Webservice-Ready Configurable Devices for Intelligent Manufacturing Systems

Jiří Faist, Milan Štětina

▶ To cite this version:

Jiří Faist, Milan Štětina. Webservice-Ready Configurable Devices for Intelligent Manufacturing Systems. IFIP International Conference on Advances in Production Management Systems (APMS), Sep 2015, Tokyo, Japan. pp.476-483, 10.1007/978-3-319-22759-7_55 . hal-01431135

HAL Id: hal-01431135

https://inria.hal.science/hal-01431135v1

Submitted on 10 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Webservice-Ready Configurable Devices for Intelligent Manufacturing Systems

Jiří Faist and Milan Štětina

University of West Bohemia, New Technologies for the Information Society,
Department of Cybernetics
Pilsen, Czech Republic
{faist,mstetin2}@ntis.zcu.cz
http://www.ntis.zcu.cz/en

Abstract. This paper takes closer look at inner structure of an *eScop* RTU device, a highly configurable PLC-like embedded device developed in Node.js environment for low level physical layer of Intelligent Manufacturing Systems.

Keywords: structured text, control system, embedded device, Web Service, IEC 61131-3

1 Introduction

Embedded system for Service-based Control of Open manufacturing and Process automation (*eScop*) is a system that aims to overcome the current problems of production systems integration at shop-floor control level by semantically integrating embedded devices. The main idea behind the *eScop* architecture is to use embedded devices together with an ontology-driven service-oriented architecture (SOA). This approach allows the system to be automatically configured based on the ontology using embedded devices at the physical layer. This reduces setup time during which the manufacturing system stops.

2 *eScop* Device

eScop system is based on embedded devices – *eScop* devices. Profile of these devices is introduced in [1] where basic behavior and REST interface of device are described. Purpose of this article is to give closer insight into inner structure of *eScop* device.

Components of *eScop* Device (or *eScop* Remote Terminal Unit – RTU) are depicted in figure 1. Apart from the REST-API, *eScop* device consists of three components that enable the device to execute scripts written in Structured Text. Structured Text Language(STL) is one of five languages supported by the IEC 61131-3 standard and is designed for programming PLC devices.

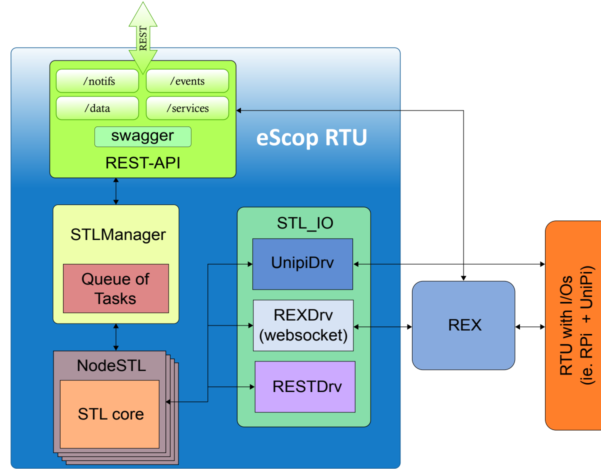

Fig. 1. *eScop*RTU device

3 STL Parser/Interpreter

One of the main goals to reach when developing *eScop* device was to create easily reconfigurable PLC like device. Configuration of the device is done through scripts written in Structured Text.

An STL Parser/ Interpreter was created using Bison and Flex. Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic or generalized parser [2]. Flex is a tool for generating programs that perform pattern-matching on given text [3].

Detailed description of STL Parser/Interpreter interface can be found in table 1. First part of the interface contains functions for manipulation with STL files in interpreter like loading new file to interpreter or removing file from interpreter. Second part of the interface contains functions for calling STL functions. STL interpreter currently also supports functions with parameters that are passed into the function by reference. These references can be created and destroyed by functions PushArray() and PopArray(). This mechanism is necessary for calling functions with arrays as parameters or with in-out parameters that serve as input and also as output.

