Implementation of Digital Audio Broadcasting System Based on SystemC library

Master thesis performed in Electronics Systems by

Eduardo Moreno Martínez

LiTH-ISY-EX-3638-2004
Linköping 24-08-2004
Implementation of Digital Audio Broadcasting System Based on SystemC library

Master thesis in Electronics systems at Linköping Institute of Technology
by

Eduardo Moreno Martínez

LiTH-ISY-EX-3638-2004

Supervisor: Kent Palmkvist
Examiner: Kent Palmkvist
ABSTRACT

This thesis describes the design and implementation of a Digital Audio Broadcasting (DAB) System developed using C++ Language and SystemC libraries. The main aspects covered within this report are the data structure of DAB system, and some interesting points of SystemC Library very useful for the implementation of the final system.

It starts with an introduction of DAB system and his principals advantages. Next it goes further into the definition of data structures of DAB, they are FIC, MSC, and DAB audio frame, explained with MPEG and PAD packets. Later on this chapter there is an explanation of the SystemC library with special attention on the features that I used to implement the system. This features are the events used in the communication between processes and the interfaces needed for sending and receiving the data.

With all these points covered is quite easy for a reader to understand the implementation of the system, despite this point is covered in the last chapter of the thesis. The implementation is here explained in two different steps. The first one explain how is formed the DAB audio frame by means of MPEG frames that are wrote in channel by producer interface, these frames are read by consumer interface. For this purpose I have created some classes and structures that are explained in this part. The second part explain how I obtain the DAB transmission frame which is obtained creating MSC frames, that are big data structures formed by groups of DAB audio frames, therefore there are some functions that act like a buffer and add audio frames to the MSC data structure. Of independent way there is the FIC frame that is generated of random way and it is added to the transmission frame.
ACKNOWLEDGEMENTS

I want to thank to my parents and brothers for supporting me whit his love and affection in all this months living away for my home.
I want to thank the all international and no international people of Linköping who share with me all this great time in Sweden.
I would like to thank to all my Spanish friends for be my friends and stay here when I need them.
I would wish to thank my friend Rosalía Fernandez, who has helped me with grammatically reviewing this thesis.
CONTENTS

1 INTRODUCTION ........................................................................................................... 1
  1.1 MAIN OBJECTIVES AND APPROACH ................................................................. 1
  1.2 GUIDE TO THE THESIS .......................................................................................... 2

2 DAB SYSTEM .................................................................................................................. 5
  2.1 CHARACTERISTICS OF DAB SYSTEM ................................................................. 6
  2.2 DAB ARCHITECTURE .............................................................................................. 8
    2.2.1 Fast Information Channel (FIC) .................................................................... 10
    2.2.1.1 Fast Information Block (FIB) ................................................................. 10
    2.2.1.2 Fast Information Group (FIG) ............................................................. 11
  2.3 Main Service Channel (MSC) .................................................................................. 13
    2.3.1 Stream mode .................................................................................................. 13
    2.3.2 Packet mode - network level ........................................................................ 13
      2.3.2.1 Packet header ................................................................................. 14
      2.3.2.2 Packet data field .............................................................................. 15
      2.3.2.3 Packet CRC ..................................................................................... 15
  2.4 Audio coding: ......................................................................................................... 15
    2.4.1 MPEG Audio Frame Header ........................................................................ 20
    2.4.2 Programme Associated Data (PAD) ........................................................... 25

3 SYSTEM C LIBRARY ...................................................................................................... 29
  3.1 Events and dynamic sensitivity .............................................................................. 31
    3.1.1 WAIT() Method .......................................................................................... 31
  3.2 EVENT TYPES ......................................................................................................... 32
  3.3 INTERFACES ........................................................................................................... 33
    3.3.1 Interface base class ...................................................................................... 33

4 IMPLEMENTATION OF THE SYSTEM: ..................................................................... 35
  4.1 FIRST PHASE: DAB AUDIO FRAME OBTENTION ........................................... 35
  4.2 SECOND PHASE: DAB TRANSMISSION FRAME OBTENTION ....................... 41

5 RESULTS AND CONCLUSIONS ................................................................................... 47

ANEX A INSTALLATION OF SYSTEMC LIBRARY ......................................................... 49

ANEX B SYSTEMC INTERFACE EXAMPLES ............................................................... 50

ANEX C INTRODUCTION TO C++ AND CLASS HERITANCE ................................... 53
1 INTRODUCTION

The target of this project is to implement the DAB System using the functionalities that offer us the SystemC library in C++.

This library contains a helpful set of characteristics that are perfect for the implementation of synchronized systems, and this is the field goal of the use of this library.

The DAB system is a very complete system and his specifications includes since the data bits structures used for the protocol until the radio communications specifications used for transmit the radio signal. One another hand the quantity of services described in the DAB specifications are innumerable is for that that is necessary to concrete the system to only a part of this.

Although DAB can transmit audio and data the proposed implementation of the system is prepared to transmit only audio. It is important to observe that all the data structures adjust to the specifications and for further development of the system can be used.

This report initially try to get some ideas about DAB and SystemC, for later explain the system programmed, it is supposed that for the use of SystemC library it is necessary to get knowledge about Oriented Object Programming and C++ that is the used programming language, although in the end of the report there is an annex with some information about classes and heritance between classes, because this concepts was very important to develop the system.

1.1 MAIN OBJECTIVES AND APPROACH

Within the project, I am to implement a document System bases in DAB Audio System which creates audio frames together different kind of data structures used in this protocol This data structures are processed to obtain in final result the DAB transmission frame.

For help me in the development I used a C++ Library named SystemC, this library has a lot of functionalities and for all of them I used his time based signals and process to obtain a synchronized system, this is completely necessary if we have a system like DAB because we have to transmit binary data packet to keep the DAB system requirements.

To obtain this I have to do an abstraction of the real system, because DAB is a very big system, I don’t implement the energy dispersal process, convolutional codes and interleaving. The system is not a completed system but it can give a good idea of how DABS system work.
For create the system I used C++ Language, a very common and used language, that I used sometimes in my home university, is good for me to use this because I have never implemented a very big system like this with this language and I think that finally I obtain a good experience to work with it.

1.2 GUIDE TO THE THESIS
In this thesis is described the architecture and main features of the DAB based system that I have developed for my master's graduation project. Besides a high-level description of the various aspects of this system, I also present the most important details of the implementation. In the remainder description of this section is presented a general description of the contents that follows.

• **Section 2** presents what is DAB system and how it works. It presents too the data structures used in this protocol that I based for create the system.

• **Section 3** gives an overview of SystemC Library and of his principal functionalities that I used to develop the final system.

• **Section 4** describes the implementation of the system, I divide this section in to parts, that for me they are the principal field goals in the total development, in first order how to obtain the DAB frame audio system, and in second order using the results of the first part obtain the DAB transmission frame.

• **Section 5** presents the results of the system implemented. I will also discuss some advantages and disadvantages of the system built.

The target of this project is to implement the DAB System using the functionalities that offer us the SystemC library in C++.

This library contains a helpful set of characteristics that are perfect for the implementation of synchronized systems, and this is the field goal of the use of this library.

The DAB system is a very complete system and his specifications includes from the data bits structures used for the protocol until the radio communications specifications used for transmitting the radio signal. On another hand the quantity of services described in the DAB specifications are innumerable, for that is necessary to concrete the system to only a part of it. Although DAB can transmit audio and data, the proposed implementation for the system is prepared to transmit only audio. It is important to observe that all the data structures are adjusted to the specifications and for further development of the system can be used.

This report initially try to get some ideas about DAB and SystemC, to explain later the programmed system, it is supposed that for the use of SystemC library it is necessary to get knowledge about Oriented Object Programming and C++ that is the used programming language, although in the end of the report there
is an annex with some information about classes and inheritance between classes, because these concepts were very important to develop the system.
2  DAB SYSTEM

System DAB (Digital Audio Broadcasting) was born in 1987 like a European project denominated Eureka 147.

The DAB is a very robust system designed for domestic and portable receivers and, specially, for the reception of mobile receivers. It is suitable for satellite and the terrestrial broadcast and allows us to introduce data. This technology does not have the problems of FM when many signals from different points are received: transmitted signal and its reflections, dispersions and diffractions that vary in time. It is obtained with DAB system that most of these signals contribute positively to the reception, and we can obtain in the received signal a gain of reception like when we use and amplifier.

DAB technique allows introducing many channels in the spectrum, and on each channel many programs, the capacity of programs is practically multiplied using the same spectrum frequencies. In addition, the system allows emitting a great number of programs using multiplex, depending on the quality that is required.

The quality of the programs in DAB is similar to the CD audio quality but not exactly the same, although for human ear it practically sounds equal. In order to be able to emit 6 programs by same multiplex it is necessary to reduce the information, so we need to eliminate information that ear cannot hear, maintaining an admissible quality for the broadcasting.

This system is based fundamentally in two principles: MP2 source codification and COFDM channel codification.

The source codification, that originally is denominated Musicam and later was standardized denoming it MPEG2 or MP2, is a system very similar to the MP3 but it is necessary less processing capacity for MPEG2 than MP3. It is based fundamentally in the principle of reducing information that the ear can’t distinguish. When there are two very next signals in frequency and one of them is stronger than the other, the signal that has inferior level normally is masked and it is not possible to hear it. In addition, the ear has a threshold of noise below as it does not hear the sounds. This system eliminates everything what the ear is not going to perceive. By this way, we are able to reduce the bandwidth of the original signal that is needed to transmit. Reducing in 6 factor the information is possible to emit 6 programs in stead of one, using the necessary capacity for only one program.

In fact with DAB, a data container is transmitted continuously, we have two different kinds of information, for one hand the information of its content and its configuration is sent, this allow to the receiver to know in a very fast way what it is receiving and allow to select anyone of the contents (programs). On the other hand, in the container it is sent additional services and audio programs, and within each program of audio we can introduce data associated to that program, for example, a meteorological chart.

