A PORTABLE JAVA API INTERFACE TO SIMPLIFY USER ACCESS TO DIGITAL CAMERAS

Peter Corcoran, Ferenc Papai, Arpad Zoldi and Joe Desbonnet
Dept. of Electronic Engineering, University College, Galway, Ireland

Abstract - **The digital camera is one of the main successes of recent years in consumer electronics. However a typical digital camera remains tied to proprietary software on a personal computer. In this paper we describe the design and implementation of a portable Java API to simplify end user access to digital cameras and to provide interconnectivity with a new generation of intelligent home applicances.**

1. Introduction

The digital camera market has taken off during the past 18 months. Indeed there has been such explosive growth that there are now more than 20 manufacturers of competing products in this particular market niche. However, although many digital cameras are functionally similar each camera is accessed via its own proprietary software and protocols on the users personal computer.

In this paper we consider some of the issues involved in the design of a generic API for digital cameras which is independent of both the underlying operating-system and of the digital camera. Furthermore, as a proof of concept we have implemented a basic software application which is based on our prototype of a generic camera API. Our experiences in developing and implementing this API are also described.

We place particular emphasis on modularizing components of the API. Thus the core software building-blocks, or "personality" which control a camera, and provide access to its inbuilt functionality and stored data may be loaded and unloaded dynamically as required. The end goal is to allow camera "personalities" to be loaded remotely over a wide area network such as the Internet.

Finally, and looking towards the emerging new generation of home-Internet appliances, we consider how such appliances would be capable of providing access to and functional integration with such a generic camera API. This might be achieved using the Internet as a communications backbone for updating and adding new functional modules to the core system software.

2. Software Architecture & Overview

As a practical implementation of our ideas we have studied 5 of the most popular low end digital still cameras and derived from these a consensus of the main features, services and user-interface components required in a generic digital camera API. We have then implemented this API using a software architecture as illustrated in **Fig 1** below. Java is the language of choice for this implementation as it supports many of our core design goal, particularly portability and platform independence and mechanisms to support abstract interfaces and modularization of the core software modules.

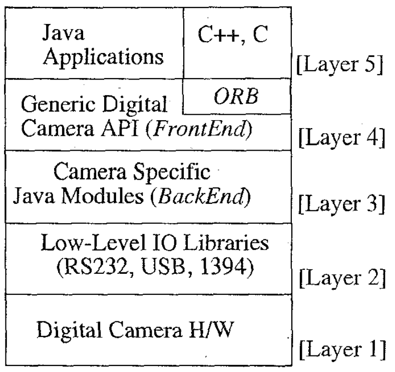
Java Applications
C++, C
[Layer 5]
Generic Digital Camera API (*FrontEnd*)
ORB
[Layer 4]
Camera Specific Java Modules (*BackEnd*)
[Layer 3]
Low-Level IO Libraries (RS232, USB, 1394)
[Layer 2]
Digital Camera H/W
[Layer 1]

Fig 1: Overview of the API Software Architecture

As can be seen from **Fig 1** there are several layers to our architecture. A key goal of this project was to hide the complexity of the lower layers from an application programmer. Thus layers 1-3 are, essentially, invisible to programmer who only requires a knowledge of layer 4 in order to integrate digital camera functionality into a software application. We will now describe each of these different layers in more detail.

2.1 The Low-Level Communications Layers

The two lower layers represent the physical interface between a camera and the hardware unit on which the camera software will run. In practice this requires that Java native methods be implemented, typically in the C language, to allow communications between the camera and the communication hardware. In today's cameras this is typically an RS-232 port, but it might equally well be USB, 1394 or a PCMCIA card interface. If a native method is available to implement low-level communications with a camera then it is practical to link this to the higher layers of this software architecture using a dynamically loadable

Manuscript received June 17, 1998

system library. In the MS-Windows programming environment, for example, this appears as a DLL file.

We note that the recently released Java Communications API addresses these issues, making access to a range of communications media more uniform across all operating systems which support Java[ref 1].

