MASTER THESIS

TITLE: Development of embedded transmitter for visible light communication system for indoor environment

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTHOR: Judit Bravo Albà

DIRECTOR: Ciprian Gavrincea

SUPERVISOR: Luis Alonso Zárate

DATE: October 27th, 2014
Overview

This project is part of a more extensive project developed by Centre Tecnològic de Telecomunicacions de Catalunya (CTTC). CTTC has developed a software defined Visible Light Communications (VLC) system based on 802.15.7 standard for low-speed communications in indoor environments.

The aim of this master thesis is to design and develop a VLC transmitter suitable for integration on microcontroller based embedded platforms. The main challenge is to optimize and adapt the previous software in order to cope with the scarce computational resources available on embedded systems.

To achieve the end, several assignments have been performed. First, a study of existing technologies, taking into accounts the advantages and shortcomings. Secondly, we focused in detail in the physical layer and after that, we introduced the communication frame structure and architecture based on the IEEE 802.15.7 Standard. Then, we implement the processing blocks using high level programming language. And, finally, the developed software have been integrated and tested on an embedded platform form NXP based on M3 ARM microcontroller.
TÍTOL: Desenvolupament d’un transmissor incorporat per al sistema de comunicació de llum visible per a entorns interiors

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTORA: Judit Bravo Albà

DIRECTOR: Ciprian Gavrincea

SUPERVISOR: Luis Alonso Zárate

DATA: 27 d’octubre de 2014

Resum:

Aquest projecte forma part d’un projecte més extens desenvolupat pel Centre Tecnològic de Telecomunicacions de Catalunya (CTTC). CTTC ha desenvolupat un software definit com a Visible Light Communication system (VLC) basat en l’estàndard 802.15.7 per a baixa velocitats en ambients interiors.

L’objectiu principal d’aquesta tesi és dissenyar i desenvolupar un transmissor VLC adequat per a ser integrat en plataformes embegudes basades en microcontroladors. El repte principal és optimitzar i adaptar el programari anterior per tal de fer front als escassos recursos computacionals disponibles en sistemes encastats.

Per aconseguir aquest objectiu, s’han realitzat diverses tasques. En primer lloc, un estudi de les tecnologies existents, tenint en compte els avantatges i inconvenients. En segon lloc, ens centrem en detall a la capa física i després d’això, presentem l’estructura de la trama de comunicació i l’arquitectura basada en l’estàndard IEEE. Tot seguit, hem implementat en llenguatge de programació d’alt nivell. I, finalment, el programari desenvolupat s’ha integrat i provat en una plataforma integrada de NXP basada en el microcontroaldor M3 ARM.
Thanks to my thesis director Ciprian Gavrinca for guiding me during this project.

Thanks to my family for all their support.
INDEX

INTRODUCtion ................................................................. 9

CHAPTER 1. IEEE 802.15.7 STANDARD ................................. 13

1.1 FRAME STRUCTURE ..................................................... 14
   1.1.1 Preamble ............................................................ 15
   1.1.2 PHR ................................................................. 18
   1.1.3 PSDU ............................................................... 20

1.2 ARCHITECTURE .......................................................... 22
   1.2.2 RS encoder ......................................................... 24
   1.2.3 Interleaving and puncturing block ............................ 26
   1.2.4 Convolutional encoder .......................................... 26
   1.2.5 RLL encoder ...................................................... 28

CHAPTER 2. PRACTICAL IMPLEMENTATION ON EMBEDDED DEVICES. 29

2.1 SYSTEM ARCHITECTURE ............................................... 29
   2.1.1 Preamble ............................................................ 30
   2.1.2 PHR ................................................................. 30
   2.1.3 PSDU ............................................................... 31

2.2 CRC ........................................................................... 31

2.3 Convolutional encoder ................................................. 32

2.4 Interleaving and Puncturing ........................................... 38

2.5 Reed-Solomon Encoder (RS) .......................................... 42

2.6 Run-Length Limited (RLL) ............................................ 43

CHAPTER 3. SYSTEM INTEGRATION ...................................... 45

3.1 Board ....................................................................... 45
   3.1.1 Board Specifications .............................................. 45
INTRODUCTION

In the last century, technology has been suffered an exponential growth, entering into new areas of investigation that provide us new methods and technologies.

These new technologies related to our environment, are streamlining, optimizing and perfecting some activities we do in our day to day life. Communication today is something that has far advanced. Each century has seen a new addition to the ever-growing list of means of communication. In the 15th century the invention of printing press made possible the transfer of documents from one place to another. Then, at 19th century the Telegraph appeared and shortly after the Telephone. Both technologies are using electrostatic signals transmitted through a wire. In early 20th century, arise two new means of communication, the Radio and Television. It was the beginning of short-wave communication technique followed by Cell phone. Finally, at the end of the century, Internet appeared as a basic form of connecting group of computers to each other and share information. Nowadays, wireless communication has become fundamental to our lives and we transmit a lot of data every day. The most used technology to transfer data wirelessly is called Wireless Fidelity (Wi-Fi). It uses electromagnetic waves, in particular radio waves, which are received and interpreted by wireless card inserted into most mobile device today. But, while the Wi-Fi mean breakthrough for transferring data wirelessly, it also has some shortcomings because of restricted spectrum availability and interference. Nowadays, radio spectrum is crowded and it is difficult to find radio capacity to support media applications.

For this reason, new solutions for mitigating the RF spectrum are being investigated and new technologies are being proposed. One of them is Visual Light Communication (VLC). The history of VLC dates back to the 19th century when Alexander Bell invented the photophone, which transmitted speech on modulated sunlight over several hundred meters. It worth to be noted that photophone has been invented before the Telephone.
But it is now, when these technology starts to be developed. VLC is a new way of wireless communication that instead of using radio waves to transfer data, uses visible light emitted by LED based light bulbs.

This new technology has many advantages in compare with the transmission of radio waves. First of all, the bandwidth is very large and offers a huge frequency band, but is not regulated. Secondly, VLC is highly energy efficient as we can illuminate and transmit data at the same time. Furthermore, VLC transmitters and receivers are cheap and there is no need for expensive RF units.

The next stage to take into account is the human health. The transmission of power radio waves cannot be increased over a certain level because there are serious health risks for humans. As VLC is harmless for the human body, we can increase the transmission power if needed although there is a limit imposed by the illumination standard. This limitation of transmitted power is higher in comparison with radio waves transmission. Then, as light waves do not penetrate opaque objects it cannot be intercepted, so it offers a very secure communication. Moreover, we can use VLC in many places like hospitals, aircrafts without any problem of interferences with other devices.

Light Emitting Diode (LED) is a semiconductor device that can be modulated. At present, the conventional lights are being replaced by LED’s due to the efficiency and low consumption. For this reason, we will have the infrastructure of a lot of LED-based lights installed in the world; it means potential VLC transmitters and therefore, we can use them for communications.

But not all are advantages; with this technology we have also disadvantages. The VLC system is prone to interference from other light sources, like sunshine, incandescent light, fluorescent lamp, etc. Moreover, VLC requires line of sight, which means that we can only transmit data where the light is.

As the benefits of this new technology are more than the downsides, we have decided to develop and implement a transceiver, able to be integrated on an embedded platform.

This project focuses on the transmission part where we will make a detailed study of the physical layer.
This technical memory is organized in 4 main sections. In the first section, we introduce the structure of communication frame, based on the IEEE Standard, in order to know which parts compose the frame and how each part works. On the second section, we introduce the architecture described on the Standard to understand how the data is transmitted. Latter we introduce the signal processing block diagram to illustrate the main features and requirements of each processing module. On the third section, we present our implementation using high level programming language, in our case C++ code. Here we will focus on the challenges raised by the implementation on embedded platform. Finally, we will present the integration of our code on the chosen embedded platform together with a set of tests used to validate the correct behavior of our VLC transmitter.
CHAPTER 1. IEEE 802.15.7 STANDARD

The IEEE 802.15.7 standard for VLC has been published on 2011. This standard covers both, the physical layer (PHY) and medium-access control layer (MAC).

The standard is being proposed for a variety of VLC applications relating to Wireless Personal Area Networks (WPAN). The MAC currently supports three multiple access topologies; peer-to-peer, star and broadcast mode. The physical layer is divided into three types; PHY I, II and III, and these employ a combination of different modulation schemes.

There are two different modulation modes: OOK (On-Off Keying) and VPPM (Variable Pulse-Position Modulation). On-Off Keying represents digital data as the presence or absence of carrier wave. The presence of a carrier for a specific duration represents a binary one, while its absence represents a binary zero.

\[
\text{DATA} \quad 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0
\]

Fig.1.1 OOK modulation

Otherwise, Variable Pulse-position Modulation supports simultaneously illumination with dimming control and communication. It changes the duty cycle of each optical symbol, which is distinguished from others by the pulse position to encode bits. The variable term in VPPM represents the changes of the pulse width in response to the requested dimming level. The logic ‘0’ and logic ‘1’ are symbols; pulse width modulated depending on the dimming duty cycle requirement. In that figure we can see also the "unit symbol" which basically is the length in time of a data symbol.
In order to keep the compatibility with modern illumination systems, both modulations were designed to operate in the presence of dimming, which controls the brightness of the light in order to adjust the intensity.

The modulation using OOK under dimming provides constant range and variable data rate by inserting compensation time and the modulation using VPPM under dimming provides constant data rate and variable range by adjusting the pulse width.

