ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 8, No. 1, 2023

EMBEDDED SYSTEMS MULTIMEDIA FRAMEWORK FOR MICROCONTROLLER DEVICES

Yaroslav Krainyk

Petro Mohyla Black Sea National University, 10, 68, Desantnykiv str., Mykolaiv, 54003, Ukraine
Author's e-mail*: yaroslav.krainyk@chmnu.edu.ua*

https://doi.org/10.23939/acps2023.01.043

Submitted on 17.10.2022

***Abstract*: The presented paper attempts to establish a generalized approach to the development of embedded systems multimedia applications. It is formalized in the form of a framework that defines rules and recommendations for a developer on how to implement specific pieces of software that work with multimedia data. The basis for the development process is the division of the system's functionality into stages with the following development of each stage. The framework also defines how touch sensor events may be elaborated. The proposed framework has been tested in a test scenario in an application with multiple stages. The results proved that the solution is feasible for multimedia applications (specifically, with graphics processing) and can be regarded as a generalized approach to the development of embedded systems with multimedia functionality.**

***Index Terms*: embedded systems; framework; graphics; multimedia; processing.**

I. INTRODUCTION

Recently, we have observed a huge leap in the capabilities of microcontrollers and other embedded hardware platforms. Previously, they supported a rather limited set of interfaces and had weak computational power. It can primarily be explained by the fact that they were oriented on maximizing the time of autonomous work. However, the continuous development of the Internet and formalization of the Internet-of-Things (IoT) and Cyber-Physical Systems (CPS) changed the requirements for embedded devices [1]. Connection to the network became a mandatory requirement for multiple applications [2]. This requirement raised the bar of functionality implemented and accelerated the development of advanced designs. This was the main driver for the appearance of multimedia interfaces and accelerators in microcontrollers.

Contemporary microcontrollers that belong to the performance line facilitate the peripheral set that includes modules for work with multimedia (graphics, audio, and even video). For instance, microcontrollers from the STM32 F4 series provide access to the graphical accelerator Direct Memory Access 2D (DMA2D) and support the connection of the display via parallel interface LCD-TFT Display Connection (LTDC). These modules hugely extend possible use cases for the development of embedded applications. In terms of performance, those modules are not top-notch enablers for data processing, however, they still can provide a decent level of functionality.

However, this poses a challenge to multimedia application development [3]. As the effective use of the available hardware resources is significant, it is necessary to realize the performance parameters of each module in both individual and collaborative manner. On the other hand, each scenario has its peculiarities from the point of view of data organization. Some applications may require active use of animation resources while others may rely on the relatively simple user interface with basic graphics. That fact implies differences in the development process. Therefore, this paper attempts to generalize the graphics management output process and organization of the software.

II. RELATED WORK REVIEW

The appearance of the LTDC module as a compound part of STM32 microcontrollers accelerated the development of software solutions for graphical user interfaces [4]. Some of them are supplied as open-source and available for use with examples and others are delivered on the terms of a paid license. They guarantee a rich user experience with the support of familiar desktop controls and information display. However, the main idea of these frameworks is to provide ready-to-use graphical components, e.g. buttons, text fields and labels, lists, etc. On the other hand, work with custom animation and graphical components has less support from these libraries [5].

The presence of these components commenced huge research of their capabilities and novel approaches to the hardware and software organization in the CPS. As embedded systems are tightly associated with the IoT paradigm, they also served as a driver for the extension of IoT solutions in the field of multimedia. The trend that can be observed is that many research works consider multimedia [3, 8] as an indispensable component for the development of IoT systems. For instance, many of the developments facilitate not only connectivity with an exchange of small chunks of data but also the User Interface implemented based on the display and touch sensor. Moreover, conventional protocols of IoT such as MQTT [6] and CoAP [7] are regarded as the source of multime-

dia data (even on the level of streaming). From the point of view of the multimedia system organization, such systems can be divided into two types [5]:

1) content demonstrating/interpreting systems.
2) content generating systems.