The total capacity of information in a multiplex is of 2’3 Mbit/s, but in fact what we have is a container with 864 cells, that are filled with programs and data, and are emitted continuously.
System DAB uses for the codification of the transmission channel the system of modulation COFDM. It is multiplex by division of orthogonal frequencies in which DAB made a codification. On the one hand, the codification introduces redundancy to be able to detect the transmission errors and to correct them and, in addition, the system uses time division multiplexing access (TDMA), and frequency division multiplexing access (FDMA). The diversity in the time is obtained by means of time interleaving of the information, so that if there are some disturbance, when having the distributed information is possible to recover it better, avoiding continuous errors in a frame. With frequency division multiplexing access it is obtained that information will be distributed by discontinuous way in all spectrum of the channel and it is seen less affected by the disturbances, and with the division in the space, information can be sent from different emitting centers and all of them contribute positively creating a network of unique frequency, also, reflections of the signal contribute positively in the receiver.

Obtained quality is directly related with bandwidth. System DAB allows to use two compressions systems: MPEG1 (MP1) and MPEG2 (MP2). First it is the normal one for high speeds, whereas the MP2 allows to use a sampling frequency half, instead of using the 48 kHz of frequency of sampling of the 24 MP1 is used kHz, to obtain at low speeds a better quality. For example, with 20 kbit/s in MP2 obtains a quality similar to the one of 70 kbit/s with the MP1.

If we spread 6 programs with quality 192 kbit/s, and level of protection 3, kbit/s is left a small channel of to 32 to introduce data. Nevertheless, if we used 160 kbit/s and a level of protection 2, instead of 6 programs we could only spread 5 programs. That is to say, depending on the level of protection and the speed of transmission we can spread programs more or less. If of a speed of 192 kbit/s we lowered an inferior one, for example, 160 kbit/s with the same level of protection 3, we can spread to 7 programs instead of 6 programs, leaving for 224 data kbit/s, as it is indicated in the figures. Therefore, it is a flexible system but with a limited capacity, with which the capacity that we used for programs not it we will be able to use for data.

2.1 CHARACTERISTICS OF DAB SYSTEM

System DAB provides digital broadcasting multiservice of high quality, destined to mobile, portable and fixed receivers, as much for the terrestrial broadcasting as for broadcasting by satellite. It is a flexible system that allows an ample range of options of programs codification, the data associated to the radiophonic programs and of the services of additional data.

Their main characteristics are the following ones:

- Efficiency in the use of the frequency spectrum and the power with transmitters of low power.

- Improvements in the reception. By means of system DAB the effects of multipassge propagation (reflections in buildings, mountains, etc..) that
are produced in the stationary, portable and movable receivers are suppressed (information interferences and distortion). These improvements are obtained by means of the COFDM transmission that uses a codification system to distribute the information between a high number of frequencies.

- Rank of transmission frequencies: System DAB is designed to be able to work in the rank of frequencies between 30 - 3,000 MHz.

- Distribution: It is possible to be made by satellite and/or terrestrial transmission or of cable using different ways that the receiver will detect automatically.

- Quality of sound: It is equivalent to the CD audio quality. In system DAB the effect takes advantage of the psychoacoustic characteristics of the human ear to obtain this quality, because human era is not able to perceive all the present sounds in a while, and therefore is not necessary to transmit the sounds that are not audible. System DAB uses a system of compression of called sound MUSICAM to eliminate the nonaudible information, and it reduce the amount of information to transmit.

- Multiplexing: Of analogous for to chose a channel in TV, it is possible "to enter" multiplex DAB and select between several programs of audio or services of data.

- Capacity: Each block (multiplex) has a useful capacity of approximately 1.5 Mbit/s, which, for example, allows transporting 6 stereo programs of 192 kbit/s each one, with his corresponding protection and several additional services.

- Flexibility: The services can be structured and be formed dynamically. The system can accommodate speeds of transmission between 8 and 380 kbit/s including corresponding protection.

- Services of Data: In addition to the digitized audio signal, in multiplex, other information can be transmitted:
  - The information channel: It transports the configuration of multiplex, information of the services, date and hour, services of general performances like: radio searching, system of warning of emergency, system of global positioning, etc.
  - The data associated to the program are dedicated to the information directly related to the programs of audio: musical titles, author, songs lyrics in several languages, etc.

- Additional services: They are services that are provided to a reduced group of users, like for example: cancellation of robbed shipment, credit cards of images and text to electronic bulletin boards, etc. All these data are received through a screen incorporated to the receiver.

- Coverage: The cover can be local, regional, supranational and national. The system is able constructively to add the signals coming from
different transmitters in the same channel, which allows establish networks of unique frequency to cover a certain geographic area in which it is possible to use small transmitters to cover the zones.

2.2 DAB ARCHITECTURE

The DAB system is designed to carry several digital audio signals together with data signals. Audio and data signals are considered to be service components which can be grouped together to form services. This subclause describes the main transport mechanisms available in the DAB multiplex. The DAB transmission system combines three channels:

1) **Main Service Channel (MSC):** used to carry audio and data service components. The MSC is a time interleaved data channel divided into a number of sub-channels which are individually convolutionally coded.

2) **Fast Information Channel (FIC):** used for rapid access of information by a receiver. In particular it is used to send the Multiplex Configuration Information (MCI) and optionally Service Information and data services.

3) **Synchronization channel:** used internally within the transmission system for basic demodulator functions, such as transmission frame synchronization, automatic frequency control, channel state estimation, and transmitter identification.

Each channel supplies data from different sources and these data are provided to form a transmission frame. Both the organization and length of a transmission frame depend on the transmission mode. The Fast Information Block (FIB) and the Common Interleaved Frame (CIF) are introduced in order to provide transmission mode independent data transport packages associated with the FIC and MSC respectively.
Next table gives the transmission frame duration and the number of FIBs and CIFs which are associated with each transmission frame for the four transmission modes.

<table>
<thead>
<tr>
<th>Transmission mode</th>
<th>Duration of transmission frame</th>
<th>Number of FIBs per transmission frame</th>
<th>Number of CIFs per transmission frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>96 ms</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>II</td>
<td>24 ms</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>III</td>
<td>24 ms</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>IV</td>
<td>48 ms</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

In transmission mode I, the 12 FIBs contributing to one transmission frame shall be divided into four groups which are each assigned to one of the CIFs contributing to the same transmission frame.

The information contained in the first three FIBs shall refer to the first CIF, the information contained in the fourth, fifth and sixth FIB to the second CIF, and so on. All FIBs contributing to a transmission frame, in transmission modes II and III, shall be assigned to the one CIF associated with that transmission frame. In transmission mode IV, the six FIBs contributing to one transmission frame shall be divided into two groups which are each assigned to one of the CIFs contributing to the same transmission frame. The information contained in the first three FIBs shall refer to the first CIF, and the information contained in the fourth, fifth and sixth FIB to the second CIF.

The following subclauses describe the formation of the FIC and MSC.
2.2.1 Fast Information Channel (FIC)

The FIC is made up of a group of FIB frames, it is showed in figure 1 next it is important to explain, the frame structure of the FIB blocks and the different types and functions that can realize

2.2.1.1 Fast Information Block (FIB)

The general structure of the FIB is shown in figure 3, for a case when the useful data does not occupy the whole of a FIB data field. The FIB contains 256 bits and comprises an FIB data field and a CRC.

![Figure 2: Structure of the FIB](image)

Next are explained the all the fields showed in this figure that form the structure of FIB block:

- FIB data field: the FIB data field shall be organized in bytes allocated to useful data, an end marker and padding in the following way
  - the useful data occupy the whole 30 bytes of the FIB data field. In this case, there shall be no end marker and no padding bytes.
  - the useful data occupy 29 bytes of the FIB data field. In this case, there shall be an end marker but no padding bytes.
the useful data occupy less than 29 bytes. In this case, there shall be both an end marker and padding bytes.

- there is no useful data. In this case, the FIB data field shall begin with an end marker and the rest of the FIB data field contains padding bytes.

- The FIB data field is described as follows:
  - **Useful data field:** this contains one or more Fast Information Groups (FIGs).
  - **Padding:** this field shall contain the bytes required to complete the FIB data field. The padding byte field shall contain all zeroes.
  - **End marker:** is a special FIG and shall have a FIG header field (111 11111) and no FIG data field.
  - **CRC:** a 16-bit Cyclic Redundancy Check word is calculated on the FIB data field and shall be generated by a polynomial algorithm based in ITU-T Recommendation X.25.

### 2.2.1.2 Fast Information Group (FIG)

The FIG shall comprise the FIG header and the FIG data field (see figure 2). The following definitions apply:

- **FIG header:** shall contain FIG type field and the length:
  - **FIG type:** this 3-bit field shall indicate the type of data contained in the FIG data field. The assignment of FIG types is given in next table.

<table>
<thead>
<tr>
<th>FIG type number</th>
<th>FIG type</th>
<th>FIG application</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>MCI and part of the SI</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>001 Labels, etc. (part of the SI)</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>Data Channel (FIDC)</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>Conditional Access (CA)</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>In house (except for Length 31)</td>
</tr>
</tbody>
</table>

- **Length:** this 5-bit field shall represent the length in bytes of the FIG data field and is expressed as an unsigned binary number (MSb first) in the range 1 - 29. Values 0, 30 and 31 shall be reserved for future use of the FIG data field except for 31 ("11111") when used with FIG type 7 ("111") which is used for the end marker.

- **FIG data field:** this field is described in next figures that show the data structures of types 0, 1, 5 and 6.
Figure 3: FIG types and data structure
2.3 Main Service Channel (MSC)

The MSC is made up of Common Interleaved Frames (CIFs). The CIF contains 55,296 bits. The smallest addressable unit of the CIF is the Capacity Unit (CU), comprising 64 bits. Therefore, the CIF contains 864 CUs, which shall be identified by the CU addresses 0 to 863.

The MSC is divided into sub-channels. Each sub-channel shall occupy an integral number of consecutive CUs and is individually convolutionally encoded. Each CU may only be used for one sub-channel. A service component is a part of a service which carries either audio or general data.

The data, carried in the MSC, shall be divided at source into regular 24 ms bursts corresponding to the sub-channel data capacity of each CIF. Each burst of data constitutes a logical frame. Each logical frame is associated with a corresponding CIF. Succeeding CIFs are identified by the value of the CIF counter, which is signalled in the MCI.

