A Framework for User-Interfaces on Low-Cost Embedded Processors.

Master's Thesis

at

Graz University of Technology

submitted by

Ferdinand Wörister BSc.
Matriculation Number 0431184

Institute for Software Technology (IST),
Graz University of Technology
8010 Graz, Austria

30 Sept 2012

Advisor: Assoc.Prof. Dipl.-Ing. Dr.techn. Oswin Aichholzer

Eine Programmumgebung für Benutzeroberflächen auf Eingebetteten Systemen mit beschränkten Ressourcen.

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Ferdinand Wörister BSc.
Matrikelnummer: 0431184

Institut für Softwaretechnologie (IST),
Technische Universität Graz
8010 Graz

30. September 2012

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter: Assoc.Prof. Dipl.-Ing. Dr.techn. Oswin Aichholzer

Abstract

For their limited resources, embedded systems pose challenges that differ greatly from those of personal computers. In the field of user-interfaces in particular, their sparse resources impose numerous limitations.

Nonetheless, a number of lessons learned in the world of personal computers can be used to facilitate the development of software that targets the above platforms. In this thesis, a number of principles and patterns of software-architecture are applied to a tool-kit for the creation of Graphic-User-Interfaces (GUIs) that target low-cost graphics controllers.

The markup-language FUIML, a declarative language for designing GUIs, is introduced. It has been designed in the course of this project and provides an efficient, hardware-independent way of user-interface design. A compiler is presented that generates a machine-friendly bytecode from the used FUIML. The bytecode is interpreted by an embedded processor that renders the user-interface.

Kurzfassung

Aufgrund ihrer beschränkten Ressourcen unterscheiden sich die Herausforderungen bei der Programmierung von Eingebetteten Systemen stark von denen, die bei Systemen mit größerer Rechenleistung auftreten. Besonders auf dem Gebiet der Benutzeroberflächen stoßen Eingebettete Systeme schnell an ihre Grenzen.

Dennoch können Fortschritte die bei der Programmierung für den Personalcomputer erzielt wurden, die Entwicklung von Programmen für Eingebettete Systeme vorantreiben. In dieser Arbeit werden Prinzipien und Entwurfsmuster der Software-Architektur bei der Entwicklung einer Programmumgebung für Benutzeroberflächen angewendet, die speziell auf Eingebettete Systeme zielt.

Die eigens entwickelte Sprache FUIML wird vorgestellt, die es erlaubt, effizient und abstrahiert von der Plattform Benutzeroberflächen zu erstellen. Mittels eines Compilers wird aus FUIML-Dateien ein maschinenfreundlicher Code erzeugt. Dieser wird dann von einem Programm auf dem Eingebetteten System interpretiert um die Benutzeroberfläche darzustellen.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Acknowledgements

It is with immense gratitude that I acknowledge the support and help of Dipl.-Ing. Mario Schwaiger, Dipl.-Ing. Andreas Feuersinger and Dipl.-Ing. Peter Randeu from Spintower KG. They have guided and supported me throughout this project. I wish to thank my Advisor, Prof. Dr. Oswin Aichholzer, for his support and guidance in the course of writing this thesis. I thank Prof. Dr. Keith Andrews for his LaTeX-skeleton that I have used as a template for this thesis. Last but not least, I am grateful for the support I have received from my family and friends.

Ferdinand Wörister
Oftering, Austria, September 2012

Contents

Chapter 1

Introduction

In the world of embedded systems technologies change quickly, time-to-market is critical and competition is fierce. Embedded systems are often designed for one task only. Their specialized nature, limited resources and great differences in architecture and protocols that are supported offer far less standardisation than their desktop-counterparts. Software that has been developed for a specific device often cannot be transferred to another without the need for significant change. In general, this is particularly true for the low-end range of chips available. Wherever price, dimensions or power consumption are limiting factors - as it is often the case for hand-held-devices - a lack of memory and processing power severely limits the choice in libraries that leverage development via use of existing, stable source-code. For a lack of middleware, programmers must often make use of low-level, platform-specific instructions that are not available on other devices.

On modern personal computers, high-level programming languages and software-libraries are available that offer a high degree of flexibility both in terms of choice of platform and operating system as well as in terms of programming paradigms and software-architecture. Abstraction layers that manage communication between applications and the system's hardware and protocols greatly facilitate the reuse of source-code. If cross-platform operability is taken into account at planning-phase of a project, a wide range of platforms can be supported both in the present and in years to come.

Graphic User Interfaces (GUIs) have long played a special role. By nature - for being comparatively costly in terms of processing power and therefore for their close entanglement with their hardware-platform as well as for their complex nature - they are hard to transfer from one system to another. In the present however, numerous frameworks are available that offer full cross-platform operability by in some way or another relying on a common principle that can be traced back to the early days of personal computers: by separating the application logic from a hardware- and platform-specific way of implementing the user-interface - *how* the interface is rendered. Instead, it is up to the target-device to construct a visual representation of the GUI - the application defines *what* is supposed to be rendered.

One of the oldest frameworks that followed the above idea is the X-Window System which was developed in the mid 80ies of the last century [Scheifler and Gettys, 1986]. To this day it is widely used as the default window manager on UNIX/Linux based systems and is presently owned by the non-profit X.Org Foundation [Coopersmith, 2012]. At the core of its architecture lies a client-server model. The clients are individual applications that use the X-library to encode and communicate requests such as "draw line from A to B" to the server. The server is a program that via hardware-specific libraries draws the application's requests on the screen. As part of the requirements formulated in the development process, the system must be network-transparent - an application running on one machine can connect via a network to an X Server that runs on another machine, regardless of differences in architecture [Peersman et al., 2011].

The X-Window System stems from the age of mainframe-computers. Consequently the client-server architecture was in part introduced as a means of allowing applications to be run on what at the time was

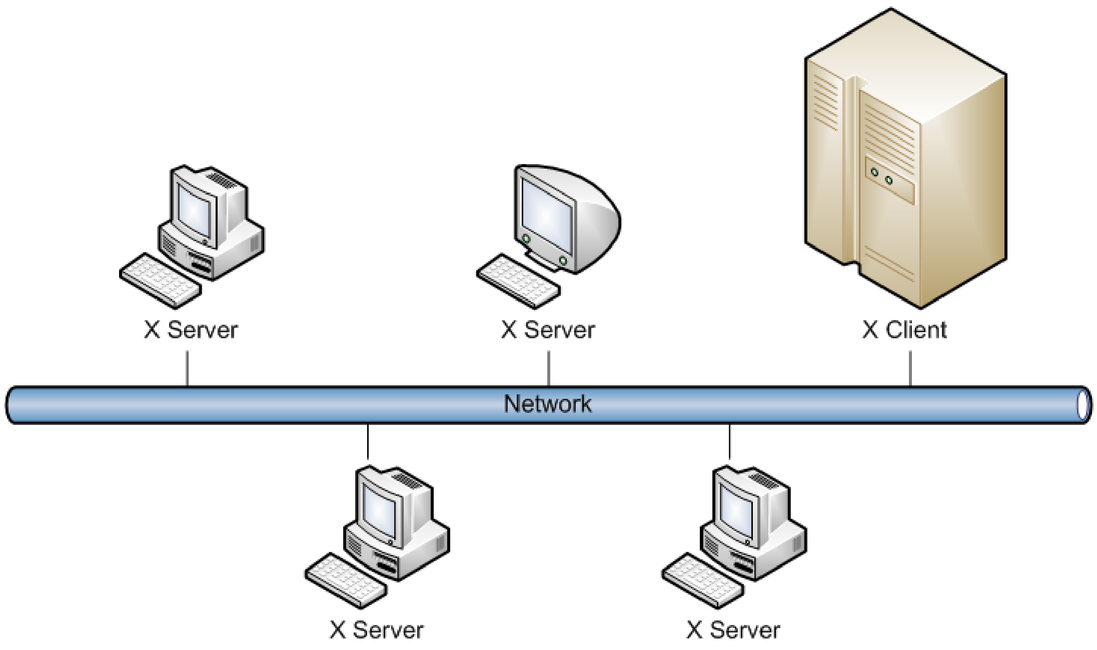


Figure 1.1: At the core of the X Window System lies a client-server model. Being network-transparent, an application running on one machine can connect via a network to an X Server that runs on another machine. (Image based on [Nye, 1994]).

considered powerful hardware with the operator controlling a less potent machine. However, the sheer fact that - in spite of criticism - the X Window System is still in use is proof of the virtues of the design. In addition, the success of other standards such as HTML have since drastically proven that the common denominator of the two - separating the *how* from the *what* can be used to create software that indeed can stand the test of time.

In the following chapter, the device that is in focus of this document will be presented along with a number of applications that will make use of the GUI-framework FlashUI. After that, the considerations that have led to the decision of implementing FlashUI on a dedicated graphics-controller that is independent of the device's main processor are elaborated. At the end of the chapter, a number of goals is formulated that constitute the core motives that have been at the heart of FlashUI's development.

Chapter 3 introduces influential patterns and principles in GUI-architecture along with a number of patterns that from today's point of view are considered deprecated. After a brief discussion about the virtues of the individual designs in the light of the context at hand, the spotlight shifts to another aspect of GUI-development: design-tools in general and ones that target embedded systems in particular.

Chapter 4 provides an overview of the XML-based markup-language *FUIML*, that has been developed in the course of this project. At first general language properties are explained, later in the chapter individual widgets that are supported by the framework are presented.

In Chapter 5, the application that is used to translate the above FUIML-user-interface into a machine-friendly bytecode is introduced.

Chapter 6 describes the firmware that is used to render compiled bytecode to the screen of the device. At first, a number of general concepts is introduced. Towards the end of the chapter the process of interpreting individual widgets is explained along with an overview of the difficulties that have been encountered in the course of the interpreter's implementation and the measures that have been taken to overcome them.

The above decision to separate the user-interface from the main-processor gives rise to the need for a number of protocols that facilitate communication between the two embedded chips - they are introduced in Chapter 7.

The focus of Chapter 8 lies on a number of tools that have been implemented along with FlashUI's core-components. They include a tool that facilitates tests of the above protocols, a tool that can be used to upload maps onto the device and another tool that has played a major role in the development-process: one that simulates the interpreter on desktop-machines in order to facilitate debugging of the chip's firmware.

In the final Chapter 9, the framework is evaluated in the light of the goals that have been stated in Chapter 2.

Chapter 2

Motivation

2.1 The Application

The objective at the beginning of this project lay in the creation of a GUI-framework for use on a family of hand-held devices. The devices are optimized for low power consumption, mobility and ease-of-use. For their small footprint, the devices can comfortably be worn on the wrist. Several applications are planned that will be based upon the framework, they include:

- **ADELE4home** - medical-aid-device for senior citizens [Schwaiger, 2012a] that gathers biometrical data such as its carrier's heart-rate and temperature and automatically contacts medical staff in case of unusual or dangerous values. Also, the device includes a shock-sensor that measures g-forces. If certain characteristics are measured that indicate a heavy fall of the person carrying the device, medical staff is contacted. To facilitate communication with physicians and other health professionals, the device can receive text-messages and initiate phone-calls. However, only a limited number of contacts can be called - the device does not offer a number-pad. Via a GPS-sensor, the device can keep track of its carrier's position, which in case of an emergency facilitates the rapid arrival of help. For the target-customership, ease-of-use as well as barrier-free-use is considered of vital importance since on the one hand, senior citizens tend to have less experience in the operation of modern electronics, on the other hand, members of the above group are more often physically challenged than other target-groups. See Figure 2.1 for an image of ADELE4home.
- **A cycle computer** for professional athletes and sports enthusiasts that like the above device gathers biomedical information about its carrier. Via the built-in GPS receiver, the device keeps track of the cyclist's progress on a pre-defined route. Both the biomedical information as well as the latter can be displayed either on the device itself as well as via a web-front-end. This enables training personnel to both observe and - via text-messages or via speech - instruct the athlete in real-time from a remote location. See Figure 2.2 for an image of the device.
- Other fields of application include manned security, event security, on-site training, mission documentation, lone-worker safety, team coordination, stress monitoring and live data tracking [Schwaiger, 2012b].

While at first glance, the above applications differ greatly in terms of domain as well as in the customership that is addressed, they share numerous requirements.

- **They are all intended for a highly mobile use**. Power-consumption must be kept at a minimum so that devices can remain operational for multiple days. In addition, the devices cannot exceed a certain size so that they do not interfere with their carrier's freedom of movement and offer a high degree of wearing comfort.

Dimensions	86mm x 52.5mm x 18.6mm
Main Processor	Low-Power Cortex M3 SoC
Weight	100 g
Display	2.0 inch sunlight-readable TFT display
Water resistance	up to 3 meters
Sensors	3D accelerometer
	GPS receiver
	3D compass
	Barometric height
	Optional: COx and VOx gas detectors
Connectivity	Quadband GPRS data transceiver
	Quadband GSM voice function
	ANT+ transceiver
	RF mesh transceiver

Table 2.1: Specifications of the Live Data Tracker [Schwaiger, 2012b]

- **Ease-of-use is essential**. Senior citizens are often overwhelmed by modern mobile phones that offer countless ways of customization and features. For this reason, a booming market has developed that caters to the special needs of the above customership. They focus on key functionality instead of gimmicks and flashy graphics. Whereas the other target groups may be more acquainted to feature-rich hand-held devices, in the particular circumstances, the device must offer the same characteristics as for the above group. While cycling, the user can spare little attention to the operation of the user-interface.

- **The less interaction the better**. Modern smartphones are designed to capture all of their user's attention. Browsing the internet, typing emails or taking pictures are processes that require a user to stop and focus on the task at hand. They all demand sophisticated ways of interaction while the planned devices are intended to be used as an infrastructure. The less they interfere with the user's activities the better. Hence, there is little need for application-specific, complex input-controls.

- **Numerous common components**. From the perspective of the user-interface, most devices focus on the display of data while interaction is limited. Common widgets for charts, maps, buttons and labels can with, little adaptations, be used on multiple devices.

These synergies suggest a common platform that can be used in multiple roles. Essentially, a hardware-platform can be developed that differs only in the user-interface that is installed. There is little benefit in implementing each user-interface from scratch when most of it will be used several times in different applications - hence a common framework can be used the minimize effort that is required for individual implementations.

Apart from the additional effort that is required for a custom-built solution for each application, a common code-base offers a higher grade of reliability and a lower cost of maintenance. Changes and fixes

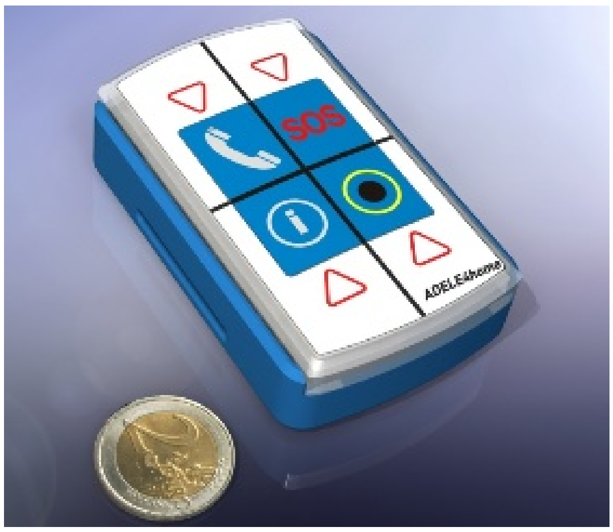


Figure 2.1: An image of ADELE4home

on one application can immediately be applied to another, stable components that have been developed for one application can be used to leverage the development of another. The other principles must be addressed in a different way.

In spite of the limited set of functionality - in comparison to those available on today's smartphones - that is boasted by the previously introduced applications, the amount of software that is required to drive the platform is considerable and differs greatly in fields of expertise. However, there are sharp boundaries between individual concerns:

- A number of web-front-ends must be created. Whereas software that is intended for use on an embedded system mostly is implemented in a low-level programming language that offers little abstraction from the nuts and bolts of the hardware, the implementation of a web-front-end requires a completely different set of knowledge and expertise.
- A large proportion of the implementation concerns the infrastructure of the device: gathering and processing sensor-data as well as other hardware-related code such as the management of data-transmissions via wireless networks.
- The on-device user-interface constitutes a component that overlaps little with other parts of the system. In this module, flexibility is particularly important and it has long been established as good-practice separating the GUI from other components. From an implementation-standpoint the user-interface is more related to the low-level nature of the above device-infrastructure, yet in many aspects a different set of skills is required.

By splitting the implementation along the above domains of concern and via a carefully designed minimal interface between them, the effort can be split into several teams. On the one hand, this allows for a division of work that potentially lowers time-to-market, on the other hand, via a modular design individual components can be modified or even exchanged with little impact on other parts of the system.

In case of the user-interface, several options arise:

- The main processor on the device is responsible for rendering the GUI on the display. In theory, this approach may offer a slightly lower price per unit since no additional chip is needed. However, it reduces the choice of available processors since on the one hand, enough program-memory must be available for additional display-drivers and graphics-libraries, on the other hand the chip must be capable of addressing and communicating with a display unit. A further downside of this

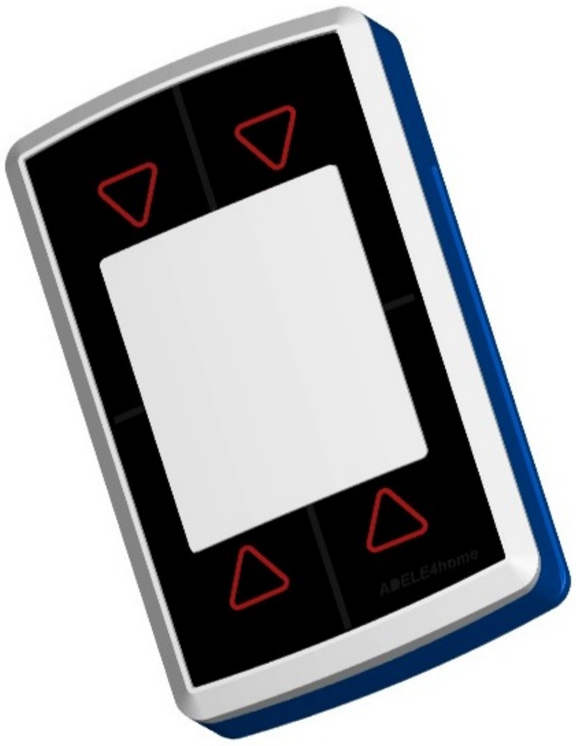


Figure 2.2: An image of the Live Data Tracker

design lies in the fact that two teams residing on different locations work on the same processor which boasts a highly limited amount of resources and lacks layers of abstraction that facilitate the creation of slender, efficient interfaces between components. Also, in case of an exchange of hardware two modules are affected instead of one.

- A separate graphics-processor is used to render the user-interface. A dedicated processor that has been designed specifically as a graphics-controller allows for a smaller main processor to fulfil the above tasks since most offer built-in drivers and functions for communication with common LCD-display units. Additionally, being a physically separated chip that communicates only with the main unit via lightweight interface allows for an exchangeability of the entire user-interface without affecting the core-unit. Potentially it is a more stable design: if the graphics-processor encounters an error, it can be reset by the main-processor without risking the loss of data. In the other design, an error in either component causes the entire system to fail.

While in terms of cost-per-unit, the one-chip option may be the better choice, for the above considerations of flexibility and separation of concerns, the second option has been chosen. The design that relies on a separate processor for the display of the user-interface in some aspects resembles that of the X-Window System and countless other successful designs such as the World Wide Web. It introduces a sharp distinction between the application itself and its visual representation. In addition, the graphics controller that has been chosen - 4D System's GOLDELOX GFX2 - boasts a number of virtues that are of paramount importance for the device at hand [4D LABS, 2011]:

- A size of only 6 by 6 mm.

- A typical power consumption of 12 milliamperes at 3.3 volts.

2.2 Goals

On its own however, in spite of its virtues the choice in hardware cannot entirely fulfil the requirements for the user-interface. Neither is it the intended focus of this document. Instead, it lies on the creation of a GUI-framework that is robust and flexible enough to be adapted to the individual requirements of the applications that were introduced before and future ones that may follow. For this to be achieved, the framework must fulfil a number of key requirements:

- **Platform Independence -** The framework must offer a platform-independent way of specifying the appearance of the user-interface. The design of the GUI must be transferable to other platforms without major modification.
- **Reuse of Components -** The framework must offer a common set of common components that while being adjustable to the distinct needs of the individual applications can be reused without modification.
- **One Framework for all Applications -** The same framework must be applicable for all applications without modification.
- **Efficient Design-Tools -** The framework must offer an abstracted, easy-to-use and efficient way of designing the GUI, that can be used by graphics-designers who lack intricate knowledge of the hardware-platform used and the implementation built upon it.
- **Modern Look-and-Feel -** The visual appearance of the GUIs that are based upon the framework must be modern and dynamic, interaction with the device must be seamless and intuitive.
- **Efficient Use of Resources -** For the limited features that are available on the target-platform, the framework must efficiently exploit those resources that are available in plenty and compensate for those that are lacking.

2.3 Applying Proven Concepts on Embedded Systems

Computer-science is still a comparatively young branch of engineering - however, its ”wild-west”-days are long gone. As the success of technologies such as the X Window System and HTML demonstrates, at least in the field of software-architecture, computer science can no longer be considered a frontier science. In the past decades a number of blueprints to good design have been established that if applied correctly, can leverage the successful implementation of new software based upon lessons learned in the past.

About a quarter of a century after the introduction of the above technologies, there is broad consensus in the world of software-engineering of a number of principles that like the above have been proven correct by the test of time. Among those that developers at Microsoft are encouraged to adhere to are [Microsoft Patterns & Practices Team, 2009]:

- **Separation of concerns** Applications should be divided into distinct features with as little overlap in functionality as possible. The important factor is minimization of interaction points to achieve high cohesion and low coupling. In the field of computer-science, the term may have been coined by Dijkstra, who considers separation of concerns a technique that facilitates ”focusing one’s attention upon some aspect” [Dijkstra, 1982].
- **Single Responsibility** Each component or module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionality.

- **Principle of Least Knowledge** A component or object should not know about internal details of other components or objects.
- **Don't repeat yourself** Functionality should not be duplicated in any other component. Instead, existing functionality should be reused among components. This principle was originally proposed by McIlroy [McIlroy, 1968] who encourages mass-production of software via the application of reusable, standardized building blocks.

While Microsoft's guidelines may not constitute an exhaustive list of valuable lessons learned in the field of software engineering during the past decades, in the author's opinion they nonetheless constitute a highly abstracted and condensed blueprint for good software design. The above principles can and *should* be applied to any implementation, regardless of the programming paradigm that is used and regardless of the type of the application.

In the world of embedded systems change happens - it happens fast and sometimes without warning. Thus, it is even more important to adhere to established design principles that have leveraged progress on personal computers. The goals that were introduced earlier in this chapter represent *what* the GUI-framework aspires to achieve, design-principles offer a foundation that can used to leverage its implementation - *how* the individual goals can be achieved. While the principles that are encouraged by Microsoft offer a more general set of advice, it is in the next chapter that patterns are introduced that specifically target GUI-architecture. Whereas these patterns target desktop-applications that - unlike the application at hand - govern large amounts of data, a number of benefits can be transferred to the FlashUI-framework. The achievement of the previously stated goals can be facilitated by adhering to established designs that have offered a blueprint to good design in countless other applications. While the X Window System and HTML vary greatly in terms of technology, they nonetheless share a set of common ideas that are an intricate part of their success-story. In the same way a part of FlashUI's road to success can be outlined by the patterns that are introduced in the following chapter.

Chapter 3

Existing work

In the following pages, at first a number of patterns is introduced that either represent the state-of-the-art in GUI-architecture, or have played an influential role in the latter's development. Since to a large extent, however, GUIs owe their success to the design-tools that are used in their development, later in this chapter a number of general approaches to GUI-design is evaluated. After that, a selection of tools that aim at the embedded-systems-market is introduced.

3.1 Patterns in GUI-Design

3.1.1 The Naive Approach

At its core, a GUI consists of a number of primitives that are arranged on screen. A naive programmer that lacks knowledge of the advances in GUI architecture over the past decades may adhere to an approach that step-by-step instructs the machine to draw individual primitives on screen. A button, so the programmer may think, consists of a border and some text or an image - a few lines of code will suffice to draw a button on screen. Consequently, whenever a button is needed the diligent programmer will write an instruction that draws a rectangle, one that inserts a label or an image into the rectangle and finally implements a routine that is called whenever the mouse-button is pressed within the rectangle. For a lack of better knowledge, the programmer will make little distinction between instructions that concern the logic of the user-interface and ones that are part of the application's domain-specific logic. Proudly, the programmer will present the application to the customer - who in turn insists that all rectangles be drawn with rounded corners.

Whereas in the this example, the effort that is needed to replace all rectangles with ones that boast rounded corners is limited it demonstrates how reuse of code leverages the implementation of any software-project. GUIs often require a common set of components that differ only in certain details from one another. The principle of reuse implies that all components that share appearance or functionality should share a common implementation. In this way, the naive programmer would only have to replace perhaps a single line, instead of laboriously changing the appearance of the rectangle in a multitude of places. To an experienced programmer living in the world of today, the inherent flaws of the naive approach are obvious:

- Time is wasted by constructing the same component from scratch over and over.
- Even minor changes may affect multiple parts of the application.

In fact, to a programmer that is used to modern GUI-frameworks, the above may seem far-fetched. However, low-end embedded graphics processors often lack a set of ready-to-use components such as buttons. Via a set of functions that render primitives such as those mentioned earlier on screen, the entire user-interface must be built from scratch. On the one hand, it is for their limited amount of program-memory that generic functionality has to be kept at a minimum, on the other hand it is for their inability to render user-interfaces that in terms of complexity rival those available on personal computers, that in the world of embedded systems the naive approach is still in use. Additionally embedded systems often cannot be programmed in languages that support object-oriented programming, which in modern GUI-frameworks is part of the foundation of the architecture.

3.1.2 Forms and Controls

While in a strict sense the Forms and Controls does not qualify for a design-pattern, in fact it is at times considered an anti-pattern - one that represents an inferior approach, it is still widely used. Its name was coined by Fowler [Fowler, 2006] who for his contributions to software-architecture will be referenced frequently in the course of the following pages. What is meant by the above euphemism is an architecture that does not separate an application's UI-components from those that contain domain-specific logic and -data. Often aided by a visual design tool, to a large degree the application consists of event-handlers that upon interaction with the GUI modify and request data. In the designs that follow, data is usually abstracted in a way that makes components of the user-interface oblivious to the nuts and bolts of both the data itself and of the infrastructure that is used to access the data. In this approach not necessarily but frequently data is accessed directly by the user-interface. It is thought of as a record-set instead of a domain-model. For a number of reasons, the pattern is considered bad-practice:

- There is a lack of separation between individual concerns which leads to an increase in the amount of dependencies among components.
- For the above separation of concerns, individual components can less easily be reused than in other designs.
- The architecture often cannot be transferred to a different platform. If the target platform lacks support for the GUI-framework, a new implementation that is based upon another framework is needed.