The function for calling STL functions `CallFunction()` has an option to specify how many instructions in STL language should the interpreter process. This enables to suspend the STL function execution and gives the option to trace the program's execution and can be used for debugging purposes. If function's execution has not finished, `CallContinue()` function must be called to continue the execution.

Last part of interface contains functions for handling events. Event is raised when the STL Interpreter reaches instruction `EVENT(X)`, where `X` is an integer that identifies the event. Raising an event will cause the interpreter to stop exe-

cution and give the programmer opportunity to react to this event. This mechanism was later generalized for defining external functions. External functions are defined as functions that are only declared in STL file but they are implemented elsewhere. External function in STL file is declared by its header with specified parameters and with an instruction `EXTERN` in its body. This specific instruction is just a macro for raising special event that denotes external function execution. Example of definition of external function can be found in code snippet 1.

Table 1. STL Parser/Interpreter interface

File manipulation interface	
AddStl()	Loads new file into the parser and returns list of functions it contains.
RemoveStl()	Removes specified file from the parser.
ResetStl()	Resets parser. Used when errors occur.
Function handling interface	
CallFunction()	Calls specified STL function.
CallContinue()	Continues execution of the function if the execution have not finished by calling CallFunction().
PushArray()	If a parameter should be passed to the function by reference than this function must be called before actual call of the function to push parameter to the interpreter and obtain reference (pointer).
PopArray()	After argument passed in by reference is not used anymore than this function should be called to clean up.
Events and Extern functions interface	
GetEventFunction()	Returns function that the event was raised in.
GetEventParam()	Returns parameters of function that the event was raised in.
GetEventArray()	Returns parameters passed in by reference.

4 Integration of STL Interpreter with Node.js

STL Interpreter is a component where computing power is very important aspect. That is also one of the reasons why this component is implemented in `C` programming language. However for the development of *eScop*device was chosen the Node.js runtime environment mainly for its event-driven architecture and a non-blocking I/O API that optimizes an application's throughput and scalability. Now the question has been raised how to integrate these two environments together.

Node.js supports two options how to use libraries written in `C/C++` from JavaScript code. First of them is to use FFI (Foreign Function Interface) mechanism and that allows to call a function from one program in another program

```
FUNCTION readRelay:INT;
   VAR_INPUT
    id:INT;
   END_VAR
   VAR_IN_OUT
    value:BOOL;
   END_VAR
   EXTERN;
END_FUNCTION
```

Code snippet 1. Declaration of external function in STL.

usually even across programming languages. Using FFI involves loading shared library at runtime and marshalling the arguments of function from one program to another before each call. This mechanism is very simple to use but not very efficient.

The other option is to write a specific so called native addon in `C/C++` that wraps functions from desired library. This addon than can be used in JavaScript code in same manner as any other JavaScript object. This approach is about hundred times faster than FFI but developer needs to handle every conversion between JavaScript types and `C/C++` types by himself.

The approach using FFI might still be useful for calling functions containing heavy calculations and/or if these functions are not called very often but for the purpose of *eScop* device its performance is insufficient. That is why it was chosen to implement native addon for Node.js that serves only as a thin wrapper for STL Interpreter. This wrapper is denoted as NodeSTL component in figure 1.

5 STL Manager

A Service Manager Core component was introduced in [1]. This component is partially implemented by the component STL Manager that manages adding and building services simply by using NodeSTL component and its STL Interpreter to load STL file and expose functions defined inside as new services.

5.1 Task management

STL functions can be designated as one of three types of services defined by [1] as *process*, *operation* and *query*. Execution of services defined in STL files must be controlled to avoid deadlocks and collisions in STL interpreter. Thus a simple task management have been introduced.

Every service invocation creates a task that is assigned a priority based on its service type. The task is than pushed to a priority queue and executed once it is popped as the first in line. Task management algorithm also uses the concept

of specifying how many instructions should the STL Interpreter execute. That enables the manager to suspend task before it has finished and execute task with higher priority.