The logical frame count is a notional count which shall be defined as the value of the CIF counter corresponding to the first CIF which carries data from the logical frame.

There are two transport modes in the MSC: one is called the stream mode and the other the packet mode.

2.3.1 Stream mode

The stream mode allows a service application to accept and deliver data transparently from source to destination. At any one time, the data rate of the application shall be fixed in multiples of 8 kbit/s. The application shall either supply information on demand, or include a method of handling data asynchronously at a lower rate. Data shall be divided into logical frames. Only one service component shall be carried in one sub-channel. The DAB audio frame typically has a duration of 24 ms and shall map on to the logical frame structure in such a way that the first bit of the DAB audio frame corresponds to the first bit of a logical frame.

2.3.2 Packet mode - network level

This is the method used in the final implementation of the system. DAB audio frames are carried in a MSC GROUP bit structure, in lines below we can see how it work.

The packet mode allows different data service components to be carried within the same sub-channel. The permissible data rates for the sub-channel shall be
multiples of 8 kbit/s. Data may be carried in data groups or transported using packets alone. The value of the DG flag indicates which mode is used.

A packet shall be identified by an address. Packets with different addresses may be sent in any order in a sub-channel. However, the sequence of packets with the same address shall be maintained.

Padding packets shall be used, if necessary to adjust the data rate to the required multiple of 8 kbit/s.

A packet shall consist of a Packet header, a Packet data field and a Packet CRC.

![Figure 4: Structure of MSC](image)

### 2.3.2.1 Packet header

The packet header has a length of 3 bytes and it shall comprise the following parameters:

- **Packet length**: four different packet data field lengths are allowed.
- **Continuity index**: this 2-bit, modulo-4 counter shall be incremented by one for each successive packet in a series having the same address. It provides the link between successive packets, carrying the same service component, regardless of length.
- **First/Last**: these two flags shall be used to identify particular packets which form a succession of packets, carrying data groups of the same service component.
- **Address**: this 10-bit field shall identify packets carrying a particular service component within a sub-channel. We can carry up to 1023 different services in a sub-channel.
- **Command**: this 1-bit flag shall indicate whether the packet is used for general data or for special commands, in the implementation every times this field is 0, because we are transmitting data information.
- **Useful data length**: this 7-bit field represents the length in bytes of the associated useful data field.
2.3.2.2 Packet data field
This field contains the useful data field and padding.
- **Useful data field**: this field shall contain the useful service component data, in our case of study this service component is directly formed by DAB audio frames.
- **Padding**: this field shall comprise the bytes required to complete the packet data field. The padding byte field shall contain all ones.

2.3.2.3 Packet CRC
The packet CRC is a 16-bit CRC word calculated on the packet header and the packet data field. The generation is based in the polynomial method following the ITU-T Recommendation X.25.

![Figure 5: MSC group](image)

2.4 Audio coding:

The coding technique for high quality audio signals uses the properties of human sound perception by exploiting the spectral and temporal masking effects of the ear. This technique allows a bit rate reduction from 768 kbit/s down to about 100 kbit/s per mono channel, while preserving the subjective quality of the digital studio signal for any critical source material.

An overview of the principal functions of the audio coding scheme is shown in the simplified block diagram of the DAB audio encoder.
The input PCM audio samples are fed into the audio encoder. A filter bank creates a filtered and sub-sampled representation of the input audio signal. The filtered samples are called sub-band samples. A psychoacoustic model of the human ear should create a set of data to control the quantizer and coding. These data can be different depending on the actual implementation of the encoder. An estimation of the masking threshold can be used to obtain these quantizer control data. The quantizer and coding block shall create a set of coding symbols from the sub-band samples. The frame packing block shall assemble the actual audio bit stream from the output data of the previous block, and shall add other information, such as header information, CRC words for error detection and Programme Associated Data (PAD), which are intimately related with the coded audio signal. For a sampling frequency of 48 kHz, the resulting audio frame corresponds to 24 ms duration of audio and shall comply with the ISO/IEC 11172-3 Layer II format. The DAB audio frame shall be fed into the audio decoder, which unpacks the data of the frame to recover the various elements of information. The reconstruction block shall reconstruct the quantized sub-band samples. An inverse filter bank shall transform the sub-band samples back to produce digital PCM audio signals.
The source encoder for the DAB system is the MPEG Audio Layer II encoder with restrictions on some parameters and some additional protection against transmission errors.

The DAB source coding algorithm is based on a perceptual coding technique. The six primary parts of such an audio encoding technique are:

1) Analysis sub-band filter.
   An analysis sub-band filter should be used to split the broadband audio signal with sampling frequency $f_s$ into 32 equally spaced sub-bands, each with a sampling frequency of $f_s/32$.

2) Scale Factor calculation.
   In each sub-band, 36 samples shall be grouped for processing. Before quantization, the output samples of the filter bank should be normalized. The calculation of the Scale Factor (ScF) for each sub-band shall be performed every 12 sub-band samples. The maximum of the absolute value of these 12 samples shall be determined. The lowest value, given by the column "Scale Factor" in table 11, which is larger than this maximum shall be used as the ScF of the 12 sub-band samples.

3) Psychoacoustic model.
   A psychoacoustic model should calculate a just-noticeable noise-level for each sub-band in the filter bank. This noise level should be used in the bit allocation procedure to determine the actual quantizer for each sub-band. The final output of the model is a Signal-to-Mask Ratio (SMR) for each sub-band. For a high coding efficiency, it is recommended to use a psychoacoustic model with an appropriate frequency analysis.
4) Bit allocation procedure.

A bit allocation procedure shall be applied. Different strategies for allocating the bits to the sub-band samples of the individual sub-bands are possible. The principle used in this allocation procedure is minimization of the total noise-to-mask ratio over the audio frame with the constraint that the number of bits used does not exceed the number of bits available for that DAB audio frame.

The allocation procedure should consider both the output samples from the filter bank and the Signal-to-Mask-Ratios (SMRs) from the psychoacoustic model. The procedure should assign a number of bits to each sample (or group of samples) in each sub-band, in order to simultaneously meet both the bit rate and masking requirements. At low bit rates, when the demand derived from the masking threshold cannot be met, the allocation procedure should attempt to spread bits in a psychoacoustically inoffensive manner among the sub-bands.

After determining, how many bits should be distributed to each sub-band signal, the resulting number shall be used to code the sub-band samples, the ScFSI and the ScFs. Only a limited number of quantizations is allowed for each subband.

5) Quantizing and coding.

A quantization process of the sub-band samples shall be applied. The following description of this process is informative, but the coding of the sub-band samples has to follow normative rules.

Each of the 12 consecutive sub-band samples, which are grouped together for the scaling process, should be normalized by dividing its value by the Scale Factor to obtain a value denoted X and quantized using a mathematic based procedure in multiplication of coefficients:

6) Bit stream formatter.

The frame formatter of the audio encoder shall take the bit allocation, ScFSI, ScF and the quantized sub-band samples together with header information and a few code words used for error detection to format the MPEG Audio Layer II [3,14] bit stream. It shall further divide this bit stream into audio frames, each corresponding to 1152 PCM audio samples, which is equivalent to a duration of 24 ms in the case of 48 kHz sampling frequency and 48 ms in the case of 24 kHz sampling frequency. The principal structure of such an MPEG Audio Layer II frame with its correspondence to the DAB audio frame can be seen in figure 24.

Each audio frame starts with a header, consisting of a sync word and audio system related information. A Cyclic Redundancy Check (CRC), following the header protects a part of the header information, the bit allocation, and the ScFSI fields. After the CRC follows bit allocation, ScFSI and Scale Factors. The sub-band samples, which will be used by the decoder to reconstruct the PCM audio signal, are the last audio data part in the MPEG Audio Layer II frame before the ancillary data field. This ancillary data field, which is of variable length, is located at the end of the MPEG Audio Layer II frame.
Figure 8: MPEG audio coding
An adaptation of the MPEG Audio Layer II frame to the DAB audio frame is performed in order to introduce:

- specific DAB Scale Factor Error Check (ScF-CRC);
- a fixed and a variable field of Programme Associated Data (F-PAD and X-PAD).

The lower part of figure 24 indicates how this additional specific information, necessary for DAB, shall be inserted into the ancillary data field of the MPEG Audio Layer II frame.

For MPEG-1 Audio the whole DAB audio frame fits exactly into a DAB logical frame. However, for LSF-coding which is standardized in MPEG-2 Audio, the DAB LSF audio frame shall be divided into two subframes of equal length and each subframe fits into two consecutive DAB logical frames.

Figure 9: Differences between MPEG Audio Layer II and DAB audio frame

2.4.1 MPEG Audio Frame Header

An MPEG audio file is built up from smaller parts called frames. Generally, frames are independent items. Each frame has its own header and audio information. There is no file header. Therefore, you can cut any part of MPEG file and play it correctly (this should be done on frame boundaries but most applications will handle incorrect headers). For Layer III, this is not 100% correct. Due to internal data organization in MPEG version 1 Layer III files, frames are often dependent of each other and they cannot be cut off just like that.

When you want to read info about an MPEG file, it is usually enough to find the first frame, read its header and assume that the other frames are the same. This may not be always the case. Variable bitrate MPEG files may use so called bitrate switching, which means that bitrate changes according to the content of each frame. This way lower bitrates may be used in frames where it will not
reduce sound quality. This allows making better compression while keeping
high quality of sound.

The frame header is constituted by the very first four bytes (32bits) in a frame.
The first eleven bits (or first twelve bits, see below about frame sync) of a frame
header are always set and they are called "frame sync". Therefore, you can
search through the file for the first occurrence of frame sync (meaning that you
have to find a byte with a value of 255, and followed by a byte with its three (or
four) most significant bits set). Then you read the whole header and check if the
values are correct. You will see in the following table the exact meaning of each
bit in the header, and which values may be checked for validity. Each value that
is specified as reserved, invalid, bad, or not allowed should indicate an invalid
header. Remember, this is not enough, frame sync can be easily (and very
frequently) found in any binary file. Also it is likely that MPEG file contains
garbage on it's beginning which also may contain false sync. Thus, you have to
check two or more frames in a row to assure you are really dealing with MPEG
audio file.

