International Journal of Engineering Research-Online
A Peer Reviewed International Journal
Articles available online http://www.ijoer.in; editorijoer@gmail.com

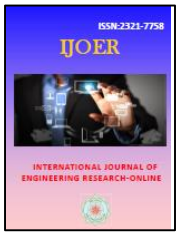
Vol.7., Issue.5, 2019
Sept-Oct

RESEARCH ARTICLE

ISSN: 2321-7758

AN ARCHITECTURE FOR THE DEVELOPMENT OF COMPUTER VISION APPLICATIONS AND DIGITAL IMAGE PROCESSING IN EMBEDDED SYSTEMS

Dr. AMAMER KHALIL MASOUD AHMIDAT[1],
MUHAMAD ABDULLA MUHAMAD ABDUSSALAM[2]
[1]Higher Institute of Medical Technology in Bani Walid.
Email:mkmasoud@hotmail.com
[2]Higher Institute of Medical Technology in Bani Walid.
Email: mrabett34@gmail.com
DOI: 10.33329/ijoer.75.42



ABSTRACT
This article presents the hardware and software framework that leads to the design of Computing Vision and Digital Image Processing technologies in Embedded Systems, especially Smart Cameras. Smart cameras are cameras which, in addition to capturing images, are capable of extracting specific information for the application. Embedded systems are systems made up of hardware and software designed to perform specific tasks integrating computing and input and output devices provided by applications. Computer Vision can be described as the ability to automatically recreate a mathematical model from one or more objects in a given scene. The mathematical or computer model is able to provide information about the scene previously recorded only in images. This is then the inverse process to Free Computing in which the computing system, from a mathematical model, generates digital images. The Digital Image Processing area is, therefore, the basis for the development of any Computer Vision application, as it is responsible for the manipulation of images, which are the first input of a Computer Vision application. The resource constraints inherent to Embedded Systems coupled with the complexity of Computer Vision hinders the construction of applications within these two contexts. The solution, proposed in this paper, is to build an architecture that helps the developer, abstracting details of the Embedded System and the Computer Vision algorithms.
Keyword: Computer Vision, Smart-cameras, Architecture

1.0 Introduction

Smart Cameras is defined as a Vision System which, besides capturing images, is able to: extract information relevant to the application; generate events based on information from the captured image; make decisions that are used in intelligent, autonomous systems They are closed systems that encapsulate to the communication interface, for example Ethernet. Noticeably more compact than personal computer-based Computer Vision Systems.

1.1 Embedded Systems

Embedded systems are systems composed of hardware and software designed to perform specific tasks combining processing and input and output devices required by applications. Usually in these systems there are resource constraints such as main memory, disk space or ash (used for rmware

storage, Operating System), input and output devices (monitor, keyboard). Thus, for Embedded Systems to meet performance requirements, a well-designed project needs to be carried out in order to reduce product cost and size while meeting application constraints[1].

Application development for embedded system devices is becoming increasingly complex and multidisciplinary as the need for complete applications is increasing. This requires that Embedded Systems application development be able to provide multi-functional and compatible applications, but at the same time simple and low cost development. Embedded System device manufacturers provide proprietary, platform-specific development environments. However, development environments do not have an established standard, at the manufacturer's discretion to provide the features and features provided. Platform independence is an important and motivating aspect of this work, as the designed architecture abstracts inherent and specific details from each platform, making it possible to exclude the development environment provided by the process manufacturer. This ensures code reuse regardless of the platform used. However, it is necessary that the device in question be able to interface with the low level, part of the architecture designed here Schmidt [2002][2].

The large amount of processing required by Digital Image Processing applications[3] and consequently the area of Computer Vision combined with the availability of processors optimized for this type of application is a reason to choose. This area of knowledge in this paper. The Black in processor[4], a hardware platform optimized for digital image and video processing, was used. This platform was used in the development of the architecture resulting from this work and described here to assist in the creation of applications and Computer Vision and Digital Image Processing aware.