1.1 FRAME STRUCTURE

Frame structure refers to the skeleton, which shows us the division of information into different fields. According to the skeleton the frame is divided in 3 main parts: a synchronization header (SHR), a PHR block and PHY Service Data Unit (PSDU).

<table>
<thead>
<tr>
<th>Preamble</th>
<th>PHY header</th>
<th>HCS</th>
<th>Optional fields</th>
<th>PSDU</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHR</td>
<td>PHR</td>
<td></td>
<td></td>
<td>Payload</td>
</tr>
</tbody>
</table>

Fig.1.3 PPDU Format
• The synchronization header (SHR) is composed by the preamble used to obtain clock synchronization.

• PHR block, composed by PHY header, HCS and optional fields. PHY header contains parameters that give us how the data has to be transmitted; the HCS part used to check possible errors and the optional fields that proportionate more specific information about other fields.

• PHY Service Data Unit (PSDU) block. It contains the payload of the frame.

1.1.1 Preamble

In the preamble two main parts can be differentiated. The Fast Locking Pattern (FLP) bits and Topology Dependent Patterns (TDP) pattern sequence.

<table>
<thead>
<tr>
<th>FLP</th>
<th>TDP</th>
<th>-TDP</th>
<th>TDP</th>
<th>-TDP</th>
</tr>
</thead>
</table>

Fig.1.4 Preamble structure

FLP bits are used to obtain clock synchronization in the phase lock loop (PLL) block of the receiver. A PLL is a control system that tries to generate an output signal whose phase is related to the phase of the input reference signal. The circuit compares the phase of the input signal with the phase of a signal derived from its output oscillator and adjusts the frequency of its oscillator to keep the phases matched.

Fast Locking Pattern contains zeros or ones. The size of FLP field must be between the minimum and maximum values that are defined in the standard.
Those are at least 64 bits and maximum of 16,384 bits. This wide range is good because if the phases are not matched we need to change the number of bits. Sometimes more bits are needed to have the phases matched.

TDP pattern is a known sequence used to signal the beginning of a new frame. This sequence is known at the receiver, who through correlation can see whether the data received and his defined patterns, are related and how strong is this relation.

If this relation is maximum, a peak is shown, meaning that a frame starts. Furthermore, this pattern gives us more information about which topology has been used.

After the FLP we have four repetitions of TDPs, each of them with length 15 bits. As we said before, this TDPs are use to be able to distinguish different PHY Topology.

TDP_Patern1: (1 1 1 1 0 1 0 1 1 0 0 1 0 0) – Independent
TDP_Patern2: (0 0 1 0 1 1 0 1 1 1 1 1 0) – Peer to peer
TDP_Patern3: (1 0 0 1 1 0 0 0 0 1 0 0 1) – Star
TDP_Patern4: (0 1 0 0 0 1 1 0 1 0 0 1 0 1) – Broadcast

In order to make an implementation we have to take into account that the preamble shall be transmitted using OOK modulation. Also, we have to take into account that FLP is not transmitted all the time; it depends on which data mode is being used.

The PHY layer supports 3 data transmission modes. Single mode, Packet mode and Burst mode.
Single mode transfers one data unit per frame, it implies to transmit more bits each time we transmit a frame because in each frame, we had to add the extra bits (preamble) in order to be sure that the receiver will be able to decode our data bits.

![Single mode diagram](image)

Fig. 1.5 Single mode

Packet mode contains multiple data unit per frame. It is used to send multiple consecutive PPDU to the same destination. With this mode we can reduce the number of extra bits corresponding to the header and we can transmit more useful bits.

![Packet mode diagram](image)

Fig.1.6 Packet mode

And, burst mode, which a device is transmitting data repeatedly without going through all the steps. It reduces the length of the preamble. With that mode we can also reduce the number of header bits and we are able to transmit more useful bits.
According to the transmission modes, for single data mode and packet data mode we had to transmit the FLP field in each frame as we can see in the figure 1.4. The main difference between two of them is that using packet data mode we reduce the number of extra bits and we are able to transmit more useful bits in each frame.

But, for the burst mode transmission, the FLP shall be transmitted only in the first frame. Subsequent frames shall not include the FLP since the receiver is already synchronized to the transmitter. With this, we can reduce the preamble length and provides higher throughput.

1.1.2 PHR

After the SHR block in which we achieve synchronization, PHR block is the responsible for the following tasks:

- Channel selection
- Data transmission and reception
- Error correction

In order to achieve it, PHY header is needed to know some parameters used in the transmission part like the mode of transmission, data rate, modulation, PHY type and optical clock used.
To detect errors a CRC-16 block is added. The input data has a short check value attached. On the receiver the calculation is repeated and if the check values do not match, it could apply a corrective action.

After the CRC part, if the transmission supports dimming, we have to add some optional fields like tail bits, compensation length, resynchronisation length, sub frame length, OFCS and channel estimation sequence.

![PHY Header](image)

**Fig. 1.8 PHR frame**

As we can see in the figure 1.8, the fields of PHY Header are composed by:

- Burst mode, we need to take into account this field because as we said before, if the mode is burst we have to reduce the preamble in some frames and we need this field to know if the mode is burst or not;
- Channel number, in order to know the band plan ID;
- MCS indication that provides information about PHY type and data rate;
- PSDU length that gives us information about the number of octets that it contains. This value is between 0 and 1023, where 1023 is the maximum size that PHY is able to receive;
- Dimmed OOK extension bit to know if dimming is available or not.
- Other reserved fields for future use.
Regarding the optional fields, tail bits shall be transmitted only when PHY I is used with an optical clock of 200 kHz. The compensation length, resynchronization length, sub frame length and OFCS shall be transmitted after the tail bits when the dimmed OOK bit is set.

### 1.1.3 PSDU

The PSDU is the data unit being sent down from the MAC layer for transmission on the wireless medium. The structure of the PSDU field is as shown in figure 1.9.

<table>
<thead>
<tr>
<th>MHR</th>
<th>MAC payload</th>
<th>FCS</th>
<th>Tail bits</th>
</tr>
</thead>
</table>

Fig.1.9 PSDU structure

MAC Frame structure is similar to the PHY frame. It has a header, MHR, which provides information, needed in the transmission, a data payload (MSDU) of variable length and also adds a Frame Check Sequence (FCS) algorithm, in our case Cyclic Redundancy Check is appended after the MAC payload in order to detect errors.

The Mac Protocol Data Unit (MPDU) at the output of the MAC sublayer passes through the PHY layer and becomes PHY Service Data Unit (PSDU) at the output of the PHY layer.

PSDU structure is composed by data coming from the MAC layer and tail bits. The size of the data coming from the MAC could be a value between 0 and 1023, where 1023 is the maximum size that PHY is able to receive.

Regarding of the six tail bits of zeros, they are attached to the end of the PSDU, if PHY I is used with data rates of 11.67kb/s, 24.44kb/s, or 48.89kb/s. These tail bits are related with the tail bits attached in the PHR frame.
After explaining the different parts that make up the physical layer, the figure below 1.10 shows us a global vision of the entire physical layer data unit. We can see each block and their parts.

Fig. 1.10 Physical layer data unit (PPDU)
1.2 ARCHITECTURE

The architecture of this Standard follows the OSI model created by International Organization for Standardization (ISO) that consists of 7 layers.

The communication functions are broken down into a hierarchical set of layers. Each layer performs a related subset of the functions required to communicate with another system. It relies on the next lower layer to perform more primitive functions and to conceal the details of those functions. It provides services to the next higher layer. The layers are defined in such a manner so that changes in one layer do not require changes in the other layers.

By partitioning the communication functions into layers, the problem at hand is much more manageable.

In our project, we have focused on PHY Layer. This layer is responsible for the following tasks:

- Activation and deactivation of VLC transceiver.
- Wavelength Quality Indication (WQI) for received frames.
- Channel selection.
- Data Transmission and Reception.
- Error correction.
- Synchronization.

The layer supports multiple PHY Types:

- **PHY I**: this type developed for outdoor usage with low data rate applications. This type uses OOK and VPPM with data rates in tens to hundreds of Kb/s.
- **PHY II**: this type developed for indoor usage with moderate data rate applications. This type also uses OOK and VPPM with data rates in tens of Mb/s.
- **PHY III**: this type developed for applications using colour-shift keying (CSK) that have multiple light sources and detectors.
In our case we worked on PHY I type. After knowing the structure of the PHY Frame we need to know how this data is processed in order to make the system more reliable and without errors.

The diagram of signal processing is the following:

![Block diagram signal processing](image)

In the figure below 1.11, we can see how the signal is processed and which are the processing blocks of PHY I.

PHY I contains two encoders to detect and correct errors. The RS encoder, that can detect and correct symbol errors and the convolutional encoder that detect and correct bit level errors.

The output of the RS encoder passes through an interleaver that organizes the digital information in a non-contiguous way to protect against burst errors. Then, depending on what data rate we want to achieve the data is punctured, it means that some of redundancy bits are removed.

Finally, the data enters to a RLL encoder that is used to encode the data over communications channel in order to make the system stronger and not have flickering.
1.2.2 RS encoder

Error identification and correction are crucial for reliable and efficient digital communication and data storage. Reed Solomon codewords are the most widely used method to provide digital error detection and correction.

Reed-Solomon code is a systematic way of building codes that could detect and correct multiple random symbol errors. In RS encoder redundant information is added to the signal to allow the receiver to detect and correct errors that may have occurred in transmission.

