System Design of a Modern Embedded Linux for In-Car Applications

Conference Paper · March 2017

CITATIONS
2

READS
1,526

3 authors, including:

Murali Padmanabha
Technische Universität Chemnitz
12 PUBLICATIONS **26** CITATIONS

SEE PROFILE

embeddedworld2017
Exhibition&Conference
...it's a smarter world

System Design of a Modern Embedded Linux for In-Car Applications

Murali Padmanabha, Daniel Kriesten, Ulrich Heinkel
Chemnitz University of Technology
Chair for Circuit and System Design
09107 Chemnitz, Germany
<murp, krid, heinkel>@hrz.tu-chemnitz.de

***Abstract*—The state of the art in-car application area that utilizes Linux as an operating system, is the infotainment application. While this market is going to mature with Linux based systems in the next few years, there are other application areas which are over looked. Dashboard and instrument clusters, Head-Up-Displays, telematics and remote vehicle interaction are applications that can be built on Linux OS. In this paper, we present an approach and the corresponding framework for building a Linux system for in-car applications. Analysis of typical requirements and state of the art definitions and solutions resulted in TUC-L4IVA, a scalable Linux system for in-vehicle applications. Different system layers are realized into independent package groups using the Yocto build system. Mapping some or all of these layers to the derived application architecture results in a Linux system image for the respective application. An instrument cluster app is designed to demonstrate one of the use cases. The resulting system is implemented on TI am335x based BeagleBone Black. Testing methodologies such as Linux Test Project, Yocto package test and Yocto automated runtime tests are implemented to test the functional coverage, reliability and security of the derived system. This paper is concluded with the achieved state of the art definition, some of the short comings and possible future enhancements.**