Frames may have a CRC check. The CRC is 16 bits long and, if it exists, it
follows the frame header. After the CRC comes the audio data. You may
calculate the length of the frame and use it if you need to read other headers
too or just want to calculate the CRC of the frame, to compare it with the one
you read from the file. This is actually a very good method to check the MPEG
header validity.

Here is "graphical" presentation of the header content. Characters from A to M
are used to indicate different fields. In the table, you can see details about the
content of each field.

<table>
<thead>
<tr>
<th>Sign</th>
<th>Length (bits)</th>
<th>Position (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11</td>
<td>(31-21)</td>
<td>Frame sync (all bits set)</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>(20,19)</td>
<td>MPEG Audio version ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>00 - MPEG Version 2.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>01 - reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 - MPEG Version 2 (ISO/IEC 13818-3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11 - MPEG Version 1 (ISO/IEC 11172-3)</td>
</tr>
</tbody>
</table>

Note: MPEG Version 2.5 is not official standard. Bit No 20
in frame header is used to indicate version 2.5.
Applications that do not support this MPEG version
expect this bit always to be set, meaning that frame sync
(A) is twelve bits long, not eleven as stated here.
Accordingly, B is one bit long (represents only bit No 19). I
recommend using methodology presented here, since this
allows you to distinguish all three versions and keep full
compatibility.

2     | (18,17)    | Layer description |
C  00 - reserved
    01 - Layer III
    10 - Layer II
    11 - Layer I

D  1        (16)  Protection bit
    0 - Protected by CRC (16bit CRC follows header)
    1 - Not protected

E  4        (15,12)  Bitrate index

<table>
<thead>
<tr>
<th>bits</th>
<th>V1,L1</th>
<th>V1,L2</th>
<th>V1,L3</th>
<th>V2,L1</th>
<th>V2, L2 &amp; L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>free</td>
<td>free</td>
<td>free</td>
<td>free</td>
<td>free</td>
</tr>
<tr>
<td>0001</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>0010</td>
<td>64</td>
<td>48</td>
<td>40</td>
<td>48</td>
<td>16</td>
</tr>
<tr>
<td>0011</td>
<td>96</td>
<td>56</td>
<td>48</td>
<td>56</td>
<td>24</td>
</tr>
<tr>
<td>0100</td>
<td>128</td>
<td>64</td>
<td>56</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>0101</td>
<td>160</td>
<td>80</td>
<td>64</td>
<td>80</td>
<td>40</td>
</tr>
<tr>
<td>0110</td>
<td>192</td>
<td>96</td>
<td>80</td>
<td>96</td>
<td>48</td>
</tr>
<tr>
<td>0111</td>
<td>224</td>
<td>112</td>
<td>96</td>
<td>112</td>
<td>56</td>
</tr>
<tr>
<td>1000</td>
<td>256</td>
<td>128</td>
<td>112</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>1001</td>
<td>288</td>
<td>160</td>
<td>128</td>
<td>144</td>
<td>80</td>
</tr>
<tr>
<td>1010</td>
<td>320</td>
<td>192</td>
<td>160</td>
<td>160</td>
<td>96</td>
</tr>
<tr>
<td>1011</td>
<td>352</td>
<td>224</td>
<td>192</td>
<td>176</td>
<td>112</td>
</tr>
<tr>
<td>1100</td>
<td>384</td>
<td>256</td>
<td>224</td>
<td>192</td>
<td>128</td>
</tr>
<tr>
<td>1101</td>
<td>416</td>
<td>320</td>
<td>256</td>
<td>224</td>
<td>144</td>
</tr>
<tr>
<td>1110</td>
<td>448</td>
<td>384</td>
<td>320</td>
<td>256</td>
<td>160</td>
</tr>
<tr>
<td>1111</td>
<td>bad</td>
<td>bad</td>
<td>bad</td>
<td>bad</td>
<td>bad</td>
</tr>
</tbody>
</table>

NOTES: All values are in kbps
V1 - MPEG Version 1
V2 - MPEG Version 2 and Version 2.5
L1 - Layer I
L2 - Layer II
L3 - Layer III
"free" means free format. If the correct fixed bitrate (such files cannot use variable bitrate) is different than those presented in upper table it must be determined by the application. This may be implemented only for internal purposes since third party applications have no means to find out correct bitrate. However, this is not impossible to do but demands lot’s of efforts.
"bad" means that this is not an allowed value

MPEG files may have variable bitrate (VBR). This means that bitrate in the file may change. I have learned about two used methods:

- bitrate switching. Each frame may be created with different bitrate. It may be used in all layers. Layer III decoders must support this method. Layer I & II decoders
may support it.

- bit reservoir. Bitrate may be borrowed (within limits) from previous frames in order to provide more bits to demanding parts of the input signal. This causes, however, that the frames are no longer independent, which means you should not cut this file. This is supported only in Layer III.

For Layer II there are some combinations of bitrate and mode which are not allowed. Here is a list of allowed combinations.

<table>
<thead>
<tr>
<th>bitrate</th>
<th>allowed modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>all</td>
</tr>
<tr>
<td>32</td>
<td>single channel</td>
</tr>
<tr>
<td>48</td>
<td>single channel</td>
</tr>
<tr>
<td>56</td>
<td>single channel</td>
</tr>
<tr>
<td>64</td>
<td>all</td>
</tr>
<tr>
<td>80</td>
<td>single channel</td>
</tr>
<tr>
<td>96</td>
<td>all</td>
</tr>
<tr>
<td>112</td>
<td>all</td>
</tr>
<tr>
<td>128</td>
<td>all</td>
</tr>
<tr>
<td>160</td>
<td>all</td>
</tr>
<tr>
<td>192</td>
<td>all</td>
</tr>
<tr>
<td>224</td>
<td>stereo, intensity stereo, dual channel</td>
</tr>
<tr>
<td>256</td>
<td>stereo, intensity stereo, dual channel</td>
</tr>
<tr>
<td>320</td>
<td>stereo, intensity stereo, dual channel</td>
</tr>
<tr>
<td>384</td>
<td>stereo, intensity stereo, dual channel</td>
</tr>
</tbody>
</table>

F 2 (11,10) Sampling rate frequency index (values are in Hz)

<table>
<thead>
<tr>
<th>bits</th>
<th>MPEG1</th>
<th>MPEG2</th>
<th>MPEG2.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>44100</td>
<td>22050</td>
<td>11025</td>
</tr>
<tr>
<td>01</td>
<td>48000</td>
<td>24000</td>
<td>12000</td>
</tr>
<tr>
<td>10</td>
<td>32000</td>
<td>16000</td>
<td>8000</td>
</tr>
<tr>
<td>11</td>
<td>reserv.</td>
<td>reserv.</td>
<td>reserv.</td>
</tr>
</tbody>
</table>

G 1 (9) Padding bit
0 - frame is not padded
1 - frame is padded with one extra slot

Padding is used to fit the bit rates exactly. For an example: 128k 44.1kHz layer II uses a lot of 418 bytes and some of 417 bytes long frames to get the exact 128k bitrate. For Layer I slot is 32 bits long, for Layer II and Layer III slot is 8 bits long.

First, let's distinguish two terms frame size and frame length. Frame size is the number of samples contained in a frame. It is constant and always 384 samples for Layer I
and 1152 samples for Layer II and Layer III. Frame length is length of a frame when compressed. It is calculated in slots. One slot is 4 bytes long for Layer I, and one byte long for Layer II and Layer III. When you are reading MPEG file you must calculate this to be able to find each consecutive frame. Remember, frame length may change from frame to frame due to padding or bitrate switching.

Read the BitRate, SampleRate and Padding of the frame header.

For Layer I files use this formula:

\[
\text{FrameLengthInBytes} = \left( 12 \times \frac{\text{BitRate}}{\text{SampleRate}} + \text{Padding} \right) \times 4
\]

For Layer II & III files use this formula:

\[
\text{FrameLengthInBytes} = 144 \times \frac{\text{BitRate}}{\text{SampleRate}} + \text{Padding}
\]

Example:
Layer III, BitRate=128000, SampleRate=441000, Padding=0
\[
\Rightarrow \text{FrameSize}=417 \text{ bytes}
\]

H 1 (8) Private bit. It may be freely used for specific needs of an application, i.e. if it has to trigger some application specific events.

I 2 (7,6) Channel Mode
00 - Stereo
01 - Joint stereo (Stereo)
10 - Dual channel (Stereo)
11 - Single channel (Mono)

J 2 (5,4) Mode extension (Only if Joint stereo)
Mode extension is used to join informations that are of no use for stereo effect, thus reducing needed resources. These bits are dynamically determined by an encoder in Joint stereo mode.

Complete frequency range of MPEG file is divided in subbands. There are 32 subbands. For Layer I & II these two bits determine frequency range (bands) where intensity stereo is applied. For Layer III these two bits determine which type of joint stereo is used (intensity stereo or m/s stereo). Frequency range is determined within decompression algorithm.
### Programme Associated Data (PAD)

Each DAB audio frame contains a number of bytes which may carry Programme Associated Data (PAD). PAD is information which is synchronous to the audio and its contents may be intimately related to the audio. The PAD bytes in successive audio frames constitute the PAD channel.

The PAD bytes are always located at the end of each DAB audio frame. With a sampling frequency of 48 kHz, the whole DAB audio frame fits into the 24 ms frame structure of the CIF, and a new set of PAD bytes is available at the receiver every 24 ms. However in the case of a 24 kHz sampling frequency, the DAB LSF audio frame is divided into two parts of equal length (i.e. an even and odd partial frame) and spread across two CIFs. In this case, a new set of PAD bytes is available only every 48 ms.

In each DAB audio frame there are two bytes called the fixed PAD (F-PAD) field. Thus, the bit rate of the F-PAD field depends on the sampling frequency used for the audio coding. The bit rate for F-PAD is 0.667 kbit/s for 48 kHz sampling frequency. In the case of 24 kHz sampling frequency, this value is divided by a factor of two.

