, such as the support of only a few standard Python modules, justified by design choices taking into account constrained embedded environments.

The MicroPython implementation features a modern, efficient, highly portable and light-weight implementation with small memory footprint and fast execution designed to be embedded in applications (see [3]).

3 IMPLEMENTATION

Scripting languages usually assume that the procedures are compiled and executed on the same platform. However, for technical and safety constraints embarking a parser and a compiler on board is not an option: their implementation is usually too complex and their execution requires too much memory and consumes too much processing power. The code that is executed on board must therefore first be compiled on ground and then uploaded from ground to board.

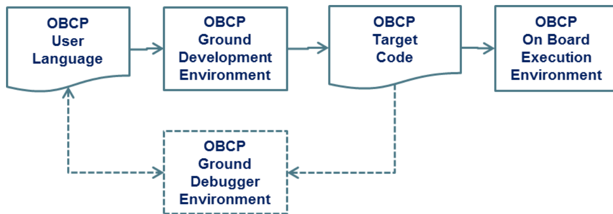

Figure 1: On Board Control Procedure Chain

The environment is therefore broken down into two parts, namely the Ground Development Environment and the On Board Execution Environment. Optionally, a Debugging Environment on Ground can complete the solution.

3.1 Ground Development Environment

The OBCP Ground Development Environment must ideally be integrated in the overall On Board Software Development Environment.

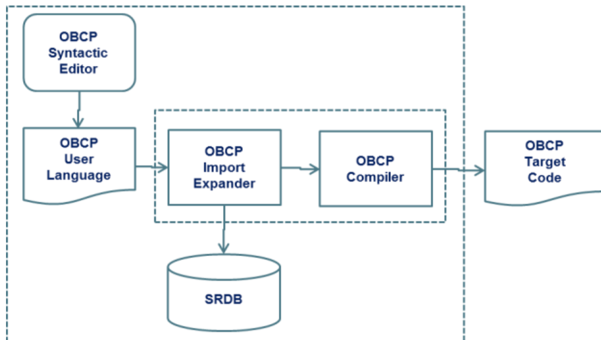

Figure 2: Ground Development Environment

It consists in a syntactic OBCP Editor to code the OBCP in high level user language and an OBCP Compiler that translates the high level source language in the target byte code.

The OBCP Editor can be any standard Python development environment.

The OBCP Compiler is provided as part of the MicroPython VM solution. Ahead of the OBCP Compilation, an “Import Expander” tool allows building a single OBCP file from multiple files importing one another – this preprocessing is necessary because dynamic import is not supported in the VM on board[4]. The tool also interfaces to the Satellite Reference Data Base (SRDB) in order to resolve symbolic naming by replacing the SRDB names used in the OBCPs by their corresponding value or address; such pre-processing prevents storing the whole SRDB on board or uploading it as part of the OBCPs.

3.2 Ground Debugging Environment

OBCPs must be exercised and debugged on ground before being loaded and executed on board.

With OBCPs written in MicroPython, the simplest, cheapest and widely available debugging environment is the native Python 3.4, supported by adequate stubs to replace any interface between the OBCP and the OBSW. In place of standard Python 3.4, it is also possible to build a native MicroPython interpreter. However, as those native interpreters are not fully representative, such tests need to be completed with adequate test campaign on the target On Board Execution Environment, either on an SVF or on a hardware test bench.

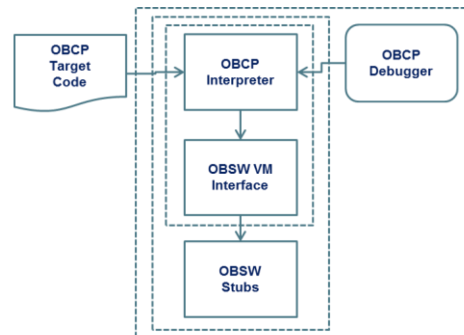

Figure 3: Ground Debugging Environment

3.3 On Board Execution Environment

The OBCPs can then be uploaded on board for execution. The On Board Execution Environment consists in an OBCP Interpreter surrounded by OBSW services. The OBCP Interpreter is implemented by the MicroPython Virtual Machine (VM). The surrounding OBSW Services that constitute the On Board Execution Environment are the OBCP Engine, the OBCP Storage, the OBCP TM/TC Interface and the OBCP VM Interface.