The first variant is supposed to serve as a receiver of multimedia data from the other devices. Typically, the data is generated by more powerful device and sent via network. The device receives it and decodes into the form that is suitable for representation in the form of the image/video content and audio data. The latter option works in a different way as the device is responsible not only for demonstration of that multimedia sources but also serves as a generator itself which implies much more computational intensity from the core. The paper deals with the second type of devices and develops the processing pipeline for multimedia generation and output on the screen. The peculiarities of the involved modules are taken into account according to ensure their efficient use.

The double buffering technique is regarded as an indispensable part of any graphics processing application regardless of the platform. It mitigates many undesirable problems that may appear on the display in the form of artifacts, unstable output images, etc. Hence, it is highly recommended to incorporate this technique into the framework to avoid possible problems with the output image. The proposed framework presumes that all operations with graphics are executed in the double buffer mode.

The graphics accelerators in modern microcontrollers typically perform a limited set of operations. The basis for all of them is a copy of one memory region to another which is the core of the Direct Memory Access (DMA) module. The module is enriched with the possibility to work with different pixel formats and further conversion, blending, smoothing, and blurring operations in comparison with general-purpose modules. These operations are suitable for two-dimensional raster processing. More complex pipelines with three-dimensional processing are too complex for the current revision of devices and are not available. Thus, the framework is oriented on the work with two-dimensional raster images to generate the output.

Many graphics frameworks for desktop applications and 3D engines have the notion of the stage as their fundamental basis. For instance, the JavaFX framework [9] uses a stage class that serves as a platform for other output. It works both with conventional user interface elements, e. g. buttons, list boxes, tables, etc. as well as with pure graphical components that are to be rendered by the engine. This approach was incorporated in the presented paper as the stage is also the basic entity for the whole rendering process organization. As the work is performed on a lower level in comparison to the JavaFX situation, the difference in the stage approach is significant.

In the work [10], the author established a network-based multimedia solution. The purpose of the node module was to receive data and transfer it to the corresponding output interfaces. No additional processing of the incoming stream was required. In contrast to the network-based approach, this paper attempts to conduct the system organization for the device that serves as a content generator and not a retranslator of the content. This demands a completely different architecture for software management which is the aim of the presented paper.

The framework [11] is a prominent example of ready-to-use graphical frameworks for embedded systems that procures graphical components and additional structures that can drastically facilitate the development process of the graphics application or some of its elements. It is advantageous of this software part that it includes solutions that are typical for the modern user interface. However, it is deficient in the intense animation scenarios that can be applied for development.

As it can be observed from the review, there is a steady trend in the development of multimedia-enabled systems. However, such systems can be regarded as reasonably complicated and resource-consuming. This is also valid for the software development process. Hence, the gap in the development strategies for multimedia applications is based on the embedded systems platforms. The proposed solution in this paper can be considered as a framework that is designed to generalize the development process.

III. OBJECTIVES OF THE PAPER

The framework presented in this paper sets the goal to organize the development of the multimedia functions of the embedded system according to the standard workflow. The framework may serve as a base for the development of complex applications that actively use multimedia resources in their work. From the developers' point of view, the framework should assist in reducing overall development time as all graphical elements can be realized similarly.

The contribution of this paper is in the following. The framework for the development of multimedia applications has been established in terms of hardware and software organization. The framework defines software architecture for individual components and provides recommendations on the implementation of typical scenarios.

IV. MULTIMEDIA FRAMEWORK DESCRIPTION

Firstly, let us provide an overview of the multimedia embedded system organization from the point of view of hardware components. The embedded system has the requirements to store graphical information in the energy-independent memory. The first option is the internal programmable Flash memory. However, this is not the best option because it limits the extensibility of the device. Flash memory facilitates up to 2 megabytes for high-performance microcontrollers. While this capacity may be enough for systems that utilize small bitmaps, usage of multiple full-screen bitmaps will result in a lack

of internal memory and may even cause problems with compilation. Hence, the solution is to use external memory to store blocks of multimedia data. The two main options for external memory to guarantee the necessary amount of memory are:

3) SD-memory card.