By adding $n$ check symbols to the data, an RS code can detect any combination of up to $x$ erroneous symbols, or correct up to $x/2$ symbols.

RS code can be described as an $(n,k)$ code where $n$ is block length in symbols and $k$ is the number of information symbols in the message. In order to know what outer RS code use, we have to take into account the data rate because depending on it, we have different generator polynomials.

The possible outer RS code for PHY I proposed by the IEEE Standard 802.11.7 are the following:

<table>
<thead>
<tr>
<th>$(n,k)$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(15,11)</td>
<td></td>
</tr>
<tr>
<td>(15,7)</td>
<td></td>
</tr>
<tr>
<td>(15,4)</td>
<td></td>
</tr>
<tr>
<td>(15,2)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1 Outer RS codes for PHY I
Also,

\[ n \leq 2^m - 1 \]  \hspace{1cm} (1)

Where \( m \) is the number of bits in a symbol, There are \( n-k \) parity symbols and \( t \) symbols can be corrected in a block, where

\[ t = \frac{n - k}{2} \text{ for } n-k \text{ even} \]  \hspace{1cm} (2)

\[ t = \frac{n - k - 1}{2} \text{ for } n-k \text{ odd} \]  \hspace{1cm} (3)

As we can see in figure 1.12, the RS encoder is based on the Linear Feedback Register (LFSR) structure. It is based on a sequential circuit that allows the generation of symbols redundancy. The number of stages is given by the difference between the parameters \( n, k \).

Fig.1.12 RS encoder diagram
1.2.3 Interleaving and puncturing block

Interleaving is a technique in which digital information is organized in a non-contiguous way to improve the performance of the system. It is used in digital data to protect against burst errors. Such occasional errors affect several consecutive bits, and invalidate the corrective properties of redundant error codes used in data transmission. By using interleaving, burst errors are distributed among several words, facilitating the work of correcting code used.

Puncturing is the process of removing some of the parity bits after encoding with an error-correction code. This has the same effect as encoding with an error-correction code with higher rate, or less redundancy. So, we have a trade-off between data rate and protection.

If we want a robust system, we have to sacrifice data rate, so we will obtain a good system in terms of redundancy but the data rate will affected and not be as high as we want. If we want a high rate, it will imply less redundancy. Depending on what we are interested in at that moment we will have to choose and sacrifice data rate or robustness.

1.2.4 Convolutional encoder

Convolutional encoder is a type of error correcting code in with each $m$-bit of information symbol to be encoded is transformed into an $n$-bit symbol, where $m/n$ represents the code rate. The code rate is the ratio between the number of bits entering the encoder ($m$) and the number of bits obtained at the output of the encoder ($n$).

We also have the constraint length parameter($K$)of a convolutional encoder, which is the maximum number of symbols in a single output stream that can be affected by an input symbol.
The generator polynomial is also very important for defining the operation of a convolutional encoder because depending on the generator polynomial we have to do some operations or other.

Convolutional coding is basically performed by using a shift register and combinatorial logic responsible for performing the sum of module 2. Shift register is implemented by concatenating a series of registers, so that, after each clock cycle the data that we have at the input of a register goes out, and thus stands at the entrance of the next register, which has done the same with the data that was in his when the input clock cycle came. As for the combinational logic, it uses a XOR gates.

In the IEEE Standard 802.11.7 the inner code is based on a rate-1/3 mother convolutional code of constraint length seven (k=7) with generator polynomial $g_0 = 133_8$; $g_1 = 171_8$; $g_2 = 165_8$ as we shown in the following figure.

![Fig.1.13 Mother convolutional encoder rate-1/3](image)

The standard allows us to obtain various code rates by applying puncturing to the data output by the mother convolutional encoder. These data rates are $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{2}{3}$. For example, once we have the mother convolutional encoder, puncturing the third bits we could obtain a $\frac{1}{2}$ rate.
1.2.5 RLL encoder

Flicker is a fluctuation of the brightness of light that, if it is perceived by humans, can cause noticeable physiological changes. To avoid this effect, an RLL encoder is used; it is responsible for eliminating any long sequence of bits of the same value.

Run-Length Limited is a line coding technique used in communications systems to increase the reliability and efficiency of the channel. It consists of sending arbitrary data over a communication channel with bandwidth limits.

Depending on the mode used for modulation, we have different RLL encoder. For VPPM mode we use 4B6B and Manchester encoding for OOK mode. Both of them consist of mapping the input bits to \( n \) output bits.

4B6B encoding scheme relies on a D.C balanced code where each message to be transmitted is sent as a plurality of symbols, each symbol having six bits, three ones and three zeros.

Manchester encoding is a form of digital encoding in which data bits are represented by transitions from one logical state to the other. Logic 0 is indicated by a 0 to 1 transition and logic 1 is indicated by a 1 to 0 transition. Manchester encoding minimizes the error rate and optimized reliability. The main disadvantage is the fact that requires more bits to be transmitted than in the original signal.

The main difference between two of them is the number of bits used to redundancy. Manchester encoding uses 4 bits of redundancy, it means duplication in number of bits, and meanwhile, 4B6B encoding only uses 2 bits of redundancy.
CHAPTER 2. PRACTICAL IMPLEMENTATION ON EMBEDDED DEVICES

This chapter focuses on explaining in detail our implemented system. We talk about the system architecture, what are the main blocks implemented and how we implemented them in order to achieve an efficient system.

2.1 SYSTEM ARCHITECTURE

As we introduced in chapter 1, the frame structure is composed of three main parts, the SHR, PHR and PSDU. In order to build the system for our transmitter we have to take into account the different blocks where the signal passes during the processing.

In the figure below, we can see on the one hand, that SHR doesn’t pass through any signal-processing block because is a time domain sequence and does not have any modulation. On the other hand, PHR and PSDU are treated but not in all the blocks. The main difference between PHR and PSDU is that PSDU don’t pass through CRC block because it incorporates CRC from the MAC layer.

![Signal processing diagram](image)

Fig.2.1 Signal processing
2.1.1 Preamble

As a preamble is a time domain sequence and does not have any modulation, channel coding or line coding, this part does not pass through the processing block. So, we build the Preamble frame taking into account the Standard Specifications.

In our case the size of the FLP is 68 bits and has the same pattern 1010...until last bit. Followed of these bits have one TDP pattern or other depending on the topology. So, the size of preamble frame is 128bits.

We recommend to use a size multiple of 8 in order to have all frames completely. It means more efficiency and waste fewer resources in terms of CPU, storage, etc when we talk about embedded systems.

So, working with multiple of 8 bits in this part, we found some problems with the TDP pattern declaration because the total length of this part is 60 bits and this number is not multiple of 8. Therefore, with the purpose of having a multiple of 8 bits, we add 1010 (0xA in hexadecimal) at the beginning of each TDP declaration pattern and we understand these 1010 bits as a part of FLP bits. Doing this, we can achieve our aim (work with multiples of 8).

2.1.2 PHR

The structure of PHR is defined in chapter 1.1.2 where we studied the main parts of the frame. For our implementation we build the PHR frame taking into account the 32 bits referring to the header and 16 bits of the corresponding HCS block. But, what we have to take into account are the six tail bits added to the end of the CRC when working in PHY I type and ‘OOK’ modulation is used. So, in our case we have to add this six tail bits.

As we said before we recommend to use a size multiple of 8 in order to have all frames completely. It means more efficiency and waste fewer resources because our structure works in a byte level.
PHR block enters into Cyclic Redundancy Check (CRC) in order to detect possible errors during the transmission. After that, the data enters into the RS encoder to be able to protect against possible channel errors. Then, to have a more robust system the data is mixed and punctured. And finally a convolutional encoder and RLL are applied.

### 2.1.3 PSDU

As we explain before in chapter 1.1.3 the data is sent down from the MAC layer. For this, in our implementation we have the data input from MAC layer. These data enters into the RS encoder in order to proportionate protection, then, the data is mixed and punctured in the interleaver and puncture block; to achieve a reliable data transfer a convolutional encoder is used and finally to avoid flicker an RLL is used.

Unlike the SHR data, PSDU does not enter in the CRC block of physical layer because as the data comes from the MAC layer, it layer has an incorporated CRC code.

### 2.2 CRC

Cyclic redundancy check (CRC) is a method of checking for errors in data that has been transmitted on communications systems. A sending device applies an $n$ bit polynomial to a block of data that is to be transmitted and appends the resulting cyclic redundancy code to the block. The receiving end applies the same polynomial to the data and compares it result with the result appended by the sender. If they agree, the data has been received successfully. If not, the sender can be notified to resend the block of data.

The IEEE standard, informs us that the CRC shall be calculated using the generator polynomial $G_{16} = x^{16} + x^{12} + x^5 + 1$. 
CRC calculation is realized with a shift register and XOR gates. In CRC-16 we start with 16 memory registers, each holding 1 input bit. All memory registers start with a value of 1. The encoder has a module-2 adders implemented with a XOR gate, and a generator polynomial. An input bit $m$ is fed into the left most register. Using the generator polynomial and the existing values in the remaining registers, the encoder output $n$ bits. Then, the bits are shifted to the right and wait for the next input bit. If there are no remaining input bits, the encoder continues output until all registers have returned to the zero state.