2.2 Camera Specific Modules - *Back-End*

The third layer implements camera specific software in Java to handle different communication protocols, camera specific instruction sets and messages and to manage access to and downloading of pictures stored in a camera. We will refer to this layer as the software *back-end.*

The Java language supports the powerful concept of an abstract software interface. This allows a programmer to define abstract data structures and methods which can be implemented by another software module. Note that the interface-module does not need to know anything about the software module which will implement the interface and about the details of the implementation. This, in turn, allows modules to be loaded dynamically and accessed only through the specific interface after the main core of a software application has been started. This is, essentially how the JCam architecture works, in that it is the *back-end* modules which implement the methods defined by the actual camera API, or *front-end.*

2.3 The *Front-End* Abstract-Interface

This fourth layer, the digital camera API, makes a broad range of cameras accessible through a common programming interface. The design principle used here is to abstract the most common camera functionality, implementing them as an interface. A call to a specific camera function happens through this interface which maps the call to the appropriate, camera specific module.

Camera specific functions, properties and parameters can also be accessed. They are implemented in a list of self describing name/value pairs. For example if a camera has a flash than it will have a flash/value pair, where the value is an integer describing the state of the flash setting with a potential range of values from 0 to 255. With such self-describing options the backend does not need to be concerned with presentation issues. It simply provides a list of options that describe additional controls and functions available in the camera device. Similarly, there are benefits to the frontend as it does not need to understand the meaning of each option. It simply provides a means to present and alter the options defined by the backend. These self-describing pairs are grouped in a list, called *properties-list.*

2.4 The Applications Layer

This will be implemented by a programmer using our API. It could be a graphical design and drawing package, an engineering CAD application or even a plug-in for a web browser. The key point is that a common set of API calls allow the programmer to access digital cameras from a wide range of manufacturers. The final application is thus camera neutral, offering the end-user a greater choice in the cameras they use.

3. The Digital Camera API

JCam is a Java package of a java application programming interface (API) that provides standardized access to any digital still camera and can, potentially, be implemented on any operating system (OS). This API is implemented as a Java interface allowing a programmer to write only one driver module for each digital camera instead of one driver for each OS. At the application level, access to each camera is through an identical set of methods. The reduction in the number of required driver modules and the provision of a common set of class methods which are shared across all digital still cameras, provides a significant saving in development time.

The digital camera API intends to provide application software with a consistent abstraction of all digital cameras and still provides access to any specific capabilities which differentiate a particular manufacturers camera. This is achieved through the use of a property list which provides a means for the back-end driver modules to export additional functionality via the main JCam API interface. These are exported in terms of properties and parameter lists that make it easy to give a user-application access to all of a cameras functionality. This, in turn, allows end-users to access a wide variety of cameras from a common user interface.

3.1 The Main Classes

There are four main classes in the JCam package. These are outlined in **Table 1** below. Of these the core class is the *DigitalCamera* class which defines the main interface to a digital camera.

Class:	Description:
ModuleLoader	*provides access to camera and detect modules remotely or locally*
DigitalCamera	*generic digital camera interface*
DetectCamera	*interface for automatic recognition of digital cameras*
Picture	*provides access to all information regarding a taken picture*

Table 1: *The Main JCam Classes*

The *DetectCamera* and the *ModuleLoader* classes are important in the context of our architecture which supports the automatic detection and loading of back-end modules. These classes are discussed later in section 4. Finally, the *Picture* class offers information about a picture taken with a digital camera and also offers functions for the manipulation of this digital picture.

3.2 The Main Methods of the *DigitalCamera* Class

Applications communicate with the camera through a well defined interface, the *DigitalCamera* class. This interface incorporates the basic and the most common operations which can be performed with all digital cameras.

The camera driver modules implement this interface, each of them in its own way, based on the camera specific communication protocol.

Method:	Description:
open()	*opens the connection*
close()	*closes the connection*
init()	*resets the camera to a known state*
getType()	*returns the camera's manufacturer and model No.*
getNumberOfPictures()	*returns the number of pictures available in the camera*
getPicture(n)	*downloads picture number n*
getThumbnail(n)	*downloads thumbnail number n*
deletePicture(n)	*delete picture number n*
deleteAll( )	*delete all pictures*
isProtectedPicture(n)	*checks if the picture is protected*
protectPicture(n)	*protect picture number n*
takePicture()	*takes a new picture*

Table 2: *Main Methods of the DigitalCamera Interface*

3.3 The *PacketListener* Methods

Applications can be notified about the download status of images, using the Java event delegation model. This requires that notification of received data packets from the camera is passed on to the application layer. This is realized by generating appropriate events. These events, with incorporated status information are accessed by a *PacketListener* object. Status information includes the amount of data downloaded and the amount remaining.