Number of instructions to execute depends on the priority of task. For example services designated as *query* should be very fast performing only small number of instructions. Thus they should have high priority and high number of instruction to be executed since these services probably should not be suspended at all. On the other hand services designated as *process* are usually long life services performing large number of instructions and thus should have small priority and small number of instructions to execute since these services should be suspendable in case that new tasks with higher priority occur.

To avoid deadlocks and collisions, it was decided to use one STL interpreter per STL file with assumption that every interpreter can have only one function running or in pending state. The STL Interpreter can be taken as a critical section and each executed task blocks every other task coupled with the same interpreter.

However this approach can lead to so called Priority Inversion Problem. That is a situation when task with higher priority is blocked by a task with lower priority. This creates a deadlock since lower priority task cannot continue its execution since a task with higher priority occurred. Thus The Basic Priority Inheritance Protocol suggested in [4] was adopted. Simply put this simple algorithm assigns the lower priority task with the highest priority of the tasks that are blocked by it. Due to this change of priority the lower priority task can be executed first and thus unlocks the interpreter for task with higher priority as soon as possible.

6 STL_IO

Since the *eScop* device should serve as PLC like device it must be able to read physical inputs and write to physical outputs. Raspberry Pi was chosen as a hardware platform in combination with UniPi.

Raspberry Pi is a low cost single chip credit-sized computer based on BCM2835 unit. It can be used in many ways as regular computer but in addition it also features GPIO (general purpose input/output) pins that serve as a physical interface between Raspberry Pi and other devices [5]. To make Raspberry Pi more PLC-like device a UniPi board was added. UniPi board was designed with Raspberry Pi in mind. It features 8 relays, 14 digital inputs, single channel 1-wire interface, 2 analog inputs, one analog output and Real Time Clock module [6].

For communication and for handling of physical signals were developed 3 drivers that a developer can use. STL_IO is a component that serves for registering and deployment of drivers to STL Interpreters. With these drivers can *eScop* device communicate with many already existing real world systems.

STL_IO drivers:

1. *UniPiDrv* - This driver serves for reading digital inputs and setting relays on UniPi board. It allows the system to read input signals and control physical processes.

2. *RESTDrv* - This driver allows to call various REST services by serializing inputs to JSON data format in request and deserializing JSON data from response. It allows the system to communicate with various systems on network.
3. *RexDrv* - This driver communicates with the system REX – Control System for Advanced Process and Machine Control [7]. It allows the system to manage real-time control processes in REX.

6.1 Drivers implementation

Drivers use the mechanism of calling external functions in STL Interpreter introduced earlier. STL file contains declaration of external function and its implementation lays in the JavaScript code. An example of external function declaration can be found in code snippet 1 and its implementation in code snippet 2. It is a function taken from *UniPiDrv* driver for reading relays on UniPi board.

The implementation in JavaScript is a function that takes 3 arguments. First argument is an array of inputs, second is an array of outputs and third is a callback that must be called after execution of external function is finished. The reason for using callback is that the function can be asynchronous as it is in our example in code snippet 2. Callback takes a return value of the function in STL as argument.

Few special attributes are set under the function implementation. They serve like annotations. First annotation is mandatory and helps to distinguish regular JavaScript functions from those implementing external functions. The rest of annotations are optional and allows to specify data types of external function parameters and return value in STL. This helps to ensure that both the STL code and JavaScript code use same data types right after compilation of STL file. This prevents from using incompatible drivers during STL function execution.

7 Pragmas

As it was stated before services should be divided by their type into tree classes: *process*, *operation* and *query*. Every service is than represented by a STL function. The information about the type of service must be presented in STL source code. It was therefore decided to introduce *pragmas* as a way to annotate STL source code. Example of service function annotated with pragmas is in code snippet 3.

Pragmas can be viewed as additional comments inserted into code but unlike comments that are ignored by the interpreter, pragmas are recognized and can affect the behavior of the program. The IEC 61131-3 only states that pragmas exist, they are delimited by curly braces ({ ... }) and contain attributes separated by comma but their definition is implementation specific.