***Index Terms*—Linux, Embedded, Specification, yocto**

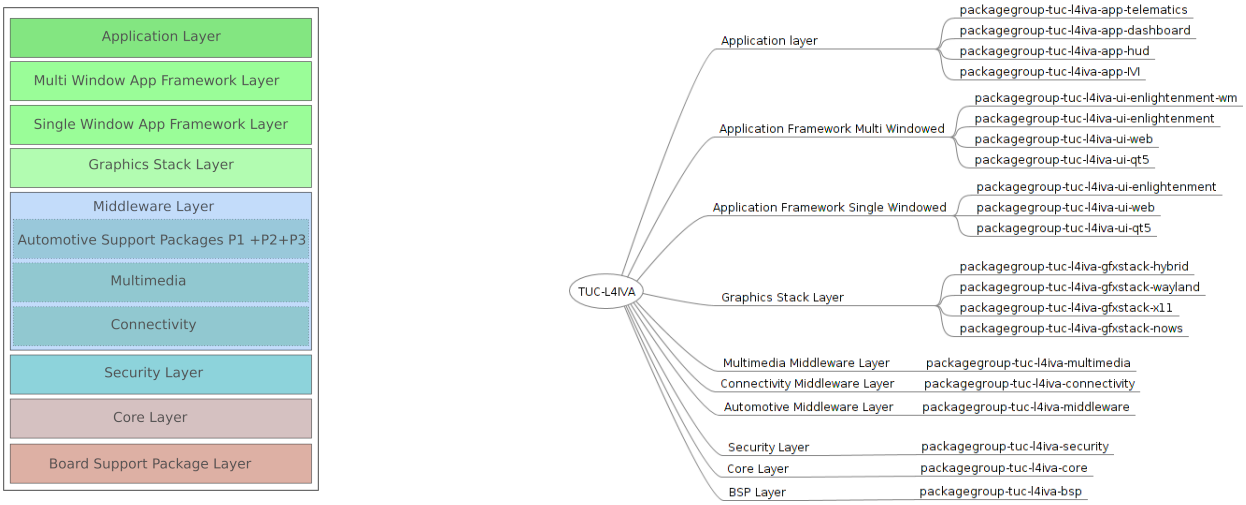
I. Motivation

Today, automotive manufacturers already including in-vehicle infotainment (IVI) systems based on Linux or derivative of Linux. This is one prove of acceptance of Linux for complex multimedia based applications. Since the features provided by most infotainment systems are similar to those provided by the mobile devices, stripped down versions of the operating system stack such as Android and Tizen are used for realizing such applications [1], [2]. A huge gap exists between the very low level critical applications implemented in traditional RTOS and the resource intensive infotainment systems featuring multimedia capabilities. This gap shows a challenging potential for both industry and academia. On full maturity of the real-time support on the Linux kernel, possibilities for Linux based automotive applications will grow tremendously [3], [4]. The potential to fill this gap is the a main motivation for the presented work.

This paper is organized as follows. After an overview on related work, we describe the basic idea of our approach in more detail. Then we discuss the actual implementation and its use in a case study. Finally, we summarize and discuss the current status of the work and derive next steps.

II. Related Work

The frequently referenced works on the subject indicate that several attempts have been made to achieve a Linux based infotainment system from existing operating system stacks or entirely from scratch. TIZEN-IVI in one of the Linux based IVI solution derived from TIZEN OS. It was customized primarily for IVI and resulted in a nearly monolithic stack with complex dependencies [2]. Automotive Grade Linux Foundation(AGL) initially developed the IVI and Heating, ventilation and air conditioning(HVAC)platform using Tizen-IVI OS but later announced to build a stack from scratch [1], [5]. GENIVI [6] also provided a demonstration platform based on Yocto which comprised of GENIVI complaint software components for integrating into other custom OS stacks. None of the afore mentioned solution was ready for use and AGL was fairly new at the time of execution of this work. This motivated us to learn from the existing work and derive an approach that enables the reuse of existing software components and achieve a non-monolithic system that we discuss in the next section.



(a) Architecture layers

(b) Layers and package-group mapping

Figure 1: TUC-L4IVA system architecture and layer mapping

III. Idea

The automotive domain provides possibilities for a wide range of applications, ranging from safety to luxury. Some of the application domains that could use a Linux OS stack are: IVI, Head-Up-Display, telematics, instrument cluster and connected cars [5]. The requirements for every application domain are slightly different. The aim is to provide a platform for all application domains using the same code base. To achieve this, it is important to understand the application requirements.

The operating system stack shall contain software and firmware components that provide the basic functionalities that every application would require, irrespective of the application domain. These components include the board support packages and the core user space components that boots up the hardware to a working state [7]. All other software components are integrated into the stack based on the application requirements.

Understanding these typical requirements helped to derive *TU-Chemnitz Linux for In-Vehicle Application* (TUC-L4IVA), a layer based stackable architecture. These layers when appropriately abstracted and contained, can be independently stacked together to achieve an architecture and hence a software stack for the application in focus. The architecture conceptualized from the requirements consists of several layers with board support package layer close to the hardware and the application layer abstracted away from it as seen in the figure 1a.

IV. Implementation

Realizing the conceptualized system architecture into software stack requires a build system that can automate most of the daunting tasks involved with compilation of software packages and satisfying their dependencies. One such state of the art build system widely used in the automotive industries is the Yocto project. With OpenEmbedded at its core, this build system not only provides automation capabilities, but also a comprehensive list of package recipes that can be easily incorporated into the resulting software stack.

Since the conceptualized layers are reusable across different application domains, these layers need to be independently packaged. The necessary software components of a layer were packaged into a package-group with at least one package-group for every layer as seen in figure 1b.

A. Integrating Core Linux Components

The software and firmware components that boots the system to a working state were made part of the BSP Layer and the Core Layer of the TUC-L4IVA architecture.

1) Boot-loaders: Today, there exists a number of bootloaders, each providing various secondary features that help during development. Some of the state of the art bootloaders used in embedded systems are U-Boot, BareBox and Gummiboot [8].