1.2 Computational Vision and Digital Image Processing

There are several of Computer Vision definitions[5]. One is the ability to automatically reconstruct the mathematical model from one image or more images from a given scene. The mathematical or computer model is able to provide information about the scene recorded in image only. This is then the inverse process to Free Computing, in which the computing system, from a mathematical model, generates digital images[6]. The Digital Image Processing area is, therefore, the basis for the development of any Computer Vision application, as it is responsible for the manipulation of images, which are the first input of a Computer Vision application. It can be defined as processing a digital image, represented by a two-dimensional function, f (x, y), where x and y are spatial coordinates in the plane and the amplitude off in any ordered pair (x, y) is called intensity of the image at the given point. An image is called digital when the values of f are discrete and sharp. There is no consensus on the boundaries between Computer Vision and Digital Image Processing, an acceptable delimitation is that Digital Image Processing deals with processes whose inputs and outputs are digital images and processes that extract attributes from digital images.

The area of Computer Vision is by nature multidisciplinary. It uses concepts, techniques and approaches from other disciplines mainly in the area of computing, but is not limited to it. Some of these disciplines are: signal processing, physics, mathematics, artificial intelligence, robotics, neurobiology (biological view), among others[7].

Due to the richness and breadth of the Computer Vision area, developing Computer Vision systems may require the programmer to mature in different areas of knowledge. Regardless of the application developed, it will inevitably be necessary to capture and process a digital image, because the process of Computer Vision begins in the acquisition of the scene image. The basic process of image acquisition and processing by applying a basic filter to reduce noise becomes constant in the development of Computer Vision applications. It is therefore useful to reuse methods that do this processing whenever you need to use them. Computational Vision application development frameworks, such as the Open CV library[8], are widely disseminated and used for this purpose. These libraries, however, require knowledge of their

functions and implementation details to generate a high performance application. The learning curve may be unfavorable and nontrivial programming, thus adding to the list of prerequisites for the development of applications in Computer Vision.

1.3 Review of Literature

1.3.1 WISDOM and YATOS

WISDOM is part of a work developed with the now defunct SensorNet research group[9] to create an operating system (YATOS) and middleware (WISDOM) designed specifically for the context of Wireless Sensor Networks. It was designed to serve as a development platform for the group and is therefore a particular case of using middleware to facilitate application development in embedded systems[10].

The designed middleware features low-level cross-platform and multi-language features, that is, it can be used to develop applications on heterogeneous platforms and languages, provided they are properly configured. It also has an intuitive graphical programming methodology, making use of the Java language[11] to be portable.

This work is similar to what is described in this paper, however its context is specific applications of sensor networks as opposed to the Computational Vision and Digital Image Processing in Embedded Systems performed in this work.

1.3.2 µCLinux

MCLinux is a Linux 2.0 kernel-based operating system. It was initially ported to microcontrollers without Memory Management Unit (MMU). MMUs are hardware devices that translate virtual addresses into physical addresses.

The µCLinux Project, however, has grown and currently provides operating systems that use Linux Kernels 2.0, 2.4, and 2.6 for many different hardware architectures. Among them is Black n, which was used as the first hardware platform of this work.

MCLinux was chosen to be used as an abstraction layer of Black n features and details because it is similar to the Linux operating system, managing memory, managing file system, implementing high level network interfaces and multitasking. These features helped in the development of the hardware and software interface, whose main benefit is the achieved abstraction.

1.3.3 TinyOS

TinyOS is an event-based operating system for sensor networks without. It was developed using the nesC programming language as described in TinyOS [2007a][12]. NesC is a variation of the C language, given in Kernighan [1988][13], optimized to deal with the memory limitations of sensor networks without.

1.3.4 Image Processing Library 98 - IPL

The Image Processing Library (IPL) is an independent C / C ++ image manipulation platform. Its purpose is to assist in the creation of new processing techniques as well as to provide standard methods for processing images. It is the basis of the OpenCV library described in the next sections.

1.3.5 Open Source Computer Vision Library - OpenCV

The Open Computer Vision Library (OpenCV) is a library produced by Intel and its focus is on real-time Computer Vision application. This library is optimized to take advantage of the MMX multimedia instructions contained in Intel processors. For this reason, it is usually used in general purpose computers.

This library is based on the IPL, it has a wide range of functions ranging from managing the image matrix in memory to creating a graphical interface for the application.

1.3.6Matlab

Matlab is a high-level, interactive modelling tool widely used in digital image processing courses because it enables the simple use of procedures used in applied mathematics disciplines. However, Matlab has several limitations such as the absence of references, unconventional syntax, and ambiguity. Also, Matlab does not focus on performance. These reasons, combined with the fact that it is closed source, make it difficult to use this feature in the proposed tool. Another important

aspect is the need for use in embedded systems, where information storage restrictions are still present. A complete tool like Matlab would not be suitable for the target platform of this work.