4) Quad Serial Peripheral Interface (QSPI) Flash memory.

While the two mentioned types of memories share in common the characteristic of data retention during a period when the system is not working (no power supply), the main difference between them is in the data provided for the system work. Apparently, the initial data for the SD-card case is written in advance as it can be used as an external drive for almost every device. The QSPI memory can be programmed with the necessary data during the flash process. Hence, all the files should be converted into a corresponding format that is suitable for the compilation and generation of a binary programming file. The retrieval of the data from those two types of memories is also different. For the QSPI, all the data is encoded in binary format. On the other hand, an SD-card can store conventional files with image, video, and sound data. However, that entails software processing of the files. Specifically, libraries to work with compressed image and video formats are compulsory to work with the data. As for the update procedure, it can be executed during the runtime providing that this functionality is implemented on the system level.

The system under consideration should also facilitate at least a display with a touchscreen sensor of any type. The display's resolution is dictated by the supported formats of the microcontroller output system. Typically, extremely large resolutions are not supported, however, it is possible to find devices that can control up to 640 by 480 pixels and even displays with resolutions of more than 1000 pixels in width and around 700 pixels in height. The resolution also depends on the target application as some devices can be effectively controlled using small resolutions while building complex graphics and displaying multiple control elements is a prerequisite of larger displays. The desirable feature associated with the display is the presence of a touch sensor that can easily substitute buttons and other controls. The touch sensor allows the developer to implement all controls on the display. That is the trend in the development of modern systems.

Another hardware component for the multimedia embedded system is an external Random-Access Memory (RAM). Usually, Single Data-rate RAM (SDRAM) is utilized to store multimedia data during the runtime of the application. The reason to have this type of memory is that the overall volume of internal RAM even for advanced microcontrollers is not enough to store multiple graphical and sound elements. This value may exceed dozens or even hundreds of megabytes. Hence, to extend the available memory range, external SDRAM is connected to the controller. In the typical case, it is used to organize a double buffer, load textures, sprites, and other graphics primitives.

Finally, the touch sensor component from the hardware point of view is represented by a thin transparent film. It has a digital interface to inform the control device about its state. In general, this type of sensor can generate an external interrupt signal when the touch event is detected and has a dedicated digital interface that sends information about the touch to the microcontroller. This way the controller can acquire necessary information about the sensor and orchestrate the reaction to the event from the point of view of graphics change, sound effects, etc.

The general scheme of architectural hardware components is demonstrated in Fig. 1.

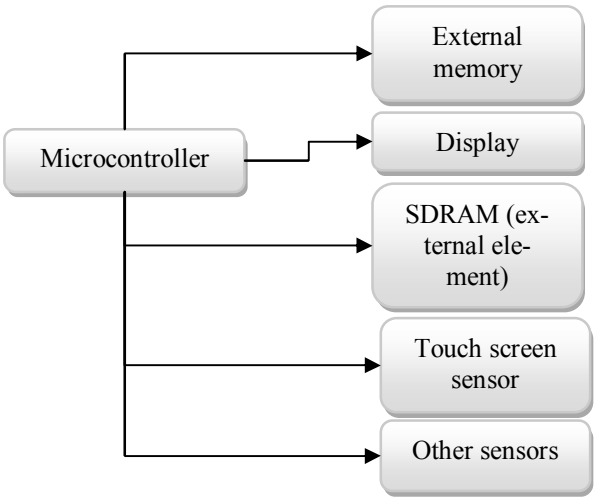

Fig. 1. General scheme of hardware elements connection in the system under investigation

The microcontroller is responsible for the initialization of the whole set of connected peripherals. After the initial configuration of the memory and controllers, they work on their own without additional interruption from the main module. At the same time, the core is responsible for the management of data flow during the next periods of the device's work. That is the part when the devised framework is activated.