U-Boot is widely used and already supports various hardware platforms by default while BareBox is the next version of U-Boot built from the same source tree. BareBox claims to fix all quirks in U-Boot and provides a Linux shell like environment. Therefore the latter one was chosen.

2) The Kernel and Kernel Modules: Due to the long life cycle nature of automotive, around 10 years, it is ideal to use a kernel with long term support. The GENIVI and AGL specifications also emphasize on the use of long term kernels for automotive applications. Some of the SoC providers maintaining their own kernel development branch or provide alternate kernel sources optimized for real time applications. Support for use of mainline and manufacturer specific long term support kernel 4.1.X and 4.4 was provided.

3) Userland Software: *Glibc*, along with components such as *udev, systemd, busybox* and *dbus* were bundled to satisfy the core requirements of all architectures. Several other software modules were included in the core and BSP layer depending on the features provided by the underlying hardware and the configured distribution. These packages were added to the image by checking the features defined in *MACHINE_FEATURE* and *DISTRO_FEATURE*. Packages that were included depending on the machine features were provided from the BSP layer of the *meta-tuc-l4iva* layer.

B. Linux Middleware

To address specific needs of different application domains, the middleware layer is split into three sub layers that provide connectivity, multimedia and automotive specific services. The connectivity layer includes software components that provide telephony, navigation and network management services. The Multimedia layer provides audio, video and camera services and was therefore divided into three sub package-groups that can be individually added to the image. Some of the state of the art automotive middleware components are available as the Yocto recipe files through the meta layers provided by the *meta-ivi* and *meta-tizen* layers. Automotive Message Broker from the *meta-tizen* layer and the Diagnostic Log and Trace from *meta-ivi* layer were included in this package-group.

C. Graphics Stack

Systems with user interfaces require the necessary infrastructure for rendering graphics and hence provide the necessary user interface. The graphics infrastructure can be implemented in many ways depending on the nature of the graphics rendered on the display devices. Some of the possible infrastructure for graphics rendering implemented in the package-group *packagegroup-tuc-l4iva-gfxstack-** are discussed here. Figure 2 shows the various components of graphics stack and the interaction between them.

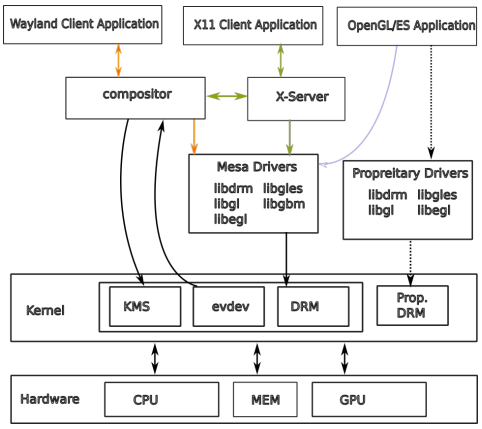


Figure 2: Graphics Stack Architecture

1) Direct Rendering with OpenGL: Applications that contain just one window with one drawing surface can efficiently utilize direct rendering by using the Open Graphics Libraries (OpenGL) or Open Graphics Libraries for Embedded Systems (OpenGL ES) libraries. This rendering infrastructure consists of the application that uses the OpenGL APIs and the OpenGL direct rendering infrastructure driver as seen in figure 2. The OpenGL Direct Rendering Infrastructure (DRI) drivers are implemented in the userland and communicate with the direct rendering manager in the kernel with *libdrm* driver. The Direct Rendering Manager (DRM) handles rendering of the graphics on the GPU, when available, providing hardware acceleration. Without windowing system, protocol engines, compositors and drivers associated, a lean graphics stack can be provided. The only drawback in using such a graphics stack is the use of application framework that supports the OpenGL/ES APIs for rendering.

