GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

Diseño y desarrollo de un equipo láser de iluminación espectacular

Autor: Alfonso Sanz Giner

Director: Javier Sánchez Ruiz

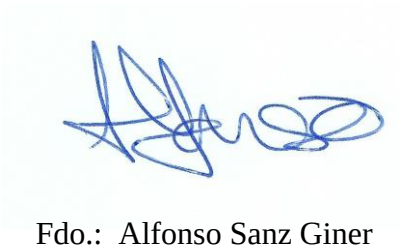
Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Diseño y desarrollo de un equipo láser de iluminación espectacular

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2019/20 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

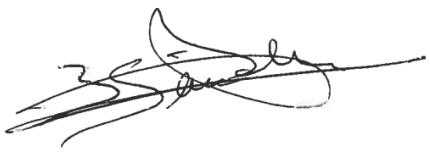


Fdo.: Alfonso Sanz Giner Fecha: 29 / 06 / 2020

Autorizada la entrega del Proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Javier Sánchez Ruiz Fecha: 29 / 06 / 2020



GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

Diseño y desarrollo de un equipo láser de iluminación espectacular

Autor: Alfonso Sanz Giner

Director: Javier Sánchez Ruiz

Madrid

DISEÑO Y DESARROLLO DE UN EQUIPO LÁSER DE ILUMINACIÓN ESPECTACULAR

Autor: Sanz Giner, Alfonso.
Director: Sánchez Ruiz, Javier.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

Este documento expone el contexto en el que surge este Proyecto, los objetivos que pretende conseguir enmarcados dentro del trabajo de fin de grado, la metodología utilizada para alcanzarlos, las herramientas y recursos a utilizar durante el mismo y el desarrollo realizado explicando en detalle todos los circuitos, componentes y programación involucrada.

Palabras clave: Programación, circuito, código, modelos de comunicación, componentes electrónicos, Trinkets, servomotores, Neopixels, XBee.

1. Introducción

Actualmente en la industria del espectáculo son muy comunes los efectos de iluminación espectacular con luz o láseres combinados con música. Sin embargo, estos suelen requerir equipos de alto coste y muy voluminosos.

2. Definición del Proyecto

El Proyecto consiste en el prototipado de un sistema de iluminación espectacular tipo "Láser Man", basado en tiras de leds y láseres controlados por servomotores empleando elementos de bajo coste y buscando una solución de reducido tamaño y gran escalabilidad.

El diseño constará de un conjunto de microcontroladores y otros elementos hardware (servomotores, displays y tiras de Leds) interconectados entre sí.

3. Descripción del sistema

El sistema contará con un microcontrolador "Máster" que mandará las instrucciones al resto de controladores "Slaves", los cuales ejecutarán en función de las mismas el control de los elementos que tengan conectados, ya sean servos o tiras de LEDs.

En el siguiente esquema se muestra el sistema con 4 controladores slaves, 1 que controla 2 servos y 3 que controlan 1 tira de LEDs cada uno.

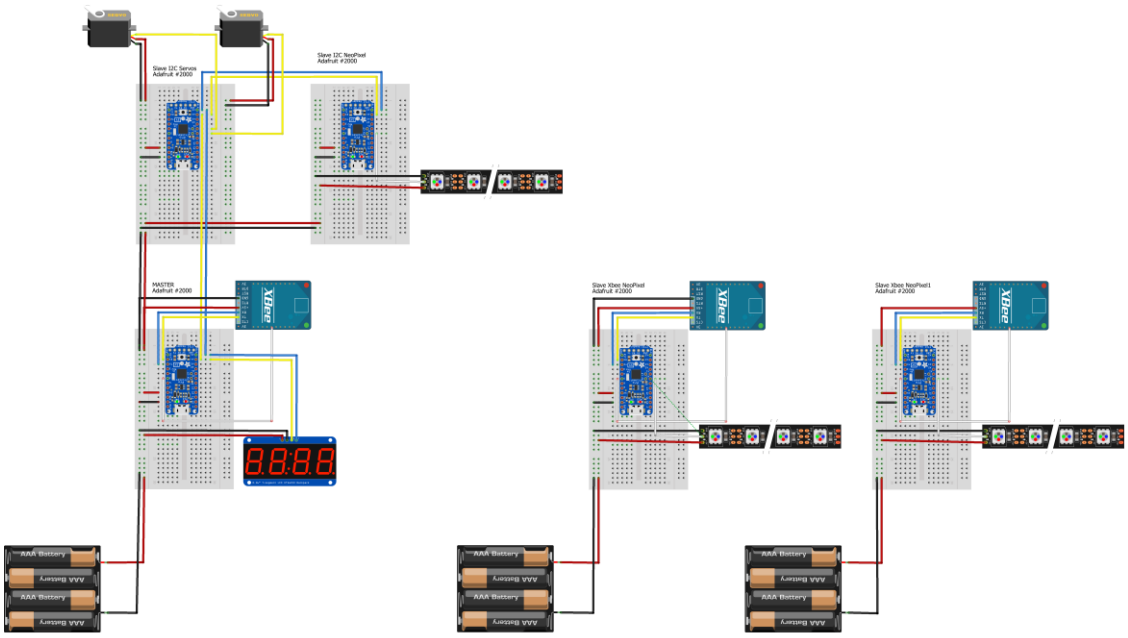


Ilustración 1 - Esquema de conexionado

Nótese la separación entre el controlador Máster (abajo a la izquierda) y los 2 slaves conectados por cable (arriba a la izquierda) con el resto de slaves comunicados por radio (a la derecha). Se entrará en más profundidad en el apartado del *Circuito*.

4. Resultados

El resultado del sistema diseñado a nivel de programación es el fácil modelado de un conjunto de efectos de luces simultáneos y organizados por medio de funciones sencillas y muy fáciles de modificar para recrear cualquier otra "coreografía".

A nivel de diseño del circuito y de hardware, el sistema admite hasta 8 servos y 51 tiras de LEDs por medio de 1 Máster y 4 slaves, siendo muy sencilla la ampliación a gran escala del número de slaves.

5. Conclusiones

Durante el desarrollo de este Proyecto se ha conseguido diseñar un modelo de conexionado fácil de replicar y una programación muy versátil que permite controlar una serie de LEDs y servos de forma sincronizada y sin riesgo de pérdida de datos durante la comunicación.

PROYECT SUMMARY

Keywords: Programming, circuit, code, communication models, electronic components, Trinkets, Sermotors, NeopPixels, Xbee.

1. Introduction

Currently in the entertainment industry, spectacular lighting effects with light or lasers combined with music are very common. However, these often require high-cost and very bulky equipment.

2. Definition of the Proyect

The Project consists of the prototyping of a “Laser Man” type spectacular lighting system, based on LED strips and lasers controlled by servomotors using low cost elements and looking for a solution of reduced size and great scalability.

The design will consist of a set of microcontrollers and other hardware elements (servomotors, displays and LED strips) interconnected with each other.

3. System description

The system will have a "Master" microcontroller that will send the instructions to the rest of the "Slaves" controllers, which will execute the control of the elements they have connected, whether they are servos or LED strips.

The final prototype consists of a system of 4 slave controllers, 1 that controls 2 servos and 3 that control 1 LED strip each.

The Master controller and the 2 slaves connected by cable will be separated from the rest of the slaves that communicate by radio. We will go into more depth in the Circuit section.

4. Results

The result of the designed system at the programming level is the easy modeling of a set of simultaneous and organized lighting effects with simple functions which are very easy to modify to recreate for any other “choreography”.

At the circuit and hardware design level, the system supports up to 8 servos and 51 LED strips controlled by 1 Master and 4 slaves, making it easy to expand the number of slaves on a large scale.

5. Conclusions

During the development of this Project, it has been achieved a design with connection model that is easy to replicate and a very versatile programming that allows to control a series of LEDs and servos in a synchronized way and without risk of data loss during communication.