The F-PAD field is intended to carry control information with a strong real-time character and data with a very low bit rate. The PAD channel may be extended using an Extended PAD (X-PAD) field, intended to carry information providing...
additional functions to the listener, such as programme related text. The length of the X-PAD field is chosen by the service provider. The use of PAD is optional. If no information is sent in the F-PAD, all bytes in the F-PAD field shall be set to zero. This also implies that no X-PAD field is present.

The PAD carried in the DAB audio frame n shall be associated with the audio carried in the following frame, n+1.

If functions in PAD are used in dual channel mode, they shall apply to channel 0 unless otherwise signalled by the application.

![Figure 10: location of the F-PAD and X-PAD fields within the DAB audio frame.](image)

The two bytes of the F-PAD field (Byte L-1 and Byte L) are located at the end of the DAB audio frame, following the Scale Factor CRC (ScF-CRC). The X-PAD field is located just before the ScF-CRC. The audio data shall terminate before the beginning of the X-PAD field.

The F-PAD channel carries a two-bit field, "X-PAD Ind", which indicates one of three possibilities for the length of the X-PAD field:

1) No X-PAD: only the F-PAD field is available. All bits in the frame up to the ScF-CRC may be filled with audio data.

2) Short X-PAD: in this case the length of the X-PAD field is four bytes in every DAB audio frame, and the entire X-PAD field lies in the better protected part of the DAB audio frame (i.e. is as well protected as the ScF-CRC). In total, 6 bytes carry PAD.

3) Variable size X-PAD: in this case the length of the X-PAD field may vary from frame to frame. The length of the X-PAD field in the current DAB audio frame can be deduced from the contents information carried within the X-PAD field. Only a part (4 bytes) of the X-PAD field is as well protected as the ScF-CRC. The remainder has a lower protection. Application data carried in the X-PAD channel may require further error protection.
Figure 11: FPAD data structure
3 SYSTEM C LIBRARY

The SystemC library extend C++ to enable the modeling of systems. This extensions permit us to use time sequenced operations, data types for describing hardware, structure hierarchy and simulation support.

<table>
<thead>
<tr>
<th>Methodology-Specific Libraries</th>
<th>Layered Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master/Slave Library, etc.</td>
<td>Verification Library</td>
</tr>
<tr>
<td></td>
<td>Static Dataflow, etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Primitive Channels</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal, Mutex, Semaphore, FIFO, etc.</td>
<td>4-valued Logic type</td>
</tr>
<tr>
<td></td>
<td>4-valued Logic Vectors</td>
</tr>
<tr>
<td></td>
<td>Bits and Bit Vectors</td>
</tr>
<tr>
<td></td>
<td>Arbitrary Precision Integers</td>
</tr>
<tr>
<td></td>
<td>Fixed-point types</td>
</tr>
<tr>
<td></td>
<td>C++ user-defined types</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core Language</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules</td>
<td></td>
</tr>
<tr>
<td>Ports</td>
<td></td>
</tr>
<tr>
<td>Processes</td>
<td></td>
</tr>
<tr>
<td>Interfaces</td>
<td></td>
</tr>
<tr>
<td>Channels</td>
<td></td>
</tr>
<tr>
<td>Events</td>
<td></td>
</tr>
<tr>
<td>Event-driven simulation</td>
<td></td>
</tr>
</tbody>
</table>

Figure 12: SystemC Language Architecture

Figure 12 shows the SystemC language architecture. The blocks with shadow are part of the SystemC language. SystemC is built on standard C++. The layers above or on top of the SystemC standard consist of design libraries and standards considered to be separate from the SystemC core language. Over time other standard or methodology specific libraries may be added and conceivably be incorporated into the core language.

The core language consists of an event-driven simulator as the base. It works with events and processes. The other core language elements consist of modules and ports for representing structure, while interfaces and channels are used to describe communication.

The data types are useful for hardware modelling and certain types of software programming.

A SystemC system consists of a set of one or more modules. Modules provide the ability to describe structure. Modules typically contain processes, ports, internal data, channels and possibly instances of other modules. All processes are conceptually concurrent and can be used to model the functionality of the module. Ports are objects through which the module communicates with other modules. The internal data and channels provide for communication between processes and maintaining module state.
Communication between processes inside different modules is accomplished using ports, interfaces and channels. The port of a module is the object through which the process accesses a channels interface. The interface defines the set of access functions for a channel while the channel itself provides the implementation of these functions. At elaboration time the ports of a module are connected to designated channels. The interface, port, channel structure provides for great flexibility in modelling communication and in model refinement.

Events are the basic synchronization objects. They are used to synchronize between processes and implement blocking behaviour in channels. Processes are triggered or caused to run based on sensitivity to events. Dynamic and static sensitivity are supported. Processes may wait for a particular event or set of events. Dynamic sensitivity coupled with the ability of processes to wait on one or more events provide for simple modelling at higher levels of abstraction and for efficient simulation.

The following describes some concepts used in SystemC that are used in the implementation of the system:

- **Modules**: SystemC has a notion of a container class called a module. This is a hierarchical entity that can have other modules or processes contained in it.
- **Processes**: Processes are used to describe functionality. Processes are contained inside modules. SystemC provides three different process abstractions to be used by hardware and software designers.
- **Ports**: Modules have ports through which they connect to other modules. SystemC supports single-direction and bidirectional ports.
- **Signals**: SystemC supports resolved and unresolved signals. Resolved signals can have more than one driver (a bus) while unresolved signals can have only one driver.
- **Rich set of port and signal types**: To support modelling at different levels of abstraction, from the functional to the RTL, SystemC supports a rich set of port and signal types. This is different than languages like Verilog that only support bits and bit-vectors as port and signal types. SystemC supports both two-valued and four-valued signal types.
- **Rich set of data types**: SystemC has a rich set of data types to support multiple design domains and abstraction levels. The fixed precision data types allow for fast simulation, the arbitrary precision types can be used for computations with large numbers, and the fixed-point data types can be used for DSP applications. SystemC supports both two-valued and four-valued data types. There are no size limitations for arbitrary precision SystemC types.
- **Clocks**: SystemC has the notion of clocks (as special signals). Clocks are the timekeepers of the system during simulation. Multiple clocks, with arbitrary phase relationship, are supported.
- **Cycle-based simulation**: SystemC includes an ultra light-weight cycle-based simulation kernel that allows high-speed simulation.
• Multiple abstraction levels: SystemC supports untimed models at different levels of abstraction, ranging from high-level functional models to detailed clock cycle accurate RTL models. It supports iterative refinement of high level models into lower levels of abstraction.

• Communication protocols: SystemC provides multi-level communication semantics that enable you to describe SoC and system I/O protocols with different levels for abstraction.

• Debugging support: SystemC classes have run-time error checking that can be turned on with a compilation flag.

• Waveform racing: SystemC supports tracing of waveforms in VCD, WIF, and ISDB formats.

3.1 Events and dynamic sensitivity

In SystemC, dynamic sensitivity allows for making processes sensitive to events. We have a set of methods that shall permit to us to realize that, they are described bellow:

3.1.1 WAIT() Method

Sensitivity lists are defined in the constructor of a module. Consider the following example.

Example 5-1: Static sensitivity.
SC_MODULE( my_module )
{
    // ports
    sc_in<int> input;
    sc_in_clk clock;
    // processes
    void proc_a();
    void proc_b();
    // constructor
    SC_CTOR( my_module )
    {
        SC_THREAD( proc_a );
        sensitive_pos << clock;
        SC_THREAD( proc_b );
        sensitive << input;
        sensitive_neg << clock;
    }
};

In the above example, there are two thread processes in the module. Process proc_a is sensitive to the positive edge of the clock, whereas process proc_b is sensitive to a change in value on input and to a negative edge of the clock.

In some cases, we want a process to be sensitive to a specific event or a specific collection of events, and this may change during simulation. This dynamic sensitivity is possible by using the wait() method. This method has
been extended to allow specifying one or more events or a collection of events to wait for. For example:

**Dynamic sensitivity with the wait() method.**

```c++
... // wait until event e1 has been notified
wait( e1 );
...
// wait until event e1 or event e2 has been notified
wait( e1 | e2 );
...
```

The wait() method can be called anywhere in the thread of execution of a thread process.

When it is called, the specified events will temporarily overrule the sensitivity list, and the calling thread process will suspend. When one (or all) of the specified events is notified, the waiting thread process is resumed. The calling process is again sensitive to the sensitivity list.

In addition to events, it is also possible to wait for time, using the SystemC time types. This can be used, as a timeout when waiting for one or more events.

```c++
// wait for 200 ns.
wait( 200, SC_NS );

// wait on event e1, timeout after 200 ns.
wait( 200, SC_NS, e1 );

// wait on events e1, e2, or e3, timeout after 200 ns.
wait( 200, SC_NS, e1 | e2 | e3 );

// wait on events e1, e2, and e3, timeout after 200 ns.
wait( 200, SC_NS, e1 & e2 & e3 );
```

### 3.2 EVENT TYPES

SystemC provides a fixed set of channels and corresponding events. To support user-defined channel types, the set of events must also be extendable.

To that purpose, it’s introduced the event type `sc_event`.

The event type `sc_event` provides the following functionality.

- **Constructor:** An event object can be created by calling the constructor without any arguments.
  ```c++
  sc_event my_event;
  ```

- **Notify:** An event can be notified by calling the (non-const) notify() method of the event object.
  ```c++
  my_event.notify(); // notify immediately
  my_event.notify( SC_ZERO_TIME ); // notify next delta cycle
  my_event.notify( 10, SC_NS ); // notify in 10 ns
  ```
In addition, functions are provided allowing a functional notation for notifying events.

```
sc_time t( 10, SC_NS );
my_event.notify( t ); // same
```

- **Cancel**: An event notification can be cancelled by calling the cancel() method of the event object.

```
my_event.cancel(); // cancel a delayed notification
```

A channel can construct any number of event objects – one for each type of event it can generate. A channel can notify an event by calling one of the notify methods of the event object.

### 3.3 INTERFACES

An interface provides a set of method declarations, but provides no method implementations and no data fields. Interfaces are used to define sets of methods that channels must implement.