2) Graphics Stack Based on Wayland: Wayland protocol along with Weston compositor provides the necessary modern graphics system for both embedded as well as personal computers. Unlike X11 based graphics stack which requires integration of several software packages, Wayland only requires a protocol package, a compositor package and the *evdev* package for capturing input device events.

3) Hybrid Graphics Stack: A modern day approach to adopt Wayland based graphics stack until fully developed, is to use the hybrid stack which uses both the Wayland and X11 protocols. The X-server and Wayland can both utilize a common compositor and the X11 clients can still render the graphics either through the compositor or by DRI. X-server now consists of *Xwayland* which provides a wrapper for the X clients to run on Wayland, enabling the hybrid graphics [9].

D. Application Frameworks for User Interface Design

State of the art embedded Linux systems with graphical interfaces are implemented using two classes of frameworks, native application frameworks such as Qt and Enlightenment foundation libraries and web application frameworks such as HTML5, Cascaded Style Sheets (CSS)3 and JavaScript with WebKit. The components for this layer are provided using the package-groups *packagegroup-tuc-l4iva-ui-** depending on the single or multi windowed application.

1) Qt Application Framework: Qt5 provides both native framework and QML which is Qt's replacement for HLTM5. It consists of three major components: Qt Essential, Qt Add-Ons and Qt Tools. The package-group recipe is structured such that the whole user interface could be modularized.

2) Enlightenment Application Framework: Enlightenment is now developed into a full blown window manager that provides a complete desktop shell. It is based on the Enlightenment foundation libraries (EFL), that can be used to build graphic rich apps with binaries less than

half the size compared to GTK+ framework. Enlightenment and EFL are used in many embedded platforms including the Tizen operating system based devices. Its wide acceptance in the embedded market emphasizes its suitability for in-vehicle applications that require graphical environments [10].

3) HTML5 based Application Framework: HTML5,CSS3 and JavaScript provide the state of the art hardware independent application framework for implementing a GUI. GUIs based on these web-thenologies can be easily ported for other platforms. A very popular and widely used rendering engine is the WebKit used in Apple's Safari and Google's Chrome browsers. A more advanced alternative used for the automotive applications is Crosswalk which is based on Chromium and Blink (both forks of WebKit). Qt also provides web-engines for rendering HTML based apps. Considering the wide use of WebKit and the rapid development of Crosswalk, both these packages were made available for this system design.

E. System Testing Framework

To test the complete Linux system, unit testing must be performed first, followed by integration and system testing. Testing individual package-groups of every layer ensures that they can be integrated with other layers and hence ensures compatibility and re-usability of the layers. The last stage of the testing was to integrate all required layers and perform integration and system testing.

1) Testing with Linux Test Project: The Linux Test Project (LTP) aims at providing the testing tools and methodologies to verify the reliability, robustness, and stability of the Linux kernel. Linux Test Project (LTP) already includes several automated test scripts that can perform kernel related tests, network related tests, user commands, etc.[11].

The Yocto project provides the recipes for the LTP as *ltp* that could be installed and executed on the target Linux system like any other binary package. Linux Test Project for Device Driver Test (LTP-DDT) is derived from LTP by Texas Instruments for testing the embedded device drivers designed for their SoCs[12]. These packages were made part of the development image by including the packages to the *IMAGE_INSTALL* variable.

2) Testing with Yocto: Yocto provides two methods of testing: package test and automated runtime testing. Package test (*ptest*) is a methodology for running test suites. It is a suitable tool for performing unit test since each software package can be bundled with the test scripts that can perform the required test on the binaries. This technique can also be extended to package-groups with custom test scripts that enables the testing and validation of the system on layer basis. [13] Unlike ptest which is performed locally on the target, the automated runtime testing is used for testing the image remotely. This method issues the commands to the target platform over the configured debug port and evaluates the results on the host machine. With the layered architecture of the TUC-L4IVA Linux system, the automated runtime testing of the images was performed on a combination of images built by adding available *tuc-l4iva-packagegroup-**** layers incrementally. This not only ensured working of the image but also the working of individual layers.

