DIGITAL AUDIO DEVICE CREATION BY THE USE OF A DOMAIN SPECIFIC LANGUAGE AND A HARDWARE ABSTRACTION LAYER

Stefan Jaritz

Dept. of Electrical Engineering and Information Technology
Ernst-Abbe-University of Applied Sciences
Jena, Germany
stefan.jaritz@fh-jena.de

ABSTRACT

The present paper deals with a framework destined to manage different aspects of the creation of digital audio devices. By means of a domain-specific language modelling aspects like signal processing and user interaction are implemented. The problem of different hardware interfaces is resolved by the definition of a hardware abstraction layer. This layer provides different types of variables and functions. A compiler translates the model referring the functions and variables defined at the hardware abstraction layer. Furthermore, the compiler is able to split the model into different parts that can be run on different hardware components. The communication needed to manage the distributed model is defined and formalized by the framework. A simple example is presented to help explain the framework's parts, as are the compiler and the execution unit.

1. INTRODUCTION

In cooperation with the local industry the Department of Electrical Engineering and Information Technology at the EAFH Jena carries out different research projects. Sometimes, developing digital audio devices is a key part of this partnership. However, developing audio devices in a short period of time has been proved challenging especially with regard to students final papers.
The main problem of device development is that it covers a large subset of aspects like hardware construction, software creation, communication, etc., and each one of these aspects may turn out rather complex. To overcome that kind of complexity different methods have been developed. The crucial point with them is that most of them focus only on one particular aspect of the device development. Thus, it is the intention of the present paper to introduce a method that connects the different approaches helping to structure, simplify and speed-up digital audio device creation.

2. ASPECTS OF DIGITAL AUDIO DEVICE CREATION

Lets consider a common audio device. An audio power amplifier can be used to demonstrate how chosen aspects of digital audio device creation work. Such device contains a lot of different sound processing elements like crossovers, limiters, equalizers, etc.. For further simplification only one aspect of the sound processing is picked out. This is done by an equalizer which itself may be considered as a cascade of simple biquad filters. The filter coefficients which are calculated from the filter parameters (described by Zölzer [1]). These are adjusted by the user through a panel containing buttons, potentiometers, LEDs and displays. All functional elements are connected to a low-cost micro-controller(μC). As soon as the user adjusts the buttons the filter coefficients at DSP are updated. The digital filtering is carried out by a DSP. In most cases the audio power amplifier is part of a larger system. In these cases a remote control device or software is needed. At the given example the audio power amplifier is controlled through a software running on a PC.
To face the challenge of device creation different views are defined. Each view can be handled with existing models and methods:

- The problem of digital signal processing may be abstracted by a functional model. A common approach is to use Matlab and the integrated toolbox Simulink. The benefit of this approach is that the model is ready to be tested and compiled to C code. In some cases this code may be deployed directly into the DSP or the micro-controller.
- Taking user interaction with the device into account, a UML use case diagram may be used to specify the user interaction. With some tools it is even possible to transfer this diagram to C++ code.
- Hardware and software connection may be defined in a specification document. This may lead to a so-called "hardware abstraction layer" that functions as a software interface between hardware and software components.
- Data exchange between hardware and software components may be described with a UML communication diagram. Furthermore, access to the communication media hardware may be specified. A common approach is to abstract the access by means of a software driver.

Dividing the problem of device creation into sub-aspects thus provokes the idea that it should be possible to carry out the different tasks of development in parallel, even if the sub-aspects are linked one to another directly and/or indirectly. As example for the connection of aspects a closer look on the modelling of signal processing is undertaken. The user interface may be developed in parallel, and the data transfer process from the user interface to the signal processing unit connects these two aspects. However, the developer should take into account that the signal processing model includes some entry points for external data. Also, the real time processing ability ought to be adjusted to handle the data transfer. By defining entry points into the model, this itself becomes hardware dependant. As a result, the interface between software and hardware should be taken into consideration as well. The aim of the present research is thus to create a platform that handles the different problem views dependencies. In the following, ideas and implementation will be discussed, referring to the example mentioned in the beginning from time to time.

3. CREATION OF A PLATFORM TO HANDLE DEPENDENCIES BETWEEN DEVELOPMENT METHODS

As mentioned in section 2 modelling digital audio devices implies different technical aspects. These technical aspects scale from little problems like realizing a simple filter till the interconnect of many different devices like a sound system for a festival. Figure 1 shows the correlation of the aspects and gives a first impression of the platform to be invented. The left side represents the digital audio device whose creation process may be classified into three categories.