The following methods are used to add and remove event listener objects to/from a module.

Method:	Description:
addPacketListener(listener)	*adds a listener object to the module*
removePacketListener(listener)	*removes the listener from the module*

Table 3: *PacketListener Methods for Picture Download*

3.4 The *Picture* Class

The pictures obtained from a camera by a *back-end* module are encapsulated, with some additional information, in *Picture* objects. The incorporated data provides information about each picture, i.e. image size, shutter speed, aperture, flash mode, quality mode, date of the picture, *etc.* in the form of a properties-list.

Details of the main methods implemented by the *Picture* class are given below in **Table 4.** Note that at the present stage we have only implemented JPEG and PNG (Portable Network Graphics) support. Additional graphics formats will be supported as, and when necessary.

Method:	Description:
getData()	*gets the picture data in the format sent by the camera*
getImage()	*gets an image which can be displayed in the current Java environment*
getImageAsJpeg()	*returns an image in JPEG format*
getImageAsPNG()	*returns an image in PNG format*
isThumbnail()	*checks if this picture is a thumbnail*
getFormat()	*returns the picture's format*
getProperties()	*returns the picture's properties list*

Table 4: *Methods of the Picture Class*

3.5 Camera Specific Features & Functionality

The digital camera API also provides access to camera specific features which can vary from camera to camera. These features (i.e. flash mode, resolution mode, date, battery status, lens mode) are grouped in a properties-list. The properties-lists are used in the API to allow access to specific values, functions and parameters in a camera and to offer more detailed information about a picture stored in the camera.

Method:	Description:
getProperties()	*gets the camera's properties list*
setProperties(new)	*sets the camera's properties*

Table 5: *The Properties Methods to provide access to non-standard features in different cameras.*

3.6 The Back-End: Camera Modules

The main function of the back-end is to implement the methods defined in the *DigitalCamera* interface for different cameras using the corresponding communication protocols. These modules have to deal with camera specific properties, proprietary image format and event generation.

From each camera's protocol we can select a number of essential commands which can be found in the general camera interface. However, to completely explore the features of a particular camera, in addition to the basic API defined above, the specific camera modules will also support additional command-sets and functionality which is not part of the basic API. These functions are encapsulated in a properties-list. The module for a particular camera will build, on request, this list by retrieving all the available additional information from the camera which cannot be accessed directly via the digital camera API. To set these camera specific features the application programmer builds a properties-list and passes it to the camera module which interprets and sets only the properties available on the connected camera. This mechanism allows the application programmer to set only the desired properties.

Some digital cameras may use a proprietary image format to store the taken pictures. In this case, the images must be converted to a standard, well known and common image format. The modules, where it is appropriate, include conversion code to well known and widely used image formats like JPEG, TIFF or PNG.

The back-end is responsible with the generation and the delivery of events. The *getPicture* and *getThumbnail* method implementations of the different camera modules are sources of events. These methods create events with embedded information about the download status and deliver them to the subscribed listener objects. A picture is sent by the camera in several packets, each of them accompanied by a checksum. Listener objects will be notified every time by an event if a correct packet or an eronate packet was received.

4. Detailed description of the architecture

There are three key components in the JCam API: general camera interface (DigitalCamera), camera detection interface (DetectCamera) and finally the module-loader (ModuleLoader). These three components provide all the interaction between the JCam API and a java application. The general camera interface provides uniform access to any digital camera, see section 3.