F. Deriving System Images

With the derivation of package-groups recipes for layers, recipes for the system images are created. A total of five image recipes is created which provides the Linux system image for headless system, dashboard/instrument cluster, HUD, IVI and finally a development image for application development. The image recipes are as listed below:

- tuc-l4iva-image-headless
- tuc-l4iva-image-dashboard
- tuc-l4iva-image-hud
- tuc-l4iva-image-ivi
- tuc-l4iva-image-dev

The possible image recipes for application domains are not just limited to the ones mentioned above but it just provides a sample set of implementations that could be easily scaled for a variety of applications. The images listed above are also incremental images, which means that the images at the bottom of the list are built on top of features already included on the images above.

V. Case Study

To evaluate the proposed architecture and the realized system stack, two applications were considered: Instrument cluster and IVI. The former system is discussed in details here. Texas instruments AM335X based Beaglebone Black embedded platform was used along with a 7" LCD touch screen for evaluation as seen in figure 4c.

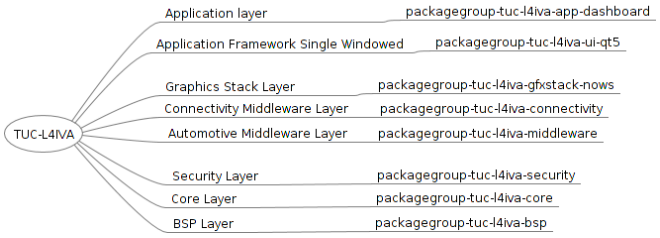
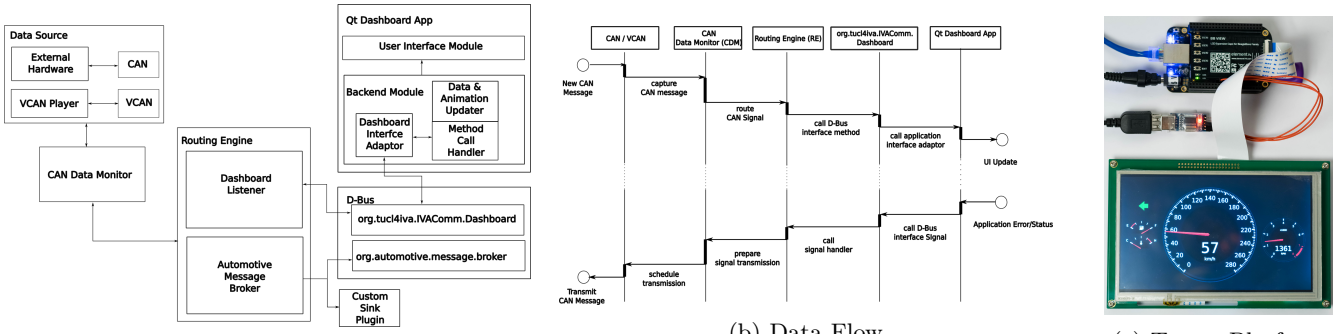


Figure 3: Mapping for Instrument Cluster/Dashboard

A. Instrument Cluster Application

The first step in preparing the operating system stack using TUC-L4IVA is to prepare the system definition with list of layers to be used. This translates to creating a Yocto image recipe with the appropriate package-groups. The relation between the architecture layer and the corresponding package-group is represented in figure 3. Ti is also necessary to integrate the appropriate BSP layer for hardware support.



(a) Application Architecture

(b) Data Flow

(c) Target Platform

Figure 4: Instrument Cluster Application

To demonstrate the working of the system image created, a full functional app was designed using Python and Qt framework provided by the system stack. The software architecture of this instrument cluster app is depicted in figure 4a.