Índice de la memoria

Índice de figuras

Capítulo 1. INTRODUCCIÓN

1.1 MOTIVACIÓN DEL PROYECTO

Este Proyecto pretende demostrar que se puede desarrollar una solución de bajo coste y reducido tamaño, con elementos sencillos (tiras de LEDs RGB, láseres controlados por servomotores y LEDs de alta potencia) sincronizados por varios microcontroladores comunicados vía radio y programados adecuadamente para conseguir efectos de iluminación avanzados y visualmente impactantes.

A lo largo del documento se expondrán los componentes teniendo en cuenta el apartado económico y comparando sus especificaciones y funcionalidad con lo que serían equivalentes de mercado de mayor precio, demostrando así la poca diferencia que existe en la práctica y lo rentable que puede ser la implantación de este sistema frente a otros ejemplos de mayor presupuesto que se mencionarán en el siguiente apartado.

Este documento hace también la labor de documentación en caso de que se desee replicar el diseño pudiendo reutilizar toda la programación y requiriendo tiempo solo para la instalación.

Capítulo 2. ESTADO DE LA CUESTIÓN

Este Proyecto se inspira en equipos de iluminación destinados al espectáculo, ya sean tipo "Láser Man" [1] en el cual se utilizan una gran cantidad de focos, láseres, motores, humo… para lograr una coreografía entre una persona y varios equipos de iluminación; o instalaciones con iluminación animada con objetivos estéticos como el "Shenzhen Light Show" [2] en el cual se iluminaron 43 edificios con una serie de LEDs sincronizados por una unidad central.

El coste de estos equipos es muy alto, y por norma general son extremadamente voluminosos. En el caso del "Shenzhen Light show" se estima en una inversión de quince millones de euros, incluyendo más de 1,5 millones de luces que cubren 43 edificios con un exterior rediseñado para esta actuación. Sin embargo, no son tecnológicamente complejos por lo que parece factible conseguir efectos similares empleando elementos comerciales de fácil adquisición situados y sincronizados adecuadamente.

La innovación en el ámbito de la tecnología LED ha sido lento desde su invención en 1927, pero su sencillez ha logrado su gran expansión en componentes electrónicos y finalmente en la vida cotidiana sustituyendo las bombillas incandescentes y focos de los coches.

Su uso decorativo es más reciente, siendo hoy en día muy común en ordenadores y edificios, en el *Anexo I* se describirá el aporte económico y de innovación que pretende el Proyecto.

Capítulo 3. DEFINICIÓN DEL TRABAJO

3.1 OBJETIVOS

El objetivo de este Proyecto es diseñar y desarrollar un sistema de iluminación espectacular tipo "Láser man" programable, escalable y versátil. Estará basado en varios microcontroladores tipo Arduino comunicados entre sí vía radio que controlen a su vez varios elementos de iluminación (tiras de LEDs, Leds RGB y láseres) para lograr los efectos deseados.

Toda la programación se realizará en lenguaje Arduino (muy similar a C++) utilizando la herramienta de Arduino IDE que permite la compilación y subida del código al microprocesador.

Los microcontroladores empleados serán Trinket Pro de 5V por su reducido tamaño y bajo consumo, ya que están basados en la arquitectura de Arduino, su programación es bastante sencilla y son compatibles con la Herramienta Arduino IDE.

Se conectarán a diferentes sistemas de iluminación como tiras de LEDs, servomotores y focos de alta potencia.

Se utilizarán módulos de radio programables para comunicar todos los microcontroladores entre sí mediante un protocolo Serie con dos cables (RX y TX).

La comunicación por radio permitirá enviar instrucciones a los distintos componentes como los efectos de luz que han de efectuar las tiras de LEDs o los movimientos de los servos.

Los servos se podrán utilizar para mover otros elementos como espejos o láseres para crear los distintos efectos de iluminación.

3.2 METODOLOGÍA

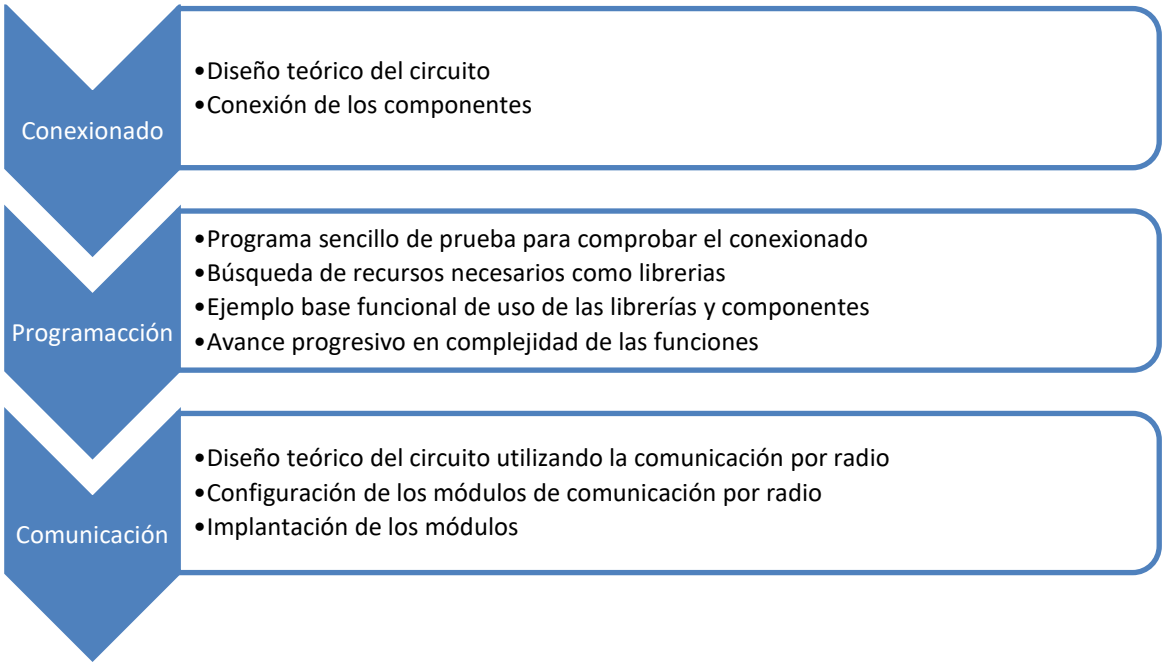
El Proyecto cuenta con varios procesos diferentes a efectuar en paralelo:

- **Programación** del conjunto de las librerías y funciones que utilizarán los microcontroladores para controlar los distintos componentes y del protocolo de comunicación vía radio.
- **Conexionado** de todos los componentes: microcontroladores, módulos de radio, y los componentes controlados, como los LEDs y servos.
- **Prueba** de los distintos componentes cableados.
- **Comunicación** de los distintos elementos por medio de los módulos de radio para posibilitar el control remoto de cada uno de los microcontroladores.

Para comprobar el correcto funcionamiento de los diferentes componentes se han de realizar los dos primeros procesos simultáneamente, comenzando con programas sencillos que requieren pocos componentes y avanzando hasta conseguir un programa final que hace uso de una completa conexión coordinada de todos los componentes.

Una vez se hayan programado los componentes de control y conectado los elementos a controlar se procederá a implementar en el sistema el protocolo de comunicación por radio. Esto requerirá volver al proceso de conexionado incluyendo los módulos de radio.

3.3 PLANIFICACIÓN



En todos los procesos se requerirá una pequeña **fase** de investigación de los componentes o herramientas seguida de un fase de testeo o prueba de los mismos (algunos componentes requerirán su construcción/soldadura). Posteriormente, se requiere el diseño base o prueba de concepto conectando y programando los componentes para asegurarse que el objetivo es posible por medio de tal diseño y, finalmente, se pulirá el diseño aumentando su complejidad tanto a nivel de programación como de hardware para lograr un sistema fiable y versátil.