In our implementation we found a more efficient way to do that process. Instead of treating each bit as a single we found a method that gives us the same results as if we do bit by bit. We processed in a byte-level. It means each 8 bits we apply them a simplified code with shifts and XOR’s, and once this bits are processed, we choose the next byte and repeated the process until the end.

### 2.3 Convolutional encoder

Encoders for error control are usually called channel encoders. In our project we implemented two of the possible code rate, the mother convolutional encoder rate-1/3 and the convolutional encoder rate-1/4.
One method to do the convolutional encoding is using a bit-serial algorithm. It consists of attending and processing one item at a time. To do that, a shift register is needed.

This method involves treating each bit as an integer. First, at the input we have the bits that we would like to process and we need to represent each of them as an integer. To do that, we store them in a shift register where it will be shifted during the operation to the LSB of a working register. Then the desired bit-level is performed using a word level operation on the working register and finally, the resulting bit is shifted into another shift register. This process is illustrated in Figure 2.4.
Another method to do the convolutional encoding is using a parallel algorithm. It signifies simultaneous processing on several object or sub-systems at the same time.

In this method, instead of using a working register, we can perform directly the word parallel operations to produce results for 32 bits in parallel in one cycle.

It means that at the input we have the bit-vector of 32 bits and at the output we have also the 32 bits corresponding to the result of the word operations. This process is illustrated in Figure 2.5.

Once we explained the two methods, we noticed that parallel algorithm is faster than the serial due to the simultaneous processing. While serial only process one item at a time, it requires more clock cycles to execute the operation and also, more space for storage the data.

As parallel algorithm is the most efficient method from the point of view of embedded systems because it is fast and use less clock cycles, we use it to perform our convolutional encoder.

In the figure 2.2 we can see the scheme for convolutional encoder of rate-1/3 with constraint length $K$ of 7. Generator polynomials are $g_0 = (1011011)$; $g_1 = (1111001)$; $g_2 = (1110101)$. Therefore, output bits are calculated as follows:
\[ y_0(m) = x(m) \oplus x(m - 2) \oplus x(m - 3) \oplus x(m - 5) \oplus x(m - 6) \] (4)

\[ y_1(m) = x(m) \oplus x(m - 1) \oplus x(m - 2) \oplus x(m - 3) \oplus x(m - 6) \] (5)

\[ y_2(m) = x(m) \oplus x(m - 1) \oplus x(m - 2) \oplus x(m - 4) \oplus x(m - 6) \] (6)

The corresponding C code implementation is the following:

\[ y_0 = x ^ (x \ll 2) ^ (x \ll 3) ^ (x \ll 5) ^ (x \ll 6) \] (7)

\[ y_1 = x ^ (x \ll 1) ^ (x \ll 2) ^ (x \ll 3) ^ (x \ll 6) \] (8)

\[ y_2 = x ^ (x \ll 1) ^ (x \ll 2) ^ (x \ll 4) ^ (x \ll 6) \] (9)

To calculate the output, after each clock cycle, the data is shifted into the next register and after that, an XOR is executed to do the combinatorial logic to make the sum of module 2.

An important problem that we found implementing that part is that, if we calculate all the outputs separately, we will make some operations twice or more times and our main goal are to have an efficient code generation such that the total number of CPU cycles for executing the convolutional encoding algorithm is minimized.

So, if the way of dealing with the outputs is individually, we need 24 logic operations: 12 bit level XOR and 12 shifts.

For this reason, we try to reduce the number of operations and we have applied the technique of Common Sub expression Elimination (CSE). [7]

To implement the optimized solution, we have to identify the patterns that appear in the first equations (4), (5) and (6), and then select the pattern and eliminate all other replicas.
In the next figure 2.6, we can see the different patterns that we have in our equations. Purple, blue and yellow are replicas used in the three equations, blue and green used in two of them and finally we have two other operations used in a single equation.

![Patterns](image)

Fig.2.6 Patterns

Taking into account these replicas we can reformulate the equations in order to compute them once. The same algorithm can be reformulated as:

\[
\begin{align*}
    w_1 &= x \cdot (x \ll 2) \\
    w_2 &= w_1 \cdot (x \ll 6) \\
    w_3 &= w_2 \cdot (x \ll 1) \\
    y_0 &= w_2 \cdot w_1 \cdot (x \ll 3) \\
    y_1 &= w_3 \cdot (x \ll 3) \\
    y_2 &= w_3 \cdot (x \ll 4)
\end{align*}
\]

(10) (11) (12) (13) (14) (15)

With this, we have achieved an optimized solution with 12 logic operations: 6 shifts and 6 XOR. So, we have a reduction of 50% in operations, which means less CPU cycles.
After analysing the different options we have chosen parallel implementation for our system because is more efficient in terms of resources and the number of CPU cycles decreases.

Once we had our implementation for rate-1/3 mother applying puncturing we can obtain other codes rates like rate-1/4, rate-1/2 and rate-2/3.

In our case, we decided to implement code rate-1/4. To achieve it, we have to perform a puncturing in the mother code (with this we have rate-1/2) and then using a simple repetition code can obtain the rate-1/4 as we can see in the figure 2.8.

![Diagram](image1)

**Fig.2.7** Puncturing pattern to obtain rate-1/2 code

![Diagram](image2)

**Fig.2.8** Repetition pattern used to obtain the effective rate-1/4 code
2.4 Interleaving and Puncturing

This block is used as an interleaver between the inner convolutional code and the outer RS code. The interleaver has a fixed height but has a flexible depth depending on frame size. After the interleaver we have a puncturing block to minimize padding overhead.

In this process, the input data is interleaved using an interleaver of size \( n \). The step of interleaving comprises the step of using a permutation between data in order to have a mix. To know how this data is mixed we used the following equation:

\[
l(i) = (i \mod D) \times n + \left( \frac{i}{D} \right); \text{ for } i = 0, 1, \ldots, S_{block} - 1
\]  

(16)

Where \( D \) is the interleaving depth calculated by equation (17); \( n \) is the RS codeword length and \( S_{block} \) is the size of the interleaver used calculated by equation (18).

\[
D = \frac{S}{n}
\]  

(17)

\[
S_{block} = n \times D
\]  

(18)

As an example, in the following table, we can show the values of the index data before being interleaved and how this data is mixed taking into account the formula above.
We can observe that the interleaver process has a pattern. It starts at symbol 0 (on the left) and makes the diagonal (in this first case, there is only one symbol in the diagonal) and then it moves until the last symbol of the first row and makes another diagonal. After that, it moves again until the second symbol, starting at left and makes the diagonal. Then, it repeats the action to move until the last symbol, but now, we have to focus on the last symbol of the second row. Once there, it makes another diagonal. This process is repeated until the end.

The following table shows the result of the interleaver. We can see how data is mixed focusing in the indexes.

<table>
<thead>
<tr>
<th>Interleaver height, $n$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1 Data before interleaved

Once we have the bits interleaved, we need to apply the puncturing block. The locations to be punctured are described by the following equation:
\[ z(t) = (n - p + 1) * D + t * D - 1; \text{for } t = 0, 1, \ldots, p - 1 \] (19)

Where \( n \) is the RS codeword length, \( D \) is the interleaving depth and \( p \) is the number of zeros RS symbols calculated in equation (20).

\[ p = n - (S \mod n) \] (20)

\( S \) is the number of symbols from the output of the shortened RS encoder calculated in the next equation (21)

\[ S = n * \frac{S_{frame}}{k} - (k - (S_{frame} \mod k)) \] (21)

\( S_{Frame} \) corresponds to the number of symbols at the input of the RS encoder and \( k \) is the number of information data symbols in the RS codewords.

\[ S_{Frame} = \frac{L_{frame} * 8}{\log_2(q)} \] (22)

In the example we know that the positions to be punctured are: 27, 30.

Regarding our example after applying the puncture we have the following table where we can see that the values 27 and 30 are punctured.
If we put our data to the example, we can show that once we have the symbols mixed and we know the puncturing positions we can replace it.

Table 2.3 Data punctured

---

If we put our data to the example, we can show that once we have the symbols mixed and we know the puncturing positions we can replace it.

Fig.2.9 All process. (A) data, (B) interleaved data, (C) puncture bits, (D) result after puncturing
2.5 Reed-Solomon Encoder (RS)

Reed-Solomon code is a block code, meaning that the message to be transmitted is divided up into separate blocks of data. Each block then has parity protection information added to it to form a self-contained code word.

Choosing different parameters for a code provides different levels of protection and affects the complexity of implementation.

In our case, we implemented a RS (15, 7) because we decided to analyse two cases; the minimum data rate and the maximum data rate.

The symbol generator redundancy encoder receives the first symbol of the input data; it passes out, and it is also operated with the data stored in the least significant register through XOR operation. The result is input through a following multiplexer to the LFSR circuit and the multiplication operation is performed in algebra finite Galois fields with each of the coefficients of the generator polynomial G(x). The result of the product is operated through a XOR symbol, with the output of the preceding register. Every k symbols entering (in our case 7), the system adds n-k symbols (in our case 8) that correspond to the value of each register in that time. Then, the registers are initialized to zero and the process is repeated until all the symbols pass out.