The *tuc-l4iva-image-dashboard* was loaded and manually tested for its working. Pre recorded CAN messages were played on the VCAN module using a Python module that fed the data to the instrument cluster application. The message/data flow between various components is visualized in figure 4b. Application executed without any performance issues, animating the CAN signal values on the GUI. Table Ia shows the overall CPU utilization when running the instrument cluster application. The single core CPU on the target hardware was nearly fully utilized with only 16.8% computing time left for other processes.

Table Ia shows the RAM utilized by the Linux system with the instrument cluster application, DLT and AMBD running. About 67.5% of the total RAM was still available for other process. The total RAM shown here is less than the actual 512 MB available on board due to the Video Random Access Memory (VRAM) allocated to GPU.

Table Ib shows the resources utilized by various threads. TUC-L4IVA instrument cluster application consists of one Python module which implements the *CAN Data Monitor* and the *Routing Engine*. This module runs as one python process thread consuming nearly 50% of the CPU time. Qt based instrument cluster UI is less CPU intensive as it depends on the GPU for rendering but utilizes nearly 12% of RAM. D-Bus daemon also shares CPU time for transferring data from *Routing Engine* to the Qt dashboard app.

1) Limitations: Some of the limitations observed during the design phase are discussed below.

a) Graphics Stack: Using Mesa for the OpenGL graphics rendering has its limitations when the underlying GPU needs to be used for hardware acceleration. Mesa does not fully support hardware acceleration on all GPUs. PowerVR SGX found in the TI am335x series of the SoC is an example of such GPU due to the closed nature of the graphics driver. When vendor specific DRI is to be used, certain adaptations were required to override the library modules provided by Mesa and additionally include the vendor specific library modules. Replacing Mesa drivers with vendor specific drivers breaks the Wayland based graphics stack as the Weston compositor relies on the Mesa provided drivers.

b) QT Framework: Qt provides platform plugins to support the rendering on different windowing systems like Wayland or X11. This however still has the dependency on the underlying GPU for hardware acceleration and hence support for OpenGl. Qt modules such as QtQuick 2 requires hardware acceleration and EGLFS is the recommended platform for embedded Linux platforms [14]. With The X protocol C-language Binding (XCB) platform plugin, Qt applications can run on X11 based windowing system, but this requires that X11 has OpenGL ES support, which might not be the case on all embedded hardware.

c) Middleware components: Even though AMB was available, there was no standard interface that could be used out of the box for routing messages. CAN to d-bus message translation was not straightforward and required AMB interface modification.

B. In-Vehicle Infotainment

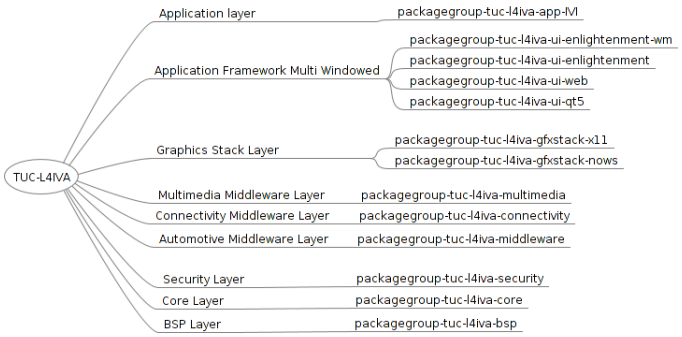


Figure 5: Mapping for in-vehicle infotainment

Another system image was generated for IVI application using the TUC-L4IVA layers. The enlightenment package was used as the application framework layer and a

CPU Utilization		RAM Utilization	
Time spent in	% CPU time	Parameter	Value
User space	67.8	MiB Total	384.6
Kernel space	15.1	MiB Free	259.9
Idle	16.8	MiB Used	38.8
Interrupt handling	0.3	MiB Buffered	85.8

(a) System

Threads	CPU %	Memory %
Data Monitor + Routing Engine	51.2	2.6
Python CAN Message Player	10.2	1.8
Qt Dashboard app (UI)	17.5	11.2
D-Bus Daemon	1.3	0.5
AMB Daemon	0.7	1.7

(b) Application

Table I: Application and System Resource Utilization

lightweight UI profile was designed to simulate an IVI interface at the application layer. The architecture and the mapped package-groups can be seen in figure 5.