Durante todo este proceso habrá que depurar los errores cuya resolución requerirá un análisis del sistema completo: funciones nuevas en la programación pueden destapar errores en el conexionado y viceversa, mejoras en el conexionado que puedan requerir el rediseño de funciones. Serán necesarias una gran cantidad de **pruebas** para encontrar la raíz de los posibles fallos que puedan surgir.

Debido a estas complicaciones que puedan surgir, la **duración** de cada fase no se puede estimar y un proceso simple puede durar horas si no se encuentra documentación sobre el error y hay que revisar todo el montaje para descartar posibles fuentes del mismo.

3.4 ESTIMACIÓN ECONÓMICA

Sin contar el coste energético del uso del ordenador durante la programación o consumo de los componentes electrónicos (este último siendo mínimo a excepción de las tiras de LEDs), el único coste asociado al Proyecto es el de los componentes electrónicos como tal.

A continuación, se enumerarán los componentes utilizados, así como su precio aproximado, en el siguiente apartado se explicará cada uno de ellos.

- Adafruit Trinket Pro 5V — 10€ x 5 unid.
- Adafruit NeoPixel Strips — 60€/m x 1m.
- DIGI Xbee S1 — 20€ x 3 unid.
- Adafruit XBee Adapter kit - v1.1 — 20€ x 3 unid.
- Waveshare XBee adaptador USB UART — 8€ x 1 unid.
- SG92R de Tower Pro — 5€ x 3 unid.
- Adafruit 1.2" 7-segment Backpack — 15€ x 1 unid.
- Cables de cobre estañado — 0.20€ x 1 m
- Adaptador FTDI 5 pines hembra a USB — 2.50€ x 2 unid.
- Protoboard Adle/Board de 1690 contactos — 20€ x 1 unid.
- Fuente de alimentación Sunpower SPS-025-05 — 30€ x 1 unid.

Soldador y herramientas (alicates corta cables, cinta aislante …) no se incluirán en el coste.

TOTAL: 323,20€

El total calculado es para las cantidades utilizadas en el prototipo, pero como se ha mencionado anteriormente uno de los principales objetivos del diseño es la escalabilidad y, por ello, es posible ampliar el número de microcontroladores (sin limitación), módulos de radio, servos (8 por microcontrolador dedicado) y tiras de LEDs (17 por microcontrolador).

Capítulo 4. DESCRIPCIÓN DE LAS TECNOLOGÍAS

4.1 HARDWARE: COMPONENTES ELECTRÓNICOS

A continuación, se exponen los componentes electrónicos utilizados, se explicará brevemente su funcionalidad y funcionamiento, así como el método de uso.

Se usarán componentes de la compañía Adafruit Industries [3] debido a su accesibilidad, precio reducido, compatibilidad entre los mismos y cantidad de código abierto en forma de librerías. Se trata de una compañía estadounidense centrada en el diseño de componentes electrónicos que se ha extendido mundialmente gracias a su venta online.

4.1.1 MICROCONTROLADORES

Los microcontroladores a utilizar son los **Adafruit Trinket Pro 5V** [4].

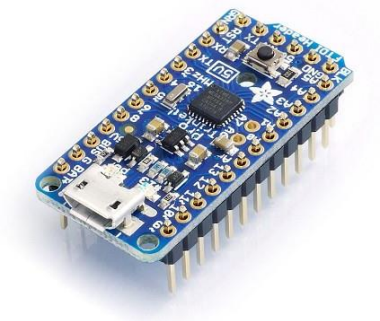


Ilustración 2 - Adafruit Trinket Pro 5V

Se trata de placas de controlador compatibles con la tecnología de código abierto de Arduino IDE (véase el *apartado 4.3.1*), es el producto equivalente al Arduino Nano [5] con ciertas diferencias y de coste muy inferior.

Los Trinkets Pro están basados en el comúnmente utilizado ATmega328 (con una velocidad de 16 MHz a 5V), así como 18 pines GPIO (general-purpose input/output), 28 Kb de memoria Flash y 2 Kb de RAM.

Disponen de una gran cantidad de pines que permitirán la conexión de los componentes necesarios, siendo los más importantes y los que vamos a utilizar:

- Los pines de voltaje: **5V**, **GND**. No serán necesarios los de BUS ni BAT+ aunque pueden ser utilizados de forma sustitutiva durante el desarrollo o la ejecución final.
- 14 pines de salida digital (del 3 al 13, RX y TX), de los cuales solo se utilizarán uno para la programación de los módulos de radio (concretamente el **pin 5**) y los pines **RX** y **TX** para la comunicación Serial (véase el *apartado 5.4.1*).
- 5 pines analógicos, que se podrán utilizar para controlar los servomotores, las tiras de LEDs, el display de 7 segmentos y la comunicación I2C (véase el *apartado 5.4.2*).

A su vez, permiten la conexión de un header FTDI en caso de que sea necesario hacer debugging [6], y de un puerto microUSB para su programación y opcionalmente alimentación (durante el desarrollo es posible alimentar algún componente a través del microcontrolador alimentado por el microUSB para evitar conectar la batería cuando hay pocos componentes, teniendo en cuenta su potencia limitada).

A partir de este apartado se designarán a los microcontroladores Trinkets para abreviar.

4.1.2 TIRAS DE LEDS

Las tiras de LEDs RGB **Adafruit NeoPixel Strips** [7] disponen de 30, 60 o 144 LEDs por metro y aportan una gran potencia lumínica: 9W, 18W y 43W respectivamente (1.8A, 3.6A y 8.6A a 5V). En este caso se utilizarán las tiras de 60 LEDs por metro pudiendo recortarse en secciones de menor tamaño.

Ilustración 3 - Adafruit NeoPixel

Estas tiras requieren solamente de alimentación y una señal de control. La señal utilizada se programará a través de una librería de Adafruit (*Adafruit_NeoPixel*), pero en este caso se utilizará además una librería aparte que nos permitiría controlar los LEDs individualmente para hacer efectos personalizados (*Ligths*).

4.1.3 MÓDULOS DE RADIO Y ADAPTADORES

Se utilizarán los **DIGI Xbee S1** [8] que se conectarán por comunicación Serial (véase el *apartado 5.4.1*) a los microcontroladores Máster y Slave y que permitirán la comunicación inalámbrica entre ellos.

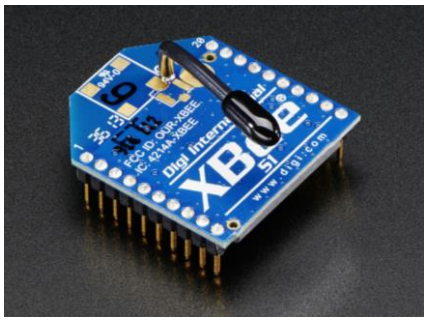


Ilustración 4 - DIGI Xbee S1

Estos módulos son programables a través del modo API (véase el *apartado* 5.5).

La configuración de estos módulos se hará a través de XCTU (véase el *apartado* 4.3.2) conectándolo al ordenador a través de un programador

Para poder conectarlos a la misma alimentación que el resto de componentes y facilitar su conexión se conectarán a través del siguiente **adaptador** [9], que requerirá de su respectivo proceso de soldadura (ver *apartado* 4.2.2):

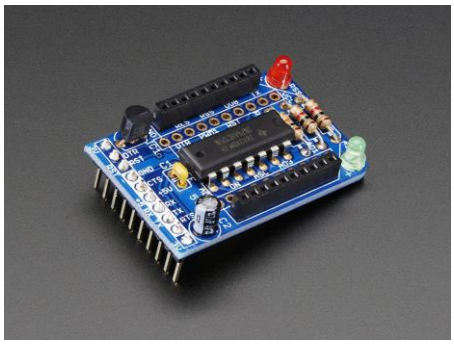


Ilustración 5 - Adaptador Xbee

Finalmente, para poder configurarlos se conectarán al ordenador por medio de un adaptador microUSB fabricado por WaveShare [10], que permitirá usar los botones de BOOT y RESET (sin tener que cortocircuitar manualmente el pin de RST a GND). Esto permitirá no solo configurarlos con XCTU sino también cambiar la versión del firmware a una anterior para que coincida en todos (para actualizarlos no hace falta, pero para hacer un "downgrade" de la versión sí).