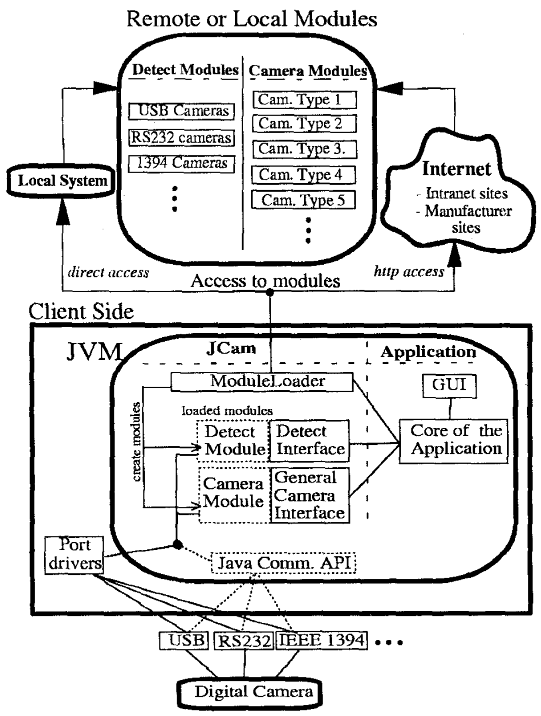

Fig 2: Detailed description of the JCam architecture

The camera detection interface provides uniform access to software modules which perform automatic recognition of the connected digital camera. There is one module for each type of I/O port which implements this interface. Each of these modules provides detection for a group of digital cameras which have the same communication port. These modules contain a minimal part of the protocol for each camera from the group, enough to recognize the cameras.

The link between the interfaces and the modules is provided by the module-loader, see **Fig 2** . Basically the module-loader creates the desired modules for the application software. The module-loader first loads the detect module for the used communication port. After that, using the recognized camera's identification number it loads the corresponding camera module. Now the camera can be accessed through this camera specific module.

The location of the camera and detection modules can vary, the modules can be on a remote site on the Internet or on the local system as well. As you can see in the given code example (section 5.1), there is no difference between working locally or remotely, except that the remote operation requires an URL (Universal Resource Locator) to be specified. The size of the modules is very small (25-30 KBytes), the download of one module from a remote site would cause only a short transfer delay.

5. An Application using the API

As an initial proof-of-concept we have implemented a prototype application based on a standard desktop PC, to demonstrate how the API allows a number of different digital cameras, each from a different manufacturer, to be accessed from a common user-interface.

In each case the core elements of the camera personality are loaded separately, as required, from the local hard-disk on the PC. However the software has been designed and may be readily modified to support the loading of these modules over a network. Thus the user-interface and application modules could also be loaded remotely in the same way. This is described in greater detail in section 4.

In a practical consumer application the PC might equally be an intelligent screen phone, set-top box, or other home-Internet appliance and the modules would be downloaded from a remote URL on a TCP/IP network. Thus an end user, by connecting his digital camera to, say, a screen phone could automatically load from the Internet -

(i) a camera module to access the camera,

(ii) a user interface to download, edit and view pictures, and

(iii) additional application modules to archive, manage and index pictures either locally, or on a remote site and to transmit selected pictures via email.

What is important is that all of these functional modules are only loaded as required by the end-user and the exact modules loaded are determined from the behaviour of the camera connected to the serial port.

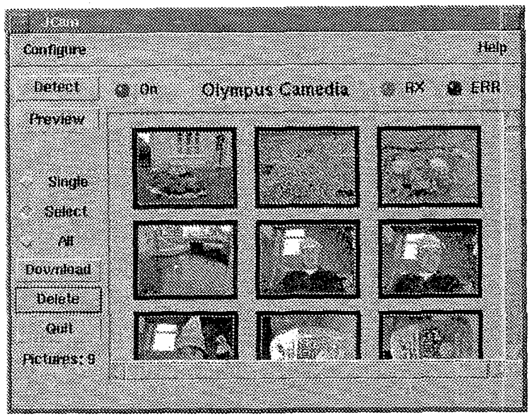

Fig 3: Example Application Based on the JCam API.

5.1 Some Practical Code Examples

The following Java code fragment illustrates the usage of the digital camera API with the camera modules located over the network and in the local file system. The only difference is resumed on the initialization of the ModuleLoader object.