First, the devised framework assumes the division of all application graphics into stages. A single stage unites graphical components, animations, and cooperation between them. The transition between different stages is controlled by the stage manager that switches them according to the application logic.

It is assumed that the individual stage is represented by a set of resources associated with it.

Each stage is presumed to execute the following sequence of activities:

1) initialization of the resources associated with a stage;

2) assignment of the function pointers values defined in the stage's files;

3) execution of the main processing loop;

4) release of the allocated resources when the stage receives a command to change.

During the initialization stage, all necessary resources are loaded into the RAM (either the internal RAM of the controller or the external SDRAM). The resources are allocated sequentially. This stage may cause some delay in the execution especially when compressed data formats are involved. When the new stage is called, it may erase/rewrite the resources of the previous instance. The reuse scenario is also possible, however, for this purpose both stages need to share the same resource map up to some extent. In this case, only deficient resources can be loaded. If we consider that each resource consumes L bytes of the SDRAM space, the start address for the i-th stage resource is calculated according to the equation

$$addr_i = addr_{i-1} + L_{i-1}. \quad (1)$$

Hence, each resource's address is calculated iteratively concerning the value of the previous resource location and its length in bytes. For image data that is stored in full resolution with a conventional RGB scheme, the value of L will be calculated as

$$L_i = 3 \cdot width \cdot height. \quad (2)$$

Each stage should define a minimal set of functions that are used to describe its behavior. That also relates to initialization. Hence, the set of functions can be generalized and regarded as a required attribute of the stage. In the proposed work, it is assumed that the function pointer mechanism is exploited to ensure the dynamic behavior of the system when an active stage changes. That is the mechanism that is available in the C programming language. In the case of C++ implementation, more advanced object-oriented techniques can be applied. The stage manager assigns dedicated pointers to the specific addresses which are associated with a particular stage instance. The list of compulsory functions for the stage to implement contains the following elements:

1) initialization;
2) foreground rendering;
3) background rendering;
4) touch sensor processing.

Those functions should be defined and accessible to the stage manager.

Stage rendering. The stage rendering process also follows the idea of unification so it can be used for all stage instances that will be implemented in the application. As the final image can be represented as a combination of background and foreground images. At first, the background is copied to the buffer memory region. Then, the foreground image is copied into the same memory region. Both operations presume to undergo chroma key filtering to remove specific pixels from the output.

The rendering process in this case implies that image data from the buffer represented by the pair of address and length $(addr_i, L_i)$ is copied to the address in the layer buffer according to the value of coordinates (x_i, y_i). As the DMA2D mechanism is activated to manage copy process, information about the width and height of the image should be stored. It is necessary because DMA2D skips memory elements that will not be overlapped by the current image according to the width and height parameters.

The efficiency of such a procedure should be quite high because DMA2D is a module that was designed for the execution of operations over graphical elements. The processor just sends information about the necessary activities that are expected from the module and then waits for their finalization. As soon as the operation is over, the next instruction can be sent to the pipeline. Some operations can be further optimized, but in a general way, this is the prominent solution to ensure a decent performance level during the graphics rendering process and unload the computational core from time-consuming operations with data arrays.

Stage manager. The stage manager part is responsible for transitions between different stages. That level of abstraction allows separating each stage and removes unnecessary connections between them. Therefore, only the instance of the stage manager should be aware of how the transition can be performed, what sequence is possible, and so on. From the implementation point of view, the stage manager may be implemented either as an instance of a separate structure/class (for C/C++ correspondingly) or encapsulate the logic in the function. Moreover, assuming more sophisticated scenarios, it may be divided into multiple functions. It depends on the application domain of the system. The stage manager instance conducts the assignments of the function pointers to specific addresses. This is the main workaround to assure the dynamic behavior of the system. A typical example of stage management is depicted in the state diagram in Fig. 2.