The "functionality" category approaches device creation from a

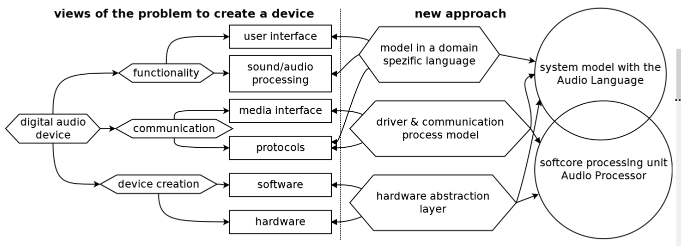
views of the problem to create a device
new approach
user interface
sound/audio processing
functionality
media interface
digital audio device
communication
protocols
device creation
software
hardware
model in a domain spezific language
driver & communication process model
hardware abstraction layer
system model with the Audio Language
softcore processing unit Audio Processor

Figure 1: *Approach to handle different aspects*

functional behaviour perspective. Referring to the example given in section 2, this involves the filtering of the input signals("sound/ audio processing") - a process which is controlled by the "user interface".

The "communication" category focuses on how data are passed inside of the devices. In the given example, the user interface passes any new coefficients of the equalizer to the DSP. The communication may be seen from a "protocol" side that focuses on the timing of the data transfer, but also includes "media interfaces" like I/O pins, CPU registers or software drivers.

The focus of the "device creation" category lies on the interaction between "software" and "hardware" elements. The several tasks of a device are dedicated to different hardware components. In many cases, a DSP is used for signal processing. If the computing power of this DSP is inadequately or it not enough I/O pins are available a μC might be the best option for the user interface.

The new approach is shown on the right side of figure 1. The division given on the left side is mixed up into several sub-categories of the new method. These categories are handled by a framework. The description of the devices behaviour is done in a textual form. The syntax necessary for that is provided by the so-called "Audio Language". This language contains common structures and keywords that are used for digital audio engineering and thus can be considered as a "domain-specific language".

All aspects of communication are abstracted into "drivers" and "communication processes". Communication data frames and the processes using them are described by means of a simple domain-specific language. The drivers associate the communication frames with the communication hardware like UARTs, Ethernet, etc.

Any connection between hardware and software elements is formalized by a "hardware abstraction layer", containing common functions and data types used to create digital audio devices. Moreover, a skeleton of the digital audio system is created. In the following this skeleton will be filled with implementations of the hardware abstraction layer elements as well as with some common code, including interpreters, message handling state machines, memory management units, etc. Together, all these elements are called "Audio Processor" which is in fact a soft CPU core. "Audio Processor" and audio device model, described by means of the "Audio Language" are connected by a compiler. This compiler translates the model into hardware abstraction layer functions. In case the model is split into pieces, these pieces will be executed on different "Audio Processors", being connected with each other and communicating via the defined communication processes.

4. ASPECTS OF THE APPROACH

The following section takes the example from section 2 and couple it with the new approach explained in section 3. This is shown in figure 2. The figure represents different views of device develop-

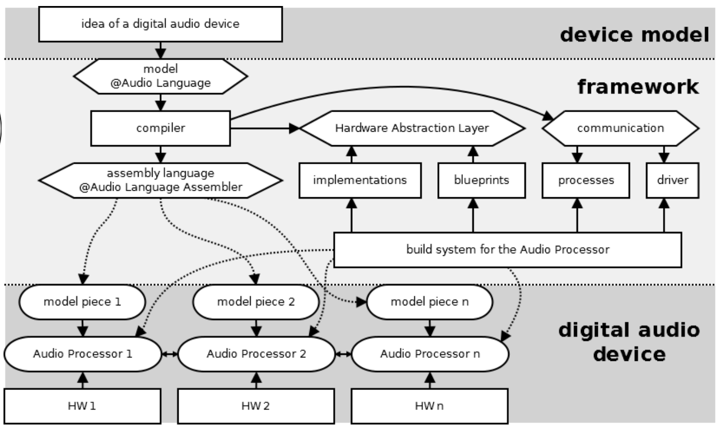

Figure 2: *Creating an audio device according to the new approach*

ment. One is about the process, starting with the idea and ending up with the package: "model piece", "Audio Processor" and hardware. The "Audio Processor" is bonded to the dedicated hardware. The other views are shown in the center of the figure. The area "framework" between the idea and the digital audio device shows the new method's parts listed in section 2. The connections between these parts visualize their interdependencies.