While for the above reasons Forms and Controls is considered an anti-pattern for large business applications, there are some benefits that cannot be offered by any of the designs that follow:

- The implementation causes little overhead in terms of effort that is otherwise required for abstraction-layers.
- It is based upon an intuitive, easily understood concept. There is no need for an expertise in object-oriented programming which makes the design particularly well suited for persons with limited experience in programming.
- For the task-oriented nature of the approach - all code that handles a particular use-case is often aggregated in a single function - the source-code can be more accessible than that relying on other patterns.

In addition to the above, for relying on software-libraries that offer a number of built-in controls and for the support for well encapsulated custom controls that can be reused in other projects, it represents a significant improvement over a from-scratch implementation of a user-interface. However, apart from the world of low-end embedded graphics-processors, specialized libraries that offer generic controls such as buttons and labels as well as the possibility of creating custom controls are features that are supported by virtually all modern programming languages and the GUI-frameworks they support. It applies for all architectures that are discussed in the following pages.

It can thus be concluded that for small-scale projects that neither rely on large amounts of heterogeneous data nor offer complex interfaces for changing the latter nor require a certain level of flexibility in terms of the infrastructure that is used to provide data, the design can at times be a good choice. *Keep It Simple, Stupid* (KISS) - some projects do not benefit from a more sophisticated architecture, whereas the adherence to a more complex pattern often causes problems of its own as well as additional effort.

3.1.3 Separated Presentation

What the previously introduced patterns have in common is a tendency of entangling presentation-logic with business-logic. In part, Parr attributes the wide-spread use of the above patterns to a fear of loss of control that is common among programmers [Parr, 2004]. Parr considers separated presentation an almost unanimously accepted principle that boasts numerous virtues. Whereas Parr discusses web-applications, most of the attributed benefits also apply to other platforms:

- **A high degree of encapsulation** of presentation and business logic that minimizes the interdependencies.
- **Clarity:** the template is immediately visible to the programmer, there is no hidden behaviour.
- **Division of Labour:** in parallel to the programming of the application, the design of the user-interface can be created by a graphics-designer.
- **Component Reuse:** designers can break down complex components into smaller templates that can be reused.
- **Single point of Change:** Instead of modifying the same aspect of the design in multiple locations, a single template that is used in multiple places can be changed.
- **Maintenance:** Instead of a change in the program, changing the look of the application can be done by changing a template.
- **Interchangeable Views:** The look of an entangled data model and display cannot easily be changed.
- **Security:** A lack of control-structures increases security.

Fowler's approach to the concept is a more general one that applies not only to web-applications. To him, a separated presentation is achieved if code that manipulates *only* manipulates presentation. He sees the acceptance of the principle as a consequence of the widespread use the following design pattern:

3.1.4 The Model-View-Controller Design-Pattern

The Model-View-Controller (MVC) pattern was first introduced by Xerox PARC in the late 1970's of the last century. As [Burbeck, 1987] points out, it was adopted in the early days of personal computers first by the "developers of the Apple Lisa and Macintosh and, in turn, by the Macintosh's many imitators." As shown in Figure 3.1, at the root of the pattern lies the triad of model, view and controller as well as the communication between them. Krasner and Pope describe the individual components of the triad as follows [Krasner and Pope, 1988]:

- *The model* manages the behaviour and data of the application domain. In the simplest case it can consist of a single base-type such as an integer or a string. On the other hand, the model can be a complex class containing numerous properties.

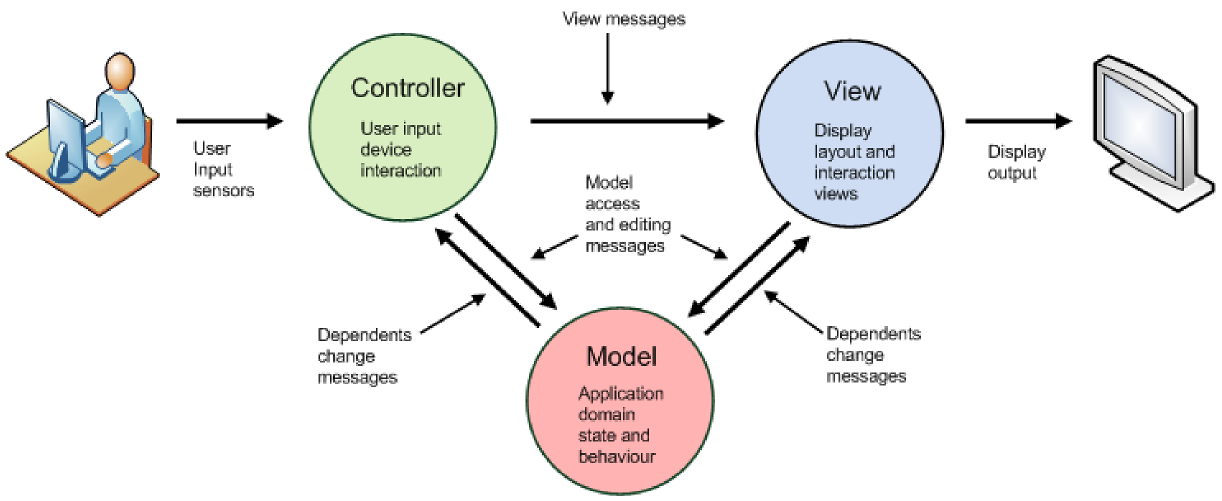

View messages
Controller
User input device interaction
View
Display layout and interaction views
User Input sensors
Display output
Model access and editing messages
Dependents change messages
Model
Application domain state and behaviour
Dependents change messages

Figure 3.1: Model-View-Controller State and Message Sending. Image based upon [Krasner and Pope, 1988].

- *The view* is concerned with everything graphical. It requests and displays data from the model. Views can be nested within each other with the root-view - in many cases the window itself or the standard system view of the window - being responsible for graphical transformations and the clipping of the content of its sub-views.
- *The controller* on the one hand manages the interface between input-devices such as the keyboard and mouse and their associated model and view, on the other hand it schedules interaction with other view-controller pairs (e.g. mouse-movement from one application view to the other).

Whereas typically, both the view and the controller have exactly one model, models can be associated to multiple views and controllers. In fact, the model should not know about its views. The controller is responsible for managing change within the model upon user-input. When the model changes, all associated views and controllers must be changed, not only those that triggered the change. The Observer-Pattern provides a widely used solution to this dilemma. Individual components may subscribe to a list of observers that are notified whenever a change in data occurs.

In the time since its introduction, countless other patterns that build upon the MVC pattern have emerged, such as the Model-View-Presenter pattern which will be explained in the next section and Microsoft's MVVP pattern. However, its core virtue remains valid to this day. For a lack of words that offer a more concise description of the design's foundations, Fowler is cited:

At the heart of MVC, and the idea that was the most influential to later frameworks, is what I call Separated Presentation. The idea behind Separated Presentation is to make a clear division between domain objects that model our perception of the real world, and presentation objects that are the GUI elements we see on the screen. Domain objects should be completely self contained and work without reference to the presentation, they should also be able to support multiple presentations, possibly simultaneously. This approach was also an important part of the Unix culture, and continues today allowing many applications to be manipulated through both a graphical and command-line interface.
[Fowler, 2006]

3.1.5 The Model-View-Presenter Design Pattern

[Fowler, 2006] describes the Model-View-Presenter (MVP) pattern as a design that unifies some of the advantages of Forms and Controls and those of the MVC pattern. Whereas Forms and Controls offers an easily understood concept it lacks the main benefit of the MVC pattern - separation of concerns.

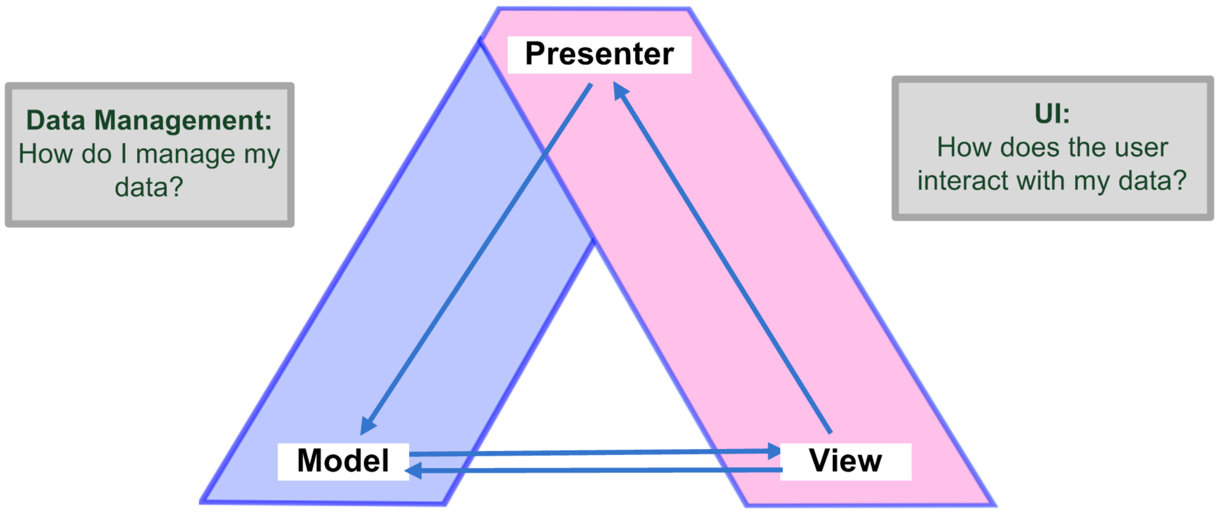


Figure 3.2: According to Potel, programming problems can be broken down into two fundamental concepts: Data Management and User Interface. Image based on [Potel, 1996]

In Fowler's words: *What it lacks, and MVP has so strongly, is Separated Presentation and indeed the context of programming using a Domain Model. I see MVP as a step towards uniting these streams, trying to take the best from each.* [Fowler, 2006].

The pattern was originally described by Potel [Potel, 1996]. Much like in the Forms and Controls design, Potel considers views as a collection of widgets that correspond to the controls of the latter design. Furthermore, he removes any distinction between the controller and the view. The handling of interactions with the above views is delegated to the presenter which in turn decides how to update the model. The view is notified about changes in the model in the same way as in the MVC pattern: via an Observer-pattern. With the advent of modern operating systems that offer built-in management of inputs such as mouse-movement and keyboard inputs, there is no need for a controller. Instead upon interaction, the view's widgets notify the presenter - the heart of the application [Bower and McGlashan, 2000] - which in turn controls interaction with the domain-model. Hence, the MVP pattern represents an adaptation of the MVC pattern to the realities of modern computing.

3.1.6 The Model-View-ViewModel Design-Pattern

The Presentation Model was first introduced by Fowler [Fowler, 2004] and has since been adopted by Microsoft for their family of XAML-based GUI frameworks - WPF and Silverlight - by the name *Model-View-ViewModel (MVVM)* [Smith, 2009]. The presentation model refers to a class that contains all information about the state of a view - e.g. the widget or window - yet does not contain references to the latter. Via a mechanism of synchronization, the view's properties are bound to fields within the above class and automatically reflect changes in the bound fields onto the user-interface. In the case of a check-box, the control is bound to a boolean-field that resides in a class that represents the presentation model. Instead of directly manipulating the check-box the value it is bound to is changed. Via this mechanism, presentation-behaviour is separated from the view which brings a number of benefits over other architectures:

- The design facilitates a strong separation of concerns between the application's presentation and the logic that drives the UI.
- Hence, both components can be exchanged and modified with little impact on one another.
- Furthermore, since much of the behaviour of individual controls is defined outside of the control itself, code-reuse is supported.

- A single application can support several presentations.
- The application can be tested without regard for the GUI.

In the context of WPF and Silverlight Microsoft calls the concept of binding a control's properties to a presentation-model *DataBinding*. In the declaration of the user-interface - individual XAML-files that define a control's presentation - a *DataContext* is assigned to each control. That of the root-control is inherited by its children. Controls reference fields within their DataContext-object by name, most of the time they are oblivious of the class of the object. Via a modified *Observer Pattern*, controls are kept in synchronization with their *DataContext* and without additional code immediately reflect changes in the presentation model.

3.1.7 Discussion

If the project that is described in this document was an application that targets personal computers, the choice would be easy: advanced design patterns such as MVVP or MVP have asserted their benefits in countless applications around the world and numerous modern Integrated-Development-Environments (IDEs) are available that include sophisticated tools for GUI-design.

The application that is presented in this document, however, differs in various key aspects from its PC-counterparts:

- Low-end processors that are suitable for the device that is used lack support for object-oriented programming which lies at the heart of most of most modern design patterns.
- In patterns such as MVC and its derivatives, perhaps the main motivation towards a more sophisticated software-architecture stems from the distinct requirements of applications that manage large amounts of complex data, whereas the applications at hand feature a limited amount of data that can rarely be manipulated via the on-device user-interface.
- For the limited resources that are boasted by embedded processors and for a lack of certain key features that are unsupported by those programming languages that are available, otherwise commonly used patterns such as the Observer Pattern are infeasible.

The application at hand requires a different approach from one that targets a more potent platform. However, while neither the MVC-pattern nor one of its more modern counterparts can be adhered to without modifications, many of their benefits can be conveyed to this project. All of the above patterns are characterized by a strong separation of concerns. The domain-specific control-flow is encapsulated from the user-interface, which in turn is separated from the infrastructure that is used to handle user-input and to render the screen. For the use of two processor on the device - one that is used to process domain-specific functions such as the control of built-in sensors as well as the administration of network connections and one that deals with all aspects of presenting the latter on screen - a design has been chosen that strongly benefits from the same virtues that are associated with MVC and its relatives. By decoupling the user-interface, not only can it be changed or even replaced without affecting other components, it also facilitates a division of work between engineers that are concerned with GUI-related tasks and ones that focus on the implementation of domain-specific parts of the system.

3.2 Approaches to GUI-Design-Tools

3.2.1 The Imperative Approach

In the early days of personal computers, before the advent of high-level programming languages, design paradigms such as object-orientation and patterns such as MVC, the entire layout, appearance and functionality of the interface was defined manually in source-code. Thus, this is arguably the oldest way of defining a user-interface.

In this approach, the user-interface is implemented in the same way as other components of the software. Hence it is particularly tempting for programmers to violate established concepts such as separated presentation in the course of a 'quick and dirty' fix for certain problems. While in modern frameworks such as Microsoft's family of XAML-based languages it is often considered bad practice to rely heavily on the *code-behind*, most tools for interface-design such as Apple's Interface Builder, the above XAML-based languages WPF and Silverlight and Windows Forms require certain aspects of the GUI to be implemented via source-code.

In the field of low-end embedded systems, the sole reliance on the imperative approach is particularly wide-spread. The market consists of a plethora of highly specialized micro-controllers, each boasting unique virtues, limitations and designs. On the one hand it is the latter differences in platforms, on the other hand the often limited number of features that are required that often make the programmatic approach the most cost-effective. Apart from the above, it is also the trade-off in performance that makes this approach particularly suited for embedded systems - in the same way as high-level programming languages are less efficient than custom-built assembler code, user-interfaces that stem from a designer that creates code often demand more resources than their 'hand-made' counterparts. In the world of embedded systems, where processing-power is a sparse resource and device-characteristics vary greatly, a 'one-size-fits-all' approach to GUI-design is often out of the question - be it for the lack of compatible third-party frameworks, or for limitations in terms resources available on the device.

3.2.2 The Drag-And-Drop Approach

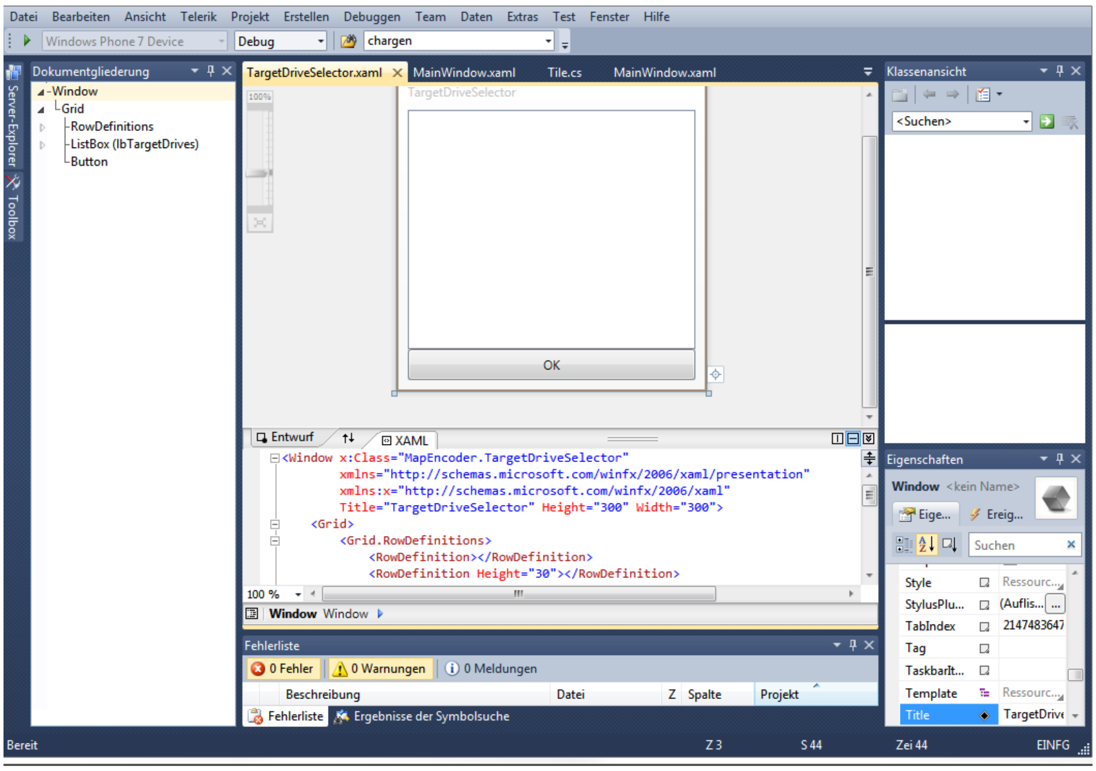
Drag-And-Drop design-tools, usually based upon a WYSIWYG (*W*hat *Y*ou *S*ee *I*s *W*hat *Y*ou *G*et) preview of the user-interface facilitate the definition of GUIs via a visual representation of the interface. Commonly used elements such as buttons, text-boxes and labels can be arranged on screen by mouse, alleviating the developer or designer from the often tedious procedure of typing countless properties such as an element's position on screen by hand. Mostly, these tools serve as an alternative to the programmatic approach - they provide easy access to a certain subset of the features available in code-behind whereas some aspects of the interface must be defined in the source-code of the program.

Similar to the field of word-processors, WYSIWYG-design tools are often considered more easy to learn and more intuitive than other approaches to GUI-design. Among the most wide-spread representatives of this approach are Microsoft's Windows Forms and Apple's Interface Builder which is considered one of the first WYSIWYG design-tools.

3.2.3 The Declarative Approach

In recent years a trend has emerged that in the author's opinion is strongly tied to the success of HTML. Instead of defining the appearance of the user-interface in a WYSIWYG-tool by arranging individual controls by mouse, a markup language - usually derived from XML - is used to design the GUI. Instead of focusing on the *how*, declarative GUI-tools emphasise the *what* [Ferguson, 2009]. As in the case of HTML, the nuts and bolts of rendering the screen are determined by the device that displays the user-interface. Instead of platform-specific controls, as it is usually the case in the Drag-And-Drop approach, an abstracted specification is used that numerous platforms can interpret in their own way.

Since by definition, markup-languages lack features that are suitable for the definition of domain-specific logic, the approach automatically enforces a high degree of separation between the application logic and the attached presentation components. Via this high degree of separation, clearly distinct roles emerge that facilitate division of work between software engineers and designers. In addition, for a lack of control-flow that is defined within the visual components of the user-interface and in turn for an enforcement of separation between concerns, the reuse of controls is facilitated.



```
<Window x:Class="MapEncoder.TargetDriveSelector"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="TargetDriveSelector" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition Height="30"></RowDefinition>
```

Style	Ressourc...
StylusPlu...	(Auflis...
TabIndex	2147483647
Tag	
Taskbarlt...	
Template	Ressourc...
Title	TargetDrive

Figure 3.3: A screen-shot of Microsoft Visual Studio's GUI-designer for XAML-based languages such as WPF and Silverlight. The user-interface can be created either by manipulating its visual representation by mouse or by modifying the XAML-code below. In addition, all widgets can be declared and arranged via source-code. Hence it constitutes a hybrid of all approaches to GUI-design that are currently in use.

3.2.4 Hybrids

Whereas most tools that are available are designed with one of the previously introduced approaches in mind, many can be used in either way. Design-tools that focus on the declarative definition of the GUI often boast support for the Drag-And-Drop method and in turn the GUI can usually be constructed in a strictly imperative way within a programming language.

Among those that support all three of the above approaches is Microsoft's XAML design tool that is an integrated part of Visual Studio from version 2008 and upwards. For a screen-shot of the above see Figure 3.3.

3.2.5 Discussion

Among the approaches to GUI-design that have been introduced in the previous pages, the imperative approach is the one that is not only supported by virtually all platforms that are available, it also requires the least effort from a short-term perspective since no additional software for the design of the user-interface must be implemented. The main downside of the approach lies in the lack of distinction between the software-infrastructure - the nuts and bolts of rendering the screen - and the actual design of the user-interface. For a lack of separation between the *how* and the *what* [Ferguson, 2009] the imperative approach leads to poorly maintainable, platform-specific implementations.

For instead of relying upon the abstract syntax of a programming language providing an intuitive, visual representation of the user-interface that can be modified by mouse it is more accessible than the imperative approach. For relying on the concept of a set of common controls that can be arranged on screen it enforces code-reuse. Whereas for these reasons, the drag-and-drop approach constitutes a step forward from the imperative one, it is not free of drawbacks: The implementation of the design-tool constitutes a considerably increased effort in comparison to both the imperative approach as well as the declarative approach. The drag-and-drop approach is widely used - not only for GUIs that target the PC, but also in the world of embedded systems, as the tools that will be introduced in the next section demonstrate.

In recent years however, markup-languages have gained more and more popularity among software-engineers and GUI-designers alike. They offer a sharp separation of the structure, layout and functionality of the user-interface and boast a more precise and compact way of representing the GUI than drag-and-drop-tools [Hanus and Kluß, 2009] - whereas in a GUI's textual representation, a widget's properties are immediately visible, in most graphical editors only the properties of the selected one are visible. In the author's opinion both types of design-tools represent viable options. From a user's perspective, it is a question of personal preference - and what in case of this project has tipped the scale in favour of an XML-based design approach is just that, the personal preference of the stakeholders involved.

3.3 GUI-Frameworks for Embedded Systems

3.3.1 MicroXWin

Apart from its role as a window manager for personal computers, the X Window System that was introduced in Chapter 1 can also be used on embedded hardware. However, while the its network-transparency boasts numerous virtues, it comes at a price: the increased overhead of the client-server model and the overhead caused by networking components leads to an increased demand in resources which constitutes a major drawback when considering its use on embedded processors. However, alternative implementations are available that claim to have a reduced footprint: The binary compatible MicroXwin can significantly reduce the demand in resources by replacing the above client-server architecture with kernel modules that facilitate communication among components. On their website, the MicroXWin-staff claims boasting twice as fast graphics and a memory use of less than 0.5 megabytes versus 29 megabytes in the case of the X.Org Server [MicroXwin, 2011].

3.3.2 X-Nano

X-Nano, formerly known as Microwindows is a window manager that is designed for hand-held devices. Although no operating system is required at all, it typically operates on top of the Linux-kernel or a derivative of Microsoft Windows such as Windows CE. User-Interfaces can be created via the imperative method introduced earlier. The framework includes built-in support for keyboard-, mouse- and touch-inputs. Its Application-Programming-Interface (API) boasts routines for window-dragging, title-bars and displaying messages. According to [Haerr, 2010] on a 16 bit platform it requires about 64 kilobytes of program memory.

3.3.3 The GIMP Toolkit

The GIMP Toolkit, also known as GTK+ is a multi-platform tool-kit for graphical user interfaces. It is licensed under the the GNU LGPL [The GTK+ Team, b]. The framework itself is written in C, yet it can be used in numerous programming languages such as Java, Javascript, Python, Lua and C/C++ [The GTK+ Team, a]. Although GTK+ can operate directly on the on the built-in graphics memory of displays, it usually builds upon a window manager such as the X-Server. User-interfaces can either be created via a declarative approach based on XML or via third-party design-tools:

- A built-in component of the framework called *GtkBuilder* can be employed to de-serialize GUIs from XML. The latter may either be loaded from a file or it may also be included as a constant string within source-code.

- Third-party tools such as *Glade* allow for the creation of GUIs via a visual design-tool by mouse. In the case of Glade, the user-interface can be exported to XML-code that is supported by the above GtkBuilder [Tristan Van Berkom, 2012].

Among popular projects that build upon the GIMP Toolkit is the GNOME desktop environment.

3.3.4 Qt

Qt is a GUI-framework that was originally developed by the Norwegian company Trolltech, which was acquired by Nokia. It is licensed under the LGPL version 2.1 [Nokia Corporation, 2012]. Qt has found wide acceptance in the field of software-engineering, not only for its GUI-library but also for other components of the framework that among other things concern networking, XML and databases [Wolf, 2007]. In part, the story of Qt's success can be attributed to its cross-platform availability: it is available for virtually all major operating systems that are designed for use on personal computers as well as countless embedded systems. Qt can be customized to reduce file-size - for a typical scenario, the package requires about 10 megabytes [Burns, 2009].

What distinguishes the Qt-framework from the others that are presented in this chapter is its support for a declarative definition of the GUI via the language QML. In contrast to most other markup-languages, QML is not derived from XML. The Integrated-Development-Environment (IDE) that is supplied by Nokia includes two design-tools:

- **The Qt Quick Designer** which is based upon the above QML allows for the declarative definition of GUIs.

- **Qt Designer** which is a design-tool that follows the drag-and-drop approach to GUI-design.