In our case, to make the multiplication operation we implemented a look-up table with $2^m$ entries. The entries values can be obtained by cyclically shifting the non-zero elements according to the index of the multiplication factor. Doing that we don’t waste resources each iteration because we done the calculations before and we save it in a table where we look iteration by iteration.
2.6 Run-Length Limited (RLL)

In our project we implemented the Manchester encoding for OOK mode. This type of encoding expands each bit into an encoded 2-bit symbol as shown in the following table.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Manchester symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2.4. Manchester

Manchester encoding is described as the process of a logical XOR combining of the data to be encoded and the clock used to establish the bit rate as we can see in the following figure:

![Fig.2.10 Hardware of RLL](image)
CHAPTER 3. SYSTEM INTEGRATION

This chapter will explain our system integration and also the tests performed in order to demonstrate the correct behavior.

3.1 Board

The LPC 1769 board is an ARM Cortex-M3 based microcontroller for embedded applications requiring a high level of integration and low power dissipation.

Cortex-M3 CPU incorporates a 3-stage pipeline and uses Hardware architecture to separate local instruction and data buses as well as a third bus for peripherals.

3.1.1 Board Specifications

- NXP Cortex-M3 LPC1769 processor
- 512 kB Flash
- 64kB data memory
- 120MHz CPU frequency
- 3.15 -3.30 V external powering
- All 1769 pins available on expansion connector (2x27 pin rows)
- USB 2.0 interface
- Serial interfaces: 2xCAN, 4xUART, 3xI2C, 2xSSP, 1xI2S
- 12 bit ADC
3.2 SPI port

The Serial Peripheral Interface (SPI) is a synchronous serial data link, which operates in full duplex mode. It is used for short distance, single master communication. Devices communicate in master/slave mode where the master device initiates the data frame.

The SPI bus specifies four logic signals:

- **SCLK**: Serial clock, used to synchronize the transfer of data across the SPI interface.
- **MOSI**: Master Output, Slave Input is a unidirectional signal used to transfer serial data form an SPI master to an SPI slave.
- **MISO**: Master Input, Slave Output is a unidirectional signal used to transfer serial data form an SPI slave to an SPI master.
- **SSEL**: Slave Select. The SPI slave select signal is an active low signal that indicates which slave is currently selected to participate in a data transfer.

We decided to use the SPI port because it is the unique port in which clock and data signal goes separately. It means that extra bits are not added in our data. It is important for us in order to receive in a good condition.

3.3 Platform

LPCXpresso is a low-cost development tool platform, available directly from NXP, which provides a quick way to develop advanced applications using NXP’s highly efficient and low-power LPC microcontrollers. It includes everything to take end users from evaluation to final production.
A major part of the LPCXpresso platform is the range of LPCXpresso boards, which provide practical and easy to use development hardware to use as a starting point for your LPC Cortex-M MCU based projects. These boards can be used with NXP's own LPCXpresso IDE, and in many cases with other tool chains and development environments.

In our project we use LPC Open libraries in order to take some developed functions and make it easy.

The main functions to program the SPI port are the followings:

- **SystemCoreClockUpdate();** It is used to update the core clock rate and is called if the system has a clock rate change.
- **Board_Init();** In order to use the board we had to initialize all the blocks and functions related to board hardware.
- **Board_SPI_Init();** This function initialize the pin for SPI interface
- **Chip_SPI_Init();** It initialize the SPI and enables the port.
- **Chip_SPI_SetFormat();** In that function we can decide the format (8bits), the clock mode, the data order (if we starts with LSB or MSB).
- **appSPIRun();** It select the transfer mode and with the function **Chip_SPI_RWFrames_Blocking();** we can read and write the frames in the MOSI port.

Taking into account that we use SPI port, we have to know the relation between the PIN’s name and their location in the board. To do that, we look at the pin out diagram of LPC. [16].
### Function, SPI Connection, Pin location

<table>
<thead>
<tr>
<th>Function</th>
<th>SPI Connection</th>
<th>Pin location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI_SCK</td>
<td>P0.15</td>
<td>13</td>
</tr>
<tr>
<td>SPI_SSEL</td>
<td>P0.16</td>
<td>14</td>
</tr>
<tr>
<td>SPI_MISO</td>
<td>P0.17</td>
<td>12</td>
</tr>
<tr>
<td>SPI_MOSI</td>
<td>P0.18</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 3.1 Pin out LPC 1769

As SPI port can work in two modes, Master and Slave, in our project we decided to configure it as a Master. For that reason, when we would like to run our project and make measurements we have to take into account what pin location is used.

If we check it in the Table 3.1 we see that in Master mode, the pin location is 11.

### 3.4 Validation test

Once we programmed the LPC board with all functions needed in PHY layer, we tested and we made measurements with oscilloscope in order to see if the transmission is was we expected.

Figure 3.2 shows us the structure of the transmitted data in which we can distinguish three main parts. First of all, thanks to the FLP bits we are able to know when a frame starts, after that; we can see the TDP pattern taking into account that we could have more than two equal bits and finally the data to be transmitted. It scheme is repeated continuously.
If we zoom the image above we can detect the FLP and TDP pattern. We recognize FLP because is a sequence of ones and zeros during a determinate length between 64 bits and a maximum defined in the standard, in our case the length of FLP is 68 bits.

After the FLP bits, we have four repetitions of Topology Dependent Patterns (TDPs) each of them with length 15 bits. In our validation test we configure peer to peer topology. So, we have the following sequence:

0 0 1 0 1 1 1 0 1 1 1 1 1 0
By zooming in on the figure above, we can observe deeply all the TDP sequence and we are able to check it.

<table>
<thead>
<tr>
<th>TDP</th>
<th>-TDP</th>
<th>TDP</th>
<th>-TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>001011101111110</td>
<td>110100010000001</td>
<td>001011101111110</td>
<td>110100010000001</td>
</tr>
</tbody>
</table>

Fig.3.4 FLP and TDP pattern part
Once we prove that it works and we detected the frame in the oscilloscope, we connected the LPC 1769 Board to transmitter. The encoded data is transformed to an optical signal which is transmitted by the light source to the receive module.

There are two ways of producing white light using LED. First one is the use of phosphor material to convert monochromatic light from a blue LED to white light. Second way is mix the entire individual LED that emitted red, green and blue together to form white light. In our case we used the first one, the phosphor material to obtain white light.

Once we connected the LPC1769 to the transmitter we obtain the following sequence. It is the same as before, when we connected the oscilloscope with the LPC 1769, but, now we have a curve effect as we can see in the following figures 3.6 and 3.7.

Fig. 3.5 TDP pattern
In figure 3.6 we can see the frame sequence after being transmitted by the LED and in figure 3.7 we can see the TDP pattern sequence. Here is where we can detect better the effect of curve. Part of this effect is caused by the long response time of yellow phosphor that converts blue light to yellow light.
CHAPTER 4. CONCLUSIONS

In this thesis we have proven that a Standard compliant VLC transmitter can be efficiently implemented in a commercial micro-controller. This can facilitate the Integration of VLC technologies in many commercial LED based lighting devices, offering them the capability of Communications which can led to the development of novel and interesting Applications.

Moreover we have presented a set of efficient solutions, in terms of computational power and data storage capabilities, for the implementation of various processing blocks encountered on the architecture of VLC transmitter.

We have shown an efficient way of implementing a CRC encoder, which can operate with bytes of data. It implies processing in a byte-level that means each 8 bits we apply a simplified code and once this bits are processed, we choose the next byte and repeated the process until the end.

We have presented two methods for the implementation of convolutional encoder: the bit-serial algorithm and parallel algorithm. Analysing both, we can demonstrated that parallel algorithm is more efficient because it process simultaneously 32 bits in one cycle. Once calculated the output of the convolutional encoder we saw that there were some patterns that were repeated. We have presented an optimization solution, which consists in eliminating the replicas and reformulating the equation in order to reduce the number of operations.

We also managed to minimize the number of operations performed in RS encoder. Building a look-up table with the multiplication operation with each coefficient of the generator polynomial, we achieve spending fewer resources. We only need to access the table on each iteration without doing extra calculations.
In the SHR and PHR part we notice that using a size multiple of 8 we can obtain more efficiency and waste fewer resources in terms of processing power and data storage which are crucial requirements when it comes to integration on embedded systems.

VLC is a promising Technology, It could be very important in the future. It is a very efficient alternative to radio-based wireless. There are several future applications for VLC.

- Hospitals: Some equipment is prone to interference with radio waves. So, VLC has many advantages in this area.

- Vehicles and transportation: Cars can communicate to each other to prevent accidents.

- Aviation: Radio waves cannot be used by passengers in aircrafts. LED-based lights are already used in aircraft cabins and each of these lights could be a potential VLC transmitter.