Ports are connected to channels through interfaces. A port that is connected to a channel through interface sees only those channel methods that are defined by the interface.

#### 3.3.1 Interface base class

All interfaces are derived from base class `sc_interface`. This class defines a method `register_port()`, which can be used by channels to do static design rule checking when binding ports to channels.

The default behavior of this method is to do nothing. The interface base class also defines a method `default_event()`, which can be used by channels to return the default event for static sensitivity. The default behavior of this method is to return a reference to an event that is never notified. The implementation of the `sc_interface` base class is depicted in next page:

---

**Pseudo code implementation of the interface base class.**

```
class sc_interface
{
  public:
  // register a port with this interface (does nothing by default)
```
virtual void register_port( sc_port_base&, const char* ) {}  
// get the default event
virtual const sc_event& default_event() const;
// destructor (does nothing)
virtual ~sc_interface() {}  
};
4 IMPLEMENTATION OF THE SYSTEM:

Target of this design is to obtain the DAB frame that it can be sent to the transmission layer for be transmitted in a radiocommunicaton system. To obtain this frame I have divided the implementation in two phases:

- The mission of the first one is to obtain the DAB audio frame, by the conversion of MPEG frames in DAB audio frames. A flow chart of this process is showed below.

Figure 13: DAB Audio frame generation scheme

- Once that we have DAB audio frames constructed we have to create the FIC and MSC frames. A group of DAB audio frames form a MSC frame that a big frame that need a buffer of DAB audio frames for be formed. On other hand FIC is generated independent of DAB audio, and it’s made up FIB blocks, it’s have to carry information about the transmission, f e Conditional Access information or interleaving information. These services are not implemented and FIC is generated randomly. A flow chart of this phase is showed below.

Figure 14: DAB transmission frame generation scheme

4.1 FIRST FASE: DAB AUDIO FRAME OBTENTION

Later to get a good vision of the DAB system and the SystemC library is time to develop a implementation of the system. Initially it is necessary to have a MPEG source, the MPEG source contains all the audio information to be carried, although DAB system is designed for
transmit different sources of MPEG this implementation of the system can carry only one source.

The best way to get the MPEG audio frames is by means of the MPEG audio libraries, there are several implementation to construct a MPEG decoder, this implementation are able to get a MPEG file and while it is reading the file the MPEG frames can be sended to a high level layer for example, like the audio filter decoder, to play his audio info. This implementations are very complicated of use for this case, and the use of this is very far from the real case of study of this project.

It is for that, that I used a random data generator of MPEG files, the data structure can be carry data sound of the same way that MPEG decoders, but this info is completely unuseful. The data structure of MPEG class is like the next:

**Header structure:**

```c
typedef struct mpeg_headerf{
  bool sync[11];
  bool mpeg_ver[1];
  bool layer_description[1];
  bool *protection_bit;
  bool *bitrate_index;
  bool *sample_frecuency;
  bool *padding_bit;
  bool *private_bit;
  bool *channel_mode;
  bool *mode_extensions;
  bool *copyright;
  bool *original;
  bool *emphasis;
  bool *crc;
}mpeg_header;
```

**Data structure:**

```c
typedef struct mpeg_datat{
  bool *bit_allocation;
  bool *scfsi;
  bool * scale_factor;
  bool * samples;
  bool ancillary_data;
}mpeg_data;
```

**Mpeg audio frame that contains a mpeg_header and a mpeg_data structures:**

```c
typedef struct mpeg_audiot{
  mpeg_header header;
```
For this purpose of generate the MPEG audio frame I created a class with the name of Mpeg, it has to different constructors, for one hand the a random generator constructor, and for another hand one that takes parameters for generate the MPEG frames.

Mpeg::Mpeg();
Mpeg::Mpeg(mpeg_audio a);

The number of bits that can be carried in a MPEG frame for audio sample depends of the sample frequency used and the bit rate that we want to use (in chapter XX is explained the possible cases of sample frequency and bit rate), is necessary for this to implement 2 functions to get them a calculate the number of bits in the samples.

int Mpeg::get_sample_frecuency();
int Mpeg::get_bit_rate();

Also I have to implement and overload of “=” operator and a function useful for can print the mpeg data frame, this function can be used for example to write the data to a file, but now only print the data in the screen.

void Mpeg::show();
Mpeg Mpeg::operator=(mpeg_audio a);

Now, with MPEG frames we can start to construct the DAB audio frames, basically DAB uses MPEG layer II whit some modifications, the principal modification is that DAB is able to transmit more information that MPEG, this can be audio or non audio information, for this purpose exists the FPAD field, that can be expanded with XPAD field (not implemented in this case).

PAD or Program Associated Data is explained in XXX, and it can transmit different kind of data, in instead of the type data field, because the audio information of mpeg is not legible, PAD information and services are generated from a random process.

For create the PAD frames like the MPEG implementation It’s created a new class with the name of Fpad, this class has the same functionalities that Mpeg class, below is wrote the data structure used to implement PAD and the more important functions implemented:

typedef struct fpad_t
{
    bool byteL1[7];
    bool byteL[7];
}fpad_f;

fpad::fpad();
fpad::fpad(fpad_f a);
Once that we have implemented Mpeg and Fpad classes we need a method to join his data structures in one data structure the best way to do it, is create a new class that has heritance of the both classes, this way this class has the same methods of this classes and the same data structures. This class is named Dab, below it is his declaration.

```cpp
class Dab: public Mpeg, public fpad{
  public:
    int size;
    Dab();
    Dab(mpeg_audio a, fpad_f b);
    Dab::~Dab();
    void Dab::show_dab();
};
```

Using the constructors and method of Mpeg and Fpad classes it is very easy to create the functions declared, for example this is the constructor using parameters of Dab class.

```cpp
Dab::Dab(mpeg_audio a, fpad_f b):Mpeg(a),fpad(b) {
}
```

Summarizing, now we have the facilities to obtain a DAB audio frame from a MPEG audio frame, with Mpeg, Fpad and Dab classes. We have to think now that DAB system is a syncronizated system and if we want that the audio sound will be continuous we have to do that all the frames are sent every 24 ms, is time to use the SystemC functionalities.

To obtain this I have developed a FIFO list with SystemC library, this fifo list consist in a buffer of MPEG audio frames that are in a producer process, the producer process send signals to a consumer process and vice versa, the consumer process get a MPEG frame every 24 ms and it converts it to a DAB audio frame. This DAB audio frame will be sent to the next layer to obtain the MSC, but I talk about it later.

For implement this with SystemC I have create two sc_interfaces one for write and one for read, write interface will be used in producer process and read interface will be used in consumer process. This interfaces are used for connect a channel that is called fifo, this channel defines the method that are declarated in the interfaces.

There are tree modules that use this interfaces, the producer module has out port, while the consumer module has a in port, one have to transmit the frames
to another. The third module is the top module it has the function of join all the modules with the channel, this way a scheme of the system would be this below:

![SystemC Communication Diagram]

**Figure 15: Communication between modules in SystemC**

The Read_event.notify() and Write_event.notify() are signals that are used to communicate and coordinate the process of write and read. The consumer read a data every 24 ms, and we obtain our initially target of read a MPEG frame by syncronized way.

Below there is some interesting things in the implementation of the system together his explanations:

**Write_if**

```cpp
class write_if : virtual public sc_interface
{
    public:
    virtual void write(Mpeg) = 0;
    virtual void reset() = 0;
};
```

**Read_if**

```cpp
class read_if : virtual public sc_interface
{
    public:
    virtual void read(Mpeg ) = 0;
    virtual int num_available() = 0;
};
```
class fifo : public sc_channel, public write_if, public read_if
{
    public:
    fifo(sc_module_name name, int size_) : sc_channel(name), size(size_)
    ~fifo()
        void write(Mpeg c)
        void read(Mpeg c)
        void reset() { num_elements = first = 0; }  
        int num_available()
    private:
        void compute_stats()
};

class producer : public sc_module
{
    public:
        sc_port<write_if> out;

        SC_HAS_PROCESS(producer);

        producer(sc_module_name name) : sc_module(name)
        {
            SC_THREAD(main);
        }

        void main()
        {
            //Code that write mpeg frames in a buffer
        }
}

class consumer : public sc_module
{
    public:
        sc_port<read_if> in;

        SC_HAS_PROCESS(consumer);

        consumer(sc_module_name name) : sc_module(name)
        {
            SC_THREAD(main);
        }

        void main()
        {
            //This code read a frame from the buffer every 24 ms, while the buffer is not empty
        }
};
class top : public sc_module
{
    public:
        fifo fifo_inst;
        producer prod_inst;
        consumer cons_inst;

top(sc_module_name name, int size) :
    sc_module(name),
    fifo_inst("Fifo1", size),
    prod_inst("Producer1"),
    cons_inst("Consumer1")
    {
    prod_inst.out(fifo_inst);
    cons_inst.in(fifo_inst);
    }
};

The form to start the system is very easy, we need only create a element of the top class, and initialize the the system:

top top1("Top1", size);
sc_start(-1);

4.2 SECOND FASE: DAB TRANSMISION FRAME OBTENTION

Before we have seeing that a transmission frame of DAB audio system is formed by two different data fields, in first place we have the FIB frame, that have information about the transmission, and in second place we have the MSC_Group that have the data information, in our case of study we have only audio information inside this frame.

This two frames are formed for a stream of another frames, for the case of FIB, we have a FIB header followed for FIC frames, that can be of different types (page XXX), and padding in case that the FIB frame was not completely full. For implement the FIB frames I have created a class named Fib.