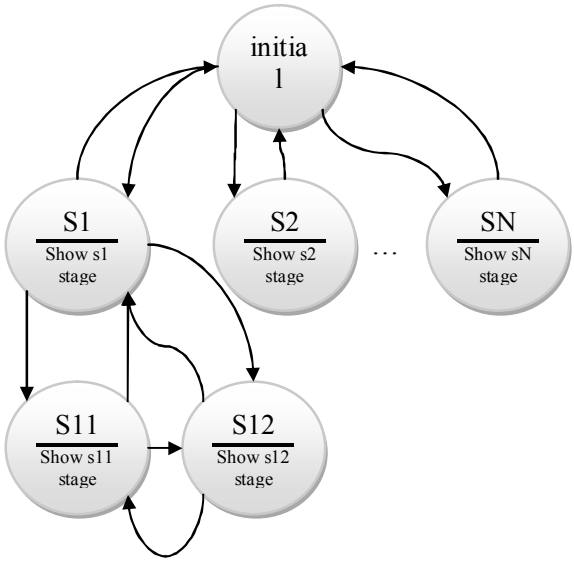

Fig. 2. Example of the state diagram for stage transition implementation

Generally speaking, two-way transitions are possible between the stages on different levels. However, it is also possible to implement logic that can handle transitions

between states on the same level of stages. The diversity of the stage graphs that is possible to implement using the proposed technique is quite large. However, it is necessary to remember that every stage causes increase in the footprint of the output binary file. Thus, generalization may be applied to the states to decrease the output in the extreme case. Other techniques are also possible. For instance, declarative stage generation according the description in the configuration file may reduce the footprint of the binary executable significantly.

The relationship between the stage manager and stages can be depicted using the following diagram (Fig. 3).

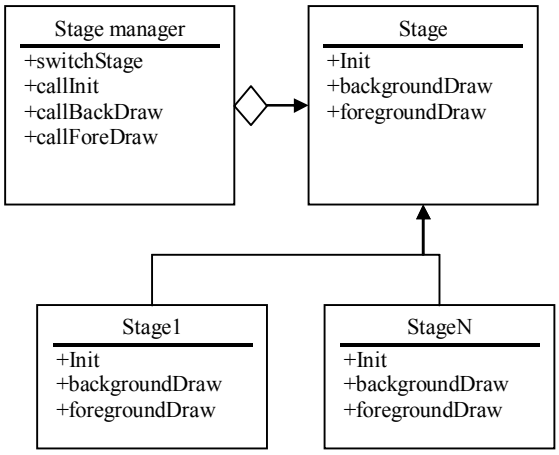

Fig. 3. Representation of the connection between the stage and stage manager in the form of a class diagram

According to the diagram, the stage is represented as an interface that all derived classes should implement. Stage manager instance has a reference to the specific stage. The calls to the stage-related functions contain the call of the functions implemented in the stage.

Touch sensor processing. There are multiple options for touch sensor processing that can be integrated within the framework. Let us distinguish the following methods:

1) interrupt-based with timer;
2) timer polling;
3) reaction in the loop.

Basically, any of them can be used depending on the complexity of the stage generated. However, to ensure the desired performance, the first two options are recommended to avoid stealing processor cycles from rendering tasks. The third option is also possible but it may lead to a mix-up of the code responsible for the basic functionality which is highly undesirable. In order to ensure the correct responses to touch events, there is a delay and disabling of the interrupt in the interrupt handler. The disabling of the interrupt guarantees that the interrupt will not be executing multiple times as a result of the pressing event which is highly probable in case of conventional pipeline. After the certain period of time, the interrupt is enabled once again. This way, the performance of the system can be regulated as well as reactivity. By decreasing the delay, more touch events can be sensed in one second. The event of enabling back the external interrupt from the touch sensor is performed in the timer update interrupt handler where the interrupt from the sensor is enabled and reverse action is applied to the timer update interrupt. Hence, there is a mutual dependency between the external interrupt and the interrupt from timer as they turn itself off and enable the opposite callback.

The possible scenario is occurrence of the multiple touch events. Contemporary sensors support such functionality and operate in similar way as with one touch. The data about multiple touch events is acquired and sent. However, from the point of view of the processing module, the situation does not change as a single interrupt occurs with all information about touches. Therefore, only logic augmentation is required in the interrupt handler but the presence of this functionality should not affect the performance in general.