- Underwater communications: RF does not work underwater but VLC can support high-speed data transmission over short distances in this environment. This could enable divers and underwater vehicles to talk each other.
IEEE STANDARD


RS


Convolutional encoder


Interleaver + puncturing


CRC


VLC


LPC Xpresso


ANNEXES

ANNEXE A. CODE IMPLEMENTED IN C LANGUAGE

Main

```c
#include "chip.h"
#include "board.h"
#include "cr_section_macros.h"
#include "stdio.h"
#include "parameters.h"
#include "string.h"
#include "math.h"

static SPI_CONFIG_FORMAT_T spi_format;

int main(void)
{
    int index;
    #if defined (__USE_LPCOPEN)
    #if !defined(NO_BOARD_LIB)
    SystemCoreClockUpdate(); // Read clock settings and update SystemCoreClock variable
    Board_Init();
    Board_LED_Set(0, false); // Set the LED to the state of "On"
    
    index=generate_SHR(buffer_output);
    index=generate_PHR (buffer_output, index);
    generate_PSDU (buffer_output, data_value1, index); //void
    
    //SPI part:
    Board_SPI_Init(true); //This function initialize the pin for SPI interface.
    
    /* SPI initialization */
    Chip_SPI_Init(LPC_SPI);
    spi_format.bits = SPI_BITS_8;
    spi_format.clockMode = SPI_CLOCK_MODE0;
    spi_format.dataOrder = SPI_DATA_LSB_FIRST;
    Chip_SPI_SetFormat(LPC_SPI, &spi_format);

    volatile static int i = 0;
    int k=0;
    while(1)
    {
        appSPIRun();
    }
    
    return 0;
}
```

SHR

```c
int generate_SHR (int SHR_preamble[])
{
    //Variables:
    extern int SHR_length;
```
int flp_bits = 0;
int TDP_pattern_length = 64;
int i = 0;

// Vectors initialization:
int glSHR_preamble[4] = {0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF};
int flp_field[2] = {0xAAAAAAAA, 0xAAAAAAAA};
int TDP_pattern1[2] = {0x28409afa, 0xecca026bf};
int TDP_pattern2[2] = {0x045bf74a, 0x8116fdd2};
int TDP_pattern3[2] = {0xfd6f419a, 0x37cd9064};
int TDP_pattern4[2] = {0xe99d2c2a, 0x5a7b4b09};

if ((data_mode == 0) && (n_frame > 1)) // if burst mode only flp in first frame
{
    for (i = 0; i < 64; i++)
    {
        flp_bits = 0;
    }
}
else // When the mode is not burst the frame is always the same.
{
    flp_bits = 64;
    memcpy (&glSHR_preamble[0], &flp_field[0], (flp_bits / 8));
}

SHR_length = flp_bits + TDP_pattern_length;

switch (topology)
{
    case 1: // visibility
        memcpy (&glSHR_preamble[flp_bits / 8], &TDP_pattern1[0], TDP_pattern_length / 8);
        break;
    case 2: // p2p
        memcpy (&glSHR_preamble[2], &TDP_pattern2[0], (TDP_pattern_length / 8));
        break;
    case 3: // star
        memcpy (&glSHR_preamble[flp_bits / 8], &TDP_pattern3[0], TDP_pattern_length / 8);
        break;
    case 4: // Broadcast
        memcpy (&glSHR_preamble[flp_bits / 8], &TDP_pattern4[0], TDP_pattern_length / 8);
        break;
}

memcpy (*SHR_preamble[0], &glSHR_preamble[0], SHR_length / 8);
return (4);

PHR

int generate_PHR (int PHR_vector[], int index)
{
    // Variables:
    int PHR_length = 0, length1 = 0;
    int temp_chann = 0;
    temp_chann = channel_num;
    int *PHR_1;
    shortint *CRC_vect;
    int index1 = 0;
    int OOK_extension = 0;

    switch (MCS_indication)
```java
| case 0: | data_rate=11670;  
| FEC_outer_codein=7;  
| FEC_outer_codeout=15;  
| FEC_inner_codein=1;  
| FEC_inner_codeout=4;  
| optical_clock=200000;  
| phy_modulation=0;  
| MCS_indication=0;  
| break; |
| case 1: | data_rate=24440;  
| FEC_outer_codein=11;  
| FEC_outer_codeout=15;  
| FEC_inner_codein=1;  
| FEC_inner_codeout=3;  
| optical_clock=200000;  
| phy_modulation=0;  
| MCS_indication=0x20;  
| break; |
| case 2: | data_rate=48890;  
| FEC_outer_codein=11;  
| FEC_outer_codeout=15;  
| FEC_inner_codein=2;  
| FEC_inner_codeout=3;  
| optical_clock=200000;  
| phy_modulation=0;  
| MCS_indication=0x10;  
| break; |
| case 3: | data_rate=73300;  
| FEC_outer_codein=11;  
| FEC_outer_codeout=15;  
| optical_clock=200000;  
| phy_modulation=0;  
| MCS_indication=30;  
| break; |
| case 4: | data_rate=100000;  
| optical_clock=200000;  
| phy_modulation=0;  
| MCS_indication=4;  
| break; |
| case 5: | data_rate=35560;  
| FEC_outer_codein=2;  
| FEC_outer_codeout=15;  
| optical_clock=400000;  
| phy_modulation=1;  
| MCS_indication=0x28;  
| break; |
| case 6: | data_rate=71110;  
| FEC_outer_codein=4;  
| FEC_outer_codeout=15;  
| optical_clock=400000;  
| phy_modulation=1;  
| MCS_indication=0x18;  
| break; |
| case 7: | data_rate=124400;  
| FEC_outer_codein=7;  
| FEC_outer_codeout=15;  
| optical_clock=400000;  
| phy_modulation=1;  
| MCS_indication=0x38;  
| break; |
```
case 8:
    data_rate=266600;
    optical_clock=400000;
    phy_modulation=1;
    MCS_indication=4;
    break;

case 16:
    data_rate=1250000;
    FEC_outer_codein=32;
    FEC_outer_codeout=64;
    optical_clock=3750000;
    phy_modulation=1;
    MCS_indication=2;
    break;

case 17:
    data_rate=2000000;
    FEC_outer_codein=128;
    FEC_outer_codeout=160;
    optical_clock=3750000;
    phy_modulation=1;
    MCS_indication=0x22;
    break;

case 18:
    data_rate=2500000;
    FEC_outer_codein=32;
    FEC_outer_codeout=64;
    optical_clock=7500000;
    phy_modulation=1;
    MCS_indication=0x12;
    break;

case 19:
    data_rate=4000000;
    FEC_outer_codein=128;
    FEC_outer_codeout=160;
    optical_clock=7500000;
    phy_modulation=1;
    MCS_indication=0x32;
    break;

case 20:
    data_rate=5000000;
    optical_clock=7500000;
    optical_clock=15000000;
    phy_modulation=1;
    MCS_indication=0xa;
    break;

case 21:
    data_rate=6000000;
    FEC_outer_codein=32;
    FEC_outer_codeout=64;
    optical_clock=15000000;
    phy_modulation=0;
    MCS_indication=0x2a;
    break;

case 22:
    data_rate=9600000;
    FEC_outer_codein=128;
    FEC_outer_codeout=160;
    optical_clock=30000000;
    phy_modulation=0;
    MCS_indication=0x1a;
    break;

case 23:
    data_rate=12000000;
    FEC_outer_codein=32;
    FEC_outer_codeout=64;
    optical_clock=30000000;
    phy_modulation=0;
    MCS_indication=0x3a;
    break;
case 24:
data_rate=19200000;
FEC_outer_codein=128;
FEC_outer_codeout=160;
optical_clock=60000000;
phy_modulation=0;
MCS_indication=6;
break;
case 25:
data_rate=24000000;
FEC_outer_codein=32;
FEC_outer_codeout=64;
optical_clock=60000000;
phy_modulation=0;
MCS_indication=0x26;
break;
case 26:
data_rate=38400000;
FEC_outer_codein=128;
FEC_outer_codeout=160;
optical_clock=60000000;
phy_modulation=0;
MCS_indication=0x16;
break;
case 27:
data_rate=48000000;
FEC_outer_codein=32;
FEC_outer_codeout=64;
optical_clock=1200000000;
phy_modulation=0;
MCS_indication=0x36;
break;
case 28:
data_rate=76800000;
FEC_outer_codein=128;
FEC_outer_codeout=160;
optical_clock=1200000000;
phy_modulation=0;
MCS_indication=0xe;
break;
case 29:
data_rate=96000000;
optical_clock=1200000000;
phy_modulation=0;
MCS_indication=0x2e;
break;
}

switch (temp_chann)
{
    case 0:
        temp_chann=0;
        break;
    case 1:
        temp_chann=8;
        break;
    case 2:
        temp_chann=2;
        break;
    case 3:
        temp_chann=3;
        break;
    case 4:
        temp_chann=1;
        break;
    case 5:
        temp_chann=5;
        break;
    case 6:
        temp_chann=3;
        break;
case 7:
  temp_chann=7;
  break;

if (data_mode==3)
  OOK_extension=32;
else
  OOK_extension=0;

int tail_bits=0;
if (MCS_indication<=4)
  tail_bits=6;
else
tail_bits=0;

PHR_length=4;
if (data_mode==0)
{
  glPHR_vector[0]=glPHR_vector[0] | 0x1
}
else
  glPHR_vector[0]=glPHR_vector[0] & 0xFFFFFFFE;

//channel number
glPHR_vector[0]=0xFFFFFFF1 & glPHR_vector[0];
glPHR_vector[0] |= temp_chann << 1;

// MCS_indication
glPHR_vector[0]=0xFFFFFC0F & glPHR_vector[0];
glPHR_vector[0] |= MCS_indication << 4;

//PSDU_length
glPHR_vector[0]=0xFD000CFF & glPHR_vector[0];
glPHR_vector[0] |= PSDU_length << 10;