1.3.7 Image Analysis - Analog Devices

The Image Analysis library from Analog Devices, a manufacturer of Black n, provides the assembly implementation for Black n of the basic image processing functions supported by this platform. It is optimized for use on Black n. Image Analysis has been translated into C language and compiled for use by µCLinux

1.3.8 Other Works

There are industry initiatives to provide middleware solutions for embedded application development. These solutions tend to be generic and therefore do not leverage hardware resources to improve application performance. However, there are few scientific references about the construction of these tools, this is another motivating factor for this work.

2.0 Objectives

This work is part of a larger project that includes building a software and hardware architecture that assists in the development of Computer Vision and Digital Image Processing applications in Embedded Systems. This paper deals only with software architecture.

The architecture as a whole was designed to: promote the reuse of basic resources (code) to ensure a developed system quality and reduce costs; isolate the hardware platform from the developed application; ensure intellectual property and security of the hardware platform; enable the development and execution of complete and real applications in heterogeneous systems; abstract details of the hardware platform; encapsulate hardware restrictions and limitations; provide an intuitive programming model.

This paper describes and describes the designed architecture. In addition, it describes the implementation of software for the translation, generation, transmission, control and execution of the user-developed program up to the Embedded System, an integral part of the designed architecture. The implementation of the hardware abstraction layer will be presented by its author as part of his master's dissertation, it complements the present work.

3.0 BUILT ARCHITECTURE

The built architecture resides between the user and the Embedded System operating system used, Figure 3.1. The architecture allows the abstraction of details of the Embedded System used as well as the making of high level applications.

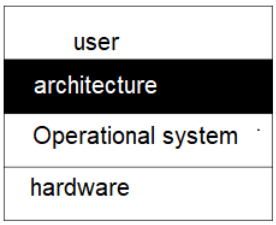


Figure 3.1. Architecture Location

The architecture was designed to be modular. To this end, its implementation was divided into two modules, tool and middleware. The scheme in Figure 3.2 shows the developed architecture segmented in its two distinct parts. The tool interfaces with the user (programmer) of applications in Computer Vision and Digital Image Processing. Its main functions are: translation and code generation provides the user with resources for application development, enabling the making of high level applications, transparent to the hardware platform used.

Transmission and execution of the program control of the transmission of the code generated when it will be executed in the chosen processor and control of the execution of this program.

Middleware works at the low level of architecture, that is, it interfaces with the Operating System, µCLinux. Its main function is to abstract the operating system layer and therefore hardware details.

The design of the architecture as a whole and the implementation of the tool have been part of this work, and are described in detail here. However, the implementation of middleware is part of the master's work of Glauber Tadeu[14]. Thus, the

middleware design features were included in this text, implementation details were at the discretion from its author. Middleware design decisions made as part of this paper is detailed in the coming part.

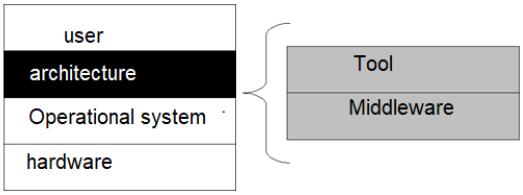


Figure 3.2. Architecture Composition: Tool and Middleware

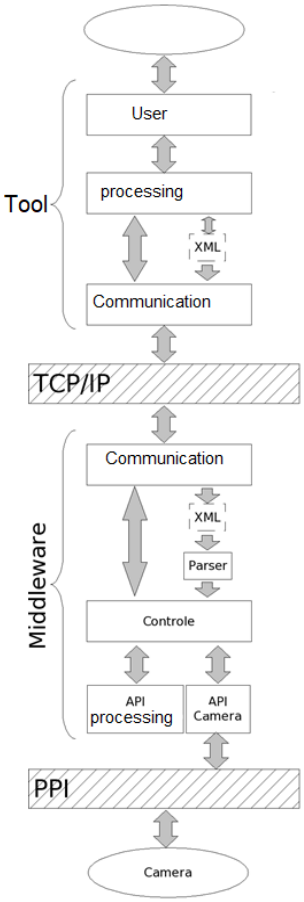


Figure 3.3. Detailed built architecture

In Figure 3.3, a detailed diagram of the projected architecture is shown. In this diagram are the interactions present and the modules responsible for each of them. The user interacts with the tool, this interaction is performed by the Tool Interface module. This module aims to assist the user in programming