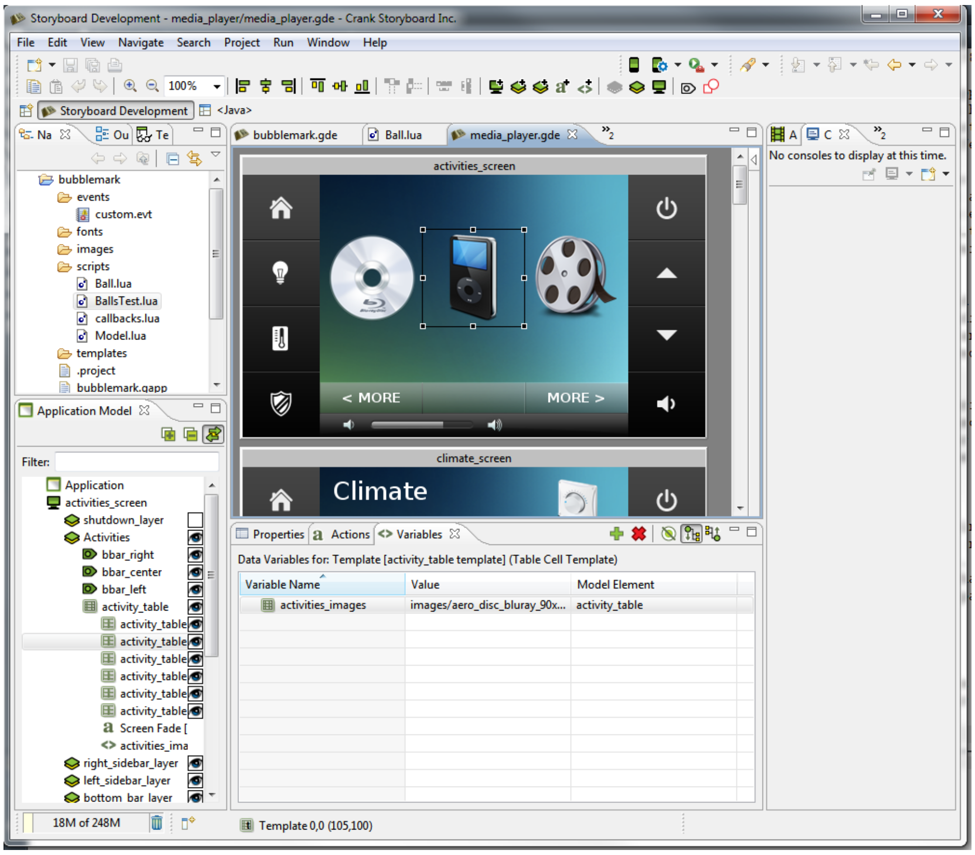
3.3.5 The Storyboard Suite

The Storyboard Suite is a Rapid-Application-Development (RAD) tool that consists of two modules:

- The **Crank Storyboard Designer** is a plug-in for the Eclipse-IDE and is thus available on most commonly used platforms. Via a WYSIWYG-design tool, prototypes of the interface can be created without the need for programming expertise, allowing UI-designers to concentrate on their core expertise. It boasts a management-system for resources such as images and fonts. Via the scripting-language Lua, programmers can implement handlers for events such as touches upon buttons. See Figure 3.4 for a screen-shot of the design-tool in use. Via a simulator-tool, the application can be previewed in real-time and an integrated debugger can be used to examine the programs in real-time [Crank Software Inc., 2012a].

- The **Storyboard Embedded Engine** is runtime-library that on the target-framework interprets and executes applications created in the above design-tool. Via a built-in library, external applications running on the device can send messages to the framework to trigger events or manipulate data. The runtime-bundle is supported on Windows CE, various Linux-distributions and the real-time operating system QNX Neutrino. Among supported processors are those of the x86, ARM and Power-PC family [Crank Software Inc., 2012b]. According to Crank Software, the footprint of the Embedded Engine ranges from 200 to 500 kilobytes [Crank Software Inc., 2012a].

As the above operating systems and target-processors imply, the suite is aimed at high-performance platforms that neither in terms of cost, power- consumption nor dimensions qualify for use in this context.



Data Variables for: Template [activity_table template] (Table Cell Template)

Variable Name	Value	Model Element
activities_images	images/aero_disc_bluray_90x...	activity_table

Figure 3.4: A screen-shot of the Crank Storyboard Designer, which relies on the Eclipse-IDE. It boasts a WYSIWYG-design-tool that allows for the manipulation of the user-interface by mouse. Components that have been declared within this design-environment can be assigned individual behaviour in source-code. Via the scripting-language *Lua*, programmers can implement handlers for events such as touches upon buttons.

3.3.6 Inflexion UI

Inflexion UI is a tool-kit based on the Eclipse-IDE. According to its distributor - Mentor Graphics - it supports a variety of processors and operating systems, among them Android and Linux. It offers built-in support for OpenGL/ES - a cross-platform Application-Programming-Interface (API) for 2D and 3D graphics. Mentor Graphics promotes the product for its ability to benefit from hardware-acceleration in the context of zooming and scaling of images as well as in 3D-effects and animations.

Like in the case of GEMstudio, which is introduced in the following chapter, Mentor Graphics claims that no coding is required for creating the user-interface [Mentor Graphics Corporation, 2011].

3.3.7 GEMstudio

GEMStudio is a WYSIWYG design-tool for platforms that build upon the GEM Graphical OS family of chips - standalone graphics processors. As the manufacturer of both the chips and the design-tool, Amulet Technologies claims - ”absolutely no coding” is needed for the GUI-design [Amulet Technologies, LLC, 2010]. After compiling the user-interface into a proprietary format, it is uploaded to the processor. Much like the architecture that has been chosen for the device described in this document, the above graphics-controllers are solely dedicated to the display of the user-interface. They rely on an external host-processor that is responsible for other tasks such as gathering sensor-data and device-specific functions. Via a serial-port or via USB, the graphics-processor can communicate with the host processor in order to request or modify variables or to call remote procedures that are executed on the host-processor. The design-tool boasts a number of widgets that are categorized as follows:

- **Object Widgets** - Images, Static Text and Animations.
- **Control Widgets** - Components that can either trigger function-calls or cause changes in data, such as buttons, check-boxes and sliders.
- **View Widgets** - Components for the display of data such as various charts and text-fields.

In contrast to the the 4D Systems Workshop which is introduced in the following section, GEMStudio is not free of charge. Amulett distributes GEMStudio via a pay-per-license model. Software that is created using GEMStudio is royalty-free.

3.3.8 C/PEG

C/PEG (Portable Embedded GUI) is a lightweight GUI-library implemented in C that boasts support for an event-driven programming paradigm that - so its distributor Swell Software, Inc. claims - offers *a superior method for creating user interface software, providing structure and order to the otherwise difficult task of responding to external system events arriving asynchronously from many sources.*
[Swell Software, Inc., 2007] The widgets that are available in the library include:

- **Primitives** - shapes such as rectangles and polygons
- **Input-elements** - such as buttons, check-boxes and text-fields
- **Layout-elements** - such as panels and lists
- and other controls such as images, a progress-bar and scroll-bars

For a lack of display-specific drivers the distributor provides templates for a number of commonly used configurations that must be modified to the specifications of individual display-models. Swell Software, Inc. provides a number of additional tools that facilitate the construction of user-interfaces
[Swell Software, Inc., 2007]:

- **PEG Window Builder** a WYSIWYG GUI design-tool that - apart from managing fonts and images - can be used to arrange widgets on screen. The tool generates all instructions that are needed to draw individual components on screen.
- **PEG FontCapture** a tool for converting TrueType- and BDF-fonts into a proprietary format. Via an included character-editor, individual characters can be customized.
- **PEG ImageConvert** an image-converter that supports BMP, GIF, JPEG, and PNG images. The output of the tool is a C-source-code file that can be compiled and linked in parallel with the application. Images can be rotated, mirrored and, via a technique that is also employed by FlashUI - Run-Length-Encoding - be compressed.

3.3.9 The 4D Systems Workshop

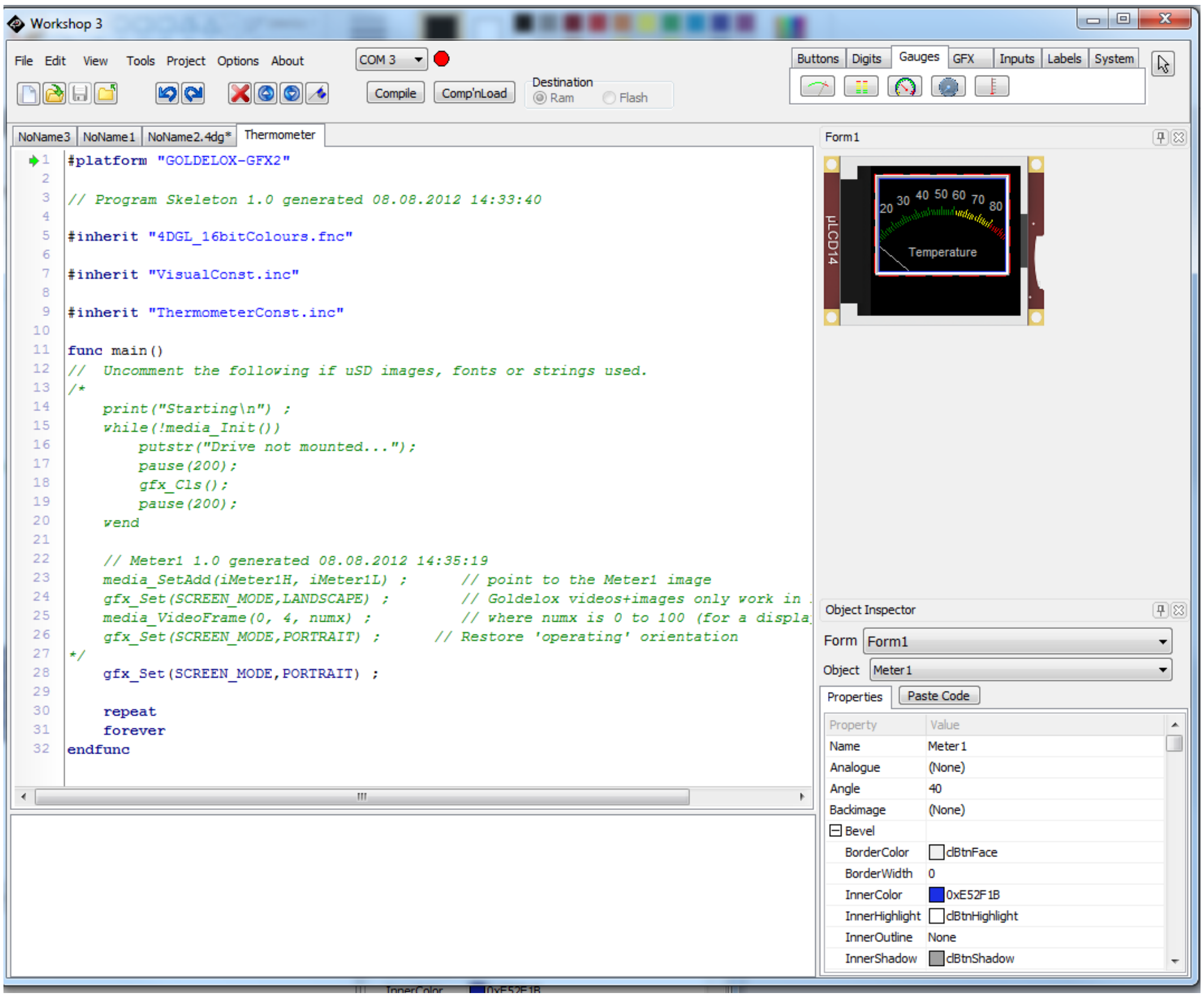
The graphics-controller that has been chosen as the hardware-platform that drives the GUI in this project is shipped with an IDE called *Workshop*. The workshop supports the creation of applications that are implemented in the 4DGL programming language, a language that for its hardware-centred, non object-oriented nature resembles C, whereas the syntax is more familiar to Pascal programmers. Via the IDE, programs can be uploaded to the GOLDELOX chip. One of the key features of the GOLDELOX processor lies in its built-in support for SD-cards, which for their large amounts of storage capacity are in stark contrast to the built-in amount of memory on the chip. The workshop includes a tool that allows for the upload of resources such as images and video-clips onto the SD-card. For the lack of a file-system, the latter can only be addressed in a raw-format - all access to the storage device must be performed by directly referencing the sector or address on the device.

At the time the framework that is described in the following chapters was implemented, among the approaches to GUI design-tools that were introduced previously, only the imperative one was supported. A WYSIWIG design-tool has since been released for testing via the company's support-forum that includes a number of built-in components such as buttons and gauges. After selecting the respective models of graphics-controller and screen from a list, a representation of the display is shown that allows for the positioning and resizing of the above controls by mouse. The tool automatically inserts those commands that are needed to draw the afore mentioned controls on screen, yet does not create any routines that associate behaviour to the visual components. In fact the controls consist of pre-rendered video-clips that must be loaded onto the SD-card. In order to change the state of a control, a distinct frame within the clip must be rendered on screen via source-code. For a screen-shot of the design-tool see Figure 3.5.

For the lack of a file-system, an image can only be drawn by directly pointing the chip to the its 32-bit-address on the storage-device. Although the Graphics Composer, a tool that is shipped with the workshop offers basic support for resizing and cropping images, the management of media is a particularly laborious task in case of the GOLDELOX platform: For relying exclusively upon the imperative approach to GUI design, all images must be located, positioned and displayed via code. As a particular caveat, a change in one image can quickly affect all images that are used within the project. Since images are distributed automatically on the SD-card, a change in size in one image can alter the address of all images that follow. See Figure 3.6 for a screen-shot of the Graphics Composer, which is the default tool that is used to upload images onto the SD-card in the software-suite.

3.3.10 Discussion

The applications and frameworks that have been discussed in the previous pages vary greatly both in terms of targeted platform as well as in their capabilities and features. The range of applications that support a hardware platform that is as limited as the chosen one is represented by X-Nano, GEM-Studio, C/PEG and the 4D Systems Workshop. Apart from offering functions that allow for the rendering of primitives on screen, X-Nano aspires to be a window-manager that conforms to the X Window System.


Workshop 3
File Edit View Tools Project Options About
COM 3
Compile
Comp'nLoad
Destination
Ram
Flash
Buttons Digits Gauges GFX Inputs Labels System
NoName3 NoName1 NoName2.4dg* Thermometer
Form1
µLCD14
20 30 40 50 60 70 80
Temperature
Object Inspector

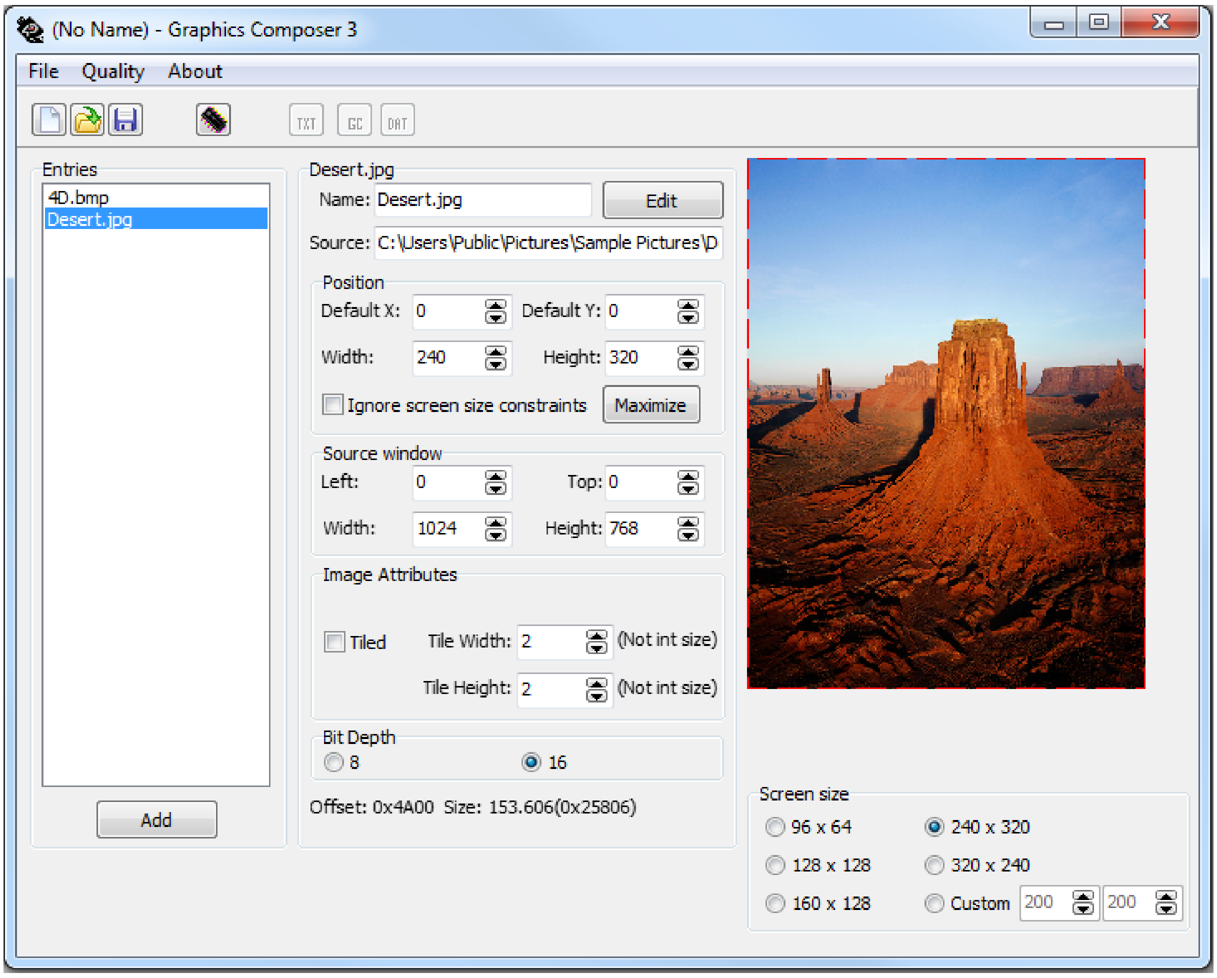
```
#platform "GOLDELOX-GFX2"

// Program Skeleton 1.0 generated 08.08.2012 14:33:40

#inherit "4DGL_16bitColours.fnc"

#inherit "VisualConst.inc"

#inherit "ThermometerConst.inc"

func main()
//  Uncomment the following if uSD images, fonts or strings used.
/*
    print("Starting\n") ;
    while(!media_Init())
        putstr("Drive not mounted...");
        pause(200);
        gfx_Cls();
        pause(200);
    wend

    // Meter1 1.0 generated 08.08.2012 14:35:19
    media_SetAdd(iMeter1H, iMeter1L) ;      // point to the Meter1 image
    gfx_Set(SCREEN_MODE,LANDSCAPE) ;        // Goldelox videos+images only work in
    media_VideoFrame(0, 4, numx) ;          // where numx is 0 to 100 (for a displa
    gfx_Set(SCREEN_MODE,PORTRAIT) ;      // Restore 'operating' orientation
*/
    gfx_Set(SCREEN_MODE,PORTRAIT) ;

    repeat
    forever
endfunc
```

Form: Form1
Object: Meter1

Properties | Paste Code

Property	Value
Name	Meter1
Analogue	(None)
Angle	40
Backimage	(None)
⊟ Bevel	
BorderColor	☐ clBtnFace
BorderWidth	0
InnerColor	🟦 0xE52F1B
InnerHighlight	☐ clBtnHighlight
InnerOutline	None
InnerShadow	▨ clBtnShadow

Figure 3.5: A screen-shot of the 4D Systems-Workshop and its built-in GUI-designer Visi. In the visual representation of the display that can be seen in the top right corner, the user may select individual components of the user-interface by mouse. In the Object Inspector below, the properties of the selected component are shown and can be modified. Most of the widgets that are available consist of pre-rendered video-clips whose individual frames can be used to represent distinct values.



Entries
4D.bmp
Desert.jpg

Desert.jpg
Name: Desert.jpg
Source: C:\Users\Public\Pictures\Sample Pictures\D

Position
Default X: 0 Default Y: 0
Width: 240 Height: 320
Ignore screen size constraints

Source window
Left: 0 Top: 0
Width: 1024 Height: 768

Image Attributes
Tiled Tile Width: 2 (Not int size)
Tile Height: 2 (Not int size)

Bit Depth
8 [✓ 16]

Offset: 0x4A00 Size: 153.606(0x25806)

Screen size
96 x 64 [✓ 240 x 320]
128 x 128 320 x 240
160 x 128 Custom 200 200

Figure 3.6: A screen-shot of the Graphics Composer, which is the default tool to upload images onto the SD-card in the 4D Systems Workshop. In the list on the left, images can be enqueued for upload to the SD-card of the device. Via a series of text-fields additional properties of an individual image can be specified such as its position on screen. After uploading the enqueued images to the SD-card, a list that contains the addresses of the individual images is displayed.

It may provide the functionality that is needed to create a GUI that resembles one that was made using one of the other tools, yet by design its strengths lie in another field.

All of the other tools boast a graphical designer, that relies upon a WYSIWIG representation of the user-interface. Whereas GEM-Studio and the 4D Systems Workshop are both designed for dedicated graphics-controllers that operate as a co-processor to other components, in one aspect their tools could not be further apart: Whereas Amulett advertises GEM-Studio as a tool that requires no programming at all, 4D System's Visi creates pre-rendered animations that lack any sort of out-of-the-box behaviour - additional code must be provided that displays the image that represents a certain value. For being proprietary tools, they both cannot be used to create platform-independent GUIs. Even minor changes such as a different display-size require significant changes in the GUI. In addition, both tools offer support for specific combinations of hardware - ahead of designing the GUI, a combination of the display and graphics-controller that is used must be selected.

In terms of functionality and in terms of philosophy C/PEG resembles the 4D Systems Workshop. Both rely heavily on the programmatic aspect of GUI-design, however, the event-based architecture of C/PEG constitutes a more modern approach to the association of behaviour to individual widgets than that used by 4D Systems.

The other tools and frameworks that have been introduced aim at a far more potent group of processors that can typically be found in smartphones.

3.4 Introducing FUIML

The framework that has been designed based upon the inspirations and patterns that have been introduced in the previous pages is called FlashUI. The name is derived from the location that is occupied by the compiled user-interface. A particular type of non-volatile memory that is used in SD-cards is often called *flash-memory*. Since the compiled user-interface resides on the SD-card of the device, the the name FlashUI was chosen.

In contrast to GEM-Studio, the 4DG-Workshop and other tools that target embedded graphics-controllers it introduces an abstraction layer between the device-independent definition of the user-interface and the hardware-specific components that are used to render the GUI on screen. At the heart of this abstraction layer lies a custom-made markup-language that is derived from XML. This markup-language, FUIML allows for the definition of the layout and of the contents of the user-interface without the need for intricate knowledge of the device that is meant to render the GUI.

Whereas in the 4DG-Workshop, all addresses on the SD-card must be managed by the programmer by hand, in the FlashUI markup-language all images and other references to a particular location on the SD-card are dynamically inserted upon compilation of the user-interface. In addition the entire user-interface including all images and animations is automatically scaled to the appropriate size. Hence, a change in display dimensions does not require a redesign of the user-interface. For relying upon a flexible set of pre-built widgets, code-reuse is strongly enforced.

The following chapter provides an in-depth description of the FlashUI markup-language and its built-in controls.

Chapter 4

FUIML Reference

4.1 FUIML in a Nutshell

FUIML is an XML-based markup-language that has been designed from scratch in the course of this project. It facilitates a device- and operating system-independent way of specifying user-interfaces for hand-held devices. An FUIML-project is organized in individual *Pages* which constitute the roots of widget-trees that define the contents of the screen. *Pages* are divided into *Tiles* - a type of widgets that on the one hand define position and size of visible elements, on the other hand they serve as buttons that link to other *Pages*. The child-nodes of a *Tile* constitute its visible content that can be seen on screen in the form of labels, images, animations and other, more complex elements.

4.2 General Language Properties

4.2.1 Datatypes

Whereas the afore-mentioned tree-structure defines the general structure of what is visible on the display of the device, it is an FUIML-node's attributes that define most of the visible actual content of the user-interface. Each attribute may belong to one of five data-types - Byte, Word, DWord, String and URI. All numbers are unsigned - however, certain text-formats cause the device to display the values signed counterpart.

Byte	an 8 bit value, no sign-bit
Word	a 16 bit value, no sign-bit
DWord	a 32 bit value, no sign-bit
String	a sequence of ASCII chars, terminated by 0
URI	the absolute or relative path to a certain resource

Table 4.1: Data-types in the FUIML language

4.2.2 Symbols

In FUIML, a Symbol is constituted by a unique name and a constant value - either a string, an integer or a URI. Whenever an FUIML-file is loaded the compiler checks if a file named Symbols.txt is present in the same directory. These Symbols can be used within the source code by typing the dollar-sign in front of the symbol-name. At compile-time, this instructs the compiler to look up the Symbol's value in the afore mentioned file.

4.2.3 Attributes

Whereas the XML-tree itself outlines the general structure of what is displayed on screen, it is the attributes of a node that define what can be seen on the screen of the device. FUIML allows for the traditional way of assigning a value to an attribute, whereby an equal-sign is appended to an attribute's name followed by its value in quotation marks. While this syntax may allow for the convenient assignment of constant properties such as an the height or width of an element, it is often necessary to alter its state dynamically based upon external influences. To cater to this need for a dynamic way of defining the properties of an element - apart from the above traditional attribute syntax - FUIML also makes use of WPF's Property-Element-Syntax: The value of an element may be defined as an XML-subtree, thus enabling the use of a more complex description of its behaviour and look [Huber, 2010].

```
<Tile width="100">
  <Image source="someImage.png">
</Tile>

<Tile>
  <Tile.width>100</Tile.width>
  <Image source="someImage.png">
</Tile>
```

Listing 4.1: Whereas the width of the above XML-node is declared in traditional syntax, that of the node below is assigned using what Microsoft calls Property-Element-Syntax. This alternative notation has found wide use in Microsoft's XAML-based languages aimed at user interface design - WPF and Silverlight. Its key advantage lies in the possibility of assigning objects in the form of XML-subtrees to an XML-attribute instead of base-types such as numbers and strings. In the example above both notations of the width-property are equivalent.

4.2.4 Triggers

Based upon the above addition to traditional XML, many FUIML-attributes can be declared dynamically using the *Trigger*-element. Using *Triggers*, the value of an attribute can be conditionally altered as a reaction to changing circumstances - most notably status-updates from the main processor of the device received via the serial-port. The *Trigger*-element boasts the following attributes:

Triggersource	The source of the left operand of the comparison - ”SD” or ”COM”
Offset	the offset of the left operand in bytes within the *Status-Array* (see Chapter 6).

Table 4.2: The attributes of a Trigger

The above attributes constitute the first half of the *Trigger*: they define the position of the operand that may be compared to various constants in the *Trigger's Triggerstates*. They represent one possible state the *Trigger* may assume. The attributes of a *Triggerstate* comprise a condition and the resulting value that is assumed by the *Trigger* it belongs to:

Result	defines the value that is returned if the *Triggerstate* evaluates to true.
Comparison	the comparison-operator to be applied to the *Trigger's* operand and the following constant
Operand	the constant that the *Trigger's* operand is compared to.

Table 4.3: The attributes of a Triggerstate

The *Comparison* attribute may contain one of the operators specified in Table 4.4.

LT	*TRUE* if the *Trigger's* operand is smaller than the constant
GT	*TRUE* if the *Trigger's* operand is greater than the constant
LE	*TRUE* if the *Trigger's* operand is smaller or equal to the constant
GE	*TRUE* if the *Trigger's* operand is greater or equal to the constant
EQ	*TRUE* if the *Trigger's* operand and the constant are equal

Table 4.4: Comparison-operators for Triggers

If the condition evaluates to *true*, then the *Triggerstate's result*-attribute is assigned to the triggered property - the one that declares the *Trigger*. See Figure 4.1 for a practical example of a *Trigger*.