Showing the benefits of this method necessitates a deep understanding of how computing devices are structured. Tannebaum [2] gives an excellent overview over that topic. Figure 2 displays the modelling of a digital audio device by means of a domain-specific language [3]. Tannebaum calls that language a "problem-oriented language" and sets it to level 5 of his computer structure. Using a compiler the model then is translated into assembler code (level 4). The new method thus includes two domain-specific languages: the "Audio Language" and the "Audio Language Assembler". The device's description in the assembler language is interpreted in the processor. Following Tannebaum this interpretation of code is divided into three levels. Level 3 corresponds to the operating system level. The instruction set architecture is set on level 2, and the micro-architecture on level 1. These levels are mixed together in the "Audio Processor" which abstracts the hardware components with the help of a hardware abstraction layer. By the use of this layer it is possible to access I/O pins of the processing unit as well as functions of the operating system (assumed the fact that an OS is run on the device). Popvici [4] describes well the approach of such a hardware abstraction layer closely connected to low cost hardware. Solving audio signal processing problems by the use of a domain-specific language and an interpreter has also been worked out by Boulanger [5]. Furthermore, PC-based platforms may benefit from tools like Pure Data, SuperCollider, FAUST, Csound, etc. In contrast, the new approach targets mainly small computing platforms like μC or DSP. Generally there is only few work that deals with planting code within non-PC hardware (like at FAUST programming language in 2006 [6] and in 2009 [7]).

Creating an "Audio Processor" is managed by a self-developed framework. This is based on a SQLite SQL database to store data and thus contains the definition as well as the implementation of the "Audio Processor" components. A gtk+-based GUI is used to provide a developer-friendly backend to fill the database. All software used for generating "Audio Processors" and "Audio Language" compilers is written in python. The compilers dealing with the "Audio Language" are implemented by means of the PLY [8] module. By doing so, benefits are drawn from python extras and the powerfulness of language creation with the tools lex and yacc. The "Audio Processor" is generated in C code. Afterwards this C code is put into a C compiler provided by the DSP, CPU or μC manufacturers. One reason to use C code is that most of the external software tools/libraries are equipped with a native C interface. Thereby it is possible to integrate them into the "Audio Processor" creation system without writing complicated wrappers.

So called "groups" should help the developer to manage dependencies between the drivers, types of variables and functions. A group is a collection of implementations of previously defined interfaces. Any implementation in the database may refer to such a group. In the example given in section 2 uses the ADSP and MSP430 groups which already includes the core functions of a DSP(ADSP) and μC(MSP430). As one result, the implementation of the "readSample" for ADSP becomes very short by using the functions provided by the ADSP group.

As mentioned before it is possible to define the hardware abstraction layer at the configuration tool. This formalized information about the interface is stored in the database and will be used to automatically generate the "Audio Language Assembler" language. The assembler generated then is able to check the assembler code against type and formal errors.

In conclusion, the new method is about running digital audio device models with different hardware. To archive these processes it is necessary to create a communication system. As shown in figure 2 communication consists of two parts. A process defines when and how data are exchanged. The data exchange itself is handled by drivers, whereby each driver provides access to the physical layer. The interface between driver and data exchange process consists of data frames defined by the developer at the framework. The set of data in such a frame is called message. Through a simple syntax the developer can use message frames to set up the processes handling these messages. In general there are two types of message processes. "RX" processes are designed to handle incoming messages. "TX" processes are started when data shall be transmitted. It is possible to automate the "RX" and "TX" processes. The "RX" process is triggered by receiving a message through a driver. Afterwards, the message is passed to what is called a "Message Handling System". This system searches the correct message handler and organizes the persistence of the data at the model interpreter. Referring to the given example, hazards are possible if the biquad filter of the equalizer is processed while coefficients change at the same time. The "Message Handling System" will prevent this. The "TX" processes are mapped into the "Audio Processor" as common C functions. These functions may be used for implementing the functions of the hardware abstraction layer or in the common code for the "Audio Processors". In the example, the PC as remote control or the MSP430 as user interface will send new coefficients of the filter to the DSP. The "update" function defined in the hardware abstraction layer uses the "TX" functions to perform this task.