3. Built Architecture

Through the availability of high-level hardware resources as well as the graphical representation (built XML tree) of the application. The interface module interacts with the processing module, whose main tasks are to maintain the language structure, generate and execute the XML code to be interpreted by middleware. The processing module interacts with the communication module to, through the protocol provide middleware with capabilities for interpreting XML. Middleware, in turn, has a communication module that implements the server part of the communication protocol; this module captures XML and transfers control to the parser that processes XML generating a list of functions. From this point on, XML is no longer used. The control module then uses built-in function list structures, APIs, and file management to execute portions of the user application delegated by the application. By design decision, only the Camera API has direct communication with the camera, simplifying the process of adapting the architecture to a new camera.

3.1 Middleware

As Geihs [2001][15], middleware masks the heterogeneity of computer architectures, operating systems, programming languages, network technologies, and facilitates application development. The sharp middleware during the present work masks these heterogeneities and is therefore allied to the tool presented in coming sections resulting in the designed architecture.

3.1.1 Definition of architecture

The main purpose of middleware is to facilitate the development of embedded systems applications through platform independence. This is done through the abstraction of hardware details made possible by the included OS layer. Figure 3.4 locates middleware against the Embedded System and its hardware / software interface; middleware acts at the interface between the operating system and the tool, abstracting the details from the lower level layers (operating system and, consequently, hardware) to the higher level layers (tool). The instance chosen by this work is to devise a

middleware that interfaces specifically to produce Digital Image Processing and Computational View applications, developed by the tool and compatible computational architectures. However, the concept used is not limited to these applications. The implemented architecture can be used for the development of general purpose applications. However, using hardware platforms optimized for this is recommended.

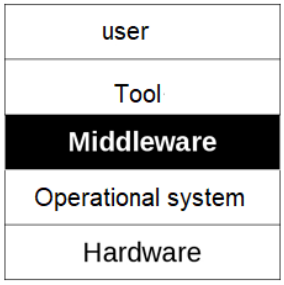


Figure 3.4. Middleware Location

The architecture designed for middleware is shown in Figure 3.5. Each platform is represented by a rectangle. At the highest level is any tool that implements the middleware communication protocol. The tool uses the abstraction layer provided by middleware to interact with each hardware platform transparently. Using this protocol allows middleware to be used by any tool capable of implementing it, and vice versa. At the lowest level are the hardware platforms that implement the Operating System communication API. To validate the concept of this work, two platforms were used: Black n and a personal computer. Using an API enables middleware to communicate with any platform that can implement it. This feature allows the abstraction and transparency of the hardware architecture provided by the middleware. Other systems are represented in the middleware architecture definition; just implement the middleware communication API to port it.

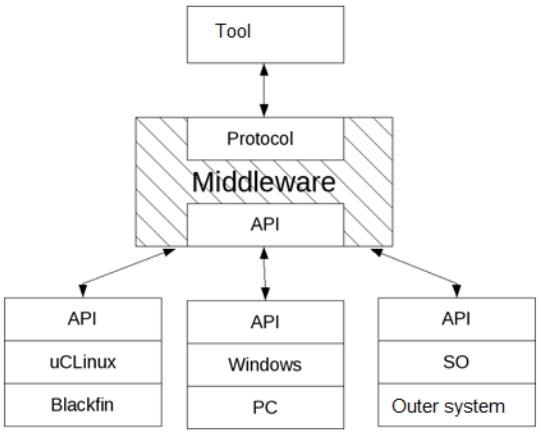


Figure 3.5. Definition of middleware architecture

3.1.2 Features

The developed middleware consists of an abstraction layer between the tool and the operating system provided by the hardware architecture in question. This layer facilitates the development of Embedded Systems Digital Image Processing applications as it abstracts the details of the configuration and operating routines of the platform in question, allowing heterogeneous hardware architectures to be programmed similarly. From middleware, we developed a software development tool for Computational Vision and Digital Image Processing for Embedded Systems, especially Black n described in Section 3.2. Middleware provides benefits to software development processes through:

Reuse of Basic Features

Code reuse is a well-known software engineering technique to ensure the quality of a system.

- Isolation
- Independence between hardware architecture and tool
- Intellectual Property Security and Assurance
- The user does not need to have access to the hardware platform details to develop their applications.

These features are achieved through the ability of the middleware, through the operating system, to abstract heterogeneities of computer architectures, operating systems, and programming languages.