//Dimmed OOK extension
glPHR_vector[1]=Calc_CRC_C (glPHR_vector[0], 4);
length1=4;
if (phy_type==1 && phy_modulation==0)//we add six tail bits
{
  PHR_length=PHR_length+3;
}
else
{
  PHR_length=PHR_length+2;
}
RS_encoder(&glPHR_vector[0],PHR_length);
PHR_length=15;
interleaver_puncturing_v1 (&glRS_vector[0], length1);
length=28;
Conv_encoder (&glinter_vector[0], length);
Long=((length)*4)/8;
manchester_encoding (&glconv_vector[0], Long);
Long=Long*2;
memcpy (&PHR_vector[index], &glmanch_vector[0], Long);
Long=(Long*8)/32;
index1=Long+index;
return (index1);
PSDU

```c
void generate_PSDU (int PSDU_vector[], int data_value[], int index)
{
//Variables:
extern int MCS_indication;
int tail_bits=0;
tail_bits_vector=0;
*PSDU_mod;
length_data=0;
*RS_vector;
//Vector initialization:
glPSDU_vector[2]={0xFFFFFFFF, 0x00000000};
PSDU_length= PSDU_length1 + tail_bits;
if (data_rate==11670) //minimum data rate
{
    length_data= 10;
    RS_vector=RS_encoder(&data_value[0], length_data);
    length_data=40;
    PSDU_mod=interleaver_puncturing_v1(RS_vector, length_data);
    Long=22*8;
    PSDU_mod=Conv_encoder (PSDU_mod, Long);
    Long=(Long*4)/8;
    PSDU_mod=manchester_encoding (PSDU_mod, Long);
    Long=Long*2;
}
elseif (data_rate==100000) //maximum data rate not RS and Conv_enco
{
    PSDU_mod=manchester_encoding (PSDU_mod, Long);
    Long=Long*2;
}
memcpy (&PSDU_vector[index], PSDU_mod, Long);
}
```

CRC

```c
unsigned short int Calc_CRC_C (int Buffer, int Len)
{
    char *y=(char*)&Buffer;
    char coge,fff=0;
    unsigned short int x=0;
    unsigned short int crc = 0xFFFF;
    while(Len--)
    {
        coge=*y++;
        x = {(crc) ^ (coge)}&0x00ff;
        fff=(x^(x<<4));
        crc = (((crc) >> 8)&0x00ff) ^ (fff << 8) ^ (fff <<3) ^ (fff>>4);
    }
    return (crc);
}
```

CONVOLUTIONAL ENCODER

```c
void *Conv_encoder (int data_entrada[], shortint Long)
{
    int p=6;
```
DEVELOPMENT OF EMBEDDED TRANSMITTER FOR VISIBLE LIGHT COMMUNICATION SYSTEM FOR INDOOR ENVIRONMENT

int m=0;
int n=0;

int w1,w2,w3,w11, w1_3, w1_1;
int y0,y1;
int y000[7]={};
int y111[7]={};
int y0000[5]={};
int y1111[5]={};

int in1;
char *i;
int *z,*ztemp, out_p;
int a=0,b=0,j=0;
int btemp;
int a0,a1,a2,a3,a4,a5,a6,a7;
int b0,b1,b2,b3,b4,b5,b6,b7;

int len=0, len1=0;
int l=1, outtemp=0,q=0,s=0,f=0, r=0, c=0 ,k=0;

while (m<Long) {
    for (k=0;k<4; k++)
    {
        a=k*3+c*13;
        i=&(data_entrada[0]);
        i=i+a;
        z=(int*)i;
        i=&(data_entrada[0]);
        i=i+a+1;
        ztemp=(int*)i;
        b=2*k;
        btemp=8-b;
        switch (k) {
            case 0:
                in1=*z;
                break;
            case 1:
                in1=((*z>>(b))&(0x3FFFFFFF))| ((*ztemp<<btemp)&0xC0000000);
                break;
            case 2:
                in1=((*z>>(b))&(0x0FFFFFFF))| ((*ztemp<<btemp)&0xF0000000);
                break;
            case 3:
                in1=((*z>>(b))&(0x03FFFFFF))| ((*ztemp<<btemp)&0xFC000000);
                break;
        }
        w11=(in1<<2);
        w1=(in1)^w11;
        w2=w1^(in1<<6);
        w3=w2^(in1<<1);
        w1_3=w1<<3;
        w1_1=w1<<1;
        y0=w2^w1_3;
        y1=w2^w1_1;
        if (j>0) {
            y000[j]=(y0>>6)&0x3FFFFFFF;
            y111[j]=(y1>>6)&0x3FFFFFFF;
        } else {
            y000[j]=y0;
            y111[j]=y1;
        }
        y0000[j]=y000[j];
        y1111[j]=y111[j];
    }
}

for (len=0; len<4; len++)
{
    a0 = y0000[j] & 0x00000001;
    outtemp = outtemp | 0x00000001<<(a0<<1);
    b0 = (y1111[j] & 0x00000001)<<2;
    outtemp = outtemp | 0x00000001<<(b0<<1);
    a1 = (y0000[j] & 0x00000002)<<3;
    outtemp = outtemp | 0x00000002<<(a1<<1);
    b1 = (y1111[j] & 0x00000002)<<5;
    outtemp = outtemp | 0x00000002<<(b1<<1);

    if (len==3 && j>0)
    {
        q = (q++) % 4;
        out_p = &glconv_vector[len1];
        out_p = (char*)out_p + s;
        *out_p = outtemp;
        s++;

        if (s>4)
        {
            s = 1;
        }

        if ((j%2==1 && j>4 && f>=4)
        {
            r = j;
            len1++;
        }

        outtemp = 0;
        break;
    }

    a2 = (y0000[j] & 0x00000004)<<6;
    outtemp = outtemp | 0x00000004<<(a2<<1);
    b2 = (y1111[j] & 0x00000004)<<8;
    outtemp = outtemp | 0x00000004<<(b2<<1);
    a3 = (y0000[j] & 0x00000008)<<9;
    outtemp = outtemp | 0x00000008<<(a3<<1);
    b3 = (y1111[j] & 0x00000008)<<11;
    outtemp = outtemp | 0x00000008<<(b3<<1);

    a4 = (y0000[j] & 0x00000010)<<12;
    outtemp = outtemp | 0x00000010<<(a4<<1);
    b4 = (y1111[j] & 0x00000010)<<14;
    outtemp = outtemp | 0x00000010<<(b4<<1);
    a5 = (y0000[j] & 0x00000020)<<15;
    outtemp = outtemp | 0x00000020<<(a5<<1);
    b5 = (y1111[j] & 0x00000020)<<17;
    outtemp = outtemp | 0x00000020<<(b5<<1);
    a6 = (y0000[j] & 0x00000040)<<18;
    outtemp = outtemp | 0x00000040<<(a6<<1);
    b6 = (y1111[j] & 0x00000040)<<20;
    outtemp = outtemp | 0x00000040<<(b6<<1);
    a7 = (y0000[j] & 0x00000080)<<21;
    outtemp = outtemp | 0x00000080<<(a7<<1);
    b7 = (y1111[j] & 0x00000080)<<23;
out_p = &glconv_vector[len1];  //Memory address
out_p = (char*)out_p + q;
*out_p = outtemp;  //write the glconv_vector

if (len1==Long)
    break;

y0000[j]=y000[j]>>B*1;
y1111[j]=y111[j]>>B*1;
outtemp=0;
len1++;
l++;

if (len1==Long)
break;
l=1;
INTERLEAVING AND PUNCTURING

```c
void *interleaver_puncturing_v1 (int data_entrada[], int Length)
{
    unsigned short int z[10] = {0};

    int i = 0, s = 0, d = 0, ss = 0, dd = 0, ddd = 0, sss = 0, it = 0, S = 0, D = 0, S_block = 0, p = 0;
    int zero = 32, one = 4, two = 32, three = 32;
    int data_data = 0;
    int S_frame = 0;
    int n = 15, k = 7, m = 4, t = 0, q = 0, bit = 0;

    data_data = Length * 8;
    S_frame = ceil((float)data_data / m);
    S = n * ceil((float)S_frame / k) - (k - (S_frame % k));
    D = ceil((float)S / n);
    S_block = n * D;
    p = n - (S % n);

    // bits that we have to eliminate in the puncture part
    if (p < 7)
    {
        for (t = 0; t < p; t++)
        {
            z[t] = (n - p + 1) * D + ((t * D) - 1); // index of data to be eliminated
            t = 0;
        }
    }

    while (i < S_block + 1)
    {
        l = (i % D) * n + floor(i / D); // position
        ddd = l / 8;

        switch (ddd)
        {
            case 0:
```
```c
sss=32-zero;
zero=(zero-4);
break;

case 1:
    if (it==1)
    {
        sss=32-one;
        one=(one-4);
    }
    else
    {
        sss=32-one;
        one=(one-4);
        it=1;
        one=32;
    }
break;

case 2:
    sss=32-two;
    two=(two-4);
break;

case 3:
    sss=32-three;
    three=(three-4);
break;
}

bit=(data_entrada[ddd]>>sss)&0xf;

if (i!=z[t] || (p>=7 && i==0)) //if i=z[t], we puncture
{
    dd=q/32;
    ss=q%32;

    if (ss==0)
    {
        glinter_vector[dd]=0x0000;
    }
    else
    {
        glinter_vector[dd]=(bit<<ss)|glinter_vector[dd];
    }
}
else
{
    t++;
    q=q-4;
}
q=q+4;
i++;
```
null
cc = bb / 2;