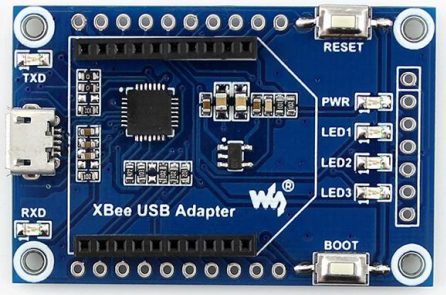


Ilustración 6 - Programador Xbee

El resto de la configuración se podrá realizar a través de la conexión FTDI.

Para evitar confundir este adaptador microUSB con el adaptador que se utilizará durante el Proyecto, se llamará a este "**programador**" de los Xbee.

4.1.4 SERVOMOTORES

Para la validación del prototipo se utilizará el modelo **SG92R de Tower Pro** [11], con un coste muy comedido (aprox. $2 cada uno) pero baja calidad en términos de precisión, fluidez y ruido.

Ilustración 7 - Servomotores SG92R

Para el control de los láseres se utilizarán servos de mayor calidad, pero debido a que los movimientos necesarios son sencillos y se trata de una prueba de concepto, este modelo es más que suficiente.

Serán controlados por una señal PWM (modulación de ancho de pulso) debido a que se utilizará una librería pública llamada *VarSpeedServo*, que hace uso a su vez de la librería integrada de Arduino para el control de Servos, pero aportando una mayor funcionalidad.

4.1.5 7SEG DISPLAY

Display de 7 segmentos de 4 dígitos de Adafruit modelo **1.2" 7-segment Backpack** [12], dispone también de 2 dos pares de puntos de minutos/horas y un punto para decimales, además de 16 intervalos de potencia para mejorar su visibilidad.

Ilustración 8 - 7 Segment Display

Se utilizará para mostrar un temporizador con el tiempo de ejecución del programa y comprobar el correcto sincronismo del resto de componentes.

Se hará uso de la librería *CountUpDownTimer* que ha sido modificada para utilizar los puntos situados entre los minutos y segundos como medios segundos.

4.1.6 CABLEADO Y SOLDADOR

Se utilizarán cables de cobre estañado recubiertos en silicona de colores (suministrados por la Universidad) para la conexión en la protoboard, junto con otros adaptadores de pines (macho-hembra) y headers soldados a estos cables para facilitar su enganche para elementos como los servos y el display.

El soldador de estaño permitirá unir estas conexiones, así como soldar ciertos componentes que vienen como elementos sueltos de fábrica (véase el *apartado 4.2*).

4.1.7 CABLES ADAPTADORES

Para la conexión de los Trinkets con el ordenador se pueden utilizar cables FTDI a USB o a través de un adaptador microUSB-USB, teniendo en cuenta que este último no permite la depuración del código en caso de errores.

Ilustración 9 - Cable FTDI a USB

Para la conexión de los Xbee y su programación se puede utilizar el mismo cable FTDI a USB o el programador mencionado en el *apartado 4.1.3*.

4.1.8 PROTOBOARD

Modelo de Adle/Board de 1690 contactos a la que se conectarán todos los componentes tanto para su conexión entre sí como alimentación, exceptuando las tiras de LEDs cuya alimentación estará soldada directamente a la batería.

Ilustración 10 - Adle/Board Breadboard

Los adaptadores Xbee se pueden conectar verticalmente a la protoboard para fijar su posición si se han soldado los pines macho inferiores (de la conexión FTDI: RX, TX, 5V y GND) de forma paralela al adaptador en vez de por los agujeros de dichos pines. Sino tendrán que conectarse con un cable de pin macho a hembra o soldando un cable de estaño.

4.1.9 FUENTE DE ALIMENTACIÓN

Fuente de alimentación Sunpower SPS-025-05 [13] de 5V y 5A para una potencia de 25W con un valor R&D de 50mV y una eficiencia de 0,75%.

Ilustración 11 - Sunpower SPS-025-05

Esta potencia es suficiente para alimentar las tiras de LEDs (18W por metro) y el 7Seg Display. El resto de componentes (Trinkets, Xbees y servos) tienen un consumo muy bajo y pueden incluso utilizar la propia alimentación de los Trinkets por microUSB.

4.2 HARDWARE: SOLDADURA Y MONTAJE

Algunos componentes vienen ya soldados de fábrica o se han soldado anteriormente para otros proyectos, pero en la mayoría de los casos los componentes electrónicos de Adafruit se venden con sus elementos (LEDs, resistencias, condensadores…) sueltos para abaratar el precio y requieren soldar el producto final.

Es el caso de los siguientes componentes:

4.2.1 TRINKETS

Estos microcontroladores no requieren de soldar ningún elemento para su funcionamiento, pero sí que se necesitarán soldar headers macho (incluidos con el producto) para poder conectarlos a la protoboard (alternativamente se puede utilizar un IC Socket compatible que vende Adafruit [14]).

Se recortarán los headers macho (que cuentan con pines por ambos lados) en 2 tiras de 12 pines que se introducirán y soldarán a ambos lados del Trinket. No se utilizarán estos headers para los pines 6 pines del header FTDI, debido a que en caso de querer utilizar esta conexión se utilizará un adaptador de FTDI a USB para conectarlo al ordenador (ver *apartado 4.1.7*).

Ilustración 12 - Tira de pines (headers) macho

4.2.2 ADAPTADOR XBEE

Los adaptadores que se han comentado anteriormente tendrán el proceso de soldadura más laborioso requiriendo soldar:

- Un regulador de voltaje lineal de 3,3V 250mA
- Un chip con varios buffers de 5V
- Un LED rojo
- Un LED verde
- 2 resistencias de 1 kΩ 1/4W
- Una resistencia de 10 kΩ 1/4W
- Un condensador cerámico de 0.1 µF
- Un condensador de mínimo 47 µF de 4V
- Los headers macho (3 pares de 10 pines, colocando los inferiores verticalmente de forma opcional para conectarlo directamente a la protoboard)
- Los headers hembra para conectar el Xbee (2 pares de headers de 10 pines)

El proceso es el que se muestra en el siguiente enlace y no se entrará en detalle debido a su simplicidad a pesar del número de elementos [15].

4.2.3 CABLES

Debido a que se utilizarán cables cortados para las conexiones a las protoboard conviene en ciertos casos soldar el final de uno de los cables a un pin macho para fijar su posición.

También se soldarán cables a la fuente de alimentación.

4.3 SOFTWARE: PROGRAMAS

Los programas necesarios para configurar o programar los componentes son:

4.3.1 ARDUINO IDE (ENTORNO DE DESARROLLO INTEGRADO) [16]

Programa enormemente extendido para la programación de microcontroladores con el mismo nombre. Como se ha mencionado anteriormente, los Adafruit Trinket Pro son compatibles con el mismo debido a la utilización del microchip ATmega328P.

Permite la programación de un código (en "sketchs") en C++, su compilación y subida del mismo a la memoria de los Trinkets. También permite la depuración del código, pero en nuestro modelo de Trinkets requerirá la conexión FTDI, no se permite a través del microUSB y, por lo tanto, sólo se utilizará en caso de haber errores difíciles de localizar.