The middleware was designed to utilize the features provided by the µCLinux operating system at first. MCLinux, in turn, is running on the Black n processor. BlackFin was chosen because it is optimized for digital processing of images and videos and, therefore, adapting to the scope of applications referenced in this work (Digital Image Processing and Computer Vision). The choice of µCLinux was because it is an open source operating system and is similar to the Linux operating system and thus is possible because the written code for it is portable to standard Linux operating systems. It also allows performance comparison between the standard Linux system running on a computer and the Embedded System.

3.2 Tool

The tool implemented in this work aims to use middleware functionalities to provide the user with a high level and transparent hardware programming model. It is the function of the tool to manage the translation, generation, transmission and control of program execution by the user.

The built-in tool generates language and platform-independent code (XML) as long as the latter implements the middleware API.

3.2.1 Definition of architecture

The main objective of the tool is to provide an intuitive and transparent user interface to program the application of Computer Vision and Digital Image Processing in the chosen Embedded System. In addition, the tool is responsible for controlling the execution flow of the developed applications. That is, the tool determines which parts of the user code will be executed in the tool itself and which parts will be executed in the Embedded System, for example Black n. It locates the tool in the Embedded System and its hardware / software interface; the tool acts on the interface between the middleware and the user, providing an intuitive and transparent programming interface and controlling the execution of the built program. For this work, ImageAnalysis library functions and a subset of the OpenCV library, present in the middleware, were implemented.

The tool was developed with the following requirements in mind:

- Enable the execution of applications in a manner and aware of various embedded systems by generating code for each specific compiler. The optimizations provided by the compilers would thus be taken advantage of.
- Enable modularized application development.
- Abstract the hardware. Allowing the hardware architecture not to be exposed to the user.

Features

The developed tool assists the development of Computer Vision and Digital Image Processing applications by building a tree representative of the functions performed by the application. This tree gives rise to XML that is representative of the user's application and is independent of the hardware platform, programming language, and processing API used. In Figure 3.6 is an example of a simple linear program tree, which only applies the µOpenCv API threshold filter to the framebuer generated by the application.

```
<head>
    <name>Example</name>
    <author>Antonio</author>
    <version>1.0</version>
    <date>20081010</date>
</head>
<function>
    <api>u/gOpenCv</api>
    <name>threshold</name>
    <param>
        <value>200</value>
```

```
<type>int</type
>

</param>

</function>

</blackfinProgram>
```

Code 3.2.2 shows the XML corresponding to the tree shown in Figure 3.8. The counterpart of the program tree is XML. It does the processing process done and determines the names of APIs, functions, parameters, and returns. Thus, it is necessary to implement, in the embedded system, a server capable of decoding the information contained in XML and performing the necessary operations. In addition, other information, such as the XML header, or program author, date, and version information through the head tag.

For application development, the user chooses the order of constructs made possible by the language and the API functions with their respective parameters and adds them one by one to the function tree. Once created the tree is generated XML, used by the tool to execute the program. Middleware is responsible for executing only the processing APIs. The tool is responsible for the execution of the other language constructs, ie, control of frames and frames. In Figure A.1 is the application build window and its function tree. In more complex constructions it is possible to interact with the nodes of the application's structured tree to detail each construct, such as constants, parameters, expressions, etc. In Appendix A are the screens built for the tool.

You can choose between two distinct APIs and processing, or one of the language constructs.

analog
ImageAnalysis by Analog Devices
• sobel
• erode
• dilate
• skeleton
• median
• perimeter
uopencv
subset inspired by Intel's OpenCV library
• threshold
• linepro le
• negative
• histogram
• countWhite
• getRect
• ood ll
• conv
control

A key feature of the tool, and of the architecture as a whole, was that of defining the frame model used. The framework is a frame list list accumulator model, that is, there is the main frame (accumulator). All operations are performed on and written to this framebuer. In addition to the accumulator, there is a list of framebuffer that is used to manipulate the accumulator data into memory. For example, if the user wants to save the framebuer before a given operation, simply copy the accumulator to the list via the push operation

3.2.2.1 Language

An application development language is used that is used to compile XML. The language was clear and in order to meet the minimum requirements of making complete and real applications possible. Thus, the following general purpose constructions have been defined. The nida language here is hereinafter referred to as XML language.