if (bb % 2 == 0)
{
    buffer12 = (buffer_12[cc] & 0x0f);
    buffer14 = (buffer_14[cc] & 0x0f);
    buffer6 = (buffer_6[cc] & 0x0f);
    buffer13 = (buffer_13[cc] & 0x0f);
    buffer4 = (buffer_4[cc]) & 0x0f;
    buffer3 = (buffer_3[cc]) & 0x0f;
    buffer9 = (buffer_9[cc]) & 0x0f;
}
else
{
    buffer12 = (buffer_12[cc]) & 0xf0;
    buffer14 = (buffer_14[cc]) & 0xf0;
    buffer6 = (buffer_6[cc]) & 0xf0;
    buffer13 = (buffer_13[cc]) & 0xf0;
    buffer4 = (buffer_4[cc]) & 0xf0;
    buffer3 = (buffer_3[cc]) & 0xf0;
    buffer9 = (buffer_9[cc]) & 0xf0;
}

first_register = buffer12;
second_register = (buffer14 ^ ant_first_register);
third_register = (buffer6 ^ ant_second_register);
fifth_register = (buffer13 ^ ant_third_register);
six_register = (buffer4 ^ ant_fifth_register);
seven_register = (buffer3 ^ ant_six_register);
eight_register = (buffer9 ^ ant_seven_register);

if (bb % 2 == 0)
{
    first_register = (first_register & 0x0f) | (first_register << 4);
    second_register = (second_register & 0x0f) | (second_register << 4);
    third_register = (third_register & 0x0f) | (third_register << 4);
    forth_register = (forth_register & 0x0f) | (forth_register << 4);
    fifth_register = (fifth_register & 0x0f) | (fifth_register << 4);
    six_register = (six_register & 0x0f) | (six_register << 4);
    seven_register = (seven_register & 0x0f) | (seven_register << 4);
    eight_register = (eight_register & 0x0f) | (eight_register << 4);
}
else
{
    first_register = (first_register & 0xf0) | ((first_register >> 4) & 0x0f);
    second_register = (second_register & 0xf0) | ((second_register >> 4) & 0x0f);
    third_register = (third_register & 0xf0) | ((third_register >> 4) & 0x0f);
    forth_register = (forth_register & 0xf0) | ((forth_register >> 4) & 0x0f);
    fifth_register = (fifth_register & 0xf0) | ((fifth_register >> 4) & 0x0f);
    six_register = (six_register & 0xf0) | ((six_register >> 4) & 0x0f);
    seven_register = (seven_register & 0xf0) | ((seven_register >> 4) & 0x0f);
    eight_register = (eight_register & 0xf0) | ((eight_register >> 4) & 0x0f);
}

// we save here the values of the registers at t-1
ant_first_register = first_register;
ant_second_register = second_register;
ant_third_register = third_register;
ant_forth_register = forth_register;
ant_fifth_register = fifth_register;
ant_six_register = six_register;
ant_seven_register = seven_register;
ant_eight_register = eight_register;

if (i == 3 + xx && k6 == 0 && k != 0)
{
    if (yy == 0)
    {
        xx = xx + 3;
        yy = 1;
    }
elseif (yy==1)
{
    xx=xx+4;
    yy=0;
}

z=i+offset;
k=10;

if (iter==1)
{
    glRS_vector[z]=data_in;
    glRS_vector[z+1]=(0x0f & ant_eight_register)|(0xf0 & ant_seven_register);
    glRS_vector[z+2]=(0x0f & ant_six_register)|(0xf0 & ant_five_register);
    glRS_vector[z+3]=(0x0f & ant_fourth_register)|(0xf0 & ant_third_register);
    glRS_vector[z+4]=(0x0f & ant_second_register)|(0xf0 & ant_first_register);

    ant_first_register=0;
    ant_second_register=0;
    ant_third_register=0;
    ant_fourth_register=0;
    ant_five_register=0;
    ant_six_register=0;
    ant_seven_register=0;
    ant_eight_register=0;
    iter=0;
}

else
{
    glRS_vector[z]=(0x0f & data_in)|(ant_eight_register&0xf0);
    glRS_vector[z+1]=(0x0f & ant_seven_register)|(0xf0 & ant_six_register);
    glRS_vector[z+2]=(0x0f & ant_five_register)|(0xf0 & ant_fourth_register);
    glRS_vector[z+3]=(0x0f & ant_third_register)|(0xf0 & ant_second_register);
    glRS_vector[z+4]=(0x0f & ant_first_register);

    ant_first_register=0;
    ant_second_register=0;
    ant_third_register=0;
    ant_fourth_register=0;
    ant_five_register=0;
    ant_six_register=0;
    ant_seven_register=0;
    ant_eight_register=0;
    iter=1;
}

offset=offset+4;
}
k=k+1;
    if (k==11)
    {
        k=0;
    }
}

if (i==3+j)
{
    glRS_vector[i+offset]=(0xf0 & data_in)|(glRS_vector[i+offset]& 0xf0);
    j=j+7;
}
elseif (i!=6+w)
{
    glRS_vector[i+offset]=data_in;
}
else
{
    glRS_vector[i+offset]=glRS_vector[i+offset];
    w=w+7;
}
}

j++;
RLL ENCODER

```c
void *manchester_encoding (int datos[], int Long)
{
    int i=0, d=0, s=0, p=0, e=0, f=0, g=0, bits=0, bits1=0;

    while (i<Long*8)
    {
        d=i/32; // the word you have to read from data
        s=i%32; //how many bits you have to shift

        bits= (datos[d]>>s) & 0x0001;
        p= g*2;
        e=i/16;
        f=i%16;

        if (f==0)
        {   
glmanch_vector[e]=0x0000;
g=0;
p=0;
        }
    }
```
if (bits==1) 
{ 
    bits1=2; 
} 
else 
{ 
    bits1=1; 
} 

glmanch_vector[e]=(bits1<<p)|glmanch_vector[e];
i++; 
g++; 
return (&glmanch_vector[0]);

GLOBAL VARIABLES

//Variables:
int data_mode=1;
    // data_mode=0 burst
    // data_mode=1 single
    // data_mode=2 packet
    // data_mode=3 dimmed OOK
int phy_modulation=0;
    // phy_modulation=1 VPPM
    // phy_modulation=0 OOK
int MCS_indication=4;
int MCS_indication_vector[6]={0,0,0,0,0,0}; // because we are working in PHY I
int n_frame=2;
int topology=2;
    // 1 visibility
    // 2 P2P
    // 3 star
    // 4 broadcast
int channel_num=3;
    //001  62nm
    //010  48nm
    //011  45nm
    //100  46nm
    //101  47nm
    //110  54nm
    //111 reserved
int phy_type=1;  // PHY I/ PHY II/ PHY III
int FEC_outer_codein=0;
int FEC_outer_codeout=0;
int FEC_inner_codein=0;
int FEC_inner_codeout=0;
int optical_clock=0;
int data_rate=0;

//SHR_function:
int SHR_length=0;
int glSHR_preamble[4];
int fip_field[2];

//PHR_function:
int PHR_1;
int glPHR_vector[20];

//PSDU_function:
int PSDU_length=50;
int PSDU_vector[30];
int PSDU_field=0;
int glPSDU_vector[2];
int PSDU_length1= 64;
int tail_bits_vector;

//Buffer_general:
int buffer_general[7];
int buffer_output[25];
int length=4;

//interleaver+puncturing
int interleaver[9];
int interleaver1[9];
int glinter_vector[50];

// convolutional encoder
shortint Long;
int glconv_vector[50];
int

data_value1[10]={0x00000000,0xffffffff,0x11111111,0x22222222,0x33333333,0xbbbbbbbb,0x88888888,
0x55555555,0xcccccccc,0xdddddddd};

//RS_encoder
int RS_vector[5];
char glRS_vector[50];

//RLL_encoder
int glmanch_vector[100];
PARAMETERS.H

#ifndef PARAMETERS_H_
#define PARAMETERS_H_
#endif

//FUNCTIONS:
extern int generate_PHR (int PHR_vector[], int index);
extern void generate_PSDU (int PSDU_vector[], int data_value[], int lenght_data);
extern int generate_SHR (int SHR_preamble[]);
extern unsigned short int Calc_CRC_C (int Buffer, int Len);
extern void *interleaver_puncturing_v1 (int data[], int Length);
extern void *manchester_encoding (int datos[], int Long);
extern void *Conv_encoder (int data_value[], shortint Long);
extern void *RS_encoder (int data_input[], int index);

//Variables:
extern int data_mode;
extern int phy_modulation;
extern int phy_type;
extern int MCS_indication;
extern int MCS_indication_vector[6];
extern int n_frame;
extern int topology;
extern int FEC_outer_codein;
extern int FEC_outer_codeout;
extern int FEC_inner_codein;
extern int FEC_inner_codeout;
extern int optical_cClock;
extern int data_rate;

//SHR_function:
extern int SHR_length;
extern int glSHR_preamble[4];
extern int fIp_field[2];
extern int TDP_pattern[60];

//PHR_function:
extern int glPHR_vector[20];

extern int channel_num;
extern int PHR_length;
extern int PHR_1;

//PSDU_function:
extern int PSDU_length;
extern int PSDU_vector[30];
extern int PSDU_field;
extern int PSDU_length1;
//interleaver
extern int glinter_vector[50];

//convolutional encoder
extern int encoder_buffer[20];
extern short int Long;
extern int data_value[10];
extern int glconv_vector[50];

//RS encoder
extern char glRS_vector[50];

//RLL encoder
extern int glmanch_vector[100];

//Buffer_general:
extern int buffer_general[7];
extern int buffer_output[25];
extern int length;