VI. Conclusion

In this paper we have shown how to use the TUC-L4IVA architecture and state of the art build tools to achieve a standard system definition for varying application requirements. We realized the conceptualized architecture as independent reusable system layers that could be combined appropriately to generate the software stack on which the application can be developed. The layers were defined using Yocto package-groups with system components from various sources after careful analysis. We proposed a mapping approach to integrate the appropriate package-groups for a specific application domain. Using this, system image definitions were achieved for some of the possible application domains. As a case study, we developed an instrument cluster application and integrated it to the software system stack. The application was evaluated for functionality, performance and reliability using both manual and automated testing approaches. The testing methodologies that we proposed for both unit and integrated testing were evaluated. As an extension we also integrated another software stack for IVI application using enlightenment framework to demonstrate the scalability of the architecture.

Acknowledgment

The presented work is part of the project Generic Platform for System Reliability and Verification (GPZV) (03IPT505X), funded by the German Ministry for Education and Research (BMBF) within the "InnoProfile — Unternehmen Region" framework. Further information concerning this project is to be found here [15].

References

[1] A. G. Linux. (2016). Agl specification, [Online]. Available: https://www.automotivelinux.org/sites/agl/files/pages/files/agl_spec_v1.0_final_0.pdf (visited on 02/07/2016).

[2] T. D. Org. (2016). Architectural overview, [Online]. Available: https://developer.tizen.org/tizen/ivi/architectural-overview (visited on 02/07/2016).

[3] I. Automotive. (2016). Linux to take the lead in automotive infotainment operating system market, [Online]. Available: http://press.ihs.com/press-release/design-supply-chain-media/linux-take-lead-automotive-infotainment-operating-system-mar (visited on 02/07/2016).

[4] M. Graphics. (2013). Linux will not be adopted in automotive, [Online]. Available: https://blogs.mentor.com/embedded/blog/2013/01/04/%E2%80%9Clinux-will-not-be-adopted-in-automotive%E2%80%9D/ (visited on 02/07/2016).

[5] A. G. Linux. (2015). Agl architecture decission, [Online]. Available: https://wiki.automotivelinux.org/project-create-agl-distro/phase_1 (visited on 02/07/2016).

[6] G. Alliance. (2016). Genivi open source project, [Online]. Available: http://projects.genivi.org/projects (visited on 02/07/2016).

[7] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. O'Reilly Media, 2005.

[8] C. Hallinan, *Embedded Linux Primer: A Practical Real-World Approach*. Prentice Hall, 2006, ISBN: 0131679848.

[9] F. Org. (2016). X clients under wayland (xwayland), [Online]. Available: https://wayland.freedesktop.org/xserver.html (visited on 02/07/2016).

[10] E. Org. (2016). Efl, [Online]. Available: https://www.enlightenment.org/about-efl (visited on 02/07/2016).

[11] P. Larson, "Testing linux® with the linux test project", in *Ottawa linux symposium*, 2002, p. 265.

[12] T. Instruments. (2012). Ltp-ddt, [Online]. Available: http://processors.wiki.ti.com/index.php/LTP-DDT (visited on 02/07/2016).

[13] T. Y. Project. (2015). Yocto mega manual, [Online]. Available: http://yoctoproject.org/docs/current/mega-manual/mega-manual.html (visited on 02/07/2016).

[14] Q. Docs. (2016). Qt for embedded linux, [Online]. Available: http://doc.qt.io/qt-5/embedded-linux.html (visited on 02/07/2016).

[15] GPZV Projektteam. (Apr. 2012). Generische plattform für systemzuverlässigkeit und verifikation, [Online]. Available: http://www.tu-chemnitz.de/etit/sse/szee/gpzv/index.html.