decl

Declaration of variables; reserve space in main memory. In the implementation adopted every variable is a vector. Variables are of byte type. Statements should be written at the beginning of the program. They are not counted as execution thread functions (cannot jump to a statement).

assign

Variable flag: assigns a value to the previously declared variable

go to

Unconditional jump; jumps to the clear position in the application execution range. In implementation the goto's are nothing more than if the condition is always true. The address is absolute (it is not possible to jump back using a negative address).

if

Conditional jump; change which will be next construct in the execution thread if the expression is true. The expressions used were based on C's sharpness pattern.

function

API function; performs a function of one of the nidas APIs available. Functions have parameters and language returns, if any. This build is performed by middleware.

return

Return of the application; marks which variable will be returned when the halt operation is performed.

halt

End of application; marks the end of the application.

These constructions were defined based on the C language. Thus, it is possible to draw a parallel between the two languages. Table 3.1 shows the constructions in C and their counterpart in the language of definition.

Table 3.1 Parallel between XML and C: General Purpose Constructs

C	XML
5*int var;	<decl>
	<var>var</var>
	<size>1</size>
	<type>int</type>
	</decl>
6*var = 0;	<assign>
	<var>var</var>
	<position>0</position>
	<value>0</value>
	<type>int</type>
	</assign>
2*out:	<goto>
goto out;	<address>9</address>
	</goto>
3*if (var > 8192)	<if>
	<expression>var > 8192</expression>
goto out;	<goto>9</goto>
	</if>
13*var = function(128)	<function>
	<api>api</api>
	<name>function</name>
	<param>
	<value>128</value>
	<type>int</type>
	</param>
	<return>

	<var>var</var>
	<pos>0</pos>
	<type>int</type>
	</return>
	</function>
7*return var;	<return>
	<var>var</var>
	<pos>0</pos>
	<size>1</size>
	<type>int</type>
	</return>
	<halt/>

instruction	XML
4*capture	<capture> <height>512</height> <widht>512</widht> </capture>
3*push	<push> <pos>0</pos> </push>
3*pop	<pop> <pos>0</pos> </pop>
3*save	<save> <name> lename</name> </save>
3*load	<load> <name> lename</name> </load>

Table 3.2. Parallel between XML and C: specific purpose constructs with

3.3 Interface between Tool and Middleware

The interface between the implemented tool and the designed middleware is performed through a network layer. The TCP / IP protocol was used to establish the connection between the tool and the middleware. All communication is performed through the application level communication protocol, it is defined as bellow.

Communication protocol

The implemented protocol is based on ASCII (American Standard Code for Information Interchange) . The client-server model Silberschatz and Galvin [2000][16] is used in which the tool is the client and the middleware the server. The main use case consists of a client-server transaction. A transaction consists of exchanging messages in ASCII code between client and server to complete an operation. The list of possible transactions is listed below. The complete protocol is shown in detail in Table 3.3.

Reset

Restores the system to its initial default settings.

Stream file

File transfer from client to server.

Update Version

Streaming server version.

Refresh Image

Transmission of the image from the accumulator to the customer.

Upload image

Loads from client image to accumulator.

Capture image

Captures image from sensor attached to server to accumulator.

Select XML

Selects which XML file to load on the server.

Load XML

Loads the XML program to run on the server.

3. Built Architecture 34

Run XML

Runs the XML program loaded on the server.

4.0 RESULTS

The architecture is designed for the intuitive development of code-generation and aware applications for various Embedded Systems that can support the design of embedded systems applications.

Some features and features offered by the architecture include:

- Constraint abstraction of the details of the platform used.
- Code generation for execution on each desired platform in a transparent way for the user.
- Providing an intuitive programming model ideally graphical.

4.1 Case Study: Presence or Absence

This was the first application written using the software and hardware architecture developed in this paper. The purpose of this program is to answer the question: "Is there an object in the given image?".

For didactic reasons, the first step is to define the execution speed of the application. In Figure 4.1, the execution flow of this application is set. Initially, you must initialize the variables and structures that will be used in the application. The second step is to binarize the image. The third step is to count the amount of white pixels in the image. The fourth step is to decide whether or not the amount of white pixels is sufficient to consider that the object is present in the image. The fifth and last step is to return the result.