First, the ModuleLoader object is created in two ways, one for remote access and one for local access to the camera modules:

```
ModuleLoader loader;
loader = new ModuleLoader("www.jcam.com");
```

or,

```
ModuleLoader loader;
loader = new ModuleLoader();
```

The next block of code gets the detect code for the serial port ("COM1") and detects the connected camera. Based on a camera identification number returned by the detect module it loads the camera specific module which can work with the connected camera:

```
DetectCamera detectCam;
detectCam = loader.getDetectModule("COM1");
int cameraID = detectCam.detect();
DigitalCamera camera;
camera = loader.getCameraModule(cameraID);
```

The following lines show some basic camera operations. First, the camera is opened on the serial port "COM1". It is asked its make/model and the number of taken pictures. It then downloads picture number 2. Finally the connection is closed.

```
camera.open( "COM1" );
String type = camera.getModel();
int nrPics = camera.getNumberOfPictures();
Picture picture = camera.getPicture( 2 );
camera.close()
```

The following lines save the downloaded image into a file. Note: if the camera has proprietary image format, the Picture class with the help of the camera module performs all the necessary conversions.

```
FileOutputStream out;
out = newFileOutputStream( "pic2.jpg" );
out.write( picture.getPictureAsJPeg() );
out.close();
```

Accessing a property of a camera is presented below. The code manipulates the camera's internal date if it has. First, the date is read and after is set. The 'camera.date' and all the other keywords, representing properties, are defined by the API.

```
Properties list = camera.getProperties();
String date = list.getProperty( "camera.date");
if( date != null ){
   // date available in 'date' variable in a
   // format like 10.JAN.1998
   list = new Properties();
   list.put( "camera.date", "2.JUL.1998" );
   camera.setPropeties( list );
} else {
  // it's not possible to acces the camera's internal
  // data or there is none
}
```

6. Future Enhancements

The Digital Camera classes provide application software with a consistant abstraction of all digital cameras. Application software can use a camera without specific knowledge of what resolutions, encoding standards, shutter speeds, and focusing mechanisms are available and without having to know the details of the communication protocol.

In order to allow existing applications written in languages such as C and C++ to use this proposed software API infrastructure, we suggest that a CORBA ORB should be incorporated into the higher API layers. We also envisage that the inclusion of an ORB will simplify the development of network aware applications and distributed applications for home and business use.

The camera specific layer implements all the necessary code to communicate with a specific digital still camera. This layer is designed so as to allow new camera personalities to be added.

The I/O layer is a platform specific library to enable communication via RS232 (in our prototype), PCMCIA and ultimately via Universal Serial Bus or 1394 serial connections when suitable cameras become available.

7. Conclusions

The Java API interface was designed to provide a generic interface to communicate with a wide range of digital still cameras from a common user interface and common software API.

As a proof-of-concept we have developed software that demonstrates all aspects of the API architecture discussed in this paper and which works with several of the commonest digital still cameras available from leading manufacturers today

References

[1] http://www.javasoft.com.

Biographies

Peter Corcoran received the BAI (Electronic Engineering) and BA (Maths) degrees from Trinity College Dublin in 1984. He continued his studies at TCD and was awarded a Ph.D. in Electronic Engineering in 1987 for research work in the field of Dielectric Liquids. In 1986 he was appointed to a lectureship in University College Galway. In 1994 he founded POD Concepts Ltd, an R&D company developing ATE systems for the International Textile Industry. In 1995 POD Concepts became a major partner in the first Sino-Irish Joint Venture, based in Beijing, PRC. He is currently teaching on a part-time basis at University College Galway, and is a visiting Professor in Telecommunications at the Technical University of Cluj-Napoca, Romania. His research interests include embedded computing applications, automated test equipment, instrumentation and telecommunications technologies. He is a member of the IEEE.

Ferenc Papai received his B.S. degree in Mathematics and Informatics from "Petru Maior" University Tirgu-Mures, Romania, in 1997. Currently he is completing the M.S. degree in Applied Science at University College Galway, Ireland.

Arpad Zoldi received his B.S. degree in Mathematics and Informatics from "Petru Maior" University Tirgu-Mures, Romania, in 1997. Currently he is completing the M.S. degree in Applied Science at University College Galway, Ireland. His major interests include OOP and Java programming, Internet technologies, networking and operating systems.

Joe Desbonnet received his B. Sc. degree in Applied Physics and Electronics from University College Galway in 1991. He is presently working as a Research Associate at University College Galway and is a director of a Galway based Internet services and software development company. His interests include Java programming language, the Linux operating system and web computing.