El programa cuenta con una serie de librerías integradas como la del control de Servos y los protocolos de comunicación Serial y I2C. Pero también se utilizarán una serie de librerías públicas o propias para ayudar al manejo de los componentes.

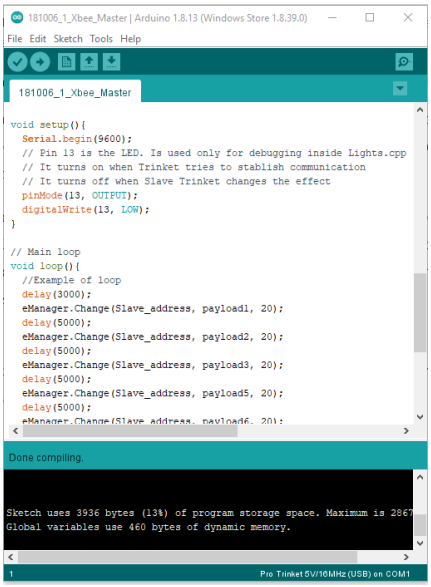


Ilustración 13 – Sketch de Arduino IDE

4.3.2 XCTU [17]

Programa gratuito y de código abierto creado por la propia compañía de los módulos de radio Xbee: Digi. Permite la configuración, control de versiones del firmware y diseño de la arquitectura de los Xbee mediante una interfaz gráfica sencilla.

Se explica en detalle en el apartado de *Configuración de los Xbee*.

4.3.3 AUTODESK TINKERCAD [18]

Herramienta de diseño y simulación de circuitos y programación de Arduinos.

Inicialmente se iba a utilizar esta misma para el esquema del sistema pero la falta de componentes y su incompatibilidad en la simulación obligó a descartarse.

Se realizaron los siguientes diseños básicos con su respectiva programación:

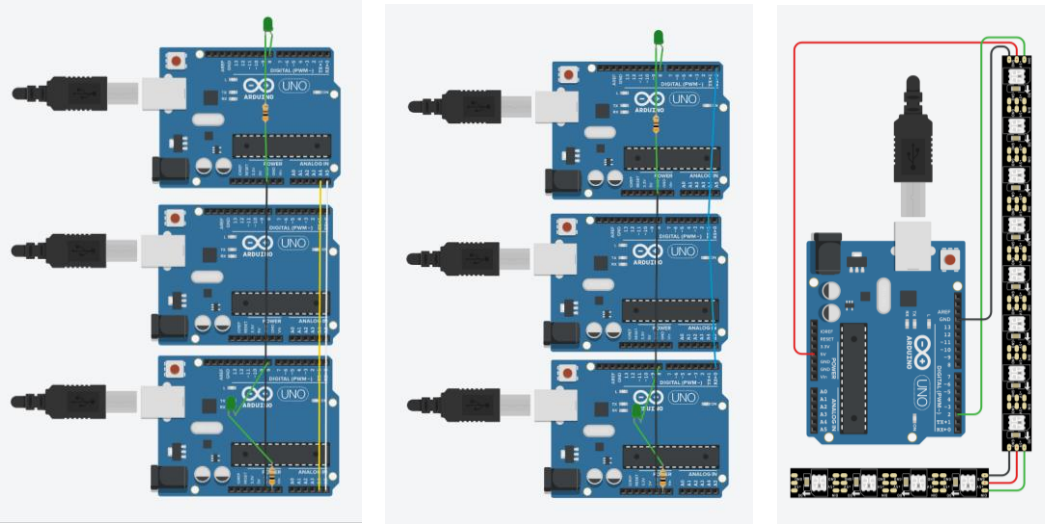


Ilustración 14 - Diseños Autodesk TinkerCAD

El primero es una prueba de la comunicación *I2C*, el segundo por *Serial* y el tercero del control de *NeoPixeles*. La programación de estos se utilizó con alguna modificación al inicio del Proyecto en *Arduino IDE* adaptándola a los Trinkets.

4.3.4 FRITZING [19]

Herramienta de diseño de circuitos con la que se ha realizado el esquema teórico del conexionado para su exposición de manera visual.

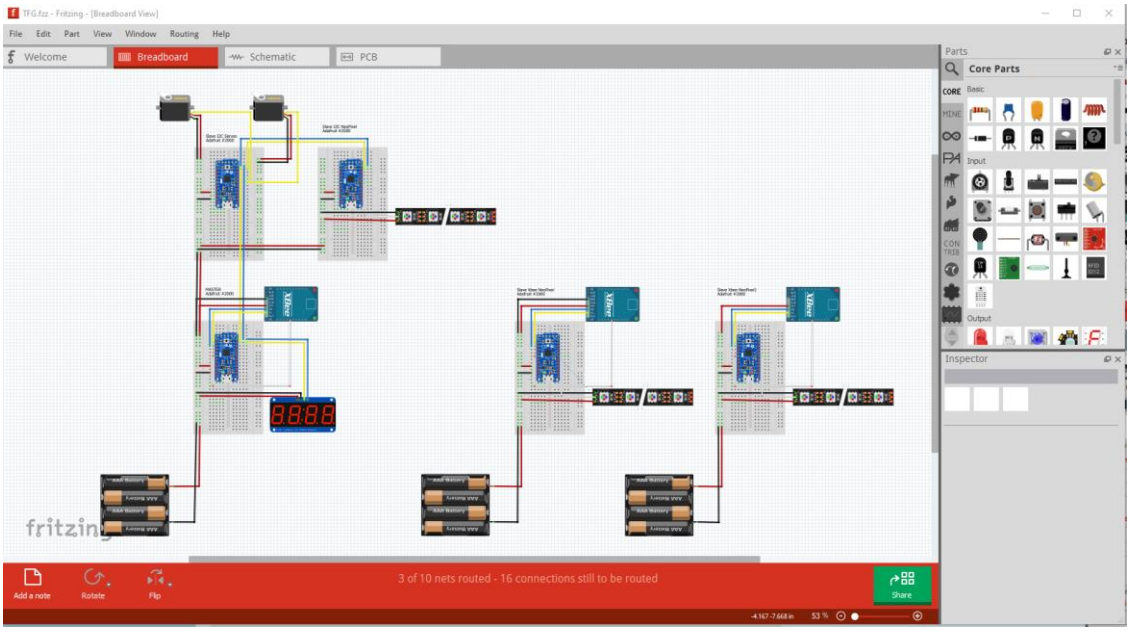


Ilustración 15 - Fritzing

4.3.5 HYPERTERMINAL [20]

Emulador para Windows de un terminal que permite comprobar el funcionamiento de los Xbee, siendo posible la configuración de los mismos por medio de comandos, aunque no será necesario gracias a la interfaz mucho más sencilla y cómoda de XCTU.

Capítulo 5. SISTEMA/MODELO DESARROLLADO

Como se ha indicado anteriormente, se utilizará el modelo asimétrico de comunicación "Master-Slave", con 1 Trinket como Máster actuando como centro de control del resto de Trinkets llamados slaves.

Durante toda la explicación del modelo se diferenciarán los Trinkets por 4 tipos a nivel tanto de conexionado como de programación; según su función Máster/slave, elementos de control (servos o NeoPixeles) y modelo de comunicación (Serial por Xbee o I2C por cable):

- Máster
- Slave de control de servos por I2C
- Slave de control de NeoPixeles por Serial
- Slave de control de NeoPixeles por I2C

5.1 CIRCUITO

En este apartado se describe el conexionado de cada componente. El siguiente esquema muestra el sistema completo, mientras que los siguientes apartados describen cada parte del mismo, siendo todas ellas muy parecidas y sencillas dado que son componentes electrónicos de alto nivel (no resistencias, transistores…) y la complejidad de su uso radica en la programación.