This Fib class has a randomly constructor that initially generate the header and later start to create randomly FIC frames, if this frames fit in the frame they are added, in case that not, the frame is not a added and we fill the rest of the frame with padding information.
Data structure used in the class is showed below, basically it consist in Boolean arrays for the header and in a pointer array of FIC elements. FIC elements are defined in a data structure and for increase the system requirements of the system I used dynamic memory allocation for add the FIC frames.

```cpp
typedef struct figt {
    bool fig_type[2];
    int length; // length in bytes of fic it's a number between 1-29 bytes
    bool *fig_data_field;
}fig;

typedef struct fib_framet {
    fig * fig_frames;
    int number_of_fig_frames;
    int sizeof_fig_frames;
    bool * padding;
    bool * crc;
    bool marker[6];
}fib_frame;
```

I only implement for this class constructor, destructor and for can test that frames are created correctly a function to visualize the frames.

```cpp
Fib::Fib();
Fib::~Fib();
void Fib::show();
```
The constructor can be explained with this basic pseudo-code:

- First generate the Fig structures, the number total of bytes it has to be minor than 30 bytes.
- Each Fig structure has a maximum length of 29 bytes and a minimum length of 1 byte.
- The Fig structures can be of 4 different types:
  - TYPE 0: 000 | XXXXX | C/N | OE | P/D | Extension (5 bits) | Type 0 field
  - TYPE 1: 001 | XXXXX | Charset (4 bits) | OE | Extension (3 bits not used: 111) | Type 1 field
  - TYPE 5: 101 | XXXXX | D1 | D2 | TCId (3 bits) | Extension (3 bits not used: 111) | Type 5 field
  - TYPE 6: 110 | XXXXX | First / Last (2 bits) | Extension (6 bits not used: 111111) | Type 6 field
- Construction of fig_frames array [Dynamic memory assignation (malloc, realloc)]
- While exit!=1
  - Generate a number between 1-29
  - sizeof_fig_frames = sizeof_fig_frames + This_number
  - if sizeof_fig_frames > 29
    - sizeof_fig_frames = size_fig_frames - This_number
    - Padding bits = (29 - sizeof_fig_frames)*8
    - Exit = 1
  - Else
    - a number between 1-4
    - Case 1: Type 0
    - Case 2: Type 1
    - Case 3: Type 5
    - Case 4: Type 6

Later of that we are able to create the FIB frames, this frames are sent along with the MSC frame, in the next part of this document I am going to explain how is implemented the MSC.

MSC is a group of information frames, in this case only audio frames, the MSC is a macrostructure that can contain 55256 bits of information, this information is divided in frames and in the case of audio one of this frames correspond to the whole DAB audio frame. It means that one MSC frame can contain more than 40 DAB audio frames.

Since we have DAB audio frames that are arriving to the system each 24 ms we need a method for packet this frames into the MSC, this have been implemented using a function that add DAB audio frames into the MSC and return a value depending of if the MSC has space or not for add this frame.

The constructor function only generate by randomly way the MSC header and reserve memory for the first DAB audio frame.

CRC is generated later have the complete MSC created.
Because we have a limit of space in the MSC_group frame it is necessary to know exactly in every moment the available memory that we have, for this purpose it is created a function in mpeg class that permit us to know the information carried in the frame. This function can be used from the dab class because if we remember class dab has inheritance from mpeg and fpad classes. The total bits of the DAB frame is the value returned for this function plus 16 that is the information carried for FPAD, in case that we implemented XPAD we need a function to calculate the value of number of bits of XPAD.

For generate the MSC_Group I have created a class named Msc_Group, this class have the next data structures:

```c
typedef struct msc_packet_header {
    bool * packet_lenght;
    bool * continuity_index;
    bool * first_last;
    bool * address;
    bool command;
    int useful_data_length;
} msc_packet_header;
```

```c
typedef struct msc_group_struct {
    msc_packet_header header;
    bool * crc;
    //dab_struct * dab_stream;
    struct mpeg_audiot * mpeg;
    struct fpad_t * fpad;
    int bits;
    int number_of_dabs;
} msc_group_struct;
```

The structures mpeg_audiot and fpad_t are used in the mpeg and fpad classes respectively. I am implemented the next functions in the class

```c
Msc_Group::Msc_Group();
Msc_Group::~Msc_Group();
int Msc_Group::add_dab_frame(struct mpeg_audiot a, struct fpad_t b);
void Msc_Group::calculate_crc();
```

Constructor only generate the header of the Msc_Group and reserve memory for the first DAB audio frame element.

Calculate_crc function is used to generate the CRC, for maintain concordance crc it is not created in the constructor because it have to be calculated from all the frame, is for this, that we generate the crc when we have completed the MSC frame.

The add_dab_frame function has like mission to add the DAB frames to the MSC_Group. We can obtain those structures with two functions created for this purpose in mpeg and fpad classes, this functions are:
On another hand the add_dab_frame function return a int value this value will be 0 if the frame is correctly added and return a 1 value if there is no free space for add the frame, in this case we have to send the frame to a high level layer and create a new Msc_Group.

The add_dab_frame initially check if there is free space available and in case that there is copy both structures to the Msc_Group, reserve memory for the next element and return 0 value, in opposite case it return 1.

```c
int Msc_Group::add_dab_frame(struct mpeg_audiot a, struct fpad_t b)
{
    //Calculate free space in msc_group (CIF is 55296 bits)
    // if there is free space add_frame return 0 and realloc memory
    // for the next dab frame;
    // if there is not free space return 1;

    int size_of_dab;
    size_of_dab=a.bits_size;
    printf("Tamaño de la trama: %d",size_of_dab);
    if ((msc_group.bits + size_of_dab)>55296)
    {
        //there is not free space
        return 1;
    }
    else
    {
        //insert the dab frame in the stream
        printf("ADDING DAB FRAME...\n");
        // REST OF THE CODE FOR COPY FRAMES
        //once that frames are added we need to reserve memory for
        //the next frame and increase the variables
        msc_group.bits=msc_group.bits+size_of_dab;
        msc_group.number_of_dabs++;
        msc_group.mpeg=(mpeg_audio *)realloc(msc_group.mpeg,
        (msc_group.number_of_dabs + 1)*sizeof(sizeof(mpeg_audio)));
        msc_group.fpad=(fpad_f *)realloc(msc_group.fpad,
        (msc_group.number_of_dabs + 1)*sizeof(sizeof(fpad_f)));
        printf("Number of bits: %d Number of frames %d\n",
        msc_group.bits,msc_group.number_of_dabs);
        return 0;
    }
}
```

To finalize the explanation of the system is time to see how the frames are arriving to the system and how we construct the transmission frame.

We do it in the main file of the system in the SystemC interface, exactly in the consumer module, it is receiving de dab frames in the execution each 24 ms, when a DAB frame arrive to the module, we extract the structures of the frame
(mpeg_audio, fpad_f) and with them we use the function add_dab_frame for add this frame to the MSC_Group.

This is implemented of the next form:

```c
    c = data[first];
    fpad fpad1;

    mpeg_audio audio;  //EXTRACT STRUCTURES
    c.get(audio);

    fpad_f pad;
    fpad1.get(pad);

    Dab dab1(audio,pad);
    dab1.show();
    dab1.show_fpad();

    out=Msc_group_list[i].add_dab_frame(audio,pad); //ADD FRAME
    if (out==1) //If there is not added we reset the system for
             {// for the next MSC_Group Frame
           i++;
           number_of_msc_groups++;
           printf ("NUMBER: %d",number_of_msc_groups);
           Msc_group_list[i-1].calculate_crc();
           Fib Fib1;

           //send(Msc_group_list[i-1],Fib1);

           printf("Added %d DAB audio frames in
                  Group_MSC[%d]",counter,i-1);
           counter=0;
        }
        counter++;
```

This Fib class has a randomly constructor that initially generate the header and later start to create randomly FIC frames, if this frames fit in the frame they are added, in case that not, the frame is not a added and we fill the rest of the frame with padding information.
5 RESULTS AND CONCLUSIONS

This section contains a critical review of the implementation in relation to the assumptions of the project.

The system was very hard to implement, not for the fact of that System C is difficult, but yes for the fact of that DAB system is a very complex system, and documentation about it is not excessive.

Finally I could implement a DAB system based in SystemC Library, but I have to recognize that this system is very limited, there are several functionalities that are not implemented in the system like CRC generation or convolutional code process of the signal, because this process are enough extensive to make another Thesis Project, however I start reading information about it.

On another hand, the MPEG fed of the system, is based in a randomly generation, in the first of the project I tried to use MPEG libraries to fed MPEG frames in the system from a MPEG file, I was studding two different implementations of this, but it was very difficult to use it because the data structures of this implementations are different of my, although the information was the same and use this would make that I create a process to convert different data structures or that I use this structures in my implementation.
ANEX A INSTALLATION OF SYSTEMC LIBRARY

System Requirements For install SystemC in a Linux operative system:

- Linux (Redhat 7.2) with GNU C++ compiler version gcc-2.95.3, I used Suse 9.0 distribution and it is valid for installation.
- GNU C++ compiler version gcc 2.95.3 or higher.
- GNU Make (gmake), you can get gmake from the next url:

To install SystemC on a Linux, do the following steps:

1. Change to the top level directory (systemc-2.0.1)
2. Create a temporary directory, e.g.,
   > mkdir objdir
3. Change to the temporary directory, e.g.,
   > cd objdir
4. Set the following environment variable(s):
   > setenv CXX g++
5. Configure the package for your system, e.g.,
   > ../configure
   While the 'configure' script is running, which takes a few moments, it prints messages to inform you of the features it is checking. It also detects the platform.
6. Compile the package.
   > gmake
7. Install the package.
   > gmake install
8. You can now remove the temporary directory, .e.g,
   > cd ..
   > rm -rf objdir
   Alternatively, you can keep the temporary directory to later uninstall the package. To clean up the temporary directory, enter:
   > gmake clean
   To uninstall the package, enter:
   > gmake uninstall
ANEX B SYSTEMC INTERFACE EXAMPLES

Below, the implementations of three example interfaces are given. The first example interface defines a read interface. The second example interface defines a write interface. Both the read interface and the write interface derive from interface base class `sc_interface`, which is described in the next section. The third example interface defines a read/write interface by deriving from the read interface and write interface. Note that all interface methods are pure virtual methods, i.e., they don’t have an implementation.