[4] The loading of modules is a functionality supported by MicroPython that has been de-scoped from the qualification. It is not used by the OBCP Engine.

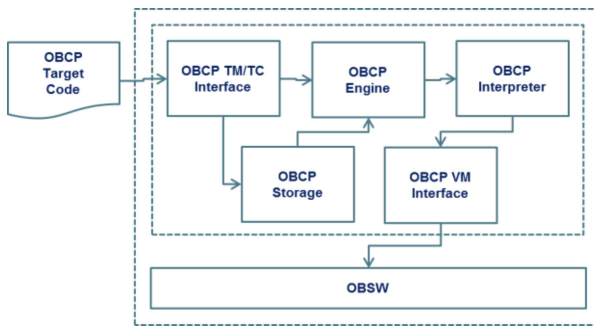

Figure 4: On Board Execution Environment

These software components are described further down in this paper.

a. Virtual Machine

The MicroPython Virtual Machine is a C library. It is purely passive, with one entry point, and it must be provided with a pointer to the bytecode to be executed.

The VM executes bytecode and is highly optimised for speed. Since the operations are dispatched to the runtime, most operations of the VM executor are short, the exception being those that deal with MicroPython exception handling. The VM implementation of exception uses C language's non-local-return (NLR), which is essentially a stack of state buffers where the VM can quickly return in case of exception.

When executing a script, the VM can be controlled by external entities through two mechanisms:

- A hook mechanism that regularly calls a user-defined C function while executing the MicroPython script.
- User-defined extensions to the MicroPython language can be developed. When called from a MicroPython script, a user-defined C function is also called.

These two mechanisms are sufficient to integrate, control and monitor the execution of OBCPs in an On Board Software.

Beside the storage of the bytecodes, the execution of MicroPython scripts requires memory. Several kinds of memory are used by the VM (on top of the static code and data areas):

- A dedicated heap to allocate and free dynamic Python objects when needed.
- A Python stack to allocate objects and data corresponding to a local context.
- The C stack required for the execution of the VM itself.

The VM executes a given bytecode from the start to the end (normal exit or uncaught Python exception) unless an abort is required from the OBSW. Several instances of VM can coexist, each one having its own task, state and memory. This allows running multiple OBCP in parallel, possibly with different priorities (see *OBCP Engine* below).

The MicroPython objects are represented by a word. Depending on the representation choice, a word may represent a pointer to a MicroPython object structure, a signed integer with a defined range, an interned string or a floating-point number. The object representation also has impacts in term of performance and memory usage. For instance, the shortest object representation using only 32 bits cannot hold a 64-bits floating point value; hence such an object needs to be allocated on the heap.

b. OBCP TM/TC Interface

This component implements the standard space-to-ground TM/TC interface of the OBCP Interpreter as per ECSS-E-70-41A standard [2], with an additional telecommand to upload an OBCP from a file.

As defined in the ECSS-E-70-41A, the PUS Service 18 provides standard service requests and reports for uploading OBCPs, controlling the execution of these OBCP and monitoring their status. To this end, it interacts with the OBCP Storage and the OBCP Engine. The improvements brought by the revision ECSS-E-70-41C are mainly to be found in the alignment with the ECSS-E-ST-70-01C, in the upload from files and in the definition of aggregated commands that can be used as actions.

c. OBCP Storage

The OBCP Storage is in charge of handling the organisation of OBCP accesses in memory. It manages a collection of OBCPs potentially located on different memory areas (RAM, EEPROM…).

d. OBCP Engine

The OBCP Engine is in charge of controlling and monitoring the execution of the OBCPs. It allows to start/stop suspend/resume/abort… the execution of an OBCP as required by the ground TC. It also makes observable the current status (OBCP state, step being executed).