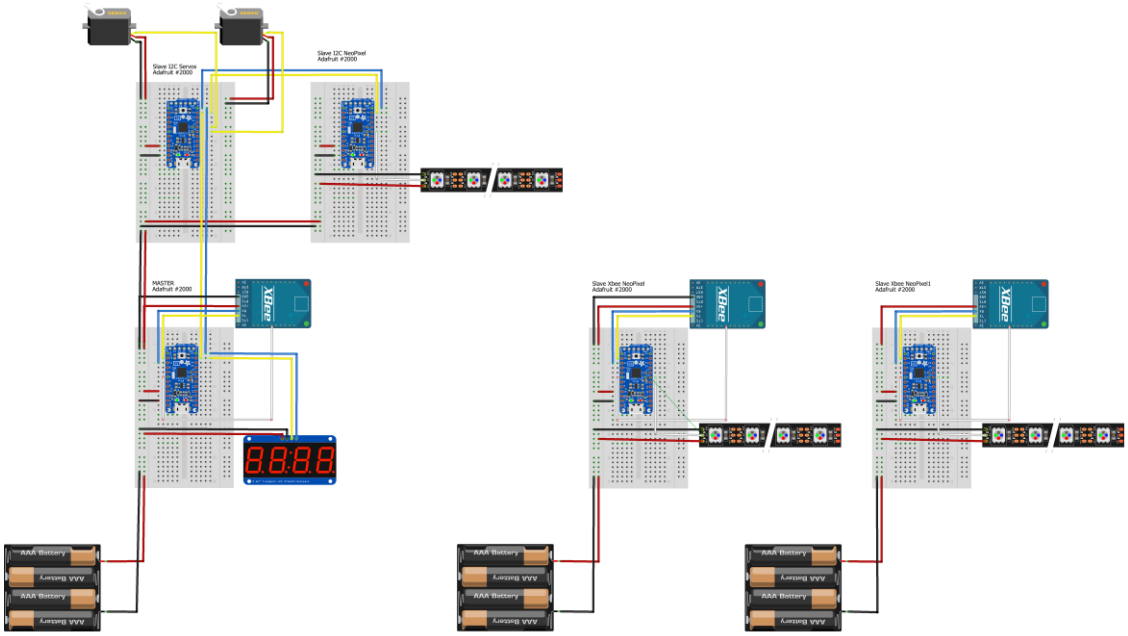


Ilustración 16 - Esquema del sistema de conexionado

En el esquema se muestra cada Trinket en una protoboard para su visualización, pero para el prototipo desarrollado se ha utilizado una placa protoboard en vez de múltiples.

Todos los componentes (Trinkets, servos, Xbee y display) tendrán conectados los cables de alimentación (mostrados como +5V en **rojo** y GND en **negro).** Además, según la función de cada uno de los Trinkets, tendrán conectados los elementos que se describen a continuación.

5.1.1 CONEXIÓN DEL MÁSTER

El Máster tendrá además conectados:

- Los cables de la comunicación Serial con el Xbee al RX en **azul** y al TX en **amarillo**
- Los cables de comunicación I2C con el resto de Trinkets y el display de 7 segmentos por los cables SDA en **amarillo** al pin A4 y SCL en **azul** al pin A5
- El cable de programación del Xbee en **blanco** conectado al pin digital 5

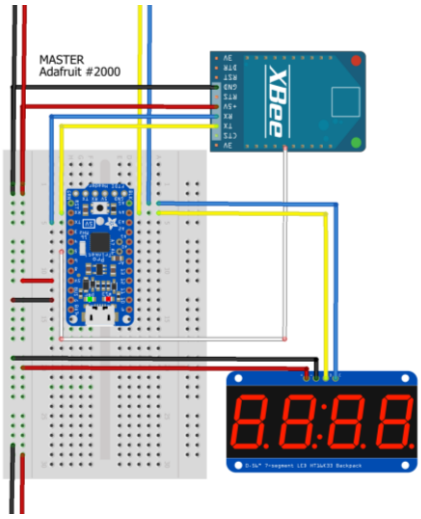


Ilustración 17 - Esquema de la conexión del Máster

5.1.2 CONEXIÓN DEL SLAVE DE SERVOS

El slave de control de los servos tendrá conectados:

- Los cables de comunicación I2C con el resto de Trinkets por los cables SDA en **amarillo** al pin A4 y SCL en **azul** al pin A5
- Los cables de control de los servos por señal PWM en los pines A0, A1, A2, A3 y tantos pines digitales como se requieran (hasta alcanzar un máximo de 8 servos)

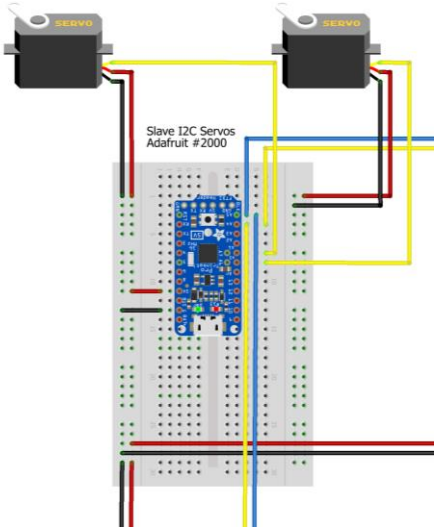


Ilustración 18 - Esquema de conexión del slave de servomotores

5.1.3 CONEXIÓN DEL SLAVE DE NEOPIXELES POR SERIAL

Los slaves de control de los NeoPixeles por Serial tendrán conectados:

- Los cables de la comunicación Serial con el Xbee al RX en **azul** y al TX en **amarillo**
- Los cables de control de los NeoPixeles en cualquier pin analógico o digital
- El cable de programación del Xbee en **blanco** conectado al pin digital 5

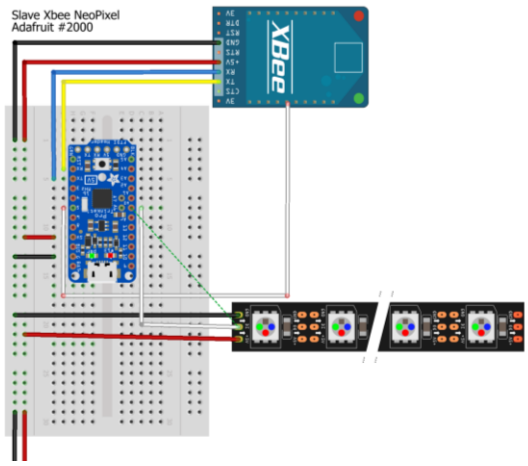


Ilustración 19- Esquema de conexión del slave de NeoPixeles por Serial

5.1.4 CONEXIÓN DEL SLAVE DE NEOPIXELES POR I2C

Los slaves de control de los NeoPixeles por I2C tendrán conectados:

- Los cables de comunicación I2C con el resto de Trinkets por los cables SDA en **amarillo** al pin A4 y SCL en **azul** al pin A5
- Los cables de control de los NeoPixeles en cualquier pin analógico o digital

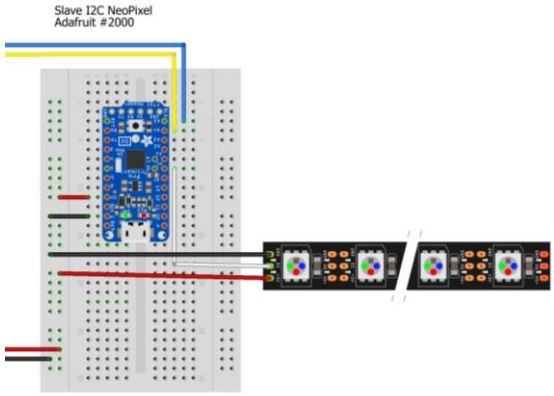


Ilustración 20 - Esquema de conexión del slave de NeoPixeles I2C

5.2 PROGRAMACIÓN

Los únicos elementos a programar serán los Trinkets diferenciando nuevamente a los 4 tipos con diferentes códigos y librerías asociadas.