Pragmas attributes:

- *name* – Specifies an alias to identify service. Use only if it is different than name of function.

```
STLExtern_Unipi.prototype.readRelay = function(inputs,
    outputs, callback) {
    unipi.readRelay(inputs[0].value, function(err, value) {
        if(err){
            console.log('Reading relay failed.');
            callback(-1);
        }else{
            outputs[0].value = value;
            callback(0);
        }
    });
};
STLExtern_Unipi.prototype.readRelay.isSTLExternFunction =
    true;
STLExtern_Unipi.prototype.readRelay.inputs
    = [ new STLParameter({type:'INT', name:'id'})];
STLExtern_Unipi.prototype.readRelay.outputs
    = [new STLParameter({type:'BOOL', name:'value'})];
STLExtern_Unipi.prototype.readRelay.rtnValue
    = new STLParameter({type:'INT'});
```

Code snippet 2. Implementation of external function in JavaScript.

- *description* – Should contain user friendly description of service.
- *type* – Specifies type of service (*process*, *operation*, *query*). This attribute can also contain type '*event*' that denotes that this function is not a service but an event.
- *schedule* – Specifies the scheduling of the service in case it is a *process*.
  - periodic – Service is called periodically by the device in the defined period.
  - continuous – Service is called cyclically without sleep between calls.
  - init – Service is called upon device initialization.
  - exit – Service is called upon termination of the device.
  - error – Service is called when error happens on the device.
- *period* – Specifies the period of calling the service in *ms*.
- *externFunction* – Specifies the identifier of external function in JavaScript if the name differs from the name of function declared in STL. This allows to use one external function from driver for multiple STL functions.

Pragmas can also be used to annotate parameters of function however no specific format was defined for it yet.

8 Conclusion

The purpose of this article is to give closer insight into inner structure of *eScop* device. These units' functionality is controlled by scripts written in Structured

```
1 {type='process',schedule='periodic',period=1000,
      externFunction="postService",description="myDescription"}
2 FUNCTION serviceName;
3     VAR_INPUT
4         {description="this is my input"};
5         value:INT
6     END_VAR;
7   .  .  .
8 END_FUNCTION
```

Code snippet 3. Example of function with *pragmas* annotation.

Text language. This article describes components developed for this functionality such as STL Parser/Interpreter for interpreting Structured Text code, STL-Manager for task management and collection of drivers for control of physical signals and for communication with other devices. *eScop* Device framework can be used for managing and seamless expanding of Intelligent Manufacturing Systems. Software was tested on Raspberry Pi and will be part of INCAS pilot application described in [8].

9 Acknowledgement

This work was supported by the ARTEMIS Joint Undertaking grant No. 332946 and by University of West Bohemia – SGS-2010-036.

References

1. O. Severa, R. Pišl, INDIN 2015 IEEE International Conference on Industrial Informatics *(not published yet)* (2015)
2. Free Software Foundation, *GNU Bison manual* (2015). URL `http://www.gnu.org/software/bison/manual/bison.html`
3. The Flex Project, *Lexical Analysis With Flex* (2012). URL `http://flex.sourceforge.net/manual/`
4. L. Sha, R. Rajkumar, J. Lehoczky, Computers, IEEE Transactions on **39**(9), 1175 (1990). DOI 10.1109/12.57058. URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=57058&tag=1`
5. Raspberry Pi Foundation, *RASPBERRY PI Documentation* (2015). URL `http://www.raspberrypi.org/documentation`
6. UniPi Technology, *UniPi technical documentation* (2015). URL `http://unipi.technology/wp-content/uploads/manual_en.pdf`
7. REX Controls. REX Control system. `http://www.rexcontrols.com/www/produkty.php?id_produkt=7` (2015)
8. P. Balda, M. Štětina, INDIN 2015 IEEE International Conference on Industrial Informatics *(not published yet)* (2015)