Example 7-1: A read, write, and read/write interface.

```cpp
// -----------------------------------------
// An example read interface: sc_read_if
// this interface provides a `read` method
// -----------------------------------------
template <class T>
class sc_read_if
: virtual public sc_interface
{
public:
// interface methods
virtual const T& read() const = 0;
};
// ------------------------------------------
// An example write interface: sc_write_if
// this interface provides a `write` method
// ------------------------------------------
template <class T>
class sc_write_if
: virtual public sc_interface
{
public:
// interface methods
virtual void write( const T& ) = 0;
};
// ---------------------------------------------------
// An example read/write interface: sc_read_write_if
// ---------------------------------------------------
template <class T>
class sc_read_write_if
: public sc_read_if<T>,
public sc_write_if<T>
{};
```

The read interface provides a read() method. The write interface provides a write() method. The read/write interface is derived from the read interface and the write interface. This allows, for example, connecting a port with a read interface to a channel with a read/write interface, but not vice versa.

LAYERED COMMUNICATION EXAMPLE

The concept of interfaces is very useful when we want to model layered communication. At present, this kind of communication is implemented, e.g., in
the On Chip Bus (OCB) defined by VSIA. In such a communication model, there are several layers of abstraction. For instance, in the OCB backbone there are three levels: transaction, packet, and cell. Using interfaces, we can define three types as shown in Example 7-2. With this approach, we can define in an Optimum way the information that must be considered at each level of abstraction. For instance, the information carried by “EOP5” is meaningful at packet level, but is not visible at transaction level. Hence, the user cannot specify this information when he is using the OCB_trans_if interface.

Example 7-2: Layered interfaces for the OCB backbone.

// pseudo code; some parts are not shown
// INTERFACE : OCB_cell_if
template <class T>
class OCB_cell_if
: virtual public sc_interface
{
public:
// interface methods
... 
virtual void set_eom( bool ) = 0;
virtual bool get_eom() = 0;
virtual void set_eop( bool ) = 0;
virtual bool get_eop() = 0;
};

// INTERFACE : OCB_packet_if
template <class T>
class OCB_packet_if
: public OCB_cell_if
{
public:
// interface methods
...
};

// INTERFACE : OCB_trans_if
template <class T>
class OCB_trans_if
: public OCB_packet_if
{
public:
// interface methods
...

virtual void set_eom( bool ) = 0;
virtual bool get_eom() = 0;
private:
// not to be defined
virtual void set_eop( bool );
virtual bool get_eop();
};
Looking at the implementation, all necessary methods are defined pure virtual inside the base class, OCB_cell_if. These methods are accessible both by OCB_packet_if and OCB_trans_if. Since some of the methods are meaningless at certain levels of abstraction, it is necessary to declare them private. For instance, the methods get_eop() and send_eop() are declared private in OCB_trans_if.

The layered approach, modelled in SystemC by using interfaces, allows also interconnecting a module at different levels of abstraction. In this way, we can attach either modules that need a very close control around the clock (i.e., at cell level), or modules that work at transaction level.

Hence, the concept of interfaces provides all the necessary freedom to the user to model its intellectual property (IP) at different levels of abstraction.
Classes in object-oriented programming attempt to model real world entities. Just as an entity has properties and abilities, a class has data members and methods. Both entities in the real world, and classes in the worlds our programs create have various relationships. These relationships help to simplify the task of understanding the systems we create and the entities of which they are constructed.

One such relationship is inheritance. In C++, inheritance defines an "is a" relationship. The derived class is a type of its base class. For instance, a "cat" class could be derived from an "animal" class. A cat is an animal. A cat is a type of animal. A derived class inherits both the data members and methods of its base class. It may also define additional members and methods that support specialized functionality. All of this is best understood by studying some simple examples.

**Definition:** A derived class inherits members from its base class. The members and methods of the base class will become part of the derived class. The type of inheritance, public, private or protected, determines access to the inherited members and methods in the derived class.

**Inheritance**

Let's see how inheritance is implemented in C++. As an example, let's develop classes and code to model the twenty plus pets that live with me in my small basement apartment.

```cpp
#include <iostream>
#include <string>
using namespace std;

class Pet
{
    // Constructors, Destructors
    Pet(): weight(1), food("Pet Chow") {}
    ~Pet() {}

    //Accessors
    void setWeight(int w) {weight = w;}
    int getWeight() {return weight;}
    void setfood(string f) {food = f;}
    string getFood() {return food;}

    //General methods
    void eat();
    void speak();

    protected:
        int weight;
        string food;

};
```
void Pet::eat()
{
    cout << "Eating " << food << endl;
}

void Pet::speak()
{
    cout << "Growl" << endl;
}

class Rat: public Pet
{
public:
    Rat() {}
    ~Rat() {}

    //Other methods
    void sicken() {cout << "Spreading Plague" << endl;}

};

class Cat: public Pet
{
public:
    Cat() : numberToes(5) {}
    ~Cat() {}

    //Other accessors
    void setNumberToes(int toes) {numberToes = toes;}
    int getNumberToes() {return numberToes;}

private:
    int numberToes;
};

int main()
{
    Rat charles;
    Cat fluffy;

    charles.setWeight(25);
    cout << "Charles weighs " << charles.getWeight() << " lbs. " << endl;
    charles.speak();
    charles.eat();
    charles.sicken();

    fluffy.speak();
    fluffy.eat();
    cout << "Fluffy has " << fluffy.getNumberToes() << " toes " << endl;

    return 0;
}
Let's examine this example. There is a Pet base class and two derived classes, Rat and Cat. Notice the syntax in the class declarations of the derived classes.

class DerivedClass : public BaseClass {
    ....
};

The keyword after the colon indicates the type of inheritance. Public inheritance is type inheritance. The subclass is a type of the base class. It inherits the public interface of the base class and can extend this interface. Next, notice that the data members of the Pet base class are declared as protected. Protected indicates that publicly derived subtypes of the base class will be able to directly access these variables. If they were declared to be private, only the Pet class could directly access them; subclasses could not. If they were declared public, any class or part of code could accessed them. This would defeat a key goal of object-oriented design: encapsulating data within a class and exposing the data only through the public interface. To summarize, the parts of a class that are declared public can be accessed by the class itself, publicly derived subtypes, and other classes, functions and code (the whole world). The parts of a class declared as protected can be accessed by the class itself and by publicly derived subtypes. The parts of a class declared as private can be accessed only by the class itself. C++ also allows two other types of inheritance, private and protected, which are less often used and will be discussed elsewhere. Public inheritance, as shown in the example, usually suffices.

Notice that the Rat and Cat objects inherit the methods of the base class, Pet. We can call the speak method and the eat method on objects of type Rat and Cat. These methods were defined only in Pet and not in the subclasses. Also, notice the each subclass extends the base class by adding methods and members. The Rat class has the "sicken" method. The Cat class has methods and members related to the number of toes an individual cat object has. I added this in because I have seen strange inbred cats with six and seven toes. The subtypes are more specialized than the base class. The base class contains common members and methods used by the subclasses.

The last thing to notice in this example is that constructors, including copy constructors, and destructors are not inherited. Each subtype has its own constructor and destructor. The reason for this is that each object of a subtype consists of multiple parts, a base class part and a subclass part. The base class constructor forms the base class part. The subclass constructor forms the subclass part. Destructors clean up their respective parts. To show this, let's modify the previous example to include print statements in constructors and destructors to better see how objects are created and destroyed.

#include <iostream>
#include <string>
using namespace std;

class Pet {
    public:
// Constructors, Destructors
Pet(): weight(1), food("Pet Chow")
{
    cout << "Pet Constructor" << endl;
} ~Pet()
{
    cout << "Pet Destructor" << endl;
}

// Rest of code unmodified from first example ......

};

class Rat: public Pet
{
public:
    Rat()
    {
        cout << "Rat Constructor" << endl;
    } ~Rat()
    {
        cout << "Rat Destructor" << endl;
    }

    // Rest of code unmodified from first example ......
};

class Cat: public Pet
{
public:
    Cat(): numberToes(5)
    {
        cout << "Cat Constructor" << endl;
    } ~Cat()
    {
        cout << "Cat Destructor" << endl;
    }

    // Rest of code unmodified from first example ......
};

int main()
{
    Rat charles;
    Cat fluffy;

    //charles.setWeight(25);
    //cout << "Charles weighs " << charles.getWeight() << " lbs. " << endl;
    //charles.speak();
    //charles.eat();

    return 0;
}
// charles.sicken();

// fluffy.speak();
// fluffy.eat();
// cout << "Fluffy has " << fluffy.getNumberToes() << " toes" << endl;

return 0;
}

Notice that first the Rat object, charles, is created.
  Pet Constructor
  Rat Constructor

Then the Cat object, fluffy, is created.
  Pet Constructor
  Cat Constructor

Then the Cat object is destructed.
  Cat Destructor;
  Pet Destructor;

Then the Rat object is destroyed
  Rat Destructor;
  Pet Destructor;
Implementation of Digital Audio Broadcasting System based on SystemC Library

Title
Eduardo Moreno Martínez

Abstract
This thesis describes the design and implementation of a Digital Audio Broadcasting (DAB) System developed using C++ Language and SystemC libraries. The main aspects covered within this report are the data structure of DAB system, and some interesting points of SystemC Library very useful for the implementation of the final system. It starts with an introduction of DAB system and his principals advantages. Next it goes further into the definition of data structures of DAB, they are FIC, MSC, and DAB audio frame, explained with MPEG and PAD packets. Later on this chapter there is an explanation of the SystemC library with special attention on the features that I used to implement the system. This features are the events used in the communication between processes and the interfaces needed for sending and receiving the data. With all these points covered it is quite easy for a reader to understand the implementation of the system, despite this point is covered in the last chapter of the thesis. The implementation is here explained in two different steps. The first one explain how is formed the DAB audio frame by means of MPEG frames that are wrote in channel by producer interface, these frames are read by consumer interface. For this purpose I have created some classes and structures that are explained in this part. The second part explain how I obtain the DAB transmission frame which is obtained creating MSC frames, that are big data structures formed by groups of DAB audio frames, therefore there are some functions that act like a buffer and add audio frames to the MSC data structure. Of independent way there is the FIC frame that is generated of random way and its added to the transmission frame.

Keyword
Digital audio broadcasting system, DAB, SystemC System C Library, C++ programming, MPEG audio frame layer II
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en länge tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydda mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Eduardo Moreno Martinez