After taking a close look at several parts of the method the next paragraph describes how the model of the audio device is build in the example. This is done in a special domain specific language called "Audio Language", which recognizes typed variables and functions. Each variable has a context and may describe global, local or functional parameters. There are two types of functions. One type is called "main". In the example there are three main functions. Among them is one which is called "ADSP" and used for signal processing. This process is performed on an ADSP 21369 DSP from Analog Devices. Another main function is labelled "MSP430" and describes the behaviour of the user interface. It has been built to run on a MSP430-169STK kit from Texas Instruments. The last function labelled "PC" implements a remote control and has been designed to run on a standard PC.

Listing 1 shows the complete code for the signal processing carried out by the DSP.

```
global {
  biquad    filter[2];
}
main (ADSP) {
  local {
    rational  fs;
    rational  f;
    rational  x[512];
  }
  code {
    fs = 48000.0;
    f = 1000.0;
    biquad[obj=filter, index=0, type=LP, fs=fs, fc=f];
    biquad[obj=filter, index=1, type=LP, fs=fs, fc=f];
    for [;;] {
      sampleIO[operation=read ,dest=x, channel=1];
      x = filter * x;
      sampleIO[operation=write ,src=x, channel=2];
    }
  }
}
```

Listing 1: *DSP code*

Signal processing starts with the definition of a global variable. Its name is $filter$, and it represents a vector with two elements of the biquad type. It must be global because it is changed by the "MSP430" and "PC" main functions. In contrast, the main function "ADSP" defines three local variables. fs stands for the sample frequency of the system. f represents the cut-off frequency of the lower pass filter. x is a vector of 512 single precision float values which are used to store the samples of the input channel, resulting from the signal processing. The code consists of two parts. One of them initializes the filter variable as low pass. The other one represents an endless loop, reading 512 samples from the first channel of the system. Finally, the x sample vector is convoluted with the $filter$ biquad vector. The result of the operation is stored at x and written to the second output channel.

Listing 2 shows the "UI" function. This function is called by the two main functions "MSP430" and "PC".

```
function UI {
  interface {}
  local {
    panel     p;
    button    bLeft;
    button    bMiddle;
    button    bRight;
    led       lRed;
    led       lGreen;
    display   d;
    string    tstr;
    rational  fs;
    rational  f;
    rational  f2;
  }
  code {
    // init UI
    // the panel
    ui[obj=p, index=0, func=dim, x=0, y=0, xle=550, yle=500];
    ui[obj=p, index=0, uuid=1, func=initPanel];
    // the display
    ui[obj=d, index=0, func=dim, x=10, y=10, xle=500, yle=300];
    ui[obj=d, index=0, uuid=11, func=initDisplay, parent=p, pIndex=0];
    ...
    // open string
    tstr = "left=LP LP;middle=HP LP;right=HP HP";
    // init constants
    // set sample frequency to 48kHz
    fs = 48000;
    // UI behavior
    ui[obj=d, func=print, text=tstr];
    ui[obj=lRed, func=LED, on=0];
    ui[obj=lGreen, func=LED, on=0];
    for [;;]{
      checkButtonPressed(bLeft) {
        f = 70.0;
        biquad[obj=filter, index=0, type=LP, fs=fs, fc=f ];
        biquad[obj=filter, index=1, type=LP, fs=fs, fc=f];
        ui[obj=lRed, func=LED, on=1];
        ui[obj=lGreen, func=LED, on=0];
        update(filter);
      }
      checkButtonPressed(bMiddle) {
        f = 1000.0;
        f2 = 2000.0;
        biquad[obj=filter, index=0, type=HP, fs=fs, fc=f];
        biquad[obj=filter, index=1, type=LP, fs=fs, fc=f2];
        ...
        update(filter);
      }
      checkButtonPressed(bRight) {
        f = 4000.0;
        biquad[obj=filter, index=0, type=HP, fs=fs, fc=f];
        biquad[obj=filter, index=1, type=HP, fs=fs, fc=f];
        ...
        update(filter);
      }
    }
  }
}
```

Listing 2: *User interface code*

The "UI" function has no interface parameters, so consequently the "interface" section is left empty. Seven variables are declared under the "local" section. They are used to provide controls for the user interface. These declarations are followed by some helper variables which are used as parameters in several functions. The "code" section contains two code parts. During the endless loop the user interface is defined and initialized, and some action handlers for the controls are implemented. First a closer look is taken at the initialization of the user interface. Every single user interface control may be initialized through two functions. The "init..." functions assigns one parent control as well as one unique identifier(uuid) to every control. By means of the uuid and the parent control it is possible to connect one to another. Figure 3 shows the evaluation board of the MSP430 μC which serves as user interface. Controls are labelled with numbers. The displays uuid is 11.