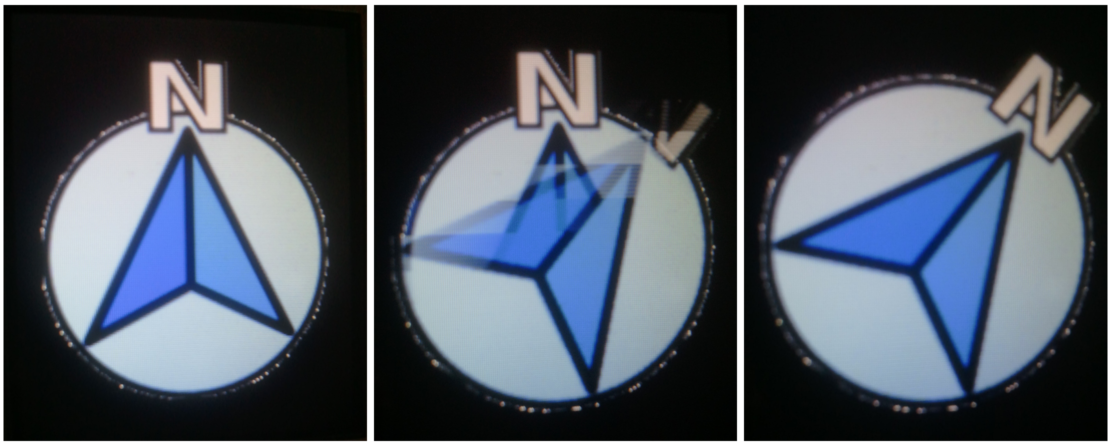


Figure 4.1: In the above screen-shots, a Trigger is used to rotate the orientation of a compass-image. Whenever the corresponding angle is updated, a different image is selected for display by the Trigger.

Example

```
<Image>
 <Image.Src>
  <Trigger offset="3" triggersource="COM">
   <TriggerState
     result="battStr01.png"
     comparison="LT"
     operand="20"/>
```

```
      <TriggerState
         result="battStr02.png"
         comparison="LT"
         operand="60"/>
       ...
    </Trigger>
  </Image.Src>
</Image>
```

Listing 4.2: This example demonstrates the use of *Triggers* in FUIML. The source-property of an *Image* is assigned a *Trigger* instead of a constant URL. At runtime, the *Trigger* selects from top to bottom among the specified *Triggerstates* the one that first fulfils a certain criterion- in this case the Status-Array at index three is compared to certain constants. The *result*-attribute of the chosen *Triggerstate* is assigned to the triggered property *Source*.

4.3 Page

Description

Pages constitute the root nodes of an FUIML-document. They may declare a background- and a foreground-color that serves as a default on the page. Individual child-elements may locally override these settings. Via the *link*-attribute of *Tiles* and via the declaration of *Alerts*, the elements of one *Page* may reference others.

Example

```
<Page foreground="#FFFF" background="#0000"
      width="128" height="192">
  ...
</Page>
```

Attributes

width	the width of the *Page* in pixels
height	the height of the *Page* in pixels
foreground	default foreground color (RGB565-encoding)
background	default background color (RGB565-encoding)

Child Nodes

- Tile
- Stackpanel

4.4 Alert

Description

The *Alert* declares a *Page* that is shown only upon a request by the main processor. For this purpose, it must be assigned an identification-number that must be included in the request of the main-processor. As a second argument, the location of the FUIML-*Page* that is shown in case of the alert must be supplied. Whereas *Alerts* can be defined in any *Page*, for their application-wide scope they must be unique.

Example

```
<Page foreground="#FFFF" background="#0000">
   ...
  <Alert id="$onPhoneCall" url="PhoneCall.FUIML">
   ...
</Page>
```

Attributes

id	The identification-number of the *Alert*, as it is used on the main-processor.
url	The location of the FUIML-*Page* that is shown in case of the alert.

Child Nodes

-

4.5 Tile

Description

Tiles define the position and size of screen-elements. Images and Animations are scaled to fit inside their parent-Tile, Labels are cropped at the borders of the *Tile*. All *Tiles* can be used as buttons. Upon activation, they may perform an associated action - either initiate the switch to another page via the *link*-attribute or perform a remote-procedure-call on the main processor via the *rpc*-attribute.

Example

```
<Tile width="80" height="80"
     x="0" y="0"
     link="someOtherPage.FUIML">
   ...
</Tile>
```

Attributes

x	The X-coordinate of the top left corner of the *Tile*
y	The Y-coordinate of the top left corner of the *Tile*
width	The width of the *Tile*
height	The height of the *Tile*
link	A URI to to another FUIML-file
rpc	The id of a certain procedure that is called on the main processor.

Child Nodes

- Image
- Animation
- Textbox
- Map
- Graph

4.6 Stackpanel

Description

Stackpanels facilitate the positioning of *Tiles*. They automatically arrange their children either in a horizontal or vertical way, thus alleviating the user from manually calculating the coordinates of individual *Tiles*. As the below example demonstrates, *Stackpanels* can be nested to automatically position elements horizontally and vertically.

Example

```
<Stackpanel orientation="Vertical">
  <Stackpanel orientation="Horizontal">
    <Tile><Image src="NorhWest.png"/></Tile>
    <Tile><Image src="NorthEast.png"/></Tile>
  </Stackpanel>
  <Stackpanel orientation="Horizontal">
    <Tile><Image src="SouthWest.png"/></Tile>
    <Tile><Image src="SouthEast.png"/></Tile>
  </Stackpanel>
</Stackpanel>
```

Attributes

orientation	Either horizontal or vertical arrangement of children

Child Nodes

- Tile
- Stackpanel

4.7 Image

Description

Displays an image that is scaled to scaled to the size of the parent-*Tile* at the parent's position.

Example

```
<Tile width="40" height="25" x="0" y="0">
 <Image>
  <Image.Src>
   <Trigger offset="3" triggersource="COM">
    <TriggerState result="battStr01.png"
             comparison="LT"
             operand="20"/>
    <TriggerState result="battStr02.png"
             comparison="LT"
             operand="60"/>
     ...
   </Trigger>
  </Image.Src>
 </Image>
</Tile>
```

Attributes

src	the URI of the image that is displayed

Child Nodes

-

4.8 Animation

Description

Defines a sequence of images that is iterated through. To preserve processing power, only one *Animation* can be active per *Page*. If the *Animation* is inactive, the first image in the sequence is displayed. All images within the sequence are automatically scaled to the size of the parent-*Tile*.

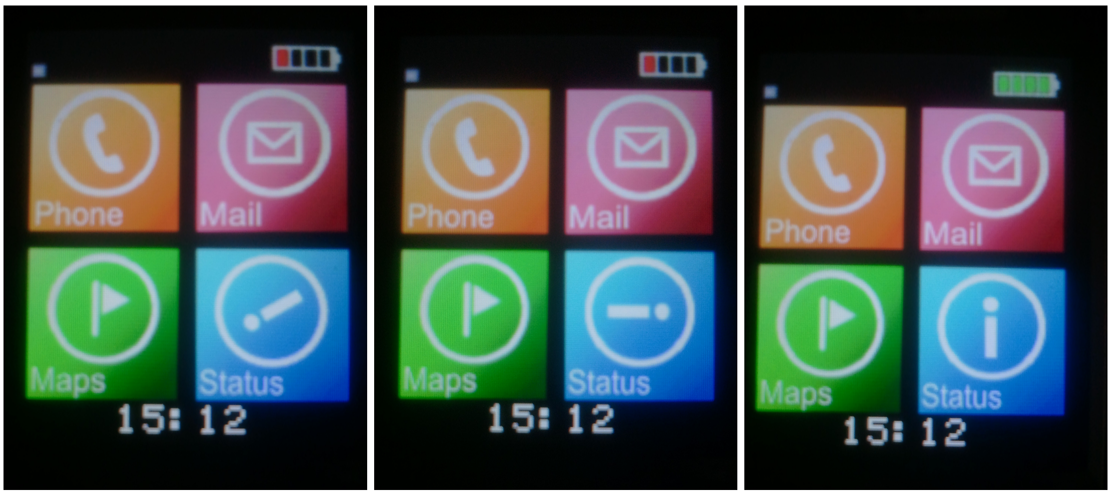


Figure 4.2: In the above screen-shots, the status-icon is animated in order to provide a more lively look-and-feel.

Example

```
<Animation interval="100">
  <Image src="img00.png"></Image>
  <Image src="img01.png"></Image>
  <Image src="img02.png"></Image>
</Animation>
```

Attributes

interval	The timespan an image is displayed in the *Animation* in milliseconds.

Child Nodes

- Image

4.9 Textbox

Description

The *Textbox* declares a text-area on the screen. The actual text that is displayed as well as its visual appearance is defined by the child-nodes of the *TextBox*, described in the rest of this chapter.

Example

```
<Textbox>
  <String text="Hello World"/>
</Textbox>
```

Attributes

-

Child Nodes

- Byte
- Word
- DWord
- String
- LongText

4.10 Byte, Word, DWord

Description

These FUIML-Nodes represent integer values (8, 16 or 32 bits) that can be displayed within the above *Textboxes*. Their visual appearance and number-format can be specified via attributes such as *background*, *font* and *format*.

Example

```
<Textbox>
  <String text="The current time is: "></String>
  <Byte foreground="$White"
      background="$Black"
      value="$COM[2]"></Byte>
  <String text=":"></String>
  <Byte foreground="$White"
      background="$Black"
      value="$COM[3]"></Byte>
</Textbox>
```

Attributes

value	The integer-value to be displayed
format	The number-format of the value
foreground	The text-color in RGB565 format
background	The background-color in RGB565 format
style	The font-style of the text encoded in a 16 bit integer
opaque	If TRUE, the background is filled with the above background color
font	The id of the font used
fontWidth	A multiplier applied to the width of the font
fontHeight	A multiplier applied to the height of the font

Child Nodes

-

4.11 LongText

Description

Whereas the afore mentioned elements may be adequate for a short text, this widget is suitable for text that far exceeds the maximum amount of characters that can be displayed at once. It automatically inserts line-breaks and for an improved orientation within the text a scrollbar is shown.

Example

```
<Textbox>
   <LongText></LongText>
</Textbox>
```

Attributes

Child Nodes

-

4.12 Map

Description

This XML-Node fills the the parent-*Tile* with a map consisting of a grid of individual images stored at a specific location on the SD-card of the device. Via the attributes positionX and positionY the screen is set to a certain cut-out of the above grid.

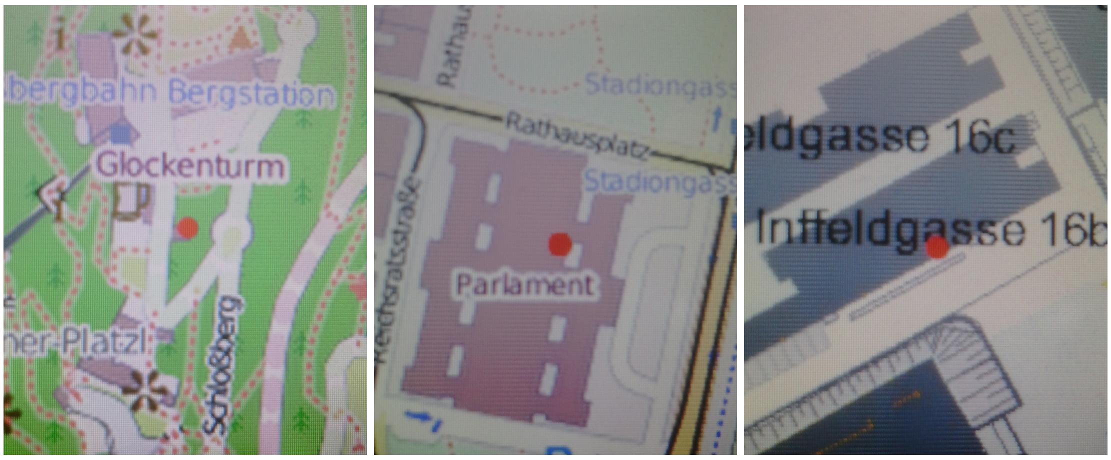


Figure 4.3: The above screen-shots demonstrate the map-component in action.

Example

```
<Tile width=”176” height=”220” link=”MainMenu.FUIML”>
      <Map positionX=”$COM[2]” positionY=”$COM[4]”></Map>
</Tile>
```

Attributes

positionX	the X-coordinate of the screen-center within the map
positionY	the Y-coordinate of the screen-center within the map

Child Nodes

-

4.13 Graph

Description

The *Graph*-widget draws a line-chart on screen. Via a background-image and via the attributes *foreground* and *background*, the visual appearance of the widget can be customized.

Example

```
 <Tile>
    <Graph   background="background.jpg"
             foreGround="$RED"
             axisColor="$BLACK"/>
</Tile>
```

Attributes

background	The URI of the background image
foreGround	The color of plot
axisColor	The color of the X- and Y- axis.

Child Nodes

-

In the following section, the compiler that is used to translate interlinked FUIML-projects is introduced. It features an automated mechanism that manages all references to images and other files that occur within a project. Hence, there is no need for manually arranging the resources that are used within the user-interface. Instead of referencing resources via an address or via the sector on the SD-card that is occupied by the resource, URIs can be used. The compiled user-interface can be uploaded directly to the device via a comfortable dialogue.

Chapter 5

The Compiler

Whereas for a human being XML may be a relatively convenient way of describing a user interface, it is far from convenient from the point of view of an embedded processor. Apart from the obvious overhead of parsing an XML-tree, limitations such as the lack of a file-system on the GOLDELOX-platform create the need to translate FXML to a more machine-friendly code.

The FXML-compiler that is presented in this chapter is a command-line-program whose main purpose lies in the automated translation of interlinked FUIML-documents into a compact bytecode that can easily be interpreted by the Goldelox chip on the target-device. Portability among operating systems has been a point of major consideration from the very beginning of the development process, hence the program is implemented in the Mono-programming-language and has been tested on Windows, Macintosh and Linux platforms.

Apart from its main task of compiling FUIML into a proprietary bytecode, the compiler also boasts a number of automated features that in many aspects surpass those of the set of tools that is supplied with the GOLDELOX-platform - most notably, it fully automates all aspects of transcoding, scaling and referencing images for the GOLDELOX-chip. These assets will be introduced in detail later in this chapter whereas the focus of the following pages will be on the afore mentioned main use of the compiler - the translation from FUIML to a machine-friendly bytecode.

Since some of the compiler's key components are shared with other tools - in particular those that concern bytecode, the output of the compiler - this command-line-tool is a mere front-end to the core library that is responsible for most of the actual processing.

5.1 Overview

In order to compile an FUIML user-interface, a number steps is required that translate FUIML into a machine-friendly bytecode. In the course of this chapter, some of the steps below will be explained in more detail:

1. When the program is launched, the main-page of the FUIML-project - which is expected as a command-line-argument - is loaded.
2. If there is a *Symbols*-file in the same directory as the *Page* it is loaded and parsed.
3. The *Page* is de-serialized into an internal object tree representing the XML-tree.
4. The compiler starts translating the above object tree into bytecode
    - The exact position of the *Page* within the bytecode is stored.

- If the program encounters any references to other *Pages*, their path and the exact location of their reference within the bytecode are stored.
- If there are any references to images, they are also stored in the same way.

5. If there are any referenced *Pages* left that have not been translated yet, the above steps are repeated for each unprocessed *Page*.
6. All references to *Pages* within the bytecode whose position has been stored before are set to the actual position of the referenced *Page* within the bytecode.
7. All referenced images are loaded, scaled to the size of their container-element, transcoded into a format readable by the GOLDELOX-platform and appended to the bytecode.
8. All references to the above images are set to the actual position of the image within the bytecode.
9. At the user's discretion, the output-file may either be dumped directly to a removable media or be uploaded to a device via the serial-port.

5.2 The Command-line Front-end

The command-line tool serves as a front-end to the actual compiler which resides in a shared library. This separation between the user interface of the compiler and its core components holds numerous benefits - most notably:

- The core features of the compiler can easily be linked into other components.
- The lack of a high-end user interface makes the library highly tolerant to the underlying operating system that is used.
- A strict separation of concerns simplifies the integration of changes within this critical subsystem.

Upon launch, the tool checks for the presence of command-line arguments:

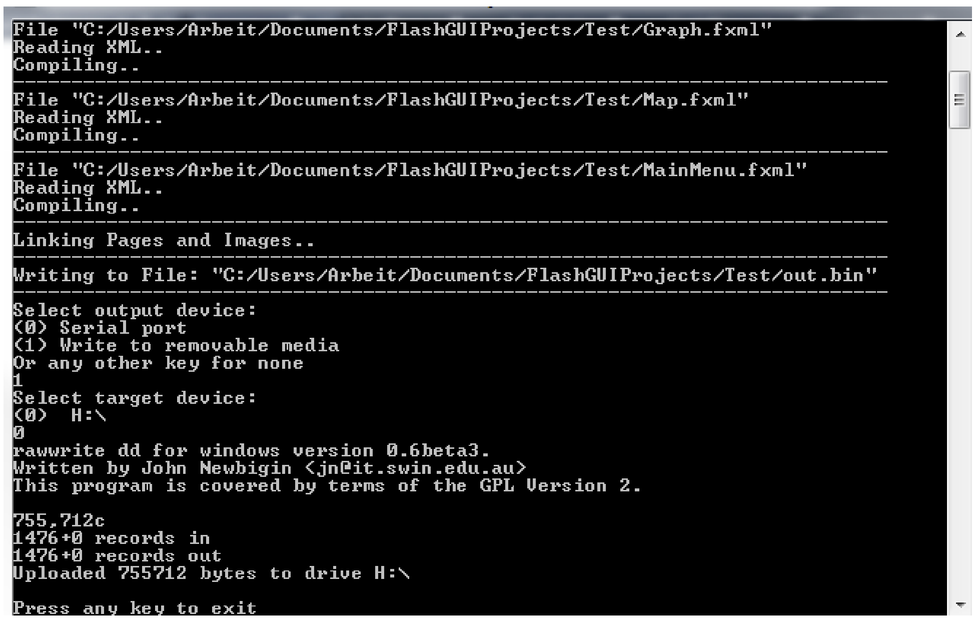
Mainpage.FUIML	required	the absolute or relative path to the user interfaces main page
Output.bin	optional	the absolute or relative path to the output file that will contain the bytecode

Table 5.1: Optional and required arguments for the compiler-tool

If no output file is specified the program assumes a default value, if no input file is specified, or if the path is invalid the tool terminates after plotting an error message. If all inputs are correct, the compiler is invoked and if successful, the bytecode is dumped into the specified output-file. After this, the user is prompted if the aforementioned code should automatically be copied either to a removable media or to a device attached via a serial-port. See Figure 5.1 for a screen-shot of the command-line front-end.

5.2.1 Copying Bytecode to Removable Media

This option causes the program to invoke an external tool that dumps the content of the compiler's output-file directly to a removable disk. For the GOLDELOX-platform's lack of a file-system, this must be done in raw-write-mode, which is unsupported by the Mono/C# programming language. Since the tool overwrites all content on the target drive from its first sector to the size of the bytecode, this is a risky procedure that may result in irreversible loss of data on the disk. Hence, as a precaution only removable media are permitted as target drives.



```
File "C:/Users/Arbeit/Documents/FlashGUIProjects/Test/Graph.fxml"
Reading XML..
Compiling..
------------------------------------------------------------------------
File "C:/Users/Arbeit/Documents/FlashGUIProjects/Test/Map.fxml"
Reading XML..
Compiling..
------------------------------------------------------------------------
File "C:/Users/Arbeit/Documents/FlashGUIProjects/Test/MainMenu.fxml"
Reading XML..
Compiling..
------------------------------------------------------------------------
Linking Pages and Images..
------------------------------------------------------------------------
Writing to File: "C:/Users/Arbeit/Documents/FlashGUIProjects/Test/out.bin"
------------------------------------------------------------------------
Select output device:
(0) Serial port
(1) Write to removable media
Or any other key for none
1
Select target device:
(0)  H:\
0
rawwrite dd for windows version 0.6beta3.
Written by John Newbigin <jn@it.swin.edu.au>
This program is covered by terms of the GPL Version 2.

755,712c
1476+0 records in
1476+0 records out
Uploaded 755712 bytes to drive H:\

Press any key to exit
```

Figure 5.1: A screen-shot of the command-line-front-end. When all pages have been compiled and linked and when the file containing the bytecode has been created, a dialogue is shown that allows for a convenient way of uploading the bytecode to the device. This can either be done via a serial-port or via an external tool that dumps the content of the compiler's output file directly to a removable disk.

5.2.2 Copying Bytecode directly to the Device

Since the on-chip-firmware - which will be presented in the following chapter - boasts an update feature for the user interface, the bytecode may be sent directly to the device via a serial-port. When a port is selected, the workstation requests the target-chip to enter ”Data-Mode” via the specified port and sends the content of the bytecode-file to the device. See Chapter 7 for details.

5.3 Parsing the Symbols File

In FUIML, *Symbols* are used as constant values that are referenced within the XML-file via their name. Since upon loading a *Page*, the compiler looks for a *Symbols*-file (Symbols.txt) that is in the same directory as the FUIML-file of the *Page*, all *Pages* may have an individual set of *Symbols*.

Within the file each line may either constitute a *Symbol* or a comment - designated by a leading #. The definition of a *Symbol* consists of its data-type followed by its name and in turn its value. A *Symbol* may be of one of the data-types listed in Table 5.2.

The compiler stores all Symbols within a hash table - its key being the *Symbol*-name, the value being the *Symbol* itself. Whenever an attribute string within the FUIML-file is lead by the dollar sign, the compiler tries to lookup the attribute string minus the $-sign within the above datastructure and instead uses the value of the *Symbol*.

5.4 Parsing FUIML Files

When the compiler has finished loading the *Symbols*-file, the FUIML-file itself is parsed. First, its XML-tree is de-serialized via a built-in feature of the .NET framework. Starting from the root of the tree, am

Keyword	Description
Byte	an 8 bit value, no sign-bit
Word	a 16 bit value, no sign-bit
DWord	a 32 bit value, no sign-bit
String	a sequence of ASCII chars
URI	the absolute or relative path to a certain resource

Table 5.2: Datatypes that are used by the compiler.

object- tree consisting of proprietary elements that represent the individual FUIML-nodes is constructed. The elements of the latter tree constitute representations of the building blocks of the FUIML-language. All elements within the tree derive from the abstract class *CodeElement* whose methods and properties are listed in Table 5.3.

5.5 Adding Properties

Since the *CodeElement* class implements most of the behaviour needed by its subclasses, in most cases only the latter two methods must be implemented in subclasses. Most of the additional behaviour of subclasses is covered via attachable members of type *XMLAttributeBase* and its derived classes. Whenever a property is assigned in FUIML, the compiler uses *CodeElement.getPropertyInfo(String propertyName)* to retrieve the targeted property within the class. The returned *PropertyInfo* constitutes a description of the target's member-field including its name, scope and data-type. This is done via the .NET framework's *Reflection* name-space, which among other features allows for a dynamic way of finding, reading and assigning member fields within a class at run-time.
The main benefit of this mechanism lies in the fully automated parsing of attributes without a child-side implementation of the latter. Thus, all properties declared by the child class can automatically be discovered by the base-class and can immediately be addressed within the FUIML file without the need for additional code. The next section introduces how the above attributes are encoded into bytecode.

5.6 Encoding Attributes

Not only does the FUIML-compiler automatically discover all assignable attributes held by FUIML-nodes, it also boasts a fully automated procedure of converting attributes into bytecode - to add a new attributes to a node, no changes within the code-base of the compiler are needed, apart from the declaration of the attribute itself.

Attributes may differ in a variety of ways, some of their characteristics - such as the size of an attribute - are determined in a hard-coded fashion within the source-code of the compiler, others are determined via the FUIML-file upon its compilation:

5.7 The Linker

Since the compiler automatically manages references to pages and images, a component is required that - in a final step after the FUIML-code has been parsed - inserts the final locations of references components in the bytecode. Hence, whenever a reference is parsed, its position and the resource it points to is stored within a data-structure in the *Linker*. As the final location of the reference within the bytecode is unknown until all resources have been encoded, the Linker inserts padding-bytes instead of an address. Whenever a referenced resource has been written to the bytecode, its address within the data-structure of the Linker is updated. At the very end of the process of compiling the FUIML-project, after all resources have been encoded, the correct addresses are inserted into the respective locations that have been saved before. In order to prevent cycles in which one page is referenced by another and vice-versa, each page is treated as a unique resource that may exist only once within the project. The same image, however, may occur multiple times in different dimensions - hence it may exist in multiple locations within the bytecode.

5.8 Encoding Images

Whenever an image is encountered, it is stored by the above Linker. In a final step, after all FUIML-files have been parsed, the referenced images are encoded. Since instead of specifying their dimensions themselves, they are determined by the container-*Tile* of the image, the compiler must trace back in the object-tree until the parent is found. It loads the file from the given location and uses a built-in function to scale the image to the appropriate size. The resulting scaled bitmap is converted into the BGR565-format - a bitmap-format that per pixel allocates 5 bits to the colors red and blue, whereas green occupies 6 bits. The resulting pixel-array is dumped to the next free sector within the bytecode.

5.9 Stackpanels

Stackpanels in FUIML are container-elements. They may be used to automatically arrange individual *Tiles* or other *Stackpanels* in an either horizontal or vertical orientation. In order to achieve this, a *Stackpanel* calculates the X- or Y-coordinate of each child-node based upon the width or height of the previously encountered child-nodes. The control itself is not encoded into the bytecode. After the individual positions of its child-nodes have been calculated, the *Stackpanel* is discarded.

5.10 The Alert-Table

The bytecode that constitutes the output of the compiler must conform to a certain structure: Within its very first 512 bytes, a table is encoded that contains the addresses of those *Pages* that are associated to an *Alert* (see previous chapter). Each entry consists of five bytes - the first contains the identifier of the *Alert*, the remaining four bytes contain the address of the *Page* that is shown in case of the alert. The sectors that follow contain the first *Page* within the application.

In the following chapter, an interpreter for the output of the compiler is introduced. It is concerned with the display of the user-interface based upon the bytecode-representation of the FUIML-source-code.

Properties

Name	*Description*
allowedChildren	A Hashset of allowed child nodes for a given node
ancestorHeight	The height of the parent-node in pixels
ancestorWidth	The width of the parent-node in pixels
hasSize	A Boolean-flag, TRUE if the node may declare the above properties for its child-nodes
width	The node's width in pixels
height	The node's height in pixels
name	The node's name, identical to its name in the FUIML-file
parent	A reference to the parent-node

Methods

CodeElement	Constructor
createToken	Static factory-method for subclasses
assignProperties	Parses and assigns XML-attributes
checkProperties	Checks properties for their datatype
getParentTag	Returns the owner-tag of Property-Element-Syntax
getPropertyTag	Returns the property-tag of Property-Element-Syntax
isParentsNameSpace	Returns TRUE if the node constitutes a property of the parent in Property-Element-Syntax
isAllowedChild	Returns TRUE if the passed childnode is valid for this element
hasProperty	Returns TRUE if the node owns a property with the passed name
getProperty	Returns a properties FUIML-subtree
getPropertyInfo	Returns a properties PropertyInfo
parseXMLNode	Abstract method, used to convert an XML-node into a CodeElement
writeBytecode	Abstract method, converts the element into bytecode

Table 5.3: Methods and properties of the abstract class CodeElement.