The pipeline of graphics processing is supposed to be executed in the loop like any other embedded application. The visualization of the graphics processing pipeline is shown in Fig. 4.

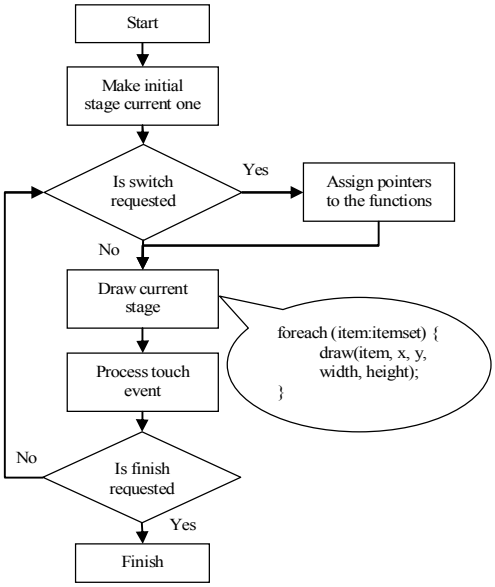

Fig. 4. The algorithm for graphics processing proposed by the framework

During the rendering stage, each graphical object is copied into the corresponding buffer with regard to the values $(x, y, width, height)$. Then, the buffers are copied completely into the area of the background or foreground.

To demonstrate the organization of the project, the diagram shown in Fig. 5 was created. Each stage is represented by a pair of source and header files. Adding a new stage assumes that two new files will be added to the project. The technique used to make the main file aware of the entities in other files is the use of an extern modifier for object declaration. Hence, the declaration appears in two files and the compiler translates it into a single instruction. The functionality that is responsible for touch sensor processing resides in the main.c file.

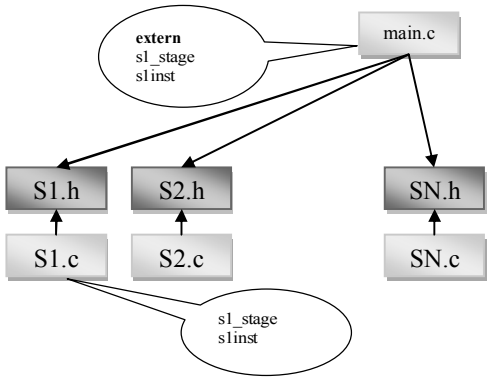

Fig. 5. The diagram of file organization according to the proposed framework

The arrows in Fig. 5 demonstrate the relation of inclusion between files. The side note elements show how stage elements are defined in the source code.

V. RESULTS AND DISCUSSIONS

The proposed approach was tested on the STM32 platform. The testbed was represented by the STM32F7 Disco development board which comprises all the necessary elements for work with multimedia data. It has a ready-to-use display with a touch sensor, audio interface, and Ethernet connectivity. All the connections between a microcontroller and peripheral devices are prepared and demand no additional manipulations. The particular interest for this work is in the LCD that has a resolution of 480 by 272 pixels and supports 24-bit color depth and an alpha component. The touch sensor is selected specifically for this display with the corresponding resolution. The board also facilitates the SDRAM element installed. The size of the internal flash memory is 1 Mbyte.

For the test purpose, based on the devised solution, the microcontroller was implemented in the STM32Cube IDE using the C programming language. Hardware Abstract Library (HAL) of STM32 and Board Support Package (BSP) for this specific board was used during the development process. Hence, the framework tries to follow practices introduced by HAL. To demonstrate the feasibility of the framework, the application contains multiple stages (6 stages; 1 main stage, and 5 complemented stages) that include different graphical objects and show different communication scenarios on the stage. The graphical resources were stored on the SD-card and grouped through a directory structure. Each stage had its dedicated directory on the SD-card.