The OBCP Engine allows for concurrent execution of several OBCPs. As already indicated, one VM only executes a single OBCP at a time. However, there can be several VM instances, each in a dedicated RTEMS task. Such a task has no specific real-time constraint and does not need to be synchronized with the rest of the OBSW. It is also not possible to anticipate its CPU usage so it should be created with the lowest priority in the system.

The OBCP Engine interacts with the VM through the hook mechanism and via the user-defined extensions to the MicroPython language to indicate that a

suspend/resume/stop request has to be executed. In case of a stop/abort, a system exception is raised in the MicroPython script, leading to its ending.

e. OBCP VM Interfaces

The OBCP Interpreter must provide the OBCPs with access to selected services of the underlying DHSW. User-defined extensions to MicroPython language are implemented in this component.

The user-defined extensions are as required by the OBCP ECSS-E-ST-70-01C standard. In particular, it provides the following functions to an OBCP as part of a built-in obcpe MicroPython module:

- Format and send TC;
- Check TC acknowledge;
- Generate a TM packet;
- Generate an event (triggering the event-report and event-action);
- Retrieve OBET;
- Retrieve OBCP parameters provided by ground TC;
- Read/write parameters from the Central Data Pool;
- Read/write authorized memory areas;
- Subscribe/unsubscribe to TM packets;
- Receive TM packet;
- Sleep;
- Wait synchronisation signal;
- Notify new step in OBCP.

Writing to memory or to parameters in the Central Data Pool from an OBCP can be considered too dangerous; it is therefore possible to inhibit such functionality.

4 VM QUALIFICATION

As already mentioned, the OBCP Engine is a classical OBSW development. It adheres to space standards for category B software. Its qualification is therefore not further described in this section, apart from the specific MicroPython VM fault containment measures that have been adopted.

The MicroPython VM has been qualified, taking into account the fact that the MicroPython VM is an existing library developed outside the space community. For this reason, several requirements from the ECSS-E-ST-40C standard have been relaxed or completed by alternative means. For instance, the code does not adhere to a strict coding standard, but is has been subject to static code analysis and code coverage. The classical approach for unit test has not been followed, but the same objectives are achieved through end-to-end testing.

The MicroPython VM qualification activities have encompassed many aspects and produced a set of plans, analyses, documents and tests, technical improvements, bringing the product toward a state compatible with stringent flight requirements.

4.1 Documentation

In line with the previous considerations, standard ECSS documentation has been produced for the OBCP Engine while a coherent set of documents have been produced for the MicroPython VM qualification, reflecting the ECSS processes that have been performed:

- Software Development Plan;
- Configuration and Data Management Plan;
- Software Verification Plan;
- Software Validation Plan;
- Software Requirement Specification;
- Software Design Document;
- Test Plans and Test Reports;
- Software User Manual;
- Software Verification Report;
- Software Release Document (incl. Software Configuration File).

Because the VM is reused software, the ECSS “Design and Implementation engineering process” has been considered mostly out of scope, together with the associated documents. All other ECSS engineering processes have been performed, including a reverse-engineering of the VM for the “Software Requirement and Architecture engineering process”.

4.2 Configuration Control

The MicroPython VM for LEON lives as a branch of the main MicroPython github project. In the frame of the qualification, however, a strict and dedicated configuration control has been put in place for this MicroPython VM for LEON. This is managed as follows.

To benefit from the Open Source approach and from the usage and implicit testing by a large community, the decision has been taken to stick to the maximum extent to the master branch. The other way round, the qualification effort and the improvements discussed above therefore also benefit to this main branch.

Agreed changes (improvements or bug fixes) that have been implemented in the MicroPython github master branch are ported on the MicroPython VM for LEON branch. Dedicated improvements or bug fixes may only concern this specific branch. At key points in the project, the corresponding code is extracted and placed under configuration of the qualification project.

This two-step approach is necessary because the MicroPython developers are not the same as the qualification team. It enables the MicroPython overall project to progress and improve without strict constraints, while ensuring that the qualified product code is well under configuration control. Last but not least, it does not cut the MicroPython VM for LEON branch from the master branch, so that future evolutions of MicroPython can still be easily ported in the qualification project.

4.3 MicroPython Configuration Selection

The MicroPython VM supports a large subset of Python 3.4. It can be configured through the selection of options amongst more than a hundred macros allowing for various trade-offs. For the purpose of the qualification, the scope has been reduced and functionality such as complex numbers or print formatting has for instance been excluded.

For the MicroPython object representation, two configurations typical of the OBSW needs have been selected and qualified:

- One with 32-bits object representation, providing the smallest memory footprint. However, this representation requires dynamic allocation on the heap when floating-point values or integers above 2^31 are used;
- One with 64-bits "NaN boxing" object representation, most efficient when using floating-point values. This technique exploits the unused bits of the floating-point representation of Not-A-Number in order to identify whether the object is an integer, a floating-point value, a pointer or an interned string. A key advantage of this representation lies in the fact that it does not require any Python heap allocation for floating-point numbers.

4.4 Technical Improvements

As part of the qualification activities, several improvements have been brought to the MicroPython VM:

- Previously sole C stack has been divided into a C stack and a Python stack. This gives more control on the stack usage and prevents stack corruptions;
- Identification of recursion in C functions, whether direct and indirect, followed by rewriting to eliminate or control such recursion (see below);
- Addition of several configuration macros, in order to reduce the code to the minimum needed by qualified MicroPython features;
- Addition of an entry-point function to start execution and report details of potential uncaught exception in a dedicated C structure;
- Improved monitoring of memory usage (heap, C stack and Python stack);
- Python exception handling implemented with NLR mechanism relying on LEON software "flush register" trap 0x83, implemented in standard RTEMS, was removed from qualified ESA RTEMS and therefore has to be reintroduced in the MicroPython VM;
- Improvement of the memory usage for the 64-bits "NaN boxing" object representation. It now supports 32-bit integer values (it was only supporting 31 bits before without heap allocation).

4.5 Code Analysis Activities

Static code analysis has been performed on the MicroPython VM code to identify potential threats and bugs, using PolySpace. The discovered issues have all been fixed or properly justified.

Code coverage measurement has been performed when executing the MicroPython VM qualification tests, and non-covered code duly justified. A particular effort has been put on the MicroPython exception handling code.

Finally, the MicroPython C code is inherently recursive. The complete call graph of the code has been established, and all cycles identified. Moreover, for each cycle, the C stack usage is verified in order to prevent memory corruption (in case the remaining C stack is too low, the running OBCP is aborted).

4.6 Fault Containment

Special fault containment measures are included in the MicroPython VM and/or OBCP Engine design to avoid propagation of errors stemming from an OBCP or from the MicroPython VM. These measures are summarized below.

a. CPU Usage

As the CPU usage of a MicroPython VM cannot be anticipated (it depends on the OBCP), the design decision is to execute MicroPython VM into low priority tasks of the OBSW. These tasks are not cyclic and do not have real-time constraints. There can be several priorities defined for the VM; VM of the same priority are executed according to a round-robin scheduling (also known as timeslicing). The only risk is that high priority OBCP's could prevent low priority OBCP from executing.

b. C Stack Usage

When started, the VM is asked to use only a part of the available C task stack. The VM dynamically checks that its C stack usage remains below the allocated range. In case of problem detected by the VM, the OBCP execution is stopped with an exception. The margin to

be provided between C task stack and VM C stack can be determined by static analysis of the code (static stack analysis). This mechanism is implemented in the VM in order to prevent any issue with recursive C function calls.

c. MicroPython Stack Usage

Any memory required for the MicroPython OBCP stack is allocated in a dedicated memory area, specified when starting the VM. In case of stack overflow, this is detected by the VM and the OBCP execution is stopped with an exception.

d. MicroPython Heap Usage

Any memory required for the MicroPython OBCP heap is allocated in a dedicated memory area, specified when starting the VM. In case of memory shortage, this is detected by the VM and the OBCP execution is stopped with an exception. As the allocation of memory in the heap is not deterministic in term of CPU usage (dynamic allocation and free), there is a specific function to lock access to the heap to the OBCP. Typically, heap access can be granted during OBCP initialisation but locked once that stage is completed.

e. Interface with OBSW

The MicroPython VM and the OBCP VM Interface perform preliminary checks of all input parameters, and raise an error before executing the requested function. This allows the VM to generate specific exceptions depending on the error (floating-point exception, memory exception, invalid type exception, etc.). As a general principle, all errors detected by the VM or by the OBCP Engine result in a MicroPython exception. Critical errors also immediately abort the OBCP execution.

f. Floating-Point Errors

Floating-point errors are mainly avoided by careful check of arguments whenever a floating-point operation is performed. However, they are nearly impossible to predict in all situations: floating-point exceptions can occur when calling mathematical library functions, but also during simple addition, multiplication or division operations.

For this reason, floating-point exceptions are treated specifically for the OBCP Engine. In case a floating-point trap (trap 0x08) occurs, if the execution context is a MicroPython VM task, then the corresponding OBCP execution is immediately aborted. If the context is not a MicroPython VM task, then the nominal exception handling is executed – typically a processor reset.

4.7 Test Approach

The MicroPython VM has then been submitted to intensive testing.

For the MicroPython VM, the following tests are executed:

- All relevant tests defined in the frame of the MicroPython VM master branch and MicroPython VM for LEON branch;
- C Python standard test suite for the core Python language (math functions, list, dictionary, etc.);
- Additional tests for improvement of the code coverage (in particular regarding exception handling)

As already indicated, the unit test objectives have been met through end-to-end testing. End-to-end testing means that a MicroPython script is produced and compiled. It is then uploaded for execution by the MicroPython VM. The output of the script is retrieved from the standard output (LEON UART line, but could be any other interface) and compared with the expected result.

All tests are automated. They are first executed on a LEON emulator (Spacebel Target Simulator). Then the tests are repeated in the EUCLID SVF (based on ESOC LEON Emulator) and in the EUCLID FUMO (hardware test bench).

A test run is performed on LEON emulator with code instrumentation in order to gather the code coverage measurements.

5 CONCLUSION

A new generation OBCP solution has been developed based on the Python Scripting Language and on the MicroPython Virtual Machine as well as on surrounding OBSW services, which implement the so called OBCP Engine.

The OBCP Engine development followed a classical OBSW development process. It adheres to space standards for category B software and it produced the standard ECSS documentation.

The MicroPython VM has been submitted to intensive testing and to static code analysis and the discovered issues have all been fixed or properly justified. Special fault containment measures have been implemented to avoid propagation of errors stemming from an OBCP or from the VM. Additional technical improvements have been brought in view of the integration and use on board. A coherent set of documents has been produced, reflecting the ECSS processes that have been performed for the MicroPython VM qualification.

The MicroPython VM has been integrated with the OBCP Engine and demonstrated in the EUCLID OBSW, on a LEON2 processor, with the ESA qualified RTEMS and Mathematical Library.

The activity results in two reusable building blocks that can be used jointly or separately:

- the MicroPython VM and
- the OBCP Engine.

EUCLID should be the first spacecraft to benefit from these new building blocks.

6 REFERENCES

1. ECSS-E-ST-70-01C – Space engineering – Spacecraft on-board control procedures, 16/04/2010

2. ECSS-E-70-41A – Ground systems and operations Telemetry and Telecommand packet utilization, 30/01/2003

2'. ECSS-E-70-41C – Ground systems and operations Telemetry and Telecommand packet utilization, 154/04/2016

3. "Porting of MicroPython to LEON platforms" David Sanchez de la Llana, Damien George, DASIA 2016

4. http://www.micropython.org