Chapter 6

The Interpreter

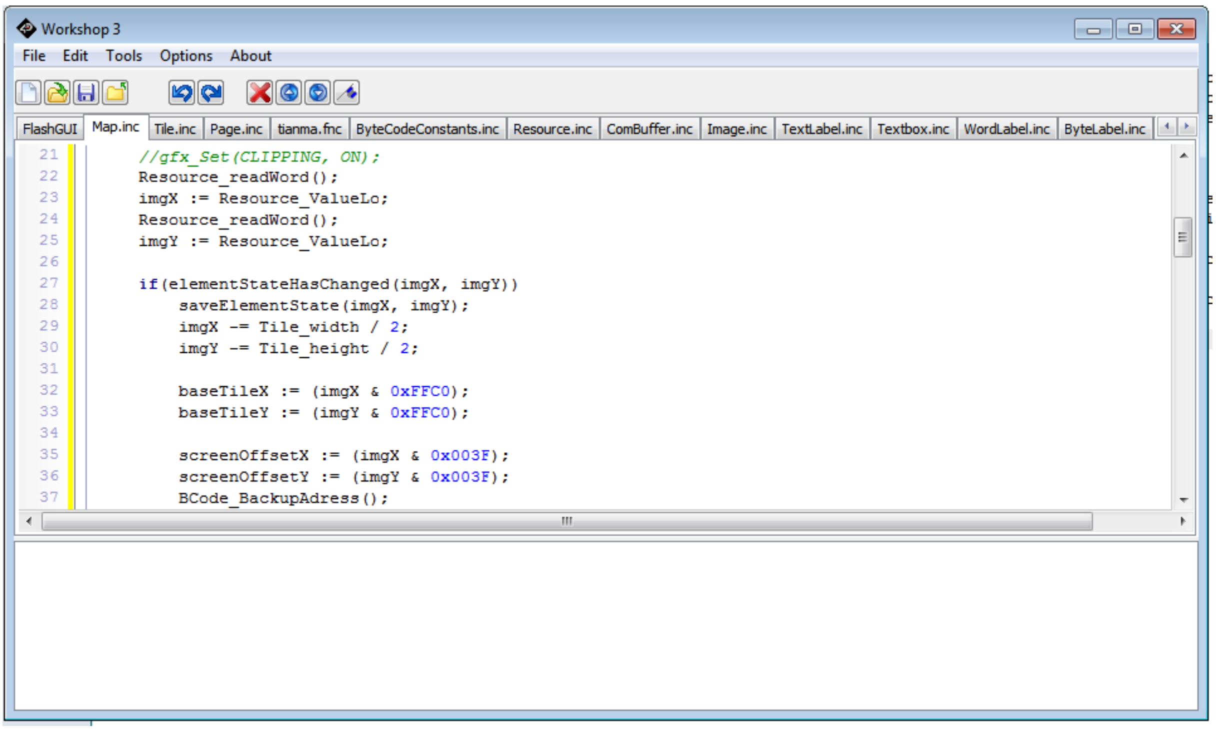
In this chapter, the bytecode-interpreter that forms the firmware of the graphics-controller is introduced. The interpreter is used to render the user-interface which is stored on the built-in SD-card of the device on screen. Perhaps the most limiting factor of the GOLDELOX-platform lies in the low amount of memory available - be it in the form of read-only program memory or be it in random access memory. However, via the use of an SD-card the amount of available memory can be drastically increased to up to 2 gigabytes. In FlashUI, the entire user-interface resides on the afore mentioned SD-card, thus only the bytecode-interpreter consumes program memory. Apart from this, the user-interface can easily be replaced at run-time without requiring a reboot of the GOLDELOX chip. Updated or newly developed user-interfaces can be uploaded via the internet to the SD-card where they replace existing GUIs.

6.1 The GOLDELOX Processor

Like its more powerful relatives - the PICASO and the DIABLO, the GOLDELOX processor is based upon the E.V.E. (Extensible Virtual Engine) virtual processor. The 4DGL programming language that the interpreter is implemented in was designed specifically for processors boasting the 4DGL's E.V.E.(Extensible Virtual Engine) engine core [4D LABS, 2012].

Dimensions	6 mm x 6mm
Program-Memory	10 kilobytes
Random-Access-Memory	510 bytes
Interfaces	An 8 bit display-interface supporting various OLED, LCD and TFT displays
	A serial port with 300 to 256K baud
	Hardware SPI port interface for uSD/uSDHC memory cards
	Two general purpose IO ports

Table 6.1: GOLDELOX-GFX2 Specifications [4D LABS, 2011]



```
    //gfx_Set(CLIPPING, ON);
    Resource_readWord();
    imgX := Resource_ValueLo;
    Resource_readWord();
    imgY := Resource_ValueLo;

    if(elementStateHasChanged(imgX, imgY))
        saveElementState(imgX, imgY);
        imgX -= Tile_width / 2;
        imgY -= Tile_height / 2;

        baseTileX := (imgX & 0xFFC0);
        baseTileY := (imgY & 0xFFC0);

        screenOffsetX := (imgX & 0x003F);
        screenOffsetY := (imgY & 0x003F);
        BCode_BackupAdress();
```

Figure 6.1: A screen-shot of the integrated development environment that is shipped with the *4D Graphics Language (4DGL)* - a proprietary programming language that was developed for the 4D-Labs family of embedded graphics processors, consisting of the GOLDELOX-, PICASO- and the DIABLO processor. The language which is influenced by the programming languages C, Basic and Pascal boasts a number of built-in functions that grant direct access to core components such as the graphics-memory (GRAM) of the display or the user-stack of the GOLDELOX-chip. The above IDE features a convenient way of uploading the compiled program via the serial-port of the GOLDELOX-processor to the target device.

6.2 The 4DGL programming language

The interpreter is implemented in ”4DGL” (4D Graphics Language), a proprietary programming language that was developed by the producers of the GOLDELOX-chip itself - 4D Labs Pty. Ltd.. According to the latter, it is influenced by the programming languages C, Basic and Pascal [4D LABS, 2012]. The compiler is shipped with an integrated development environment (IDE) that can upload the compiled program via the serial-port of the GOLDELOX-processor to the target device, see Figure 6.1. Apart from familiar instructions such as loops and conditions, the language also boasts a number of device-specific functions - for example those concerning I/O from the serial-port or the SD-card and of course a number of graphics-functions that are used to draw on the display. Furthermore, the language boasts a number of built-in functions that grant direct access to core components such as the graphics-memory (GRAM) of the display or the user-stack of the GOLDELOX-chip. The following lines contain a ”Hello World” program implemented in the 4DGL programming language.

```
func main()
  //Initialize display
  disp_Init(INIT_tbl, GRAM_sm);

  var helloWorld = ”Hello world!”;

    while(1)

    print(helloWorld);

    wend
endfunc
```

6.3 Interpreting the bytecode

After initializing the display driver, the firmware checks if an SD-card is present and establishes a connection via the COM-port. It then begins interpreting the bytecode of the main-*Page* that is located at fixed position on the SD-card.

The bytecode of a *Page* resembles the structure of its FUIML-file - the first element is the *Page*-node itself. After reading and applying the default-background and -foreground, the child-elements of the *Page* are parsed - normal *Tiles* or such that contain links to other *Pages*. When the *PAGE_END*-byte is read, the *Page* is done and the GOLDELOX chip returns to the main function where the next step consists in checking the serial-port for new messages.

The above routine is repeated endlessly. The chip keeps track of the address of the current *Page* - in the beginning the main-*Page* - when the *Page* has changed, for an example if a *Tile* that contains a link was pressed, the screen is cleared and the *Page* at the new address is parsed. in the following code-snippet, the function that parses *Pages* is shown.

```
func Page_Parse()
    var token;
    //read 1 byte from SD-card - the page token
    BCode_ReadB(/*PAGE_START*/);
    //Set default background and foreground color
    Page_SetDefaultColors(
        //read 1 word from SD-card - the background-color of the page
      BCode_ReadW(/*defaultBackground*/),
      //read 1 word from SD-card - the default foreground-color of the
          page
        BCode_ReadW(/*defaultForeground*/)
    );
    //read the first child-element of the page
    token := BCode_ReadB(/*First UI Element*/);

    //Parse childnodes until the PAGE_END-byte
    while (token != PAGE_END)
        if (token == TILE_START)
            Tile_Parse(FALSE);
        endif
        token := BCode_ReadB();
    wend
    //Return to main(), handle COM-messages.
endfunc
```

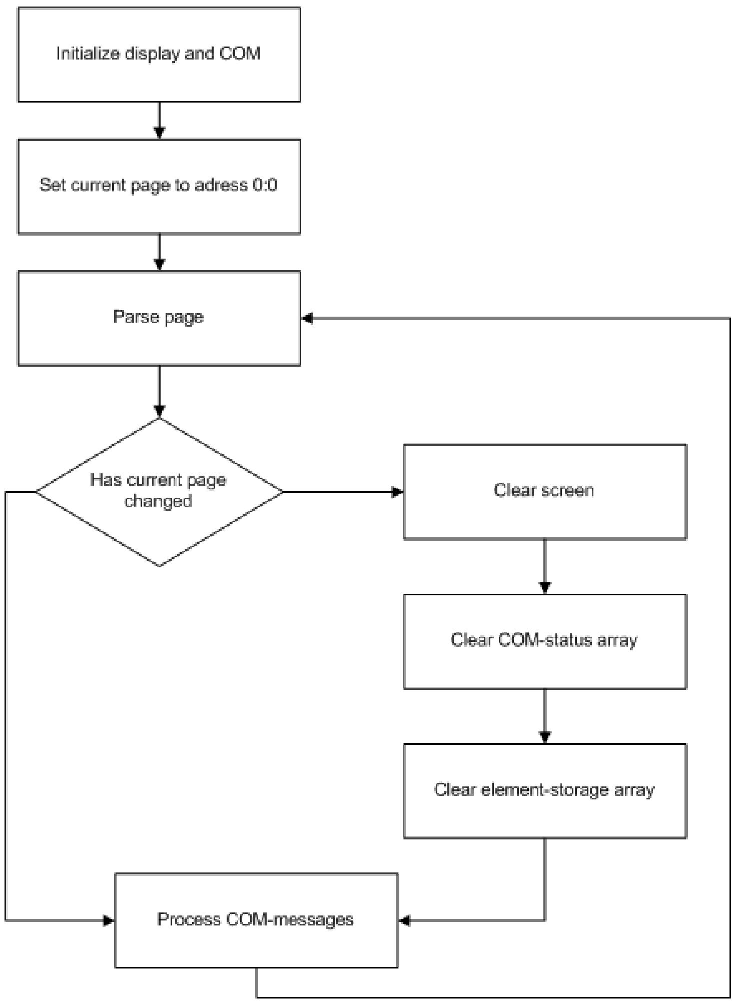


Figure 6.2: A flow-diagram of the main routine that is repeated in the interpreter. After initializing the display driver, the SD-card and the serial-port, the interpreter begins interpreting the bytecode of the main-*Page* - located at a fixed position on the SD-card. When the *PAGE_END*-byte is read, the *Page* is done and the GOLDELOX-chip returns to the main function where the next step consists in processing new messages that may have arrived on the serial-port. The above routine is repeated endlessly. The chip keeps track of the address of current *Page* - in the beginning the main-*Page* - when the *Page* has changed, for an example if a *Tile* that is linked to another *Page* was pressed, the screen is cleared and the *Page* at the new address is parsed.

After initializing the display driver, the firmware checks if an SD-card is present and establishes a connection via the COM-port. It then begins interpreting the bytecode of the main *Page*, located at sector 1, offset 0 on the SD-card. Its bytecode resembles the structure of its corresponding FUIML-file - the first element is the start-token of the *Page*. After the entire page has been parsed, the COM-buffer is checked for new messages, then the GOLDELOX chip resumes parsing the bytecode. If the current page has changed, the screen is cleared and the new page is drawn, otherwise only those elements that have changed are redrawn. See Figure 6.2 for a control-flow-diagram of the above.

6.4 Per-element storage

The chip continuously parses the page stored on the SD-card and would - if no measures were taken to prevent this - redraw each image seen on the screen with each iteration. Whereas the bytecode of a *Page* usually consumes about two or three sectors, a single 64 by 64 pixel image consumes 16 sectors (one sector being 512 bytes). Thus, sparing the GOLDELOX-chip from redrawing the same image over and over does have a major impact on the responsiveness of the user-interface. The key to counteracting the above problem lies in storing what elements already are present on the display so that time-consuming operations such as drawing an image take place only if they are truly necessary.

For the lack of support for dynamic memory management in the 4DGL-programming-language, this is accomplished by allowing elements to claim storage units within fixed-size arrays held in the on-chip-memory. Before a *Page* is parsed, the variable holding the current index within the above arrays is set to -1. When an element claims storage, the afore mentioned index is incremented and the element may use one field in each of the arrays to store information for future cycles within the main-loop of the firmware. Although in theory an unlimited number of screen-elements is supported, the length of the arrays allows only for a certain number of elements to store information.

In the case of images and animations, the address of the currently displayed image is stored. When the element is parsed, it checks if an address is present within its storage, meaning that no redraw is necessary. Otherwise, the image is drawn and its address is stored in the latter arrays. However, the various elements store a variety of different data. See Figure 6.3 for details.

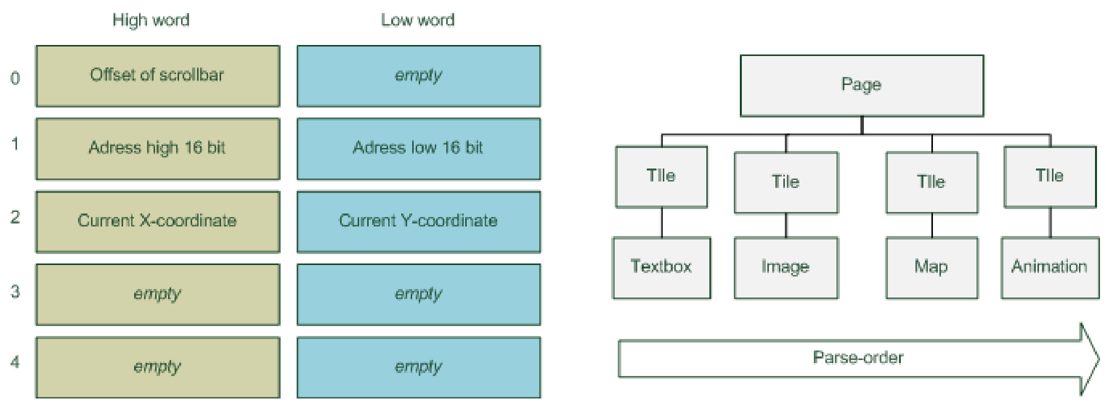


Figure 6.3: For the 4DGL language's lack of support for dynamic memory management, two fixed-size arrays - in total 32 bits - are used to store information about the current state of elements on the screen. When an element claims storage, a variable containing the index of the current position within the array is incremented. After a cycle within the main-loop that is repeated endlessly by the GOLDELOX-chip has passed, the index is set to -1.

6.5 The Status Array

To allow for a maximum of flexibility, most of the status variables used within the firmware are stored in an array mirroring the serial-port's buffer. Apart from two dedicated slots - the first two bytes of the array hold the coordinates of touches upon the display - the array can be used for a variety of purposes: the array can be used for a variety of purposes: Via the escape "$COM[index]" either individual bytes can be accessed from the FUIML file, or several bytes can be used as word-, double-word or string-values. In addition, individual pages may use the same slot within the array for different purposes.

Whenever a status-update is received via the serial-port (see Chapter 7.1), the message is copied into the status-array within the firmware. Figure 6.4 demonstrates the above in practice.

```
<Tile width="176" height="220" link="MainMenu.FUIML">
   <Map positionX="$COM[2]" positionY="$COM[4]"></Map>
</Tile>
```

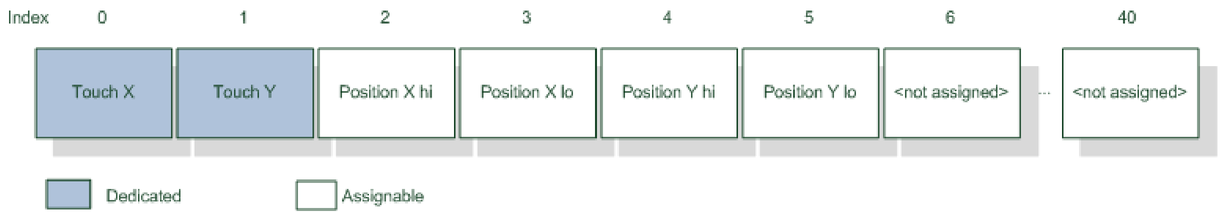


Figure 6.4: To allow for a maximum of flexibility, most of the status variables used within the firmware are stored in an array that mirrors the buffer that is available to the serial-port in order to assemble incoming messages. Apart from two dedicated slots - the first two bytes of the array hold the coordinates of touches upon the display - the array can be used for a variety of purposes: In this example, bytes 3 to 6 are used to represent the current position of the carrier of the device within the above map.

6.6 Handling Touches

As it was described in the previous section, the first two bytes within the status-array are reserved for the coordinates of touches upon the display. By default - for example whenever the current page changes - both values are set to 255, which lies outside the maximum screen size supported by the GOLDELOX-chip. Whenever a *Tile* that defines an action is parsed, it checks if a touch has occurred within its rectangle on the screen. If the touch has occurred within the bounds of the *Tile* the associated actions are executed. If the *Tile* contains a link, the address of the current page is set to that of the linked *Page*. To prevent the new page from reacting to a touch that has occurred on a previous *Page*, in the following cycle of the main-loop, the touch-coordinates are set to 255 and the new *Page* is parsed and displayed on screen. If the *Tile* has a remote-procedure-call defined, the identifier of the procedure that is referenced is sent to the main processor via the serial-port. Since the range of available coordinates is greater than the dimensions of the screen and since empty *Tiles* may be positioned outside of the screen's borders, a page may contain links to other pages that cannot be activated by the user but only via the main processor of the device. In this way, for example, a warning message can be displayed when battery-power reaches a certain minimum level.

6.7 Parsing Attributes

An attribute's value often is not defined in a constant manner within the bytecode, but may instead refer to an offset within the status-array or may be the result of a *Trigger*. In addition, the necessity of attributes whose size is greater than the 16 bit-architecture of the GOLDELOX-chip give rise to the need for a sophisticated technique of resolving the value of an attribute. Apart from some attributes that are always encoded directly into the bytecode - such as the dimensions of a *Tile* - whenever an attribute is parsed its property header is parsed. This one-byte space within the bytecode contains a number of flags upon wich the interpreter determines an attribute's

- size
- source
- encoding

- and if the attribute contains a *Trigger*.

While figure 6.5 outlines the general structure of the header-byte, the following sections will provide a more in-depth description of the individual flags.

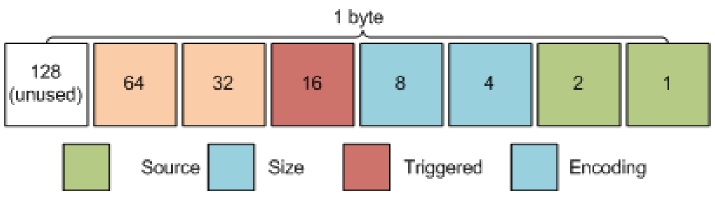


Figure 6.5: Apart from some attributes that are always encoded directly into the bytecode, such as the dimensions of a *Tile*, whenever an attribute is parsed its property header is read. This one-byte space within the bytecode contains a number of flags upon wich the interpreter determines an attribute's size, source, encoding and if the attribute contains a *Trigger*.

6.7.1 The Source-Flag

The source of an attribute is determined by the first two bytes of the property-header. There are 3 possible sources:

Source	*Description*
COM-Buffer	The value is located in the COM-Status-Array at a fixed offset. The offset is a fixed value that is located immediately after the property-header within the bytecode.
Timer	The value of attribute is determined by the internal timers of the GOLDELOX-chip.
SD-Card	The value of the attribute is contained within the bytes that immediately follow the property header.

Table 6.2: Sources that can be encoded in the source-flags of the property-header

6.7.2 The Size-Flag

Bytes 3 and 4 encode the size of the attribute's value:

Size	*Description*
TEXT	An variable-length-array of ASCII chars, zero-terminated.
BYTE	A single byte.
WORD	A 16-bit value.
DWORD	A 32-bit value.

Table 6.3: Size-flags that can be encoded in the property-header

6.7.3 Triggers

Triggers can be used to dynamically assign a value to a property at run-time instead of compile-time. This is accomplished via a series of comparisons. The first operand, located at the beginning of the *Trigger* is inserted into a number of equations consisting of an operator and the second operand. In the bytecode, each equation is followed by a result which constitutes a value that may be assigned to the triggered property.

The *Trigger* is evaluated until a comparison results in TRUE. In this case, the result is parsed and assigned to the property. After that, the interpreter skips over all other conditions until the end-flag of the *Trigger* is reached. The size of the resulting value and other properties are encoded within a header-byte which was introduced earlier in this chapter. The value of the result may as well be nested within another *Trigger*. Also see Figure 6.6.

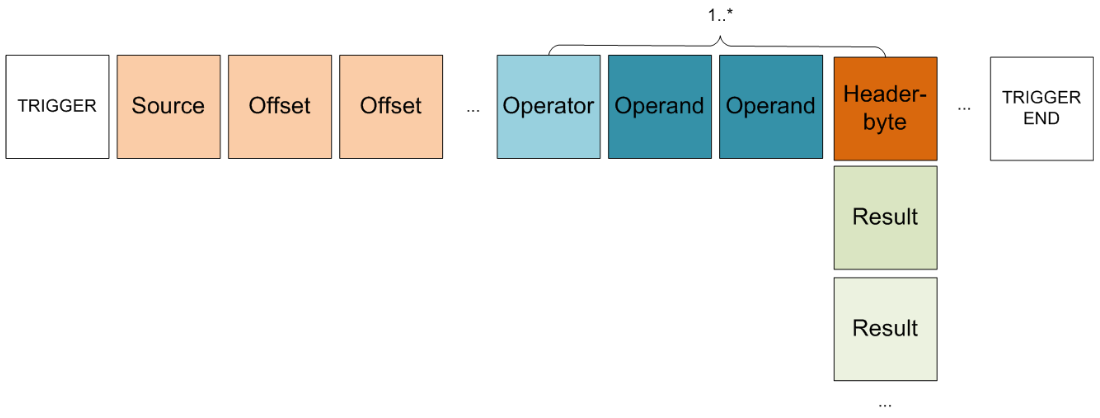


Figure 6.6: A diagram of a *Trigger's* structure within the bytecode. The first operand, located at the beginning of the *Trigger* is inserted into a number of equations consisting of an operator and the second operand. In the bytecode, each equation is followed by a result which constitutes a value that may be assigned to the triggered attribute. The size of the resulting value and other properties are encoded within a header byte which was introduced earlier in this chapter.

6.8 Animations

Animations are a sequence of *Images* that are sequentially displayed on screen by the GOLDELOX chip. In bytecode, *Animations* consist of the interval - the delay after wich an image is replaced by the next in sequence, the total count of images (one byte, thus maximally 255 individual images) and a sequence of *Images*. *Animations* are started randomly - whenever an *Animation* is parsed a random-value is generated and compared to a constant. If the comparison evaluates to TRUE, the *Animation* begins.

One of the timers that are available on the GOLDELOX-chip is exclusively reserved for animations. Every time the active *Animation* displays a new image, the timer is initialized with the afore mentioned interval. The GOLDELOX-chip automatically decrements the timer once every millisecond until zero is reached. Since the timer is evaluated only when the active *Animation* is parsed, there may be a delay between the timer reaching zero and the next image being displayed. Most of all, this delay depends upon the complexity of the current page and upon the *Animation's* width and height. Thus, the interval is to be considered a minimal bound for the time an image is displayed. In practice, intervals below 100 milliseconds can rarely be reached.

Although the above limitation may be problematic in some cases, it is a necessity that arises from the lack of glsacr:CPU-interrupts on the GOLDELOX-platform - making multi-threading impossible. It constitutes the only way of keeping the user-interface responsive while an *Animation* is active. As another measure in order to reduce the CPU-load caused by *Animations*, there can only be one *Animation* active at a time. Also see Figure 6.7 for a diagram of the structure of *Animations* within the bytecode.

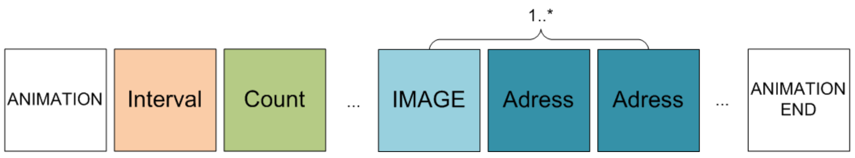


Figure 6.7: In bytecode, *Animations* consist of the interval - the delay after wich an image is replaced by the next in sequence, the total count of images (one byte, thus maximally 255 individual images) and a sequence of *Images*.

6.9 Textboxes

Textboxes serve as mere containers for labels of all datatypes. To allow for an individual appearance of the elements contained within a *Textbox*, all attributes concerning color, font-style, fontsize and number-format are declared by the *Textboxes* child-nodes. To facilitate the display of large texts - even such that exceed the amount of RAM available on the GOLDELOX-platform, a specialized component is available that also boasts a scrollbar to navigate through the text. While the latter will be introduced later, the following page concerns less complex labels.

Via the *FONT Tool* that is supplied by the manufacturer of the GOLDELOX-chip, the remaining amount of program memory that is not used by the interpreter on the GOLDELOX-chip - approximately 7 kilobytes - can be used to store custom fonts, see Figure 6.8.

6.10 Byte-, Word- and Double-Word-Labels

From the perspective of the implementation, numeric values represent the most basic of labels. As it is shown in Figure 6.9, they consist of a number of bytes that define all properties concerning a text's visual appearance that are available on the GOLDELOX-platform. These are the same for all labels - every time any label is parsed, the values are written into the corresponding registers within the GOLDELOX-chip.

In addition to the above, all numeric labels boast another property that allows for the definition of a customized number-format. It corresponds to the default way of formatting numeric values on the GOLDELOX-platform, which consists of a 16-bit value whose individual bits serve as flags representing the customized formatting. See Figure 6.10 for details. Like the above properties concerning the visual appearance of a label, the number-format is written directly to the chip without preprocessing.

Finally, the value that is meant to be displayed is parsed. Its source and other properties are encoded into a header-byte as it was introduced earlier. Thus - as shown in Figure 6.9, there is a high level of flexibility concerning the source of a value.

6.11 String-Labels

Whereas string-labels boast the same properties concerning aspects of visual appearance as their counterparts for numeric values, there is of course no need for a number-format.

In the simpler case - that of short, zero-terminated strings - a header-byte is used to define the value that is meant to be displayed on screen. Due to limitations in available screen-size and due to the platform's limited amount of available RAM, longer texts give rise to the necessity of a more complex approach than the one above.