Additionally, the testbed was equipped with a 6-axis sensor MPU6050. It comprises an accelerator and gyroscope so the reaction to the movement can be implemented by acquiring measurements from this sensor to calculate the orientation of the board.

The size of the developed firmware after the compilation with compiler optimization at level O3 was 237 kbytes. That led to the consumption of 23.7 % of the flash memory of the microcontroller. The addition of the debugging information alongside with downgrading of the optimization level will result in a higher level of program memory use.

Finally, the processor utilization was put into the test scenario. For this purpose, the code was complemented with an additional timer to measure an execution cycle T_{cycle}. The value of the timer was captured before and after the cycle execution and the difference between two values was calculated with regard to possible overflow. Then, the measurement was repeated for each individual call inside the loop. Since all operations can be divided into two categories:

1) operations require processor execution;
2) operations require only initiation from the processor.

Obviously, the second category is associated with DMA transactions. Let us denote the overall cycle time of execution as the sum of all actions from the both categories:

$$T_{cycle} = \sum_i T_{proc,i} + \sum_j T_{DMA,j} \ . \quad (3)$$

Therefore, the potential usage of the processor can be assessed as a ratio:

$$LOAD = \frac{\sum_i T_{proc,i}}{T_{cycle}} \ . \quad (4)$$

The idea of evaluating processor load in this way is in the fact that during a DMA transaction processor can execute other tasks. By applying blocking DMA call, we can measure the time that processor waits for the end of operation. This time can be reassigned to other tasks in case of non-blocking scenarios. According to the performed tests, the value of the $LOAD$ was in the range of [0.124–0.318].

The devised application had one stage controller that controls the stage switching process. The application had one main stage and several subordinate stages. The transition from the main stage can be executed to any stage of the lower level. Backward transition is also possible. Hence, the only way to access another stage of the lower level is to move back to the main stage and then one can invoke a transition. Thus, the organization of stage transition corresponds to the basic scenario with only two levels of stage hierarchy. The main stage contained multiple custom-drawn buttons to transit to the lower level stage. The other stages comprised a single button that initiated a return to the main one.

The application utilizes background and foreground layers to organize graphics representation. The background typically is a static image prepared during the loading of the file. The foreground layer contains graphical images, sprites, and text elements that are designed to communicate with each other.

As for the other aspects of the hardware platform work, other processes (which are not associated with multimedia directly, however, play a supportive role for the device, e. g. controlling battery state and signaling about it, updating multimedia data from the network resources as a part of update procedure, etc.) can exist there. Such activities may also consume resources even if they are

organized using DMA transactions. However, as the processor is not involved during the most part of the DMA transfer and just maintains the start and finish of the transfer, that should not impose significant plummet in the computational core performance. Those other background processes can take place alongside with the graphics processing pipeline. However, if the tasks are computationally intensive, that may entail some aggravation in the framerate as the processing core is involved in the interrupt handling, Thus, the recommendation is to reside maximum supportive operations in the DMA and avoid computational tasks assigned to the assisting functionality.

The other point of concern is usage of the proposed framework in the context of the Real-Time Operating System (RTOS) in comparison with bare metal applications. The OS provides additional services to ensure maintainability of the application. While the initial intent is designed and tested as a bare metal system, it still can be integrated into the solution with RTOS. For this purpose, the task for the graphics context management should be created. Typically, RTOS supports conventional interrupt processing, hence, the amount of changes to the sources code should be conceivable and may be practically transferred to the OS environment. The precaution for the RTOS usage is the correct configuration of timers that measure intervals for the system. Incorrect configuration may lead to a notable degrade in the number of frames displayed in one second which is the main metric for the system. Usage of the framework on the systems with more than one computational core is possible, however, it requires additional level of abstraction since thread safe access to the resources is not guaranteed in the current implementation of the framework. Such level of abstraction should utilize synchronization primitives as mutex to establish proper control over the system resources.

VI. CONCLUSIONS

The paper presented the framework for software development for multimedia embedded applications. The framework is based on the division of the application into multiple stages and the stage is the key notion for the devised solution. The solution allows the developer to unify the development process by following specific rules during the process. All functionality can be implemented similarly. The framework is suitable for animation organization, interface implementation, and other scenarios for the user interface. The feasibility of the proposed approach was tested on the application with 6 stages. The footprint of the whole binary file had a size of 237 kbytes. All stages were implemented similarly according to the declared interface for the stage. That should assist in the reducing development time for such applications.

Yaroslav M. Krainyk is an associate professor, Ph. D. in computer systems and components at Computer Engineering Department at Petro Mohyla Black Sea National University (Mykolaiv, Ukraine). He received a B. Sc. degree in Computer Engineering in 2011.

In 2013, he received a M.Sc. degree in Systems and Methods of Decision Making. In 2016, he was awarded a Doctor of Philosophy degree. His research interests include embedded systems, Internet-of-Things, FPGA, image processing.

REFERENCES

[1] A. Kumari, S. Tanwar, S. Tyagi, N. Kumar, M. Maasberg, and K.-K. R. Choo, "Multimedia big data computing and Internet of Things applications: A taxonomy and process model", Journal of Network and Computer Applications, Vol. 124. Elsevier BV, pp. 169–195, Dec. 2018. DOI: 10.1016/j.jnca.2018.09.014.

[2] P. Hupalo and A. Melnyk, "Acquisition and Processing of Data in CPS for Remote Monitoring of the Human functional State", Advances in Cyber-Physical Systems, Vol. 6, No. 1. Lviv Polytechnic National University, pp. 14–20, Jan. 23, 2021. DOI: 10.23939/acps2021.01.014.

[3] F. Al-Turjman, A. Radwan, S. Mumtaz, and J. Rodriguez, "Mobile traffic modelling for wireless multimedia sensor networks in IoT", Computer Communications, Vol. 112. Elsevier BV, pp. 109–115, Nov. 2017. DOI: 10.1016/j.comcom.2017.08.017.

[4] D. Ibrahim, "ARM Cortex microcontroller development boards", Arm-Based Microcontroller Multitasking Projects. Elsevier, pp. 33–45, 2021. DOI: 10.1016/b978-0-12-821227-1.00003-7.

[5] Y. Krainyk, "Regression Model of Frame Rate Processing Performance for Embedded Systems Devices", in Applications of Machine Learning. Algorithms for Intelligent Systems, P. Johri, J. Verma, and S. Paul, Eds. Springer, Singapore, 2020, pp. 257–265. DOI: 10.1007/978-981-15-3357-0 17.

[6] R. Herrero, "MQTT-Sn, CoAP, and RTP in wireless IOT real-time communications," Multimedia Systems, Vol. 26, No. 6, pp. 643–654, 2020. DOI: 10.1007/s00530-020-00674-5.

[7] W. U. Rahman, Y.-S. Choi and K. Chung, "Performance Evaluation of Video Streaming Application Over CoAP in IoT", in IEEE Access, Vol. 7, pp. 39852–39861, 2019. DOI: 10.1109/ACCESS.2019.2907157.

[8] A. Karaagac, E. Dalipi, P. Crombez, E. De Poorter, and J. Hoebeke, "Light-weight streaming protocol for the Internet of Multimedia Things: Voice streaming over NB-IoT", Pervasive and Mobile Computing, Vol. 59. Elsevier BV, p. 101044, Oct. 2019. DOI: 10.1016/j.pmcj.2019.101044.

[9] Oracle, "Class Stage", https://docs.oracle.com/. https://docs.oracle.com/javase/8/javafx/api/javafx/stage/Stage.html (accessed Oct. 17, 2022).

[10] Y. Krainyk, "Information technology of university class internet-of-things-module", in CEUR Workshop Proceedings, 2019, Vol. 2516, pp. 58–68. url: http://ceur-ws.org/Vol-2516/paper4.pdf

[11] uGFX, "uGFX – lightweight embedded GUI library". https://ugfx.io/ (accessed Oct. 20, 2022).