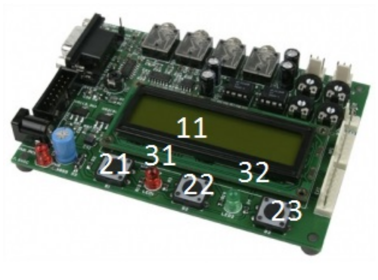

Figure 3: *User interface panel with the control uuids*

The buttons are numbered 21, 22 and 23. The LEDs between the numbers are assigned to the numbers 31 and 32. To provide the opportunity to generate a GUI out of the user interface the "dim" function is called. This function assigns dimensions in pixel to each of the controls. The MSP430 port ignores this function, but the PC port is able to create a window with buttons and edit fields. The assignment of the user interface controls is followed by the initialization of several variables and the basic setup of the user interface. This setup includes setting the displayed text as well as turning-on the LEDs.
After the setup of the user interface an endless loop will check the state of the buttons. If one of them is pressed, the biquad filter coefficients will be recomputed, and the LEDs will switch on. Finally, the global filter variable will be updated through the "update" function. This function relies on the defined TX processes which use a communication driver unique for every hardware platform.
In the following, only a short look is dedicated to the complexity of communication inside that simple system. The physical communication is realized by the use of serial ports. The ADSP serial driver takes into account that the ADSP device is only able to handle 32Bit Integer values internally. These values are stored like at the 16Bit RISC architecture of the MSP430 in big endian format. The data stream of a serial port transmits and receives 8Bit values. If a hardware component will only handle 2 or 4 Byte numbers it will become crucial that they are put in the right order on the lane. Another problem occurs on the Windows OS PC platform. The PC audio processors use a TCP/UDP-based driver to communicate with each other, and thus communicate client-server-based. Due to the limitations of Windows OS - which allows not more than one process to open a UART port - a UART to TCP/IP bridge has been developed. The challenge here was that the PC endianness is little endian. But luckily the endianness could be handled by the message handling system framework. The solving of the endianness issue demonstrates the possibilities of the driver/communication process approach. These class of problems can be solved on driver level or through "Message Handling System". Just because of this separation it is possible to create lean drivers.
Listing 3 represents the two main functions for the PC hardware and the MSP430-169STK evalboard.

```
main (MSP430) {
  local {
  }
  code {
    [] = UI[];
  }
}
main (PC) {
  local {
  }
  code {
    [] = UI[];
  }
}
```

Listing 3: *Code for the MSP430 and PC user 2interface*

There were no local variables used except the "UI" function given in listing 2. And as this function has no input or output parameter, the square brackets have been left empty.

5. CONCLUSION

The total amount of code lines written to model the problem of an equalizer that can be controlled by a μC and a PC as remote control is 132. This simple example demonstrates a new approach to handle digital audio device creation. It covers issues of hiding complex communication behind user-friendly functions as well as the reuse of code via functions. Devices which are able to execute parts of the program are generated in C code. For the ADSP this corresponds to approximately 6000 lines of code needed (compare: MSP430: approximately 7000, PC: approximately 8000). This C code is generated by a framework that combines different approaches to create digital audio devices. Compiled these codes are called "Audio Processors" and are capable to run parts of the model in the given example. As well they are capable to execute other models of different digital audio devices.

6. REFERENCES

[1] Udo Zoelzer, *Digitale Audioverarbeitung*, Teubner, 1996.

[2] Andrew S. Tanenbaum, *Structured Computer Organization (5th Edition)*, Prentice Hall, 2005.

[3] Marjan Mernik, Jan Heering, and Anthony M Sloane, “When and how to develop domain-specific languages,” *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[4] Wolfgang Ecker, Wolfgang Müller, and Rainer Dömer, *Hardware-dependent Software: Principles and Practice*, Springer, 2009.

[5] Richard Boulanger and Victor Lazzarini, *The Audio Programming Book*, The MIT Press, 2010.

[6] Robert Trausmuth, Christian Dusek, and Yann Orlarey, “Using faust for fpga programming,” in *Proc. of the Int. Conf. on Digital Audio Effects (DAFx-06), Montreal, Quebec, Canada*, 2006, pp. 287–290.

[7] Yann Orlarey, Dominique Fober, and Stéphane Letz, “Faust: an efficient functional approach to dsp programming,” *New Computational Paradigms for Computer Music*, 2009.

[8] David Beazley, “Ply (python lex-yacc),” 2001, `http://www.dabeaz.com/ply/`.