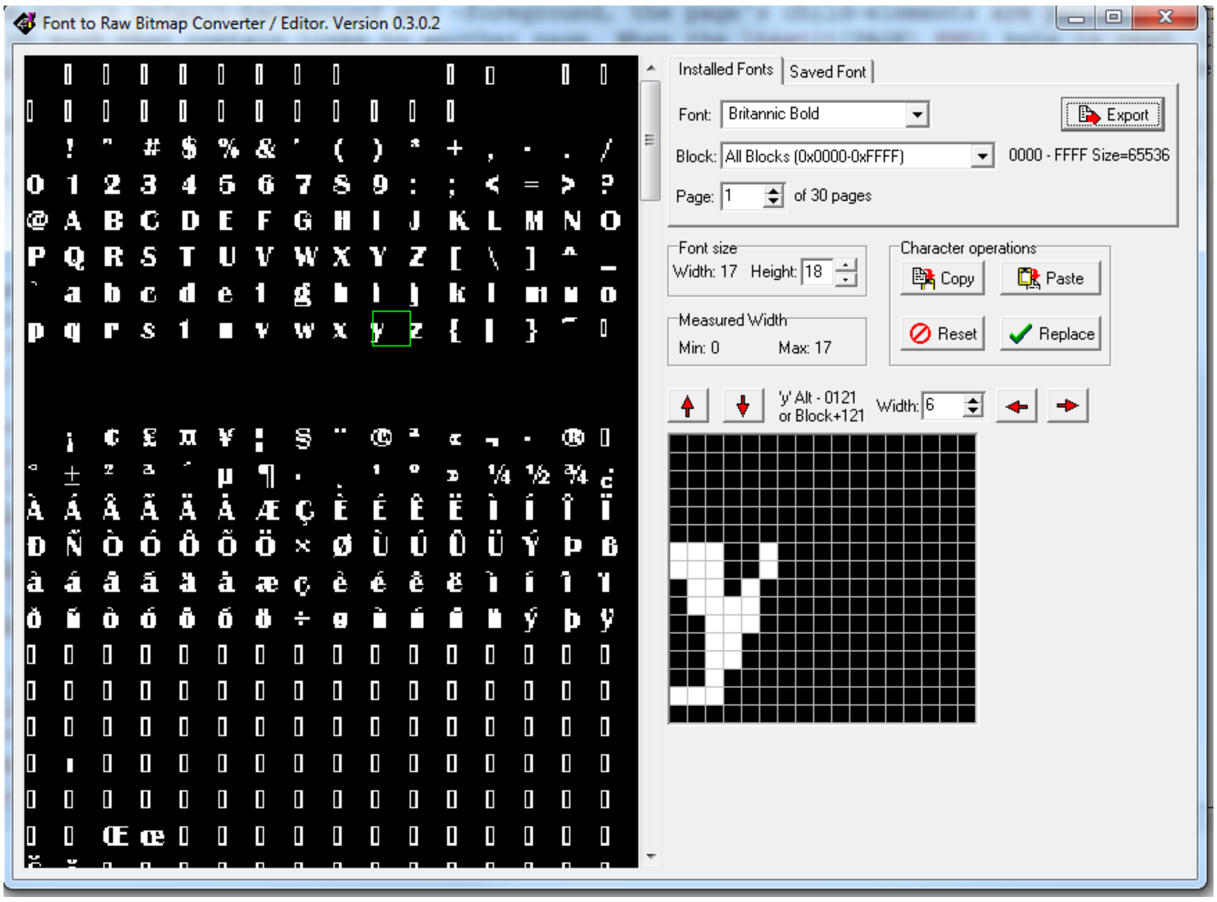


Figure 6.8: A screen-shot of 4D Systems's FONT Tool, which can be used to upload custom fonts to the GOLDELOX-chip. Via the drop-down-list in the top right corner, a font that is installed on the computer may be selected. The width and height of the individual characters, well as their margin can be adjusted. From the selected characters, the tool generates source-code that can be copied into the that of the program. Since fonts reside in program-memory on the GOLDELOX-chip, only a limited number fonts can be stored.

Instead of a header-byte, in this case the text is located at a fixed position on the SD-card. When the component is parsed for the first time, the interpreter displays all chars present on the SD-card until there is no more space available in the *Tile* that is occupied by the element. Via the platform's built-in functions to measure the width and height of a text, the interpreter automatically inserts a line-break whenever a line is full.

To address the fact that the screen can display only a very limited number of characters at a time, the component automatically displays a scrollbar if the text is too long to be displayed as a whole. By default, the scrollbar is set to the first line of text, line 0. Whenever a scrolling action occurs - a touch on the top half of the component causes the scrollbar to scroll up, one in the bottom half to scroll down - the above line-counter is decremented or incremented. If the counter starts at a line other than the first, all text is skipped over until the correct number of line-breaks has occurred - without regard for the line-break's cause, be it one that was automatically inserted or one that stems from the text itself. All remaining characters are printed on screen as long as additional space is available. In any case, the entire text must be read from its beginning to the end, so that the total number of lines can be determined - which is necessary for the scrollbar to be displayed correctly. To preserve processing time, the above procedure of redrawing text only occurs whenever the position within the text - respectively the scrollbar has changed. In Figure 6.11, the above is demonstrated in a visual way.

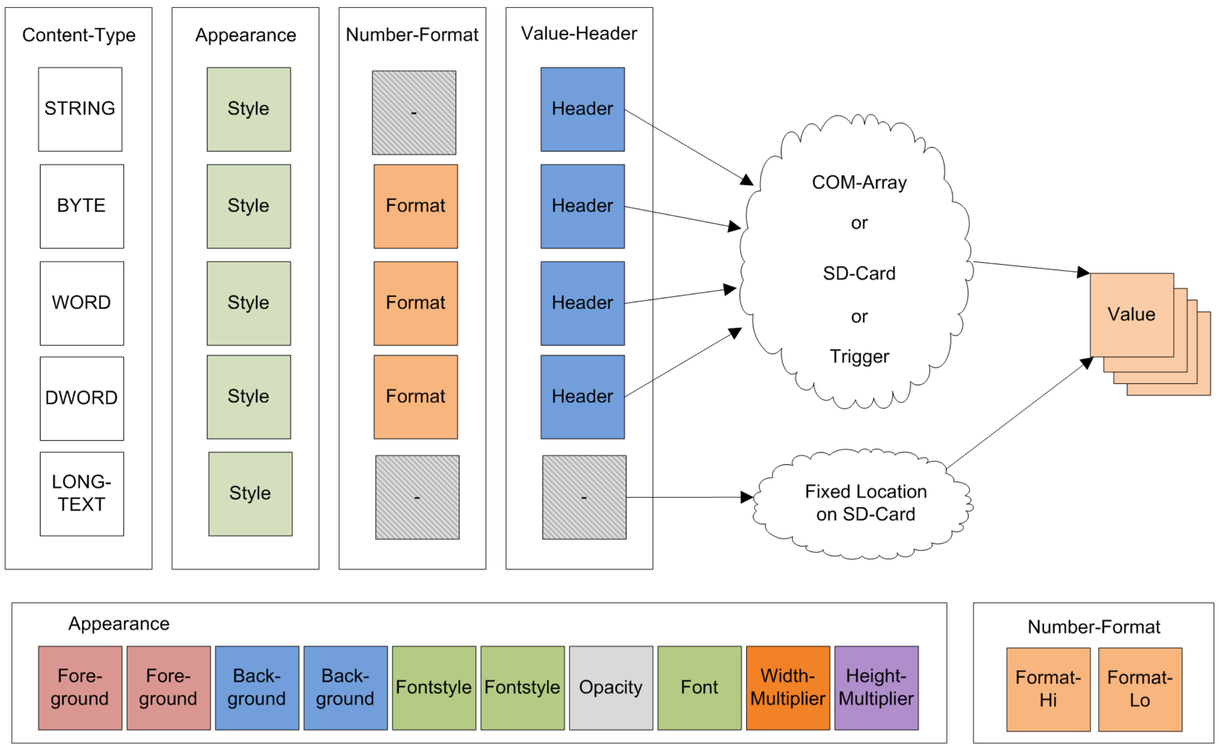


Figure 6.9: To allow for an individual appearance of the elements contained within a *Textbox*, all attributes concerning color, font-style, fontsize and number-format are declared by the *Textboxes* child-nodes. Text that exceeds the amount of characters that can be displayed at once can be shown on screen via the Long-Text-Label which includes a scrollbar.

6.12 Graph

As the saying goes - an image is often worth a thousand words. In the case of this project, the *Graph* can be used to tackle the task of presenting large amounts of data in an intuitive, visually appealing way. The first four bytes that are parsed hold the address of an image that is used as a background that fills the entire component. The next two words contain first the pen-color that is used to draw the chart's axis and that of the plotted line that represents the data. Finally, in the last 4 bytes, the address that points to the data on the SD-card is encoded.

The latter consists of one word containing the total count of data-points within the chart, followed by the Y-coordinate of each individual point stored in a single byte. This leads to a maximum of 65536

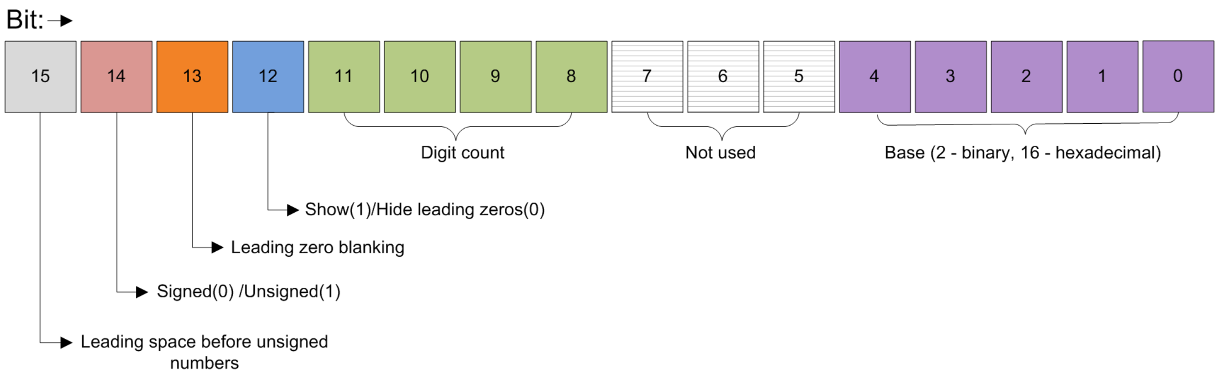

Bit:
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Digit count
Not used
Base (2 - binary, 16 - hexadecimal)
Show(1)/Hide leading zeros(0)
Leading zero blanking
Signed(0) /Unsigned(1)
Leading space before unsigned numbers

Figure 6.10: Number formatting bits supplied by format, based on [4D LABS, 2012], page 54.

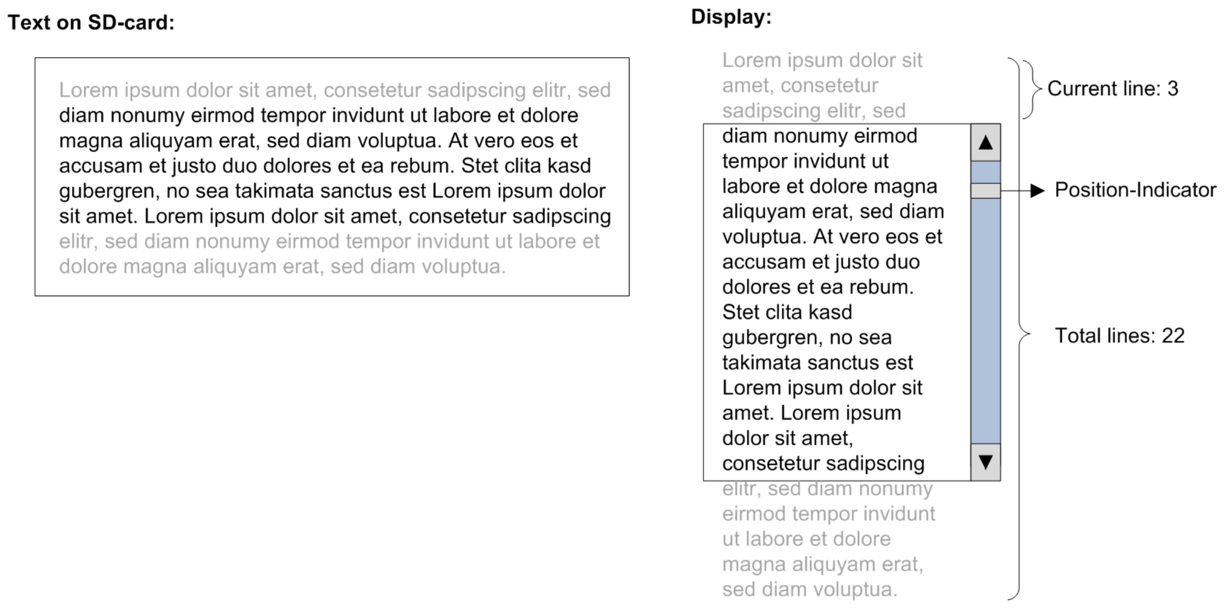

Text on SD-card:
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
Display:
Current line: 3
Position-Indicator
Total lines: 22

Figure 6.11: To address the fact that the screen can display only a very limited number of characters at a time, the component automatically displays a scrollbar if the text is too long to be displayed as a whole.

individual points. After drawing the X- and Y-axis of the graph, the data that is represented by the chart is plotted. First, a step-width is determined by dividing the width available by the number of points to be displayed. In turn a line is plotted that connects the individual data-points in the color that is specified for the plot. Also see Figure 6.12.

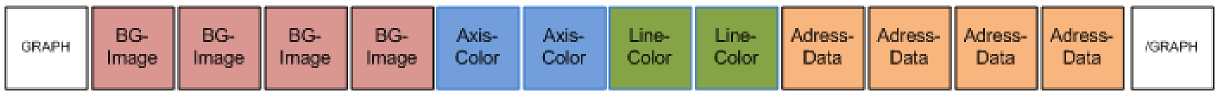


Figure 6.12: A diagram of the structure of the *Graph*-component in bytecode. It can be used to tackle the task of presenting large amounts of data in an intuitive, visually appealing way. The first four bytes that are parsed hold the address of an image that is used as a background that fills the entire component. The next two words contain first the pen-color that is used to draw the chart's axis and that of the plotted line that represents the data. Finally, in the last 4 bytes, the address that points to the data on the SD-card is encoded.

6.13 Map

This component can be used to display maps from a wide variety of sources. Whereas all maps that are shown in this document are geographic ones, the same component may as well be used to display the ground-plan of individual buildings. It contains two attributes - the X- and the Y-Coordinate on the map, starting from the top left corner. The actual map is located in a dedicated area of the SD-card.

The map consists of individual images of 64 by 64 pixels. In contrast to the default way of encoding images on the GOLDELOX platform the first six bytes, containing the width and height of an image and its colormode, are omitted. This approach holds a number of benefits:

- Each image uses exactly 128 sectors on the SD-card. The GOLDELOX platform can only display images that are located at the start of a sector. Since each image completely fills all sectors that are allocated to its storage, no space is wasted - there is no need for padding-bytes.

- In most cases, the map must display an extra row and an extra column of images (see 7.7) that are clipped at the *Tile's* borders. The larger the image, the larger the overhead for drawing parts of images in a non-display area.
- While smaller images may have virtues in terms of the above, they would require a larger index-table (see below) - leading to worse performance when locating images within the table.
- The position of a coordinate within an image as well as the image a coordinate addresses can efficiently be calculated via shift-operations.

For the limitations of the GOLDELOX platform, there is no feasible way of compressing images apart from Run-Length-Encoding (RLE). While RLE can be used for loading maps via the serial-port (see Chapter 7.4), it is not used to store images on the SD-card. Since maps consume a significant amount of memory and since in many cases - e.g. for displaying the route of a planned trip - only a small corridor is needed, the map may be of any shape. Missing sections are left out.

In bytecode, the map consists of two parts:

- The index-table containing the address of individual images and their position within the map.
- The individual 64 by 64 images that constitute the map.

The above index-table is a sorted list that is used to locate the surrounding images for a given location. Whenever the current coordinates within the map differ from those of the currently displayed map, the AND-operator is used to split the coordinates into those within the corresponding image (AND by FC00h), and in turn those that represent the position of the top-left corner of the image within the map. See Figure 6.13. In the case of the center-image, the first coordinate represents the image's negative offset from the center of the area allocated to the map. The other coordinate, that of the image within the map, is used to locate the image within the index-table. The images surrounding the center one are displayed in the same way, their coordinates can easily be calculated via a 64 pixel-offset from those of the center-image. This procedure is repeated for all adjacent tiles to the center-tile until the entire space allocated to the map is filled. If the coordinates of an image are not present within the index-table, a black rectangle is drawn instead.

6.13.1 The Index-Table

Since the index-table is a sorted array of coordinates, binary-search can be employed to determine the position of an image within the table. Through the application of the binary-search algorithm, a complexity of *O(log(n))* - n being the number of images in the index - is achieved when looking for a specific image within the table. The algorithm uses the following function to determine the relation between the coordinates currently being examined within the table and those that are supposed to be found:

```
//Compare col and row to currently read col/row.
func compare(var col, var row, var readCol, var readRow)
    if(col < readCol) return SMALLER;
    if(col > readCol) return GREATER;
    if(row < readRow) return SMALLER;
    if(row > readRow) return GREATER;
    return EQUAL;
endfunc
```

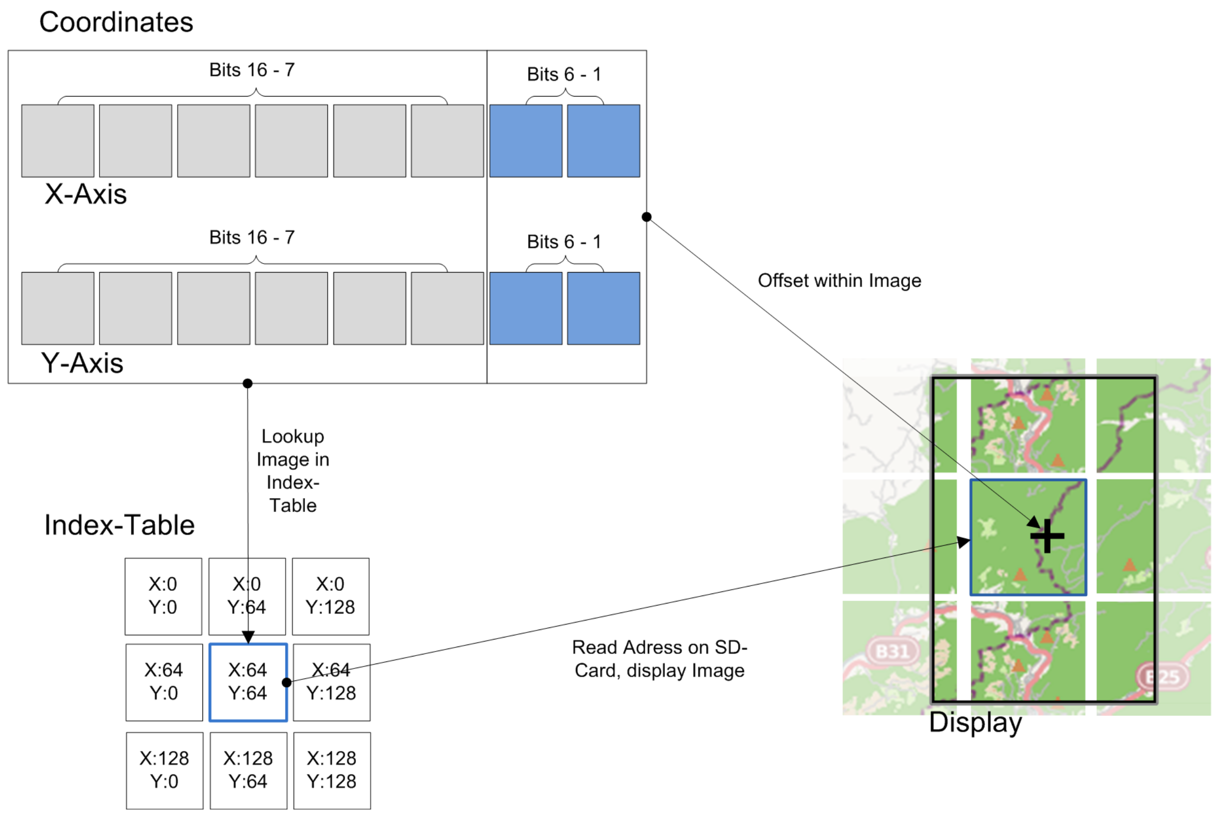


Figure 6.13: All coordinates are split on the one hand into those within the corresponding image, on the other hand those that represent the position of the top-left corner of the image within the *Map*. In case of the center-image, the first coordinate represents the image's negative offset from the center of the area allocated to the *Map*. The other coordinate, that of the image within the *Map*, is used to locate the image within the index-table. The images surrounding the center one are displayed in the same way, their coordinates can easily be calculated via a 64 pixel-offset from the center-image. This procedure is repeated for all adjacent tiles to the center-tile until the entire space allocated to the *Map* is filled. If an image is not present within the table, a black rectangle is drawn instead.

Listing 6.1: The index-table is a sorted array of coordinates. Through the application of the binary-search algorithm, a complexity of *O(log(n))* is achieved when looking for a specific image within the table. The above function is used to determine the relation between the coordinates that currently being examined within the index-table and those that are supposed to be found in the index.

As is shown in Figure 6.14, the first word within the table holds the total number of images that constitute the map. Since the latter is a 16-bit value, a maximum of 65536 images can be used in an individual *Map*.

The rest of the table describes the individual tiles of the *Map*

- The offset of the top-left corner of the image on the map's X-axis - **16 bit**.
- The offset of the top-left corner of the image on the map's Y-axis - **16 bit**.
- The address of the image on the SD-card - **32 bit**.

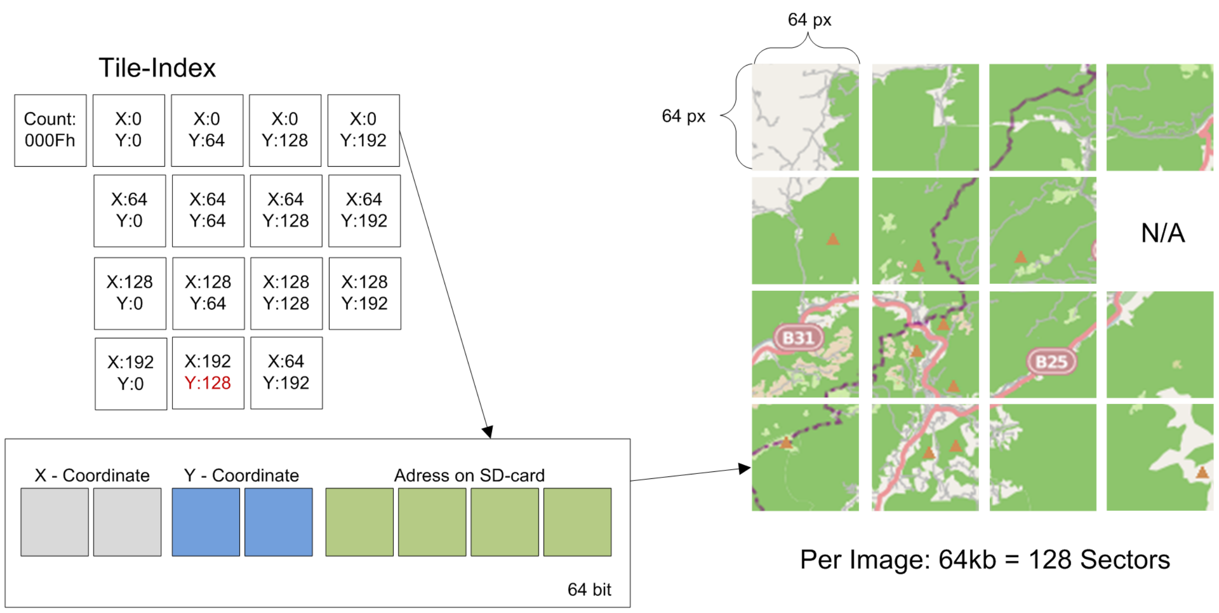


Figure 6.14: The first word within the index-table holds the total count of images that constitute the map. It is followed by a sequence of 64 bit entries that each describe an individual tile within the map. As a precondition for the binary-search algorithm that is used to locate specific entries within the table in fast way, the index must be sorted. If an image is not present within the index, a black rectangle is displayed instead.

When the corresponding image is located within the table, the GOLDELOX controller is set to the according position on the SD-card and the image is displayed. If the image is not present within the table, a black rectangle is drawn instead.

6.13.2 Limitations

Apart form the previously mentioned maximum number of images other factors pose limitations to the size of the map:

- The maximum number of images within the map is 65536.
- In both width and height, the *Map* cannot be bigger than 65536 pixels.
- Since each individual image requires 64 kilobits of storage capacity, the above 65536 images require more than 4 gigabytes of memory. The currently used SD-card may hold maps of up to 1.5 gigabytes size - approximately 24500 individual images.

To counteract the above limitations, as Figure 6.15 demonstrates each map may be of an irregular, non-rectangular format.

6.14 Minimizing Resource Consumption

As was pointed out at the beginning of this chapter, for all its virtues the GOLDELOX-controller is characterized by a number of limitations that have a grave impact on the design and implementation of the interpreter. The most relevant limitation is that of available RAM on the device. It boasts a total of one kilobyte of random-access-memory, only 510 bytes of which are at the disposal of the implementation. In combination with the lack of dynamic memory-management, the latter makes the efficient use of memory a point of critical importance. As an additional drawback, in contrast to a high-level programming

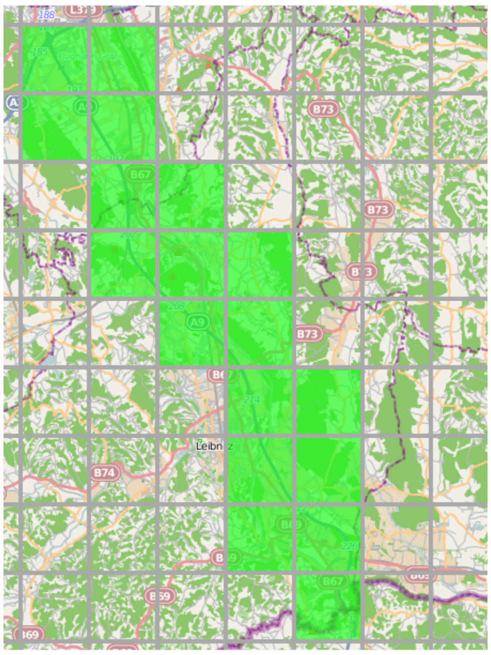


Figure 6.15: To counteract some of the limitations imposed by the GOLDELOX platform, each map may be of an irregular, non-rectangular format. In the above image, only a corridor surrounding a planned trip is selected (green). To preserve memory on the device, only the selected parts are copied upon the SD-card. This image stems from the *MapEncoder*-tool that will be introduced in detail in a later chapter.

language such as those of the .NET family, the execution does not stop when all available RAM has been used. Whereas in case of a .NET-language an exception is thrown including the execution-trace that lead to the error, the GOLDELOX-chip does not provide such a mechanism. In spite of the compromised RAM, it continues to operate - leading to an often erratic behaviour, the cause of which in most cases is hard to find. The above problem is tackled in multiple ways.

6.14.1 Avoiding unnecessary Allocation of Variables

For a programmer that is used to dealing with machines boasting multiple gigabytes of memory, the allocation of individual variables is rarely a point of major concern. In many cases, it may even be good-practice to declare helper-variables that make the code more readable and more maintenance-friendly. In case of the GOLDELOX-platform, even locally declared variables in rarely used functions noticeably decrease the remaining elbow-room in terms of free memory. The following measures were taken to minimize the impact of the above on available RAM.