Para poder comprender el código de cada Trinket es necesario explicar las funciones y clases que aportan las librerías debido a que el uso de las mismas conforma la mayor parte del código.

También se ha de mencionar que la explicación del código se ha realizado de la forma más sencilla posible pero requiere de nociones básicas de programación.

5.2.1 LIBRERÍAS

Las bibliotecas o librerías son un conjunto de recursos (generalmente funciones o clases) definidas fuera del código principal que facilitan la programación aportando una interfaz para el manejo de los componentes.

Las librerías pueden estar integradas en *Arduino IDE* de forma implícita si no es necesario incluirlas, o explícita en caso de que sí lo sea.

Además de las librerías integradas, también se hará uso de librerías externas que pueden ser públicas si son de acceso libre en interne; privadas, si no son de acceso libre o se han creado para el Proyecto, o personalizadas, si se han hecho modificando una librería pública.

A continuación, se enumerarán las librerías a utilizar y se explicarán las funciones que se usarán en el código del Proyecto:

5.2.1.1 VarSpeedServo

A pesar de que Arduino ya cuenta con una librería integrada para el control de servos, se utilizará una librería pública de GitHub [21] que permite el control de la velocidad sin necesidad de utilizar bucles con control de tiempos.

Esta librería soporta hasta 8 servos controlados simultáneamente por cada Trinket y de forma asíncrona, es decir, no es necesario que termine el movimiento de un servo para comenzar el movimiento de otro.

Esta característica con la librería integrada solo sería posible a través de interrupciones con bucles para introducir delays (ejemplo: pidiendo cada ciertos milisegundos cambiar la posición del servo en 1 grado) e introduciría problemas de sincronismo en caso de alta carga de procesamiento.

La librería es muy simple, solo requiere definir el objeto/clase `VarSpeedServo`, enlazarlo a un pin con la función `attach()` y mandarle una instrucción con la función `write()` con los parámetros de **posición, velocidad y un booleano** (true/false) que indica si se ha de pausar el código hasta completar el movimiento.

La librería se encarga de controlar automáticamente el pin digital que se haya asociado al servo transmitiendo una señal **PWM** (Modulación por ancho de pulsos), onda cuadrada con un periodo determinado y un tiempo a nivel alto variable que determina el ciclo de trabajo y, con ello, la posición del servomotor.

Para asegurar el final de instrucción debido a la falta de precisión de los servos es recomendable desconectar la señar PWM una vez acabado el movimiento. Esto se consigue por medio de la función `detach()` que no requiere ningún parámetro ya que ya dispone de un pin asignado.

5.2.1.2 Adafruit_LEDBackpack

Librería pública aportada por Adafruit [22] para el control de la matriz del LEDs del display de 7 segmentos por medio de la comunicación I2C en una dirección determinada.

Como en la librería anterior requiere la declaración del objeto `Adafruit_7segment` que se almacenará en una variable llamada `matrix`, se inicializará por medio de la función `begin()` con el parámetro de la dirección en la que se quiere que se establezca el dispositivo en hexadecimal, en este caso el 112=0x70: `matrix.begin(0x70)`.

Para escribir en la matriz se utilizarán los comandos:

- `matrix.drawColon(drawDots)` con el parámetro booleano de encendido o apagado de los puntos cada medio segundo
- `matrix.print(tiempo, DEC)` con el valor decimal del tiempo que se quiere mostrar

Ambos seguidos inmediatamente de la función:

- `matrix.writeDisplay()` para indicar que envíe la información de la variable modificada

5.2.1.3 CountUpDownTimer

Librería personalizada, modificando la librería pública con el mismo nombre de acceso público en GitHub [23], pero añadiendo la posibilidad de mostrar los medios segundos.

Permite el recuento temporal sin necesidad de recurrir a interrupciones, almacenando dicho valor en una variable y permitiendo su acceso y gestión fácilmente.

Nuevamente se define el objeto `CountUpDownTimer` almacenado en la variable `T` con un parámetro booleano (1 o UP para cuenta ascendente y 0 o DOWN para descendente), se inicializa con la función `StartTimer()`, se actualiza la variable con la función `Timer()` y se analiza el valor con las funciones:

- `ShowSeconds()`, devuelve un entero con el valor de segundos de 0 a 60
- `ShowHalfSeconds()`, devuelve un booleano con un 1 durante la segunda mitad de cada segundo
- `ShowMinutes()`, devuelve un entero con el valor de minutos de 0 a 60
- `TimeHasChanged()`, devuelve un booleano con un 1 si el valor ha variado desde la última comprobación

Además, se puede modificar el temporizador con las funciones: `StopTimer()`, `ResumeTimer()`, `ResetTimer()` y `SetTimer()`.

Para poder mostrar los medios segundos se almacena en una variable llamada Clock la cuenta de medios segundos y para acceder a segundos se divide `Clock / 2`, para acceder a los minutos `Clock / 2 / 60` y para los medios segundos el resto de `Clock % 2`.

5.2.1.4 Serial

Librería integrada implícita para el manejo de la comunicación Serial [24] (para más información de este protocolo de comunicación ver *apartado 5.4.1*) tanto por microUSB, FTDI como RX y TX. En caso de que se necesiten usar otros pines es necesario utilizar la librería pública "SoftwareSerial" en vez de esta integrada.

No es necesario incluir la librería ni declarar el objeto `Serial` en el código, solo es necesario inicializarlo con la función `begin()` con el parámetro de la velocidad de comunicación en baudios, en este caso 9600 Bd.

El envío de información se realizará por bytes, por lo tanto, no se utilizará la función `print()` que permite el envío de cadenas de caracteres sino el comando `write()` que toma como parámetro un byte. Para su lectura se utilizará la función `read()` comprobando previamente la recepción de información con la función `avalaible()` que devuelve el número de bytes que se han recibido.

5.2.1.5 Wire

Librería integrada explícita [25] muy parecida a la anterior en funciones, pero para la comunicación I2C (para más información de este protocolo ver *apartado 5.4.2*).

Esta librería sí es necesario incluirla y se inicializa como la anterior con el comando `begin()`, esta vez con el parámetro de la dirección que toma el Trinket en la comunicación, en este modelo: el Máster toma el valor 1, el slave de servos el 2 y el slave con NeoPixeles el 9.

Para el envío de bytes se requieren los siguientes comandos en dicho orden:

- `beginTransmission()` con el valor del canal al que se quiera enviar la información
- `write()` con el byte de información o con una lista de bytes y el número de elementos
- `endTransmission()` sin ningún parámetro para enviar la información

5.2.1.6 Adafruit_NeoPixel

Librería pública de Adafruit para el control de los NeoPixeles.

Esta librería no se utilizará directamente en el código, sino que se utilizará a través de la siguiente librería privada "Lights".

La librería define la clase `Adafruit_NeoPixel` que representará cada tira de LEDs con las siguientes funciones (mostrando solo las que se utilizan en el Proyecto):

- `begin()` que pone el pin asociado en modo "output"
- `show()` para enviar la información del objeto a la tira
- `setPixelColor()` para cambiar el píxel (de índice el primer parámetro) al color de valor RGB, RGBW o 32-bit asociado (en función del tipo y número de parámetros siendo 3, 4 o 1 respectivamente)
- `getPixelColor()` para devolver el valor de 32-bit asociado al color del píxel indicado como parámetro

5.2.1.7 Lights

Librería privada que hace uso a su vez de la librería *Adafruit_NeoPixel*.

La librería define 3 objetos, `EffectsManager`, `EffectsHandler` y `NeoPatterns`, de las clases con el mismo nombre respectivamente:

El **EffectManager** es el objeto encargado de enviar la instrucción con el efecto de luces nuevo que se quiera mostrar al Trinket que corresponda. De esta clase solo se utilizará la función `Change()` que toma como parámetros la dirección de 64 bits uint64_t addr64, uint8_t payload[], uint8_t len)

El **EffectsHandler** es el encargado de recibir la instrucción con el efecto y enviar la información procesada a las tiras de NeoPixeles. De esta clase interesan las funciones:

- `Init()` que inicializa un número de tiras determinado por sus parámetros, un vector con el número de LEDs de cada tira y un byte con el número de tiras.
- `Listen()` que espera la recepción de información por parte del EffectManager por medio del pin conectado al Xbee, se estará ejecutando constantemente en el loop y en caso de que se inicie la comunicación se encargará de almacenar en las variables correspondientes el payload recibido.

El **NeoPatterns** es el objeto derivado de la clase **Adafruit_NeoPixel** de la librería con el mismo nombre, pero añadiendo las siguientes funciones:

- `Update()` para cambiar el efecto de la tira
- `OnComplete()` para apagar la tira o hacer un efecto “Drop” cuando se acabe un efecto
- `Increment()` para pasar al siguiente estado de un efecto

Y los siguientes efectos con su respectiva función de inicialización (que guarda en variable el nombre del efecto y los valores de los parámetros que se le pasan como color, intervalos y pasos) y su función de “update” para cambiar el color de la tira en cada paso del efecto.

`FullColor()`, `Drop()`, `DropUpdate()`, `DoubleDrop()`, `DoubleDropUpdate()`, `Sparkle()`, `SparkleUpdate()`, `Strobe()`, `StrobeUpdate()`, `Fade()`, `FadeUpdate()`, `Rainbow()`, `RainbowUpdate()`, `Sweep()`, `SweepUpdate()`, `Fire_h()`, `Fire_hUpdate()`, `Fire_v()`, `Fire_vUpdate()`, `Breathe()`, `BreatheUpdate()`, `DimColor()`, `ColorSet()`, `Red()`, `Green()`, `Blue()`, `setPixelHeatColor()`

Ciertos efectos no requieren función "update" dado que no tienen diferentes pasos (mantiene la tira en un color constante). Los efectos que requieren update utilizarán un índice para contar los pasos del efecto o estados individuales de los pixeles. A su vez, el control de los pixeles se hace por regla general de forma individual, es decir uno a uno definiendo color e intensidad en cada paso del proceso.

Esta librería es posiblemente la más compleja que se utilizará ya que no solo controla los NeoPixeles sino también se encarga de la comunicación entre Máster y slaves por medio del modo **API** en el que se entrará en detalle en el *apartado 5.5* y permitirá cambiar la configuración del Xbee para desconectar un módulo slave y enlazar el Máster con el siguiente slave.

No sé entrará en detalle en la programación de esta funcionalidad, pero se realiza por medio de las clases `AtRemoteCommand`, `AtCommand` y `TxCommand` y sus respectivas funciones.

5.2.2 MÁSTER

El Máster será el Trinket con la programación más completa incluyendo el envío sincronizado temporalmente de instrucciones a los 3 tipos de slaves y al display, manejando también las interrupciones del timer.

El Máster incluirá las siguientes librerías: ***Lights***, ***Wire***, ***CountUpDownTimer***, ***Adafruit_LEDBackpack*** (la librería de *Adafruit_NeoPixel* no hace falta incluirla dado que ya lo hace la librería Lights).

Se inicializarán una serie de variables para guardar valores u objetos que se pueden utilizar reiteradamente, como son las direcciones de los Xbee y Trinkets.

Los **payloads** son vectores predefinidos con los datos a enviar a los diferentes Trinkets para indicar las instrucciones. Posteriormente, se pueden utilizar los efectos que se quieran, pero conviene tener definidos por defecto lo que se van a utilizar.

Para el correcto uso del timer también se requiere inicializar ciertas variables globales que se utilizarán posteriormente en el apartado de *Interrupciones*.

5.2.2.1 Setup

Todo programa de Arduino cuenta con una serie de funciones que se ejecutarán una vez definidos dentro del **setup.** En este caso se deshabilitarán las interrupciones temporalmente (para evitar la desincronización del timer), se reseteará el contador del timer1 (`TCCR1A` y `TCCR1B`) y se pondrá el fin de cuenta en `timer1_counter = 34286` para terminar la cuenta cada medio segundo (2 Hz).

Se utilizará el timer1 dado que el timer0 y timer2 son timers de 8bits mientras que el timer1 es de 16bits y, por lo tanto, tiene un límite de cuenta de hasta 65536 en vez de solo 256, pudiendo alcanzar la frecuencia de 2 Hz con un prescalado de 256:

La frecuencia del timer es igual a la velocidad del procesador (16 MHz en el caso del ATmega328) entre el prescalado: $16\ Mhz\ /\ 256\ =\ 62500\ Hz$

Como es necesaria una frecuencia de interrupción de 2 Hz se utilizará un final de cuenta de:

$$65536 - 16MHz/256/2Hz\ =\ 34286$$

Es decir, el timer decrementará el contador en 1 cada 1/62500 segundos (empezando en 65536 hasta llegar a 34286) un total de 31.250 veces por cada interrupción:

$$31250/62500\ =\ 0.5\ Hz$$

A continuación, se inicializa la comunicación Serial a 9600 Baudios y la comunicación I2C con la dirección del Master, también se establece el modo del pin 13 como "output" y se le da un valor inicial apagado. Este pin se utilizará para la depuración de la comunicación de manera visual, se encenderá cuando el Trinket intente establecer la comunicación y apagará cuando el slave reciba la instrucción y cambie de efecto.

También se inicializará el timer de *CountUpDownTimer* y la matriz de LEDs del display de 7 segmentos.

5.2.2.2 Loop

El bucle contendrá las instrucciones que se enviarán, es decir, el conjunto de efectos que realizará mandará a cada slave.

Para determinar el momento en el que se envía el efecto se utiliza un `switch` con un `delay` de 100 ms (aunque para mayor precisión se puede reducir el delay a 10 o 1 ms) que comprobará el tiempo comprendido de ejecución y, en caso de encontrarse en un segundo determinado, ejecuta las funciones de cambio de efecto que se deseen:

- eManager.Change()
- sendWireData()

5.2.2.3 Funciones

Hay dos funciones que se ejecutarán constantemente durante el bucle:

- `sendWireData()` es la función encargada de enviar un payload a dispositivo conectado por I2C. Toma como parámetros la dirección del dispositivo al que se quiera enviar la información (un byte), la información que se quiere transmitir (como array de bytes) y el número de bytes que de dicho array (en entero).
- `SevenSegTimer()` es la función encargada de actualizar el display; utiliza un booleano que se niega (cambiando de `true` a `false`) cada medio segundo (`drawDots`) y lo utiliza para dibujar el punto con la función `drawColon`. En caso de que no encontrarse en un medio segundo sino en uno completo también se actualizarán las cifras del display con la función `print` con el valor de la variable entera `tiempo` que es igual a los segundos más los minutos multiplicados por 100 para mostrarlos en la 3 cifra (notación decimal, `DEC`). En ambos casos se utiliza la función `writeDisplay()` después de modificar la variable `matrix` para mandar la información al display.

5.2.2.4 Interrupciones

Como se ha definido un periodo de interrupción de medio segundo en el timer1, se ejecutará la rutina de interrupción correspondiente, que consistirá en:

Se resetea el contador (`TCNT1`) al valor explicado en el *Setup* para conseguir una frecuencia de 1 Hz (34286) y se realiza una comprobación de antibloqueo de interrupción. Se almacena en una variable booleana (`runningInterrupt`) el estado de la interrupción y se comprueba al inicio de la misma que no se interrumpa mientras se ejecuta el código de la interrupción (problema llamado “interruