OpenIO Labs - ScriptML

ScriptML System Overview

UG401 (v2017.1) April 3, 2017

Revision History

Date	Version	Revisions
3/04/2017	2017.1	*Initial release.*

Table of Contents

Chapter 1: Introduction

This User Guide describes the salient features of the ScriptML system developed within OpenIO Labs for use within the Open Lab system. As we will see, ScriptML represent a paradigm shift in the way that computer languages are used and how they interact. With ScriptML, a user can create an application in one or more languages of choice. This application is converted into a serialised stream, referred to as ScriptML language tokens, which are transported to a destination node for execution on a target processor. The target processor does not need specific support for the originating language(s), only the support for the ScriptML intermediate language. It is from this functionality that the language name evolved, namely a Script Meta Language, or ScriptML for short.

In the remainder of this User Guide we will explore the design, operation and application of ScriptML, starting first with the structure of the OpenIO Labs System architecture, that will define the context for which ScriptML was designed.

OpenIO Labs System Architecture

The architecture of the OpenIO Labs system is explored in detail in *UG101 – OpenIO Labs System Architecture*, and briefly reviewed here. The OpenIO Labs system was designed to address the requirements introduced by advanced Machine-to-Machine Heterogeneous Networks (M2M Het-Nets) as described in UG101. Such networks are particularly prevalent in cases where both data retrieval and control are simultaneously required. There are a multitude of example deployments that can be considered, with two examples presented below.

1. Laboratory research environments which are requiring many types of sensors and actuators to be controlled and for data to be collected, tagged and stored.
2. Industrial process control systems in which real time-process data is analysed and stored, and, based on the analysis certain actions and events are triggered.

If we consider the example illustrated in Figure 1: Typical M2M Het-Net Deployment we can see a flavour of the types of sensors and devices that we need to interconnect. The sensors and devices will range from low complexity to high complexity, and in addition, the interfaces to these sensors and devices will range from relatively simple to complex.

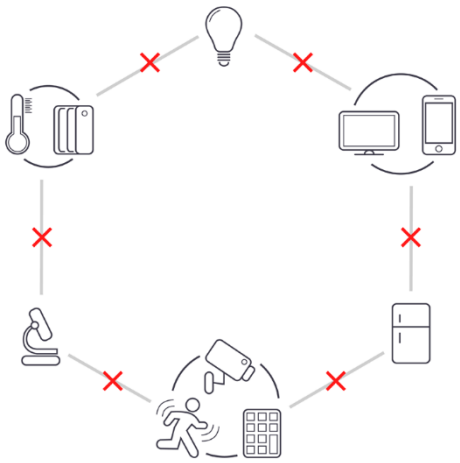

Figure 1: Typical M2M Het-Net Deployment

We can consider these sensors and devices to lie within different categories and with different interfaces as illustrated in Table 1: Examples of Sensors and Devices and Interfaces below.

Sensor / Device Type	***Example Interfaces***
Temperature, Humidity and other sensors	I2C, SPI and GPIO interfaces
Motors, Actuators, Switches and Relays	I2C, SPI and GPIO interfaces
ADC and DAC converters	I2C, SPI and parallel interfaces
Simple Instrumentation: Balances, pH meters	RS232 and USB interfaces
Standard Lab Equipment	USB interfaces with USBTMC stack
Complex Lab Equipment	USB interfaces with proprietary interfaces

Table 1: Examples of Sensors and Devices and Interfaces

What is clear from both the likely topology of the network and also the variety of the sensors and devices attached to it, is that the M2M-Het-Net needs to have the flexibility to deal with a range of devices, with a range of complexity. In these and in all other similar deployments, the requirements can be reduced to a simple set:

1. Allow the system to interface to a range of sensors, instruments and devices via a range of different interfaces and different interface protocols.
2. Place intelligence at the network edges to allow for rapid processing of data, reduction in

the transfer of data across the network, accommodate temporary network outages.

3. Utilise a common and industry standard to connect the network edges to the network core.
4. Deploy high reliability Cloud servers to offer high performance data acquisition, guaranteed availability with full network scalability.

The approach adopted by OpenIO Labs to meet these requirements is to connect devices and sensors via an Input/Output Interface (IOI). The IOI is an intelligent edge node that interfaces to the sensors and devices via the range of interfaces outlined in Table 1. The IOI is connected to OpenIO Labs Server via the industry standard Light Weight M2M (LWM2M) communications protocol. The interconnection between these devices is illustrated in Figure 2: OpenIO Labs Base Architecture below.

The structure and operation of these nodes and the interface protocol are considered in more detail below.

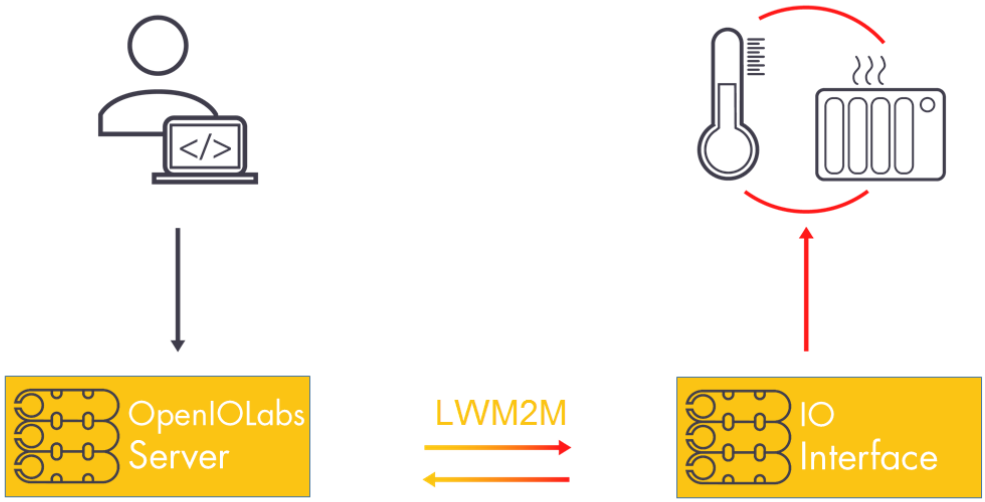

Figure 2: OpenIO Labs Base Architecture

OpenIO Labs Nodes and Protocols

The OpenIO Labs physical architecture is comprised of the three main entities illustrated in Figure 2. The Edge node (IOI), the interconnection communications network (using LWM2M) and the

OpenIO Labs Server.

IO Interface (IOI)

The OpenIO Labs IOIs are a family of edge node devices that are designed to interface to external devices and sensors. Each IOI provides a range of interfaces, with some examples of the interface types presented in Table 2: Example IOI Interface Types below.

Interface	Description
ADC	Many sensors and devices such as temperature sensors or humidity sensors interface via an analogue signal. The IOI provides an analogue interface via an ADC, processes the input signal and passes data to the OpenIO Labs server via the LWM2M interface.
DAC	Some devices such as motors, LEDs, heaters require an analogue signal to allow control of their operation. The IOI provides analogue control of the DACs via the LWM2M interface from the OpenIO Labs server.
I2C	There are many devices and sensors that utilize the simple I2C serial interface [Ref 4] between the IOI and the device. The I2C interface links the sensor or device to the IOI and then the LWM2M protocol links the IOI to the server.
SPI	The Serial Peripheral Interface (bus) [Ref 5] is another form of simple interface from the sensors and devices to the IOI. The IOI implements the client part of the SPI interface, interacts with the device and passes data to/from the server.
GPIO	The General Purpose IO interface [Ref 6] provides a simple binary interface between the IOI and the sensors and devices. Output devices as simple as an LED or as complex as a motor can be controlled via the GPIO. Input devices such as light detectors can also be interfaced to the IOI via a GPIO interface. The GPIO interface can be used in a single bit (wire) or multiple bit (bus) configuration.
UART	Universal Asynchronous Receiver / Transmitter [Ref 7] is a well established serial protocol interface that can be used with a range of input and output devices ranging from printers, GPS modules and PCs. The IOI provides the UART interface to the edge devices and links them to the server. These edge devices can be used for either input (data extraction) or output (device control).
USB	The USB interface [Ref 8] connects the IOI to a range of USB compatible peripherals. For complex instruments such as oscilloscopes or signal

	analysers, the USBTMC protocol [Ref 9] can be used to allow the IOI to control and monitor these complex instruments.

Table 2: Example IOI Interface Types

In addition to the various IO protocols, each IOI also includes high speed processors based on a multi-core ARM CPU. The CPU allows for the control and management of the edge devices and sensors, provides edge processing as well as the interface protocol back to the OpenIO Labs server.

LWM2M Interface Protocol

The interface from the IOI to the OpenIO Labs server is via the industry standard Light Weight M2M interface protocol [Ref 10]. LWM2M is a simple and robust interface protocol that has been specified specifically to service the needs and requirements presented by the Internet of Things (IoT).

The key characteristics of LWM2M are:

- Robust – Designed to operate over a range of network connections and manage issues related to the quality of the connection.
- Secure – Security of the LWM2M connection is implemented in the transport layer. This security is in addition to the other security features in the OpenIO Labs system.
- Scalable – LWM2M was designed from the outset to be scalable from tens of nodes, to many tens of thousands of nodes.
- Simple – The implementation of the LWM2M protocols was defined to allow it to be ported to the most simple micro-controller based devices.

The LWM2M interface connects the OpenIO Labs server to the OpenIO Labs IOI edge devices. Both the server and the IOI include the LWM2M stack allowing inter-operability with other commercial edge devices and servers.

OpenIO Labs Server

The OpenIO Labs Cloud server can be hosted on a range of Cloud server platforms. The server runs a virtual Linux instance and implements most of its functionality with the Node.js family of libraries and tools. The server includes both an SQL and a NoSQL data base to store the various forms of data that are to be collected and saved. The server implements a range of security mechanisms to ensure the integrity of both the server, the user and the user's data. Data security is additionally guaranteed with regular database backups. Access to the server is via the LWM2M Machine-to-Machine protocol.

The server architecture is designed to be portable across a range of physical hosting platforms. A

standard Linux distribution is used as a basis for server. Currently the OpenIO Labs Cloud server is hosted on the Amazon AWS Cloud server architecture. AWS together with Amazon's Elastic Compute Cloud (EC2) allow for a highly scalable architecture in which CPU resources, memory and data storage can be scaled according to the requirements of the deployment.

Chapter 2: ScriptML Overview

Background to ScriptML

The OpenIO Labs system, like all M2M-Het-Nets is the interconnection of a large number of devices. Each of these devices is used to either control experiments or processes or to collect, process and push data back to a server for either further analysis or storage.

In many example deployments, there is a need to add edge processing to the system. By edge processing, we mean the ability to process raw data and make process decisions on that data at the edges of the network, in the devices, rather than pushing the data back to the server and having the server perform the processing.

Within the OpenIO Labs system, we refer to this edge processing as the "application logic", the functions and processes that we need to perform on the data as it is collected, or to drive the edge devices based on the collected data or a set of algorithms defined to achieve a specific system objective. To achieve these objectives, it is recognised that a flexible method of implementing the application logic is the key to bringing the whole system together.

The specification of a framework for defining and implementing the application logic through the use of standardised languages, protocols and procedures is seen as a complex and difficult task to orchestrate across the whole M2M industry. The approach, taken by OpenIO Labs, was not to define such a structure and framework for the application logic, but rather a much simpler scheme in which we define an application logic transport protocol.

The application logic transport protocol is a mechanism that allows the application logic defined in one node or system in one language to be transported to a second system and executed on that second system within a completely different execution environment. If we take as an example, a server that is wanting to download an application logic executable to a device somewhere in the system. The server will accept the definition of the application logic written in some high-level language (as an example Python or Java). An encoder within the server will convert this high level application into an interim transport language, the language "tokens" are then transported to the edges devices using the LWM2M transport protocol and then executed on the edge device using a language decoder, or interpreter.

To achieve these goals, OpenIO Labs have defined ScriptML. ScriptML is a Meta Language that is an exact encoding of the logic defined in a source language. The mechanisms on how this achieved is described in the next chapter, but essentially consists of the application logic represented as a sequence of language independent tokens.

NOTE: ScriptML is referred to as a Meta Language *because the end user will never directly encode the application logic using ScriptML tokens. The ScriptML tokens are derived from the source languages and are executed within the decoder in the edge devices.*

The benefit of implementing the Application logic using ScriptML is that the detailed definition of a complex application framework can be avoided and instead the easier problem of defining the syntax, grammar and transportation of the ScriptML tokens can be done.

The simplest example is illustrated in Figure 3: Example of a Simple ScriptML Deployment below, in which we have the Server interacting with the edge devices attached to the OpenIO Labs IOI. The application logic encoded in the server is transported to the IOI via the LWM2M transport protocol and where the decoder interprets the application logic and implements the desired functions.

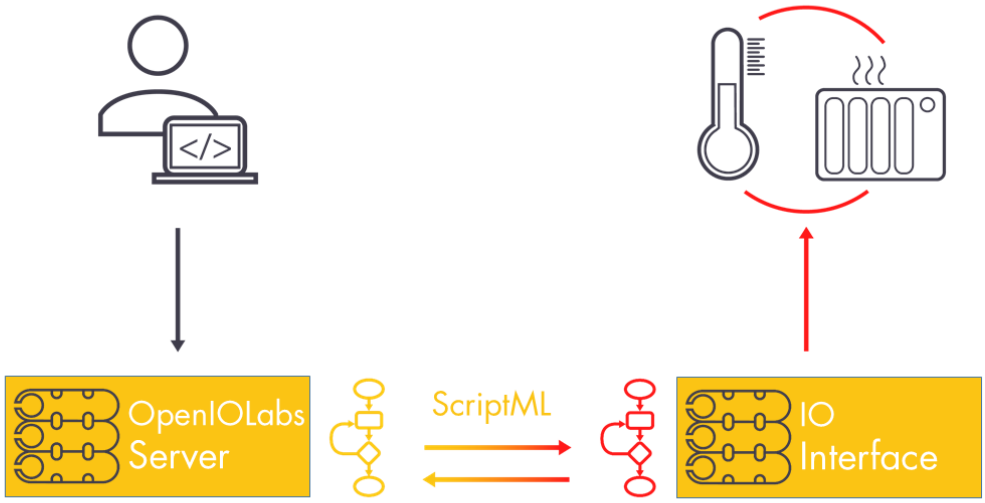

Figure 3: Example of a Simple ScriptML Deployment

A more elaborate deployment is illustrated in Figure 4: Example of a Typical ScriptML Deployment below. In this example we have a number of edge devices (temperature monitoring, light control, complex instrumentation and devices) connected to a set of IOIs (not shown). The ScriptML application logic encoding system ensures that all of these devices are managed and controlled

within the overall performance objectives for the system.

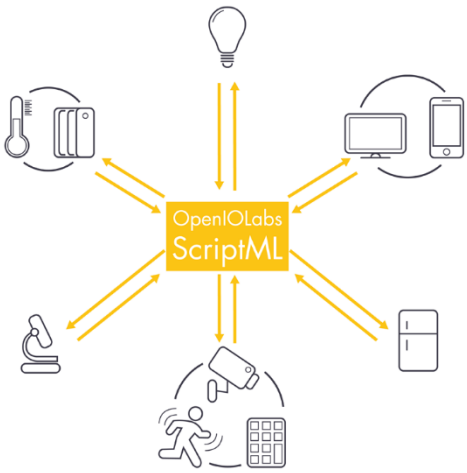

Figure 4: Example of a Typical ScriptML Deployment

In the following chapter we will turn to consider the internal structure of the ScriptML application logic transportation protocol, consider some of the details of the implementation of ScriptML, the encoder in the server and the decoder in the IOI client.

Chapter 3: ScriptML Architecture

The high-level operation of ScriptML is illustrated in Figure 3. The user of the system desires to create certain application logic to control and operate the edge devices attached to the IOI.

ScriptML Features

Following an analysis of the requirements introduced by the support of the application logic for M2M-Het-Nets, a minimum set of base features were defined for ScriptML. These features that the end user will use to implement their M2M-Het-Nets using ScriptML are listed below:

ScriptML Feature	Feature Description
Event detection	The ability to be respond to an event. The events could be the result of a reading from an edge device (e.g. temperature over threshold), or the result of some computation within ScriptML (e.g. an FFT analysis of a received signal and the detection of signals of interest). In all cases, the event can be used to trigger other processes, trigger a report to the server, or trigger the activation of some hardware response (such as the opening of a valve, the activation of a heater etc.).
Data collection	The ability to collect data from edge devices, store the data locally within the IOI and periodically (based on some algorithm) push the data to the server.
Data processing	The ability to process the data locally using ScriptML. The type of processing is limited only by the creativity of the end-user. Examples of the types of processing maybe statistical analysis of the received data, filtering functions, signal processing functions etc.
Logic control	As ScriptML was created as a "super-set" of modern processing languages, it encompasses all aspects of these languages including support for loops, switch statements, if-then-else statements etc.
Advanced processing	On occasion the capabilities of the native scripts may not be sufficient to meet tight timing constraints, for instance, in these cases, ScriptML was designed to fully interface with hardware and software accelerators. In the OpenIO Labs system the hardware accelerators are implemented using an FPGA device, and software accelerators within CPUs. The interface between ScriptML and these hardware and software accelerators is transparent to the end-user, and appears as a simple function call.

ScriptML Feature	Feature Description
Notification	Notifications are used by ScriptML integrated with the LWM2M protocol to allow the server to be notified of certain events. These notifications can be controlled from within ScriptML and could be purely time-based or based on some algorithm inherent within the scripts.
Reporting	Reporting functions allows ScriptML to be used to send data collected or processed from the edge devices via the IOIs to the server. The reporting can be continuous (e.g. streaming data from cameras) or intermittent (results from experiments when the experiment is completed).
Alarms	Often it is necessary to alter an end-user in the case of an alarm condition. The alarm condition could be due to a failure in the system or may-be due to an event that has occurred in an experimental system or process.
Hardware interface	The hardware interface is the key aspect of the OpenIO Labs system. ScriptML provides direct access to the attached hardware devices. A range of different hardware interface protocols are described in Table 2 and encompass almost all forms of standardised interface. ScriptML hides the complexities of the hardware interface to the end-user and greatly simplifies the ease in which the system can use hardware of different types and complexities.
Concurrency	Most modern languages support concurrency to some degree or other. At the outset, ScriptML was designed around the support for concurrency. Whether the concurrency be via threading (e.g. Pthreads) or co-routines or multiple processes, ScriptML can support all aspects of concurrency in a manner that is simple for the end-user to use. The use of many thousands of concurrent strands of application logic can be easily supported.
Language support	From its inception, ScriptML was designed to support multiple languages, in fact the underlying architecture of ScriptML is based on the many features of these languages, from static typed to dynamic typed, object orientated to concurrency. Languages currently supported by ScriptML include C, Python, C++, Java and Golang. The benefit to the end user, is that they can work with the language that they are most familiar with, or which gives them the features that they desire.
Object Orientation	Object oriented design is a preferred design methodology for many users to allow for the implementation of structured and re-usable code.

ScriptML Feature	Feature Description
	ScriptML supports many of Object Oriented features such as Classes, inheritance and polymorphism.
LWM2M integration	LWM2M was selected by OpenIO Labs as the preferred communications protocol between the edge devices and the server. LWM2M was defined to allow for highly scale-able networks, which when combined with edge processing using ScriptML significantly reduces the burden placed on both the inter-connecting networks and also the server.
Multi-Concurrent-Language Support	In addition to supporting multiple languages, ScriptML also supports the use of Multiple Concurrent Languages (MCL). The support for MCL falls naturally out of the design of ScriptML. Once a source language is converted to the ScriptML tokens, the issues of the concurrent language support is eliminated. The benefit to the end-user is that the language can be selected based either on preference, the availability of a hardware driver or efficiency.
RESTful Interface	In addition to the LWM2M interface to the server, the OpenIO Labs system also supports a RESTful interface to the server. The RESTful interface can also be used by ScriptML. This feature means that a whole range of applications can be created that allow ScriptML to access other resources that can only be reached using the standard HTTP protocol.

Table 1: Summary of ScriptML Features

ScriptML Language Support

ScriptML supports a number of different target languages, and these languages currently include C, C++, Python, Java and Golang. The rationale for working with a number of differing languages from the outset is twofold:

1. Adoption of ScriptML as a working methodology in M2M-Het-Nets will be enhanced with the variety of languages the end-user can utilise.
2. The internal structure and operation of ScriptML can be designed in such a way as to make the future support for other differing languages more readily achievable.

Through an examination of these five differing languages a common internal representation of a "generic" language structure can be developed and used at the core of ScriptML engine. The key aspect of this investigation was to define a minimal set of core language features that ScriptML should support and then, in addition, add in support for specific language features that are not

deemed to be core to all languages, these can be though of as extensions. The aim clearly in this process was to minimise the extent of the extensions to keep the core as broadly applicable to all languages as deemed appropriate.

Although a detailed exploration of the internal structure for ScriptML is outside the scope of this guide, a companion document UG410 examines in more detail the internal design structure for ScriptML and some of the design decisions made when coming to its current internal representation.

The next chapter will consider the operation of ScriptML in more detail, however, we present here the key aspects of the ScriptML system:

1. Parsing
2. Normalisation
3. Language extension support
4. Abstract Syntax Tree generation
5. Serialisation
6. De-serialisation
7. Interpretation

Steps (1) to (5) are referred to as Translation and occur in the OpenIO Labs server and steps (6) to (7) are referred to as interpretation and take-place in the edge node of the system (the IOI in the OpenIO Labs case).

Regarding the support for multiple languages, apart from the definition of the generic language structure mentioned previously (the parser and normalisation stages) we also need to consider the language specific extensions that may not conform generically to the other languages, In these cases the support of languages extensions is achieved through the use of what is referred to as Intrinsics. The Intrinsics can be thought of as a ScriptML language feature that is added directly to the Abstract Syntax Tree, but whose intention is to implement very specific features. Aside from the Intrinsics to support language extensions, other uses of the Intrinsics are to support hardware specific features more succinctly, such as I2C bus support.

ScriptML Entities

In Chapter 4 we will consider the use of ScriptML in the OpenIO Labs system in more detail, in this section we will introduce the basic elements that will be expanded on there.

Figure 5 illustrates the key operational steps of ScriptML within the OpenIO Labs system. The functions of ScriptML are divided between the Server (Control Node) and the IOI (End-Point-Node).

The server is responsible for steps (1) to (5) described in the previous section, and the IOI for steps (6) to (7).

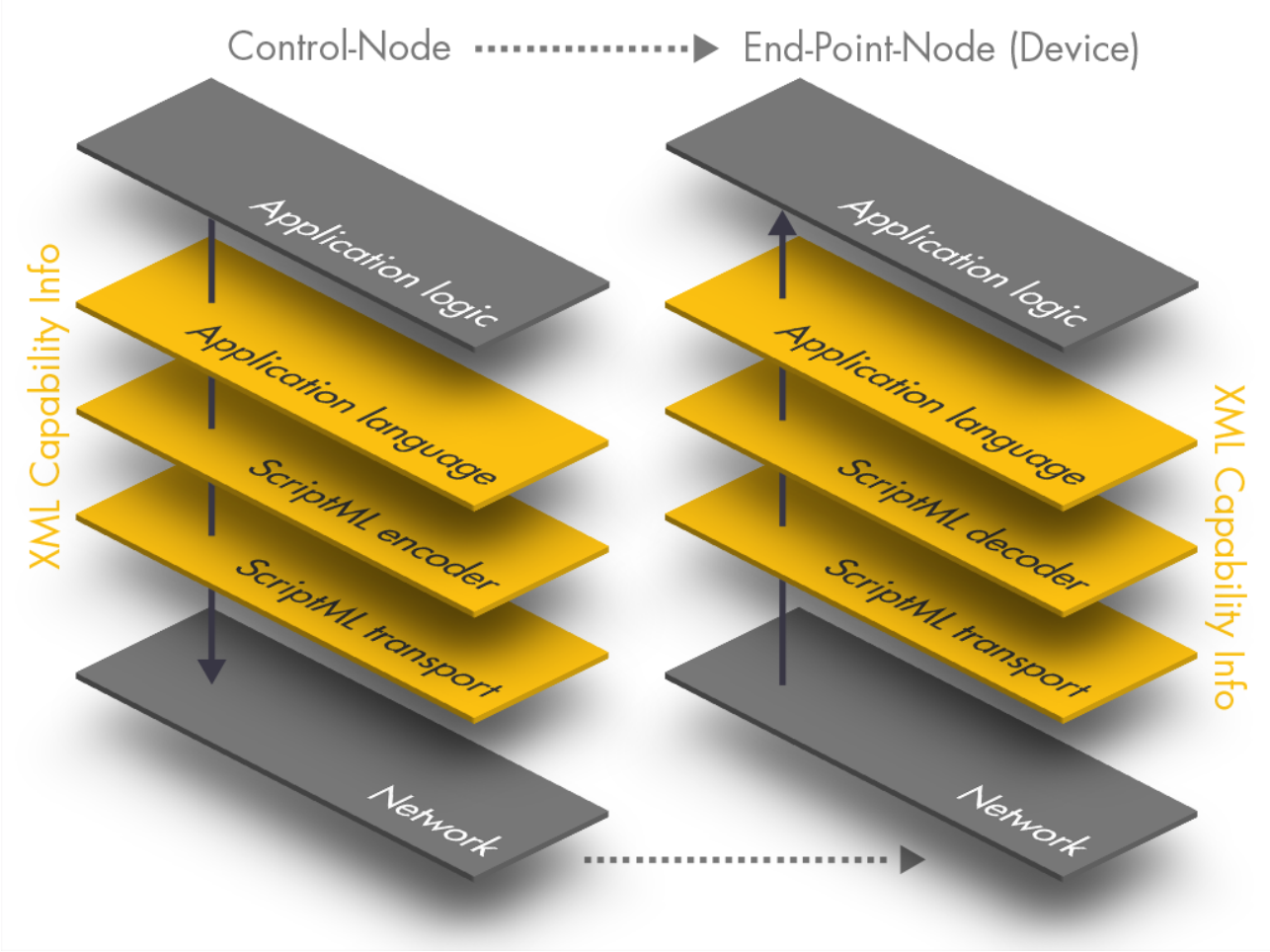

Figure 5: End-to-End ScriptML Operations

ScriptML Server Operations

The user will create and load the ScriptML scripts using the OpenIO Labs server. The "Application Logic" illustrated in Figure 5 represents the logic that the user wishes to implement within the system. The user will implement this logic using the language of their choice and load this onto the server.

The next step (illustrated as ScriptML encoder) is where the parsing, normalisation, language extensions and Abstract Syntax Tree generation occur. The output of this is the application logic represented in the form of an Abstract Syntax Tree. The next stage is the serialisation and encoding

which is represented as ScriptML transport in the diagram. The output of this is a sequence of ScriptML tokens that are collected together, formed into a JSON object and transferred using Internet tramsfer protocols to the client IOI.

Within the IOI at the network edge, a reverse process occurs. De serialisation and Abstract Syntax Tree formation occur at the ScriptML transport layer. Finally we have the ScriptML decoder, or more correctly, the interpreter. Within the interpreter, the ScriptML engine, will “walk-the-tree”, meaning it will traverse the application logic encapsulated by the ScriptML tokens and the Abstract Syntax Tree. The logic implemented within each node of the tree will be implemented within the interpreter and as a consequence, the desired application logic followed within the system.

Chapter 4: ScriptML Operation

ScriptML – Server Side

The high-level representation of the server-side ScriptML operations are illustrated in Figure 6 below. The operations within the ScriptML encoder / translator are illustrated and comprise a Scanner, a Parser and a Language Normalisation step. The input to the encoder is the application logic that the user has encapsulated in the input language of their choice. The output of the encoder are the normalised language tokens that are serialised ready for transmission across the network to the decoder in the OpenIO Labs IOI.

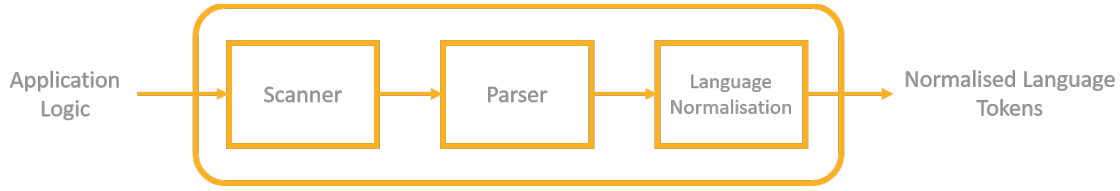

Figure 6: Server-Side ScriptML Architecture

From an implementation perspective, ScriptML is based on Flexc++ [Ref 12] and Bisonc++ [Ref 12]. Flexc++ is based on Flex, which in turn is based on Lex. Bisonc++ is based on Bison, which in turn is based on Yacc. Flexc++ is the tool that is used to generate lexical scanners which are programs that are used to recognize patterns in a source text file. The patterns are detected and turned into tokens that can then be used by the parser to generate an Abstract Syntax Tree. Bisonc++ is a parser generator for converting language grammar descriptions into C++ classes that are used to parse the defined grammars. The grammars for the languages are typically defined in a BNF like form.

Together Flexc++ and Bisonc++ allow the input languages to be parsed, tokenised and represented as an Abstract Syntax Tree. Within ScriptML an integral part of this process is the language normalisation step. With language normalisation the common parts of the different input languages can be parsed, identified and in many cases mapped to a simpler common language scheme.

An additional feature within ScriptML is the use of intrinsics. To understand what we mean by intrinsics, we can first consider the generic structure of a programming language as typified in Figure 7 below.

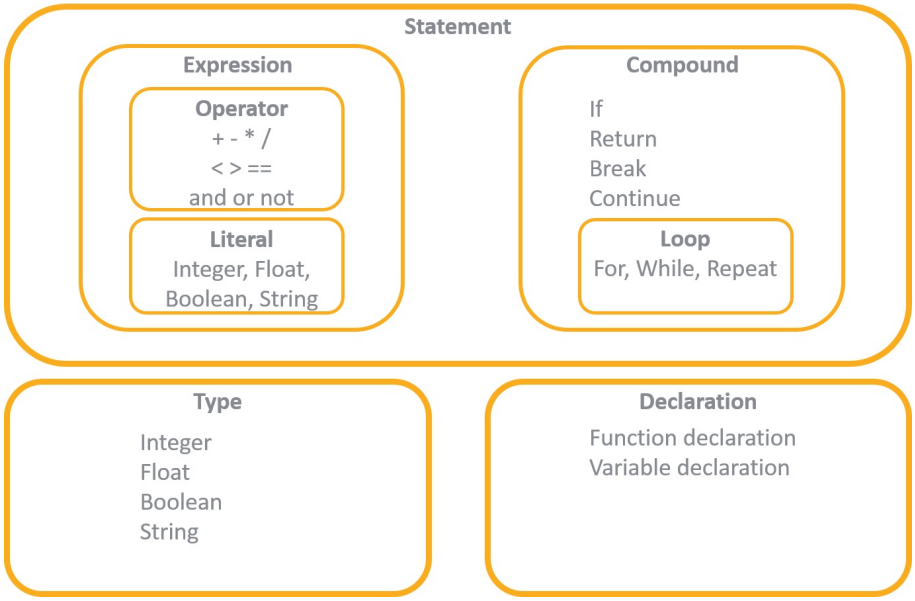

Figure7: Example Structure of a Typical Programming Language

At the top level the language is comprised of Statements, Types and Declarations. Flexc++ and Bisonc++ will be configured to identify these different language elements.

A Statement is the core of the logic used in the language. The Statement is sub-divided into expressions and compound Statements. Drilling further down we end at the keywords and operators such as {+, -, *. /} and compound statements involving keywords such as {for, while, if}. The combined scanner and parser will create an Abstract Syntax Tree which is made of nodes and terminals (the leaves of a node), with the nodes of the tree represented by keyword expression tokens, or the keyword tokens that make up a compound statement.

An intrinsic represents another type of node within the tree. We can see that the nodes are intended to represent the atomic units of a language, however, within ScriptML, the intrinsics might well represent a complex aspect of a specific language. An example for such an intrinsic could be an abstraction of the hardware interface to an I2C driver within a Linux system, for instance.

ScriptML, therefore, uses a combination of the scanner-parser, followed by the language normalisation step to define the Abstract Syntax Tree which is comprised of ScriptML language tokens similar to those illustrated in Figure 7 as well as the “special” tokens that represent these intrinsics.

ScriptML designed in this manner facilitates the conversion of many arbitrary input languages (such as C, Python, Java and Golang etc.) into a common serialise stream of ScriptML tokens.

ScriptML – Client Side

At the receiving end of the ScriptML system in the Client (or IOI within the OpenIO Labs system) there is a decoder / interpreter. An example of the functionality for this is illustrated in Figure 8 below.

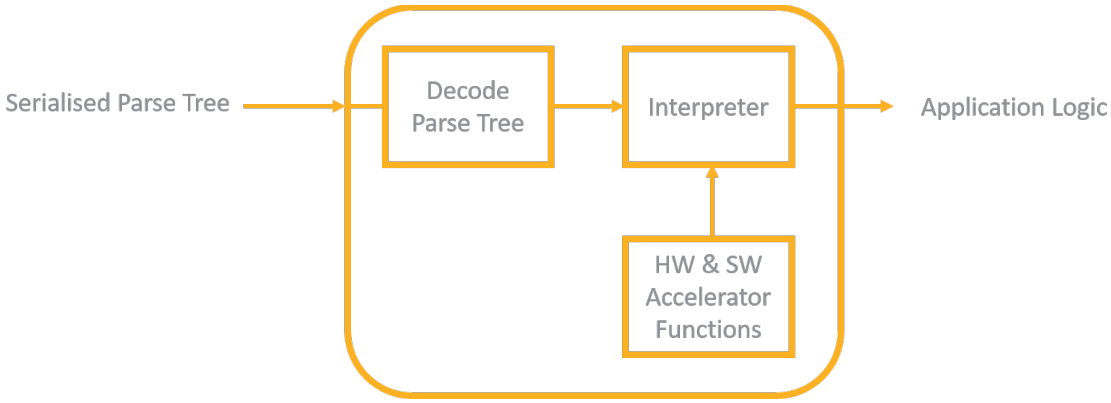

Figure 8: Example Client-Side Decoder

The client will receive a serialised version of the parse tree (that is, the Abstract Syntax Tree). The client will decode the parse tree and then "walk-the-tree" which means follow the logical flow of the application logic that is bound within the tree. This functionality is represented by the Decode and Interpreter blocks in the figure. In addition, within the OpenIO Labs implementation of ScriptML, Hardware and Software accelerator functions (or driver functions) can be linked into the interpreter functionality. The use of these additional functions and their configuration are indicated through the use of the intrinsics that were added to the ScriptML token stream at the server-side.

The complete End-to-End flow, therefore, consists of the user, creating some application logic that may well include access to hardware drivers such as I2C, GPIO, USB. The logic is defined through the language of choice of the end user before being converted into the ScriptML serialised token stream that is transported to the client-side. Within the client, the decoder performs the reverse operation. The interpreter is a matched pair to the scanner-parser and together with the intrinsics, implements the application logic that is run on the target CPU within the client.

Additional Resources and Legal Notices

References

The following documents and references are cited within this guide:

1. UG101 – OpenIO Labs System Architecture
2. UG402 – ScriptML C & Python Examples
3. UG403 – Using ScriptML in the OpenIO Labs System
4. I2C Interface – https://en.wikipedia.org/wiki/I2C
5. SPI Interface – https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
6. GPIO Interface – https://en.wikipedia.org/wiki/General-purpose_input/output
7. UART Interface – https://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter
8. USB Interface – https://en.wikipedia.org/wiki/USB
9. USBTMC – https://en.wikipedia.org/wiki/Virtual_Instrument_Software_Architecture
10. LWM2M – https://en.wikipedia.org/wiki/OMA_LWM2M
11. UG403 – ScriptML Internal Structural Design
12. Flexc++ – https://fbb-git.github.io/flexcpp/manual/flexc++.html
13. Bisonc++ – https://fbb-git.github.io/bisoncpp/manual/bisonc++.html