- Constants are preferred to variables since they do not consume memory.
- Whenever possible, locally declared variables are used instead of static, globally declared ones.
- Variables are declared only within the scope where they are needed .
- Where possible, the encoding of the bytecode has been arranged in a way that minimizes the requirement of storing individual values for later use while parsing.
- Each variable consumes 16 bits of memory - yet often only one byte is needed. Thus the allocated memory may be shared to store two individual values.

6.14.2 Careful use of Function-Calls

Whereas function-calls are an integral part of any well-designed software, if used without consideration they may significantly affect performance not only on embedded systems but even on the world's most powerful supercomputers. On the one hand, at least the return-address of the call must be stored, if the call requires arguments, they must also be pushed on the stack. On the other hand, function-calls may cause an unnecessary overhead of locally declared variables that remain allocated longer than needed.

In the case of the *Map*, the binary-search algorithm that is used to locate the appropriate image for a given coordinate may either be implemented in a recursive or in an iterative fashion. The latter is in-place - no additional memory is needed regardless of the number of elements. A recursive implementation, however, leads to an overhead that is imposed by the need to store additional data on the stack. Whereas on a desktop-computer the above may not have a noticeable impact on the performance of the algorithm, it cannot be ignored on the GOLDELOX- platform.

Chapter 7

The Serial Interface

Whereas interpreting the bytecode constitutes one major part of the device-side implementation, the other lies in the handling of communication via the serial-port of the GOLDELOX-processor. The port uses a dedicated area of memory to buffer and assemble incoming messages. In the case of this project, a 40 byte buffer is used and the port operates at the processor's default baud-rate of 115200 symbols per second.

Every time the interpreter has finished parsing the bytecode of the currently active page, the serial-port is tested for new messages using a built-in function that returns the number of bytes available within the above buffer. Since a number of different in-house protocols is supported, the first byte is used to determine the mode of operation for a given message:

NAME	DESCRIPTION
COM_STATUS_MODE	All bytes read are copied directly into the status-array, introduced in the previous chapter. The array may be referenced from within FUIML-files.
COM_ALERT_MODE	In case of an external event such as an incoming message or call, this message mode instructs the user-interface to display the appropriate page.
COM_DATA_MODE	In this mode, all data received is stored on a certain position on the SD-Card.
COM_RLE_DATA_MODE	The same as above, but the data is compressed using Run-Length-Encoding.
COM_COMMAND_MODE	Allows external devices to execute certain pre-defined functions within the GOLDELOX-processor.
COM_SDVERBOSE_MODE	Used to read out the content of the SD-card at a certain location via the COM-port.
COM_SCREENSHOT_MODE	Used to send the content of the display as an image via the serial-port.
COM_FIRMWARE_MODE	Used in the process of updating the main processor of the device.

Table 7.1: A list of protocols that are available for communication via the serial-port.

Most modes require additional information to be processed such as an address on the SD-card or the number of packages that will be transmitted:

```
//busy-wait until <count> bytes available in buffer,
//or COM_WAIT_FOR_MESSAGE_TIMEOUT reached.
func ComBuffer_WaitFor(var count)
    *COM_TIMER := COM_WAIT_FOR_MESSAGE_TIMEOUT;
    while(com_Count() < count)
        if(*TIMER3 == 0)
            return FALSE;
        endif
    wend
    return TRUE;
endfunc
```

As the above code-snippet demonstrates, if the number of bytes available in the buffer is less than that required, the chip waits for the missing parts until a time-out is reached. One of the available timers is initialized with a constant value that is automatically decremented by the chip. To allow for the recovery from errors, if the number of available bytes in the buffer is less than that required and if the timer has reached zero, the above function returns FALSE. In this case, *COM_ERROR* - the number 255 - followed by an error-code is sent to the main processor and the port is reset. The following table contains all possible error-codes and a brief explanation of their meaning.

NAME	DESCRIPTION
COM_ERROR	Something went wrong.. precedes the error-code.
COM_UNKNOWN_MODEFLAG	The specified mode-flag (see previous page) is unsupported or unknown.
COM_MODEFLAG_TIMEOUT	A time-out has occurred while waiting for the appropriate number of bytes for the chosen mode.
COM_CANCEL_ARGCOUNT	The supplied number of arguments does not suffice for the chosen mode.
COM_PORT_ERROR	A possibly hardware-related error has occurred on the serial-port (error in built-in port-management).
COM_FULL	The port's buffer is full, the incoming message could not be processed in time.
COM_NACK	The counterpart to *ACK* (ASCII 06) is sent if an unexpected response is read on the COM-port.

Table 7.2: A list of error-flags that may be sent via the serial-port.

In the following pages, each mode is presented in detail.

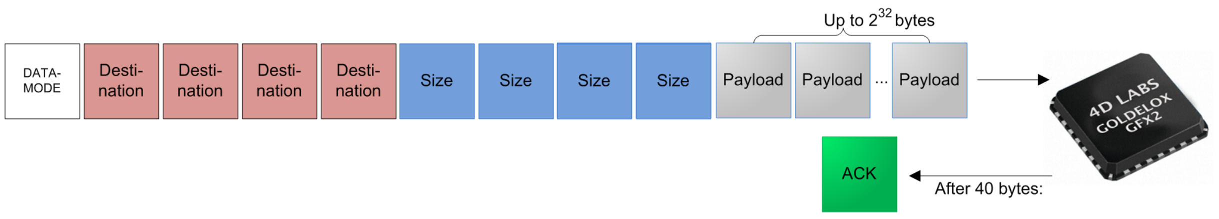


Figure 7.1: The data-mode is used to store chunks of data on the SD-card. In theory, up to 4 gigabytes of data can be transmitted.

7.1 The Status-Mode

In previous chapters, the status-array has frequently been referenced. It is a 40 byte-array that can be used to transmit a variety of information from the device's main-processor to the GOLDELOX-chip. Apart from the first two bytes which contain the coordinates of the latest touch upon the display, the content and arrangement of the individual fields can freely be chosen. Whenever this type of message is received, the entire message is copied into the status-array. From there the individual fields can be referenced by FUIML-components via the $COM[] escape-sequence. Updates are only necessary if the state of one of the above fields has changed.

7.2 The Alert-Mode

Whenever an unexpected event such as an incoming call occurs, the user-interface must display an appropriate page. Hence, in the FUIML-definition of the user-interface *Pages* can be associated to an alert-code. The above code is an identifier that is known to the main-processor. It is contained in the second byte of the message. After looking up the exact address of the requested page using the supplied identifier, the requested page is shown.

7.3 The Data-Mode

This transmission-mode is used to store chunks of data on the SD-card. The first four bytes in the protocol-header contain the destination-address on the SD-card. The following 4 bytes contain the total number of bytes that will be sent. Therefore - in theory - up to 4 gigabytes of data can be transmitted. After the above bytes have been read and processed, all incoming bytes are written directly to the SD-card. Since when using high baud-rates, the additional effort of writing to the SD-card can cause an overflow in the port's buffer, an optional feature can be used to make sure that the GOLDELOX chip can handle all incoming data: Whenever a total of 40 bytes has been read from the port, the chip sends a single *ACK* to its counterpart. For the latter, this is meant to indicate that the GOLDELOX-chip is ready for more data to be received. Also see Figure 7.1. The remaining number of bytes is stored throughout the transmission. Whenever a byte is read from the port, the count of remaining bytes is decremented. When all bytes have been read, the chip resumes parsing the the user-interface.

Since in case of an error the chip would wait forever for the remaining bytes to arrive, a timer is used that is reset whenever a byte has been read from the serial-port. If a time-out occurs, an error-message is sent and in turn normal operation is resumed.

7.4 The Run-Length-Encoded Data-Mode

This transmission-mode is a variant of the above data-mode. Run-length-encoding is a simple algorithm for data-compression. Instead of sending each byte indiscriminately, the data that is sent is examined for the occurrence of sequences of the same byte. If a sequence is found, the byte itself and the number of its occurrence is sent instead of sending the entire sequence. The standard-version of the algorithm is a form of lossless compression. Via an additional parameter that is not present in the normal data-mode, the type of algorithm to use is specified:

- **The loss-less-mode**: This mode is suited for data such as the bytecode of the user-interface. Whenever a loss of data cannot be tolerated, it is the algorithm of choice. in the case of data that contains a high entropy and hence few consecutive sequences of the same byte, the overhead of sending the count of occurrence along with the actual bits may lead to a transmission that is in a worst-case-scenario twice the size of the actual data.

- **The image-mode**: This mode exploits a special property of the image-format that is used on the GOLDELOX-platform. Since 16 cannot be divided by three, the green-channel of each pixel occupies 6 bits whereas red and blue are encoded within 5 bits. In this mode however, green is allocated only 5 bits, whereas the remaining bit is used as a flag that stores whether the color will occur multiple times. In case of the latter, the following byte represents the number of occurrence, whereas otherwise the following word represents the next pixel. In this way it can be assured that the transmission is at least no longer than the original data. However, the size of the transmission is reduced at the cost of color-depth.

In an empiric test of the second version using an image that displays a map, the algorithm boasted a compression-ratio of less than a quarter of the size of the original image.

7.5 The Command-Mode

Some of the built-in functions of the GOLDELOX-processor must be available to the main-processor of the device. Via the command-mode, the latter can be executed remotely by sending the appropriate command. Functions that can be requested include:

- Resetting the chip.

- Sending the display in sleep-mode in order to preserve battery-power.

The message consists of two bytes - the mode-flag and the code of the pending operation.

7.6 The Readout-Mode

Apart from its use in the process of flashing the main processor's firmware which will be introduced later, this feature has proven particularly valuable in the course of the implementation of the project as a debug-tool. It causes the GOLDELOX-controller to read-out all data at a given location on the SD-card. After the mode-flag, four bytes designate the address that is meant to be read. Another four bytes contain the total number of bytes that shall be read. After all required information was received the readout of the content of the SD-card at the specified location is commenced. When the requested number of bytes has been sent, normal operation is resumed.

7.7 The Screenshot-Mode

This mode allows for capturing the content of the display which is then sent via the serial-port. It consists of a single byte - no additional arguments are required. It uses built-in functions of the GOLDELOX-chip to access and read the graphics-memory of the display. Pixel by pixel, the image is transmitted until the entire screen has been sent. The stream of bytes that is received on the other side of the port constitutes a BGR565-encoded image - the same encoding that is used by the compiler to encode images for display on the target-device.

As a result of storing images in 18 bit color-depth - whereas the GOLDELOX-platform is a 16 bit processor - the currently used display is incompatible to the screen-shot-mode, since the above built-in functions to read out individual pixels return scrambled 16bit- colors. All colors that are sent to the display are converted into the above format and are displayed correctly. For this unfortunate circumstance, all images of the display that are shown in this document have been recorded by camera.

7.8 The Firmware-Mode

Since currently, the GOLDELOX-chip is the only component within the target-device that has access to large amounts of storage space in the form of the SD-card, firmware updates for the main processor of the device must be re-routed via the serial-port to the GOLDELOX chip. The entire firmware is stored on the SD-card and is then sent back to the main processor.

The process is initialized via a message that contains the corresponding mode-flag, the count of individual packages that constitute the firmware and a time-period that the GOLDELOX-chip must wait for before sending a package back to the caller. After this, a confirmation message is sent and the chip waits for the individual packages to arrive. Apart from the destination-address being omitted, each package is transmitted in the same way as described in Section 7.1. The above destination-address is calculated automatically based upon a fixed position on the SD-Card. When all packages have been received without error, the chip waits for the specified timespan until the signal *ACK*, *'R'* arrives. Each package is sent via the port using the readout-function introduced before. In between two packages, the chip also waits for the above *ACK*, *'R'* - a delay that is needed for the main processor to overwrite its existing firmware with the one stored by the GOLDELOX-controller. In case of an error after the appropriate error-message has been sent or after the last package has been sent via the port normal operation is resumed.

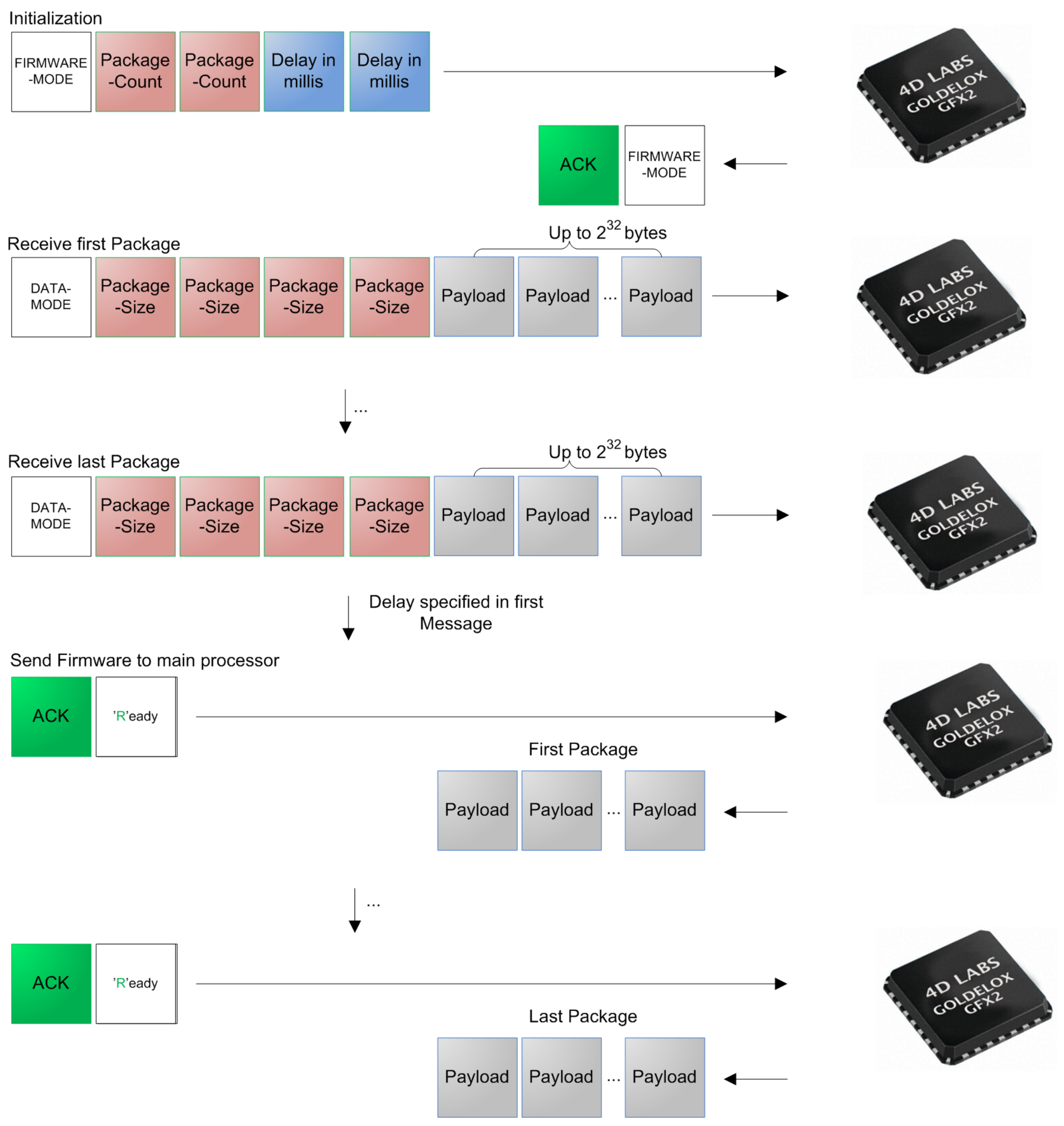


Figure 7.2: A visualization of the protocol that is used in order to update the firmware of the main-processor. Since currently, the GOLDELOX-chip is the only component within the target-device that has access to large amounts of storage space in the form of the SD-card, firmware updates for the main processor of the device must be routed via the serial-port to the GOLDELOX chip. The entire firmware is stored on the SD-card and is then sent back to the main processor.

Chapter 8

Tools

Whereas in the previous chapters, the main components of FlashUI have been introduced, the spotlight of this chapter focuses on a number of additional tools that have been created in parallel to the above. While some facilitate the use of certain features within the platform - such as the *MapEncoder* - and are still in use, others have contributed in the process of implementing and debugging the afore mentioned core components.

8.1 The Serial-Port Tool

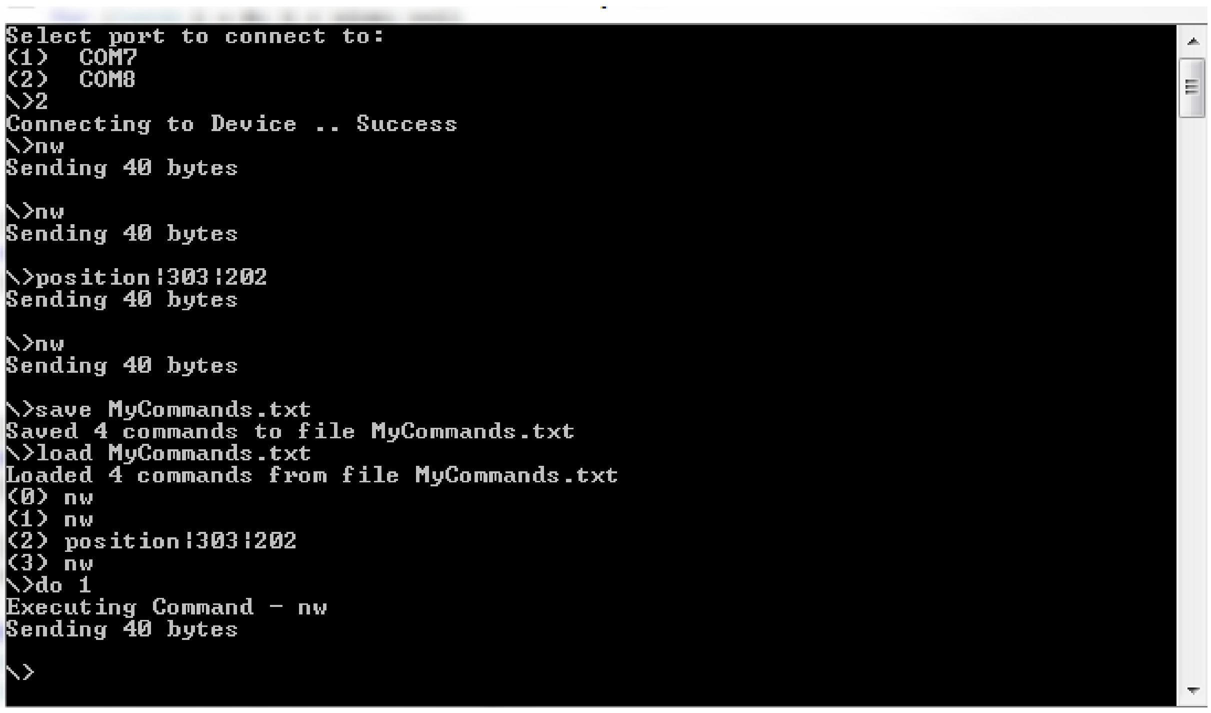
Since the program that is used on the main processor is not developed by the author, the need for a tool that facilitates testing the communication via the serial-port of the GOLDELOX-chip has arisen in the process of implementing the features and protocols that where introduced in the previous chapter. Some protocols, such as that used to flash the firmware of the device's main processor can only be tested programmatically via a tool that on the one hand automatically generates significant amounts of data and on the other hand handles the requirements of the underlying protocol. Apart from the above, the serial-port tool also greatly reduces the number of commands that must be entered manually in order to test certain components of the implementation.

Along with all components that are not intended to be executed on the GOLDELOX-chip, the tool is written in the programming language *C#*. It consists of two components:

- A class-library that provides an automated handling of core aspects concerning communication via the serial-port not only for this but also for other tools that will be introduced later in this chapter.
- A command-line front-end that makes the above library available for use on various platforms and operating systems.

Upon launch, the tool displays a list of available ports to connect to. After either entering the name of the port as displayed or the number in brackets preceding the name of the latter, connection is established to the selected port and the command-prompt is ready to operate. The tool uses an array of bytes that mirrors the status-array within the GOLDELOX-firmware. This facilitates the use of shorthand-commands that affect only certain indices within the array, whereas the entire array has to be transmitted to the chip every time a command is issued. Among the above shortcuts are commands such as:

- *NW,NE,SW,SE* which represent touches on the center of the north-western-, north-eastern-, south-western- or south-eastern quadrant of the display.
- *POSITION* which sets the device's position on the map to the supplied coordinates.



```
Select port to connect to:
(1)  COM7
(2)  COM8
\>2
Connecting to Device .. Success
\>nw
Sending 40 bytes

\>nw
Sending 40 bytes

\>position|303|202
Sending 40 bytes

\>nw
Sending 40 bytes

\>save MyCommands.txt
Saved 4 commands to file MyCommands.txt
\>load MyCommands.txt
Loaded 4 commands from file MyCommands.txt
(0) nw
(1) nw
(2) position|303|202
(3) nw
\>do 1
Executing Command - nw
Sending 40 bytes

\>
```

Figure 8.1: A screen-shot of the serial-port tool. After a port has been selected by the user, connection is established to the device. In turn a number of commands is entered and they are saved to a file. To access commands that were saved in a previous session, they must first be recalled via the *load*-command. After that, they can be automatically executed by typing *do* followed by the number of the command.

Additionally, there are also shorthand commands that can allow for the use of the data-mode introduced in the previous chapter to write an arbitrary number of bytes to the supplied address on the device's SD-card and various functions that allow for the automated execution of stress-tests for certain components, among the latter, a test for the protocol used to flash the main processor on the device. It sends several thousands of packages of a random size to the chip and automatically verifies the content of the received packages on their round-trip from the device.

For additional convenience, all entered commands are stored in memory and can be saved to a file. At a later point in time the stored commands can be loaded from the file and in turn are ready for use in the new session:

- *SAVE 'filename'* saves all commands executed so far to the file.
- *LOAD 'filename'* loads the stored commands into memory.
- *LIST* prints a list of available commands that have either been loaded from a file or been entered in this session.
- *CLEAR* removes all of the above commands from memory.
- *DO 'index in list'* executes the selected command. The index corresponds to that displayed when using the *LIST* command.

For a brief example of the use of the above commands in practice, also see Figure 8.1.

8.2 The MapEncoder-Tool

This tool provides an easy way of creating FUIML-compatible maps from a number of wide-spread image-formats. Just like the other tools presented in this chapter, the *MapEncoder* is implemented in C# and consists of a class-library that exposes key functions to other components. Since the encoder relies heavily on operations that are also needed in the process of compiling FUIML-code to bytecode, it uses the compiler-library for tasks such as image-processing and bytecode-manipulation. There are two front-ends available:

- A command-line version for use on platforms lacking support for WPF user-interfaces.
- A graphic-user-interface available on Microsoft Windows-platforms boasting the .Net Framework version 3.5 and upwards.

The basic process of encoding a map is the same for both user-interfaces:

1. An image that contains the map must be loaded.
2. The image is split in individual tiles of 64 pixels width and 64 pixels height.
3. A blank index-table that in the end will contain the coordinates and address of each tile is allocated using classes that are available in the compiler-library.
4. Each of the above images is converted into the BGR565-image-format that is used on the GOLDELOX-platform.
5. The image is encoded in the bytecode-object that is supplied by the compiler.
6. The coordinates and address upon the SD-card of the image are entered in the index-table at the corresponding position.
7. When all images have been processed, the bytecode is dumped into a file.
8. At the user's discretion, the file is copied to the SD-card - either directly or via the serial-port.

8.2.1 The Graphical Front-end

The tool's graphical front-end is a WPF-application that in contrast to its command-line-pendant allows for the selection of individual tiles to be included in the encoded map. As shown in Figure 8.2, a large portion of the window is allocated to the image that contains the map. Via scrollbars, the user may pan to a certain position within the map. On top of the image, a grid is drawn containing cells of 64 by 64 pixels. Each cell represents a tile within the map that can be selected for encoding. Whenever the mouse-pointer enters a cell and either mouse-button is pressed, the cell's state of selection is inverted - selected cells are de-selected and vice-versa. In this intuitive way, those portions of the image that are of interest in the user's context - for example for a planned trip - can be selected.

When the map is encoded, only those tiles that have been selected by the user are processed. In order for binary-search to work, the index-table must be sorted. This is accomplished in the process of encoding the map which is the same as in the command-line-version.

After encoding the map, a dialogue that contains a list of targets for saving the map - either devices connected via a serial-port or removable media - is shown. See Figure 8.2 for a screen-shot of the graphical front-end.

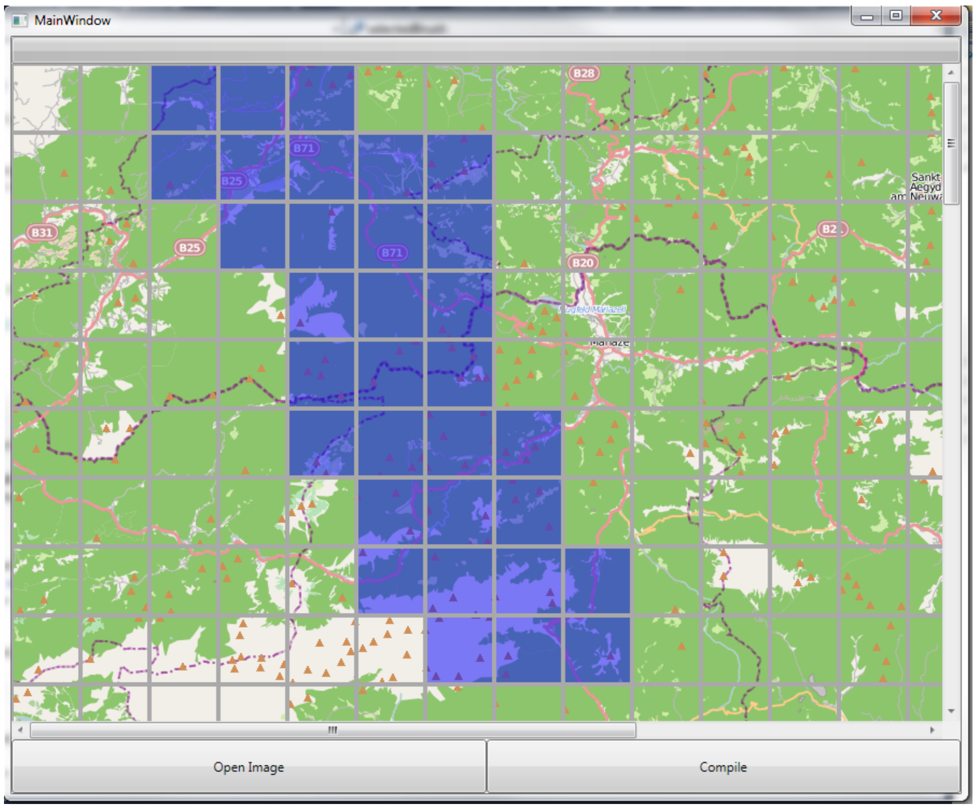
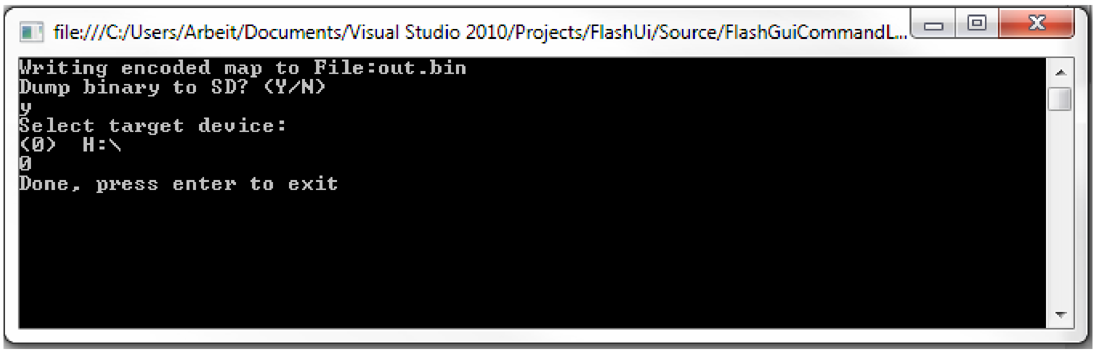


Figure 8.2: The graphical front-end is a WPF-application that allows for the selection of individual tiles to be included in the encoded map. Whenever the mouse-pointer enters a cell and either mouse-button is pressed, the cell's state of selection is inverted - selected cells are de-selected and vice-versa. When the map is encoded, only those tiles that have been selected by the user are processed. After encoding the map, a dialogue that contains a list of targets for saving the map - either devices connected via a serial-port or removable media - is shown.


file:///C:/Users/Arbeit/Documents/Visual Studio 2010/Projects/FlashUi/Source/FlashGuiCommandL...

```
Writing encoded map to File:out.bin
Dump binary to SD? (Y/N)
y
Select target device:
(0)  H:\
0
Done, press enter to exit
```

Figure 8.3: In contrast to the graphical front-end, the command-line-version is also available on operating systems other than Microsoft Windows. When the tool is launched and its command-line-arguments have been verified, the map is encoded and written into the output-file. After this, a dialogue is shown that containing a list of targets for uploading the map - either devices connected via a serial-port or removable media.