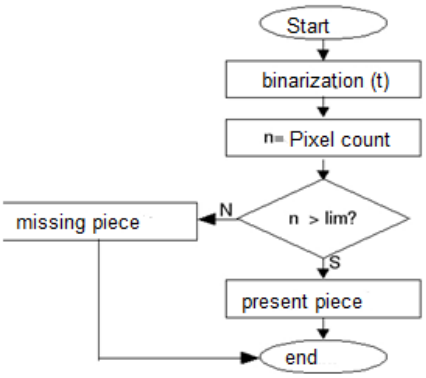


Figure 4.1. Presence or Absence Application Execution Flow

To serve as a control, the application described above was coded in C language and using the architecture of this work presented in Figure 4.2.

1 present () 2 {

3 intnumWhite = 0, r and t;

4 threshold (1 2 8);

5 numWhite = countWhite ();

6 if (numWhite> 8192) 7 r and t = 1;

8 else

9 ret = 0;

10 returnret; 11}

One can see the similarity of both C and Architecture codes to the execution flow. Both

implementations happen almost directly from the execution stream coding. Table 4.1 of the mapping between execution thread, C code and Architecture code.

In Figure 4.2, the data for each tag in the XML is omitted. However, they can be obtained by analyzing the XML code generated from the definition of the architecture application.

Figure 4.2. Presence or Absence in Architecture

Table 4.1. Mapping between approaches: presence or absence

C line	Architecture element	XML Line	Flow
3	0-1	10-22	First step
4	2	23-30	Second step
5	3	31-38	Third step
6-9	4-8	39-58	Fourth step
10	9-10	59-65	Fifth step

4.2 Case Study: Presence or Absence with Error Control

This application was chosen because it shows the specific purpose framebuster control features implemented in the language. It consists of applying Presence or Absence added error control, that is, an object is considered to be present only if it is within a preset count interval. That is, hysteresis.

For didactic reasons the first step is to define the application execution flow. In Figure 4.3, the execution flow of this application is set. Initially, it is necessary to initialize the variables and structures that will be used in the application. The second step is to binarize the image using a variable threshold. The third step is to count the amount of white pixels in the image. The fourth step is to decide whether or not the amount of white pixels is sufficient to consider that the object is present in the image, if the judgment is indecisive, turn to the beginning by raising the threshold and trying again. The fifth and last step is to return the result.

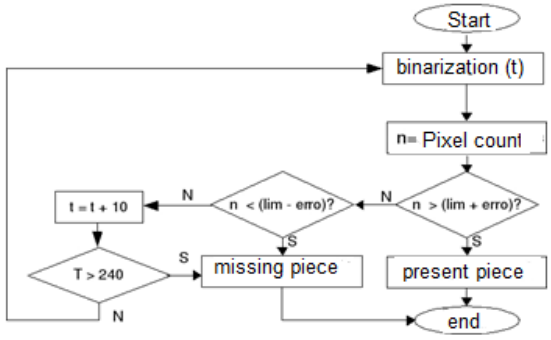


Figure 4.3. Presence or absence application execution flow with error control

To serve as a control, the application described above was coded in C language and using the architecture of this work presented in Figure 4.4.

1 int presence () 2 {

3 int numWhite = 0, r and t = 0, t h r and s h o l d; 4 push (0);

5 threshold = 118;

6 while (threshold <240)

4. Results 44

7 {

8 pop (0);

9 threshold = threshold + 10;

10 if (threshold> 240)

11 return 0; // missing

12 push (0);

13 threshold (threshold);

14 numWhite = countWhite ();

15 i f (numWhite> 8192 + 4096)

16 return1; // gift

17 elseif (numWhite> 8192 - 4096) 18;

19 else

20 return 0; 21}

22 return ret; 23}

Both implementations (C code and Architecture code) almost directly reflect the execution thread coding. Table 4.2 of the mapping between execution thread, C code and Architecture code.

In Figure 4.4, the data for each tag in the XML is omitted. However, they can be obtained by analyzing the XML code generated from the definition of the architecture application.

Figure 4.4. Presence or Absence with Error Control in Architecture

Table 4.2. Mapping between approaches: presence or absence with error control

C line	Architecture element	XML Line	Flow
3-5	0-3	10-37	First step
6-13	4-8	38-60	Second step
14	9	61-68	Third step
15-20	10-11	69-76	Fourth step
21-22	12-17	77-98	Fifth step

5.0 Conclusion

This paper designs an architecture that spans software layers, through the human-computer interface, to the low-level hardware driver layer. The work also comprises the implementation of the highest-tier architecture, consolidating user interface, translation and code generation functionality.

The architecture was designed to support Computer Vision and Digital Image Processing applications in Embedded Systems. However, the end result was a general purpose architecture. It can be used for the development of any type of application, provided the processing APIs implemented in middleware. The architecture implements a general purpose language, which has no conceptual constraints when using regular, non-embedded systems. In particular for Linux systems, simply compile the middleware for the target architecture in question.

The architecture as a whole has the features of: providing an intuitive programming model through the graphical programming interface implemented by the tool; promote the reuse of basic resources (code) to ensure the quality of the system developed through the use of processing APIs; ensure the intellectual property and security of the hardware platform by including middleware; isolate the hardware platform from the application developed by adopting the µCLinux Operating System; enable the development and execution of complete and real applications in heterogeneous systems; abstract details of the hardware platform;

encapsulate hardware restrictions and limitations by adopting an Operating System.

For future work on architecture, it is recommended to improve the communication protocol implementation between the tool and the middleware, using the construction of true transmission control protocol (TCP) packets instead of using an ASCII-based protocol. , because it inevitably has an overhead because the information in this case is coded using only the ASCII textitbyte. Another possibility is to automate the manipulation of the middleware libraries by the tool, ensuring the user the ability to use any APIs they know without having knowledge of the architecture.

Specifically about the tool, part of the implementation described in this paper, there are four tasks: user interface, translation, code generation and application execution. All of these tasks are not the critical path of the designed architecture. Once the application is developed it is downloaded through middleware and executed in isolation. Thus, the performance of the program development tool is not linked to the performance of the application since the tool is not in the optimization context.

For future work on the tool, it is recommended to analyze the quality of the code generated by the tool in the target embedded system. As the code generated by the tool is interpreted, its performance is expected to be degraded. The transition from interpreted code to its compiled counterpart coupled with optimization can provide performance gain for critical applications in this regard. Another possibility is the adoption of a diagram-based programming system, further improving the ease in the adopted programming model. In addition, the application execution architecture can be improved by enabling distributed application execution by using more than one embedded system at the same time, introducing parallelism.

References

1 Malek, S .; Seo, C. and Medvidovic, N. (2006). Tailoring an architectural middleware platform to a heterogeneous embedded environment. In SEM '06: Proceedings of the 6th International Workshop on Software Engineering and Middleware, pp. 6,370, New York, NY, USA. ACM Press.

2 Schmidt, D.C. (2002). Middleware for real-time and embedded systems. Commun.ACM, 45 (6): 43 48.

3 Gonzalez, W. (2002). Digital Image Processing. Prentice Hall, 2nd edition edition.

4 Devices, A. (2005). Adsp-bf537 black n processor - hardware reference. http: // www. analog.com/processors/blackfin/.

5 Trucco, E. and Verri, A. (1998). Introductory Techniques for 3-D Computer Vision.Prentice Hall PTR, Upper Saddle River, NJ, USA.

6 Coatrieux, J.L. (2005). Computer vision and graphics: frontiers, interfaces, crossovers and overlaps in science. Engineering in Medicine and Biology Magazine, 24 (1): 16 19.

7 Trucco, E. e Verri, A. (1998). Introductory Techniques for 3-D Computer Vision.Prentice Hall PTR, Upper Saddle River, NJ, USA.

8 Corporation, I. (2007). Open cv library. http://www.intel.com/technology/ computing/opencv/.

9 SensorNet. Sensornet http: //www.sensornet.dcc.ufmg. br / index.html.

10 Vieira, L.FM (2004). Middleware for Embedded Systems and Sensor Networks. Dissertation (Master). UFMG Federal University of Minas Gerais.

11 Deitel, H.M. and Deitel, P.J. (2001). Java How to Program. Prentice Hall PTR, Upper Saddle River, NJ, USA

12 TinyOS, A. (2007a). nesc - network embedded systems c. http://www.tinyos.net.

13 Kernighan, B. W. (1988). The C Programming Language. Prentice Hall Professional Technical Reference.

14 de Sousa Carmo, G. T. (2009). DSCam: A hardware-software platform for Computer Vision operations. Dissertation (Master). UFMG Federal University of Minas Gerais.

15 Geihs, K. (2001). Middleware challenges ahead. Computer 34 (6): 24 31

16 Silberschatz, A. and Galvin, P. B. (2000). Operating System Concepts. John Wiley & Sons, Inc., New York, NY, USA