8.2.2 The Commandline Frontend

Since the above graphical front-end is based upon Microsoft's WPF-framework - unsupported by the Mono-project - it is not available on operating systems other than Microsoft Windows. As an alternative a command-line-version of the graphical tool can be used to encode and upload maps to the target-device. It requires two arguments:

- The path of the image containing the map.
- A file-name for the output-file.

When the tool is launched and the above arguments have been verified, the map is encoded and written into the output-file. After this, a dialogue is shown containing a list of targets for uploading the map - either devices connected via a serial-port or removable media. See Figure 8.3 for a screen-shot of the command-line version.

8.3 The screen-shot-Tool

In the previous chapter, a protocol for sending a screen-shot of the display on the target-device via the serial-port has been introduced. This tool can be used to simplify the process of capturing the screen and decoding the image. When the tool is started, a dialogue is shown that allows for the selection of a port to connect to. When the connection has been established, a click on the button labelled 'Capture' causes the tool to request a screen-shot from the device, which upon its arrival is displayed in the panel above the button. See Figure 8.4 for a screen-shot of the tool.

8.4 The Screen-Simulator

For the low-level nature of the implementation and for a lack of tools that allow for the examination of control-flow within the interpreter at run-time, debugging the firmware of the GOLDELOX-platform has frequently turned out difficult. For this reason and for other reasons of convenience, a program that simulates the bytecode-interpreter has been implemented for use personal computers. The simulator relies upon the same code as it is present in the interpreter running on the GOLDELOX-chip, translated into the C# programming-language. Numerous components of the interpreter were implemented first for the screen-simulator and then adapted for use on the GOLDELOX-platform. The primary reason for

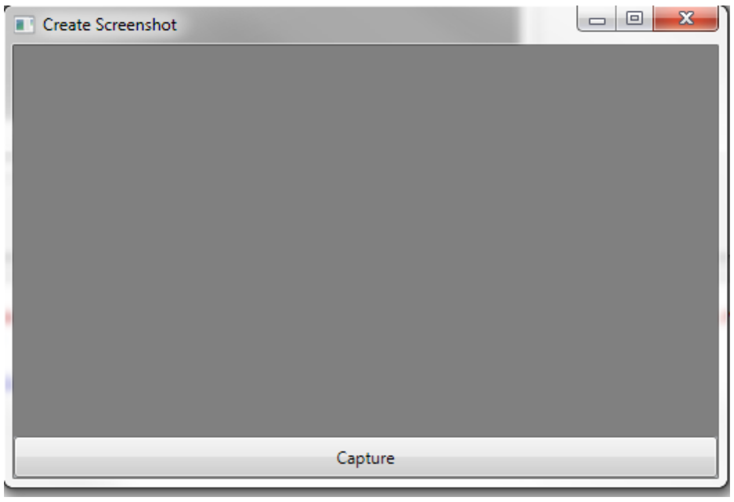


Figure 8.4: The Screen-Shot-Tool simplifies the process of capturing the screen and decoding the image. Upon clicking the *Capture*-button, a screen-shot is requested from the device which upon arrival is displayed by the tool.

this additional effort can be found in the advanced debug-features boasted by modern development environments that far surpass those available for the 4DGL-programming language. Although the simulator was a valuable aid in the process of implementing the bytecode-interpreter, its further development was terminated after all complex components such as the map had reached a stable state. See Figure 8.5.

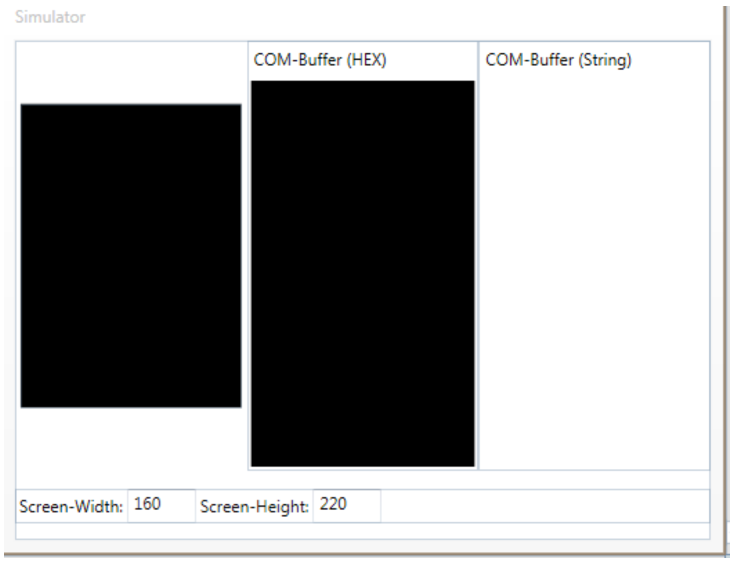


Figure 8.5: A screen-shot of the Screen-Simulator. The tool relies upon the same code as the interpreter running on the GOLDELOX-chip. On the left, the content of the display is rendered, in the middle and on the right, the content of the COM-buffer of the device is shown.

Chapter 9

Summary

FlashUI is a GUI framework that facilitates the device-independent creation of user-interfaces for a family of applications for a hand-held device that boasts a high degree of mobility. Via a custom-made markup-language called FUIML, user-interfaces can be designed in a declarative way - instead of focusing on *how* the user-interface is drawn on screen, the designer may focus on *what* is displayed. For a lack of entanglement between the user-interface and the application layer, the design follows in the footsteps of other successful designs such as HTML and the X Server System that abstract the presentation-specific components of the GUI from application-specific components. The common idea - Separated Presentation - ensures a high degree of flexibility in terms of the hardware that is used. For a lack of device-specific instructions in most of the framework's components, future changes in the specifications of the device can be reflected in a single place instead of affecting multiple components.

Among the design-patterns that were introduced at the beginning of Chapter 3, the design resembles the MVC-pattern. The MVC-pattern stems from a time when the management of input-devices was an intricate part of the application. In today's frameworks that target desktop-applications, much of the above has been shifted into the responsibility of the framework. Hence patterns that have been designed in more recent years may focus on a more fine-grained view of components that are associated to the management of data. In the device at hand, however, the complexity of the data is limited - most of it consists of either individual numbers or one-dimensional arrays.

In addition, the device is designed for tasks that focus on the *display* of data. Its applications require a different approach than one that aims at desktop-computers. Whereas the latter may be designed to capture all of the user's attention, the applications at hand are designed as an *aid* for specific tasks - the user's focus remains at the task itself.

For this reason and for the limited amount of interaction that is boasted by the device, manipulations upon the data that is gathered are less complex and occur less frequently than in other applications. Hence, the benefits that can be achieved by making use of more sophisticated techniques such as the Observer-pattern that is advocated by the MVC-pattern are limited.

Nonetheless, the design that has been chosen manages to incorporate many of MVC's core features. The triad that lies at the heart of MVC is reflected by the hardware of the device and by its software-architecture. The main processor that is responsible for the management of on-device sensors and wireless connectivity resembles MVC's model. The controller is reflected within the firmware that resides within the graphics-controller. A dedicated component of the firmware manages all communication via the serial-port - including user-input. The design of FUIML is based upon MVC's concept of views and sub-views, in concert with the bytecode-interpreter that is used to render the screen of the device it represents the triad's last component. Whereas the author does not claim that the application implements the MVC-pattern, it nonetheless transfers most of MVC's benefits to a different domain than that which MVC was originally intended for. Perhaps the most important benefit that is gained through the design's resemblance to MVC is a strong separation of concerns.

FUIML introduces an abstraction layer that facilitates not only the platform-independent design of user-interfaces but also alleviates the designer from tedious, hardware-related tasks such as the management of the arrangement of images on the SD-card as well as the scaling of images. Instead, the compiler introduced in Chapter 5 that translates FUIML to a more machine-friendly bytecode takes care of the above tasks so that GUI-designers may focus on their actual field of expertise.

The efficient implementation of the interpreter that constitutes a large part of the firmware of the graphics-controller compensates for much of the limitations that are present on low-end graphics-controllers. Widgets such as the *Map* boast a set of features that can usually be found on more potent hardware such as modern smartphones.

In Chapter 2 a number of goals have been formulated that the FlashUI-framework aspires to fulfil. In the following pages, FlashUI is evaluated in the light of the above requirements.

9.1 Platform Independence

The FUIML-language boasts a highly abstracted way of defining user-interfaces. For a lack of device-specific features that are available only on a certain platform, it supports virtually all hardware platforms, regardless of display-size and graphics-controller.

The same can be said about the bytecode that is generated by the FUIML-compiler introduced in Chapter 5. It forms an intermediate language that is more machine-readable than the original FUIML files. Indeed without modification, it can be used for the definition of user-interfaces on any platform that has access to a significant amount of storage-capacity - be it in the form of a built-in SD-card or on-chip memory.

What is needed in case of a change in platform is a re-implementation of the bytecode-interpreter, introduced in Chapter 6. By nature, the interpreter must make use of device-specific functions and hence cannot boast platform independence. However, instead of requiring a re-implementation of *multiple* GUIs, only one component must be re-implemented for the new platform which constitutes a major advantage over other designs such as ones that rely upon the use of proprietary tools such as 4D System's Visi, introduced in Section 3.3.9.

Apart from the above, it is also the software tools that have been implemented in the course of this project that ensure platform independence. Whereas most of the design-tools that were introduced in Chapter 3 exclusively target Microsoft Windows, the core-components of this project are designed to support all major operating systems that are currently in use. In addition, all tools consist of an application-specific core-library that can be used by a variety of front-ends if the requirement for a different user-interface arises. It is also for the XML-based approach for the design of the user-interface, that the GUI can be created on any platform without the need for proprietary software.

9.2 Reuse of Components

Code-reuse is strongly enforced by the framework. The existing widgets provide a common set of functionality that can leverage the implementation of multiple applications. For lack of interdependencies among components and for a lack of entanglement between a widget's visual properties and application-specific behaviour, FUIML boasts a strong encapsulation of the user-interface, its individual components and of the underlying control-flow. Instead of designing a framework that drives *one* particular application, it has been designed from ground-up to facilitate the creation of a *family* of applications that build upon a common foundation of functionality.

9.3 One Framework for all Applications

The same bytecode-interpreter that is used to render the FUIML source-code on the screen of the target-device can be used by all applications - extensions and upgrades that arise from the distinct needs of *one* application immediately become available for the benefit of *all* applications. As it is the case in 4D System's Visi, much of the interface can be created from pre-rendered graphics that are selectively displayed at the discretion of a *Trigger*. This flexible approach allows for the design of functionality via tools for editing images. Other controls such as the *Map* represent features that must be implemented in the on-chip firmware of the graphics-controller, where they are made available for use in other applications.

9.4 Efficient Design-Tools

FUIML features an abstracted, easy-to-use and efficient way of designing GUIs that is no more complex than designing websites - typically a graphics-designer's field of expertise. In the same way as writing HTML requires little knowledge about the nuts and bolts of modern browsers, FUIML user-interfaces can be created without either programming skills or familiarity with the target system.

Whereas in the 4DG-Workshop, all addresses on the SD-card must be managed by the programmer by hand, in the FUIML-language all images and other references to a particular location on the SD-card are dynamically inserted upon compilation of the user-interface. In addition the entire user-interface including all images and animations are automatically scaled to the appropriate size. Hence, by defining the position and size of individual components via *Symbols* and via the automated arrangement provided by the *Stackpanel*-control - both introduced in Chapter 4 - the entire user-interface can automatically be adapted to different screen-dimensions.

9.5 Modern Look-and-Feel

Most of the user-interface can be created from pre-rendered graphics that rely upon the superior processing-power of modern desktop-computers. Hence, for a large part the *look* is determined by the skill of the designer that is responsible for the user-interface and by the tools that are used to draw the graphics.

A modern *feel* is achieved via the use of animations that allow for a dynamic appearance of the GUI instead of static graphics. It is for the techniques that have been implemented as a measure to prevent unnecessary delays such as redrawing images that are already visible on screen, that the GUI boasts quick response-times upon user-interaction.

9.6 Efficient Use of Resources

The main limiting factor both in the platform at hand and in other platforms such as those that are supported by Amulett's GEM Studio is RAM and program-memory. FlashUI compensates for this limitation by using the comparatively massive amount of storage capacity that is available on the built-in SD-card of the device. Instead of storing the user-interface within the chip's program-memory, it is located on the SD-card - hence preserving space for additional fonts which in case of the GOLDELOX-chip must also be stored in program-memory.

Whereas in desktop-applications, recursive programming is often chosen over iterative implementations, it is unsuitable for the platform at hand. Since every function call causes an overhead - at an absolute minimum, the return-address must be stored - the depth of the call-stack must be kept at a minimum. For this reason, the interpreter that has been introduced in Chapter 6 boasts an implementation that refrains from using unnecessary function calls.

Algorithms such as Binary Search, which in case of the *Map* is used to locate a specific image in a large list, and Run-Length-Encoding which can be used to transmit large amounts of data via the serial-port, offer a significant benefit over more basic approaches.

Bibliography

4D LABS [2011]. *GOLDELOX-GFX2 Embedded 4DGL Graphics Controller Advance Information.* http://www.4dsystems.com.au/downloads/Semiconductors/GOLDELOX-GFX2/Docs/GOLDELOX-GFX2-DS-rev3.0.pdf. [Online, accessed 12-August-2012].

4D LABS [2012]. *GOLDELOX-GFX2 Internal 4DGL Functions.* http://www.4dsystems.com.au/downloads/Semiconductors/GOLDELOX-GFX2/Docs/GOLDELOX-GFX2-4DGL-Internal-Functions-rev5.pdf. [Online, accessed 12-August-2012].

Amulet Technologies, LLC [2010]. *COMPLETE HMI SYSTEM SOLUTIONS Taking you beyond the GUI software challenge.* http://www.amulettechnologies.com/images/Downloads/BrochureWeb.pdf. [Online, accessed 16-August-2012].

Bower, Andy and Blair McGlashan [2000]. *Twisting the triad: The evolution of the dolphin smalltalk mvp application framework. ESUG tutorial.*

Burbeck, Steve [1987]. *How to use Model-View-Controller (MVC).* http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html.

Burns, Charles N. [2009]. *Building Qt Static (and Dynamic) and Making it Small with GCC, Microsoft Visual Studio, and the Intel Compiler.* http://www.formortals.com/build-qt-static-small-microsoft-intel-gcc-compiler/. [Online, accessed 31-July-2012].

Coopersmith, Alan [2012]. *About the X.Org Foundation.* http://www.x.org/wiki/XorgFoundation. [Online, accessed 31-July-2012].

Crank Software Inc. [2012a]. *Crank Storyboard Suite Product Overview.* http://www.cranksoftware.com/_images/pdfs/Product_Overview.pdf. [Online, accessed 16-August-2012].

Crank Software Inc. [2012b]. *Crank Storyboard Suite Technical Datasheet.* http://www.cranksoftware.com/_images/pdfs/Product_Overview_Technical.pdf. [Online, accessed 16-August-2012].

Dijkstra, Edsger Wybe [1982]. *On the role of scientific thought (EWD447).* In *Selected Writings on Computing: A Personal Perspective*, pages 60–66.

Ferguson, Arron [2009]. *Creating a declarative XML UI language Build a UI and the accompanying framework in the Java language. IBM developerWorks.* http://www.ibm.com/developerworks/web/library/x-decxmlui/index.html.

Fowler, Martin [2004]. *Presentation Model.* http://martinfowler.com/eaaDev/PresentationModel.html. [Online, accessed 8-July-2012].

Fowler, Martin [2006]. *GUI Architectures.* http://martinfowler.com/eaaDev/uiArchs.html. [Online, accessed 5-August-2012].

Haerr, Gregory [2010]. *Nano-X Frequently Asked Questions*. http://www.microwindows.org/faq.html. [Online, accessed 31-July-2012].

Hanus, Michael and Christof Kluß [2009]. *Declarative Programming of User Interfaces*. In Gill, Andy and Terrance Swift (Editors), *PADL*, *Lecture Notes in Computer Science*, volume 5418, pages 16–30. Springer. ISBN 978-3-540-92994-9. http://www.informatik.uni-kiel.de/~mh/publications/papers/PADL09.pdf.

Huber, Thomas Claudius [2010]. *Windows Presentation Fondation Das umfassende Handbuch*. 2nd Edition. Galileo Computing. [German].

Krasner, G. and S. Pope [1988]. *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*. *Journal of Object Oriented Programming*, 1(3), pages 26–49. http://citeseer.ist.psu.edu/krasner88description.html.

McIlroy, Doug [1968]. *Mass-Produced Software Components*. In Naur, P. and B. Randell (Editors), *Proceedings of NATO Software Engineering Conference*, pages 138–155. Garmisch, Germany.

Mentor Graphics Corporation [2011]. *Mentor Embedded Inflexion UI Solutions DATASHEET*. http://www.mentor.com/embedded-software/events/esc/upload/inflexion-ui-esc-ds.pdf. [Online, accessed 16-August-2012].

Microsoft Patterns & Practices Team [2009]. *Microsoft® Application Architecture Guide, 2nd Edition*. 2 Edition. Microsoft Press. http://msdn.microsoft.com/en-us/library/ff650706.aspx.

MicroXwin [2011]. http://www.microxwin.com/. [Online, accessed 16-August-2012].

Nokia Corporation [2012]. *Qt Licensing*. http://qt.nokia.com/products/licensing. [Online, accessed 31-July-2012].

Nye, Adrian [1994]. *Xlib Programming Manual for Version 11 of the X Window System*. 3 Edition. O'Reilly & Associates, Inc.

Parr, Terence John [2004]. *Enforcing Strict Model-View Separation in Template Engines*. In Feldman, Stuart I., Mike Uretsky, Marc Najork, and Craig E. Wills (Editors), *Proceedings of the Thirteenth International World Wide Web Conference*, pages 224–233. ACM Press, New York, NY.

Peersman, Hans, Jeroen van der Velden, Nikos Mitilinos, and Renato Hijlgaard [2011]. *X Window System*. http://computingscience.nl/wiki/pub/Swa/CourseLiterature/arch-D.pdf. [Online, accessed 12-August-2012].

Potel, Mike [1996]. *MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java*. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

Scheifler, Robert W. and Jim Gettys [1986]. *The X window system*. *ACM Trans. Graph.*, 5(2), pages 79–109. http://doi.acm.org/10.1145/22949.24053.

Schwaiger, Mario [2012a]. *ADELE4home mit Sicherheit selbstständig*. http://www.spintower.biz/img/Flyer_ADELE.pdf. [Online, accessed 12-August-2012, German].

Schwaiger, Mario [2012b]. *LDT LIVE DATA TRACKER*. http://www.spintower.biz/img/Flyer_LDT.pdf. [Online, accessed 12-August-2012, German].

Smith, Josh [2009]. *WPF Apps With The Model-View-ViewModel Design Pattern*. *MSDN Magazine*. http://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

Swell Software, Inc. [2007]. *C/PEG Product Brief*. http://www.swellsoftware.com/pdfs/cpegbrief.pdf. [Online, accessed 16-August-2012].

The GTK+ Team [a]. *Language Bindings*. http://www.gtk.org/language-bindings.php. [Online, accessed 16-August-2012].

The GTK+ Team [b]. *What is GTK+, and how can I use it?* http://www.gtk.org/. [Online, accessed 16-August-2012].

Tristan Van Berkom [2012]. *Glade User Interface Designer Reference Manual*. http://developer.gnome.org/gladeui/3.6/. [Online, accessed 16-August-2012].

Wolf, Jürgen [2007]. *Qt 4 GUI-Entwicklung mit C++*. Galileo Computing. [German].

Glossary

.NET Framework is a framework for software-development introduced by Microsoft. Via a technology called the Common Language Infrastructure(CLI), multiple languages can be used to develop platform independent applications.

ANT+ a technology for wireless transmission of sensor data.

baud symbols per second.

binary-search is an algorithm that is used to locate specific entries within a sorted list.

bytecode is an intermediate code that is neither readable for humans nor directly executable by a processor. It requires a parser and interpreter to execute.

C# is a programming language introduced by Microsoft that is part of the .NET family of languages.

COM-port see *serial-port*.

control see *widget* plural.

display driver a driver that defines interaction between graphics-controller and display on embedded systems.

dynamic memory-management is technique that allows for the allocation and de-allocation of memory at run-time .

Eclipse is an IDE that boasts support for a variety of programming languages. For its extensible nature, numerous third-party modules are available that offer additional functionality.

firmware a software that is installed in the program-memory of an embedded system. It provides base-functionality an cannot be easily replaced or modified be the user.

footprint in the world of embedded systems, the term *footprint* refers to the total amount of resources needed by a certain software.

front-end a subsystem of a software that is responsible for interacting with users and/or data gathering.

GNOME desktop environment is a desktop-environment for UNIX-like systems. It usually builds upon the X-Window System.

graphics memory is a kind of memory that on embedded systems contains the visible content of the screen pixel-by-pixel..

graphics-controller a hardware-component that manages interaction between software and display-unit on an embedded system.

hardware-acceleration in the context of graphics-processing, hardware-acceleration means that certain operations are performed by dedicated hardware which often results in improved performance.

high-level the term *high-level* refers to code that is strongly abstracted from the platform(s) it targets. Through this abstraction, an often more convenient way of programming is achieved and platform-independence is facilitated.

Interface Builder is a WYSIWIG design-tool for creating user-interfaces introduced by Apple.

interrupt interrupts allow for the definition of signals that cause processors to pause normal operation and instead process other code.

kernel the core of an operating system. It governs most interaction between applications and hardware.

low-level in computer-science, the term *low-level* refers to code that directly accesses the resources available on a specific platform without the use of abstraction-layers.

mainframe-computer in the days before the advent personal computers, a powerful computer used for complex data-processing was called a mainframe-computer.

middleware is a software that offers a platform-independent interface that can be used by programs to access the resources of variety of operating systems in a common way.

Mono is an open-source implementation of Microsoft's C# programming language.

multi-threading is a technique that allows for multiple threads of execution on a single processor .

Observer Pattern is a design pattern that is often used to synchronize the user-interface with the model it represents. Observers to an observable are notified upon changes in the observable's properties.

OpenGL/ES is a cross-platform API for 2D and 3D graphics.

program memory is a part of memory that is read-only for applications. It stores the executable code that describes an application.

Property-Element-Syntax is an alternate way of assigning properties to an XML-node. The value of the node is defined as an XML-subtree.

raw-write-mode a process in which a storage medium is written to without the use of a file-system but via addresses or via physical characteristics of the medium.

real-time operating system is an operating system that ensures that a certain task requires a fixed amount of time.

remote-procedure-call is a function call from one software to another that does not reside in the same address space and/or machine as the caller.

RGB565 is a color format that allocates five bits to the color red, six to the color green and five to blue .

run-length-encoding is a simple technique for lossless data-compression. If a sequence of the same byte is found, the byte itself and the number of its occurrence is encoded instead of storing the entire sequence.

runtime-library is a software-library that that is not compiled into the applications that make use of it. Instead it is shared by a number of applications and is located at run-time.

scripting-language is a programming language that is not compiled ahead of execution but is instead interpreted at run-time.

SD-card a solid-state storage medium.

sector a subdivision of the space available on a storage device.

serial-port a communications-interface that supports the serial transmission of data - one symbol at a time.

shared library a single software that is compiled in a way that allows multiple other software to make use of its functions.

Silverlight is a framework for creating browser-based applications that was introduced by Microsoft. Like WPF, it relies on XAML for the definition of individual user-interfaces.

stack is a part of RAM that is among other things used to store the arguments and return-addresses of function-calls .

TrueType is a vector-based format for the description of fonts.

Visual Studio is an integrated development environment (IDE) marketed by Microsoft that supports a variety of languages and technologies. Among them those of the .NET framework.

widget an individual component of a user-interface.

window manager is a program that offers an infrastructure for GUIs. It abstracts the process of drawing content on the screen and manages the creation and display of individual windows.

Windows CE is an operating system developed by Microsoft that targets embedded systems.

Windows Forms is a GUI-framework that was introduced by Microsoft with the release of the .NET Framework.

WPF Windows Presentation Foundation, a GUI-framework that is part of Microsoft's .NET framework.

XAML Extensible Application Markup Language, an XML-based markup-language introduced by Microsoft that is predominantly used for GUI-design.

Acronyms

API Application Programming Interface.

ASCII American Standard Code for Information Interchange.

CLI Common Language Infrastructure.

CLR Common Language Runtime.

CPU Central Processing Unit.

FUIML FlashUI Markup Language.

GNU a recursive acronym for GNU's Not Unix.

GPS Global Positioning System.

GRAM Graphics Random Access Memory.

GSM Global System for Mobile Communications.

GTK+ GIMP Toolkit.

GUI Graphic User Interface.

HTML HyperText Markup Language.

IDE Integrated Development Environment.

LGPL Lesser GNU Public License.

MVC Model - View - Controller.

MVP Model - View - Presenter.

MVVM Model - View - ViewModel.

PC Personal Computer.

QML Qt Meta Language.

RAD Rapid Application Development.

RAM Random Access Memory.

RF	Radio Frequency.
SPI	Serial Peripheral Interface.
TFT	Thin Film Transistor.
UI	User Interface.
URI	Universal Resource Identifier.
USB	Universal Serial Bus.
WYSIWYG	What You See Is What You Get.
XML	eXtensible Markup Language.

List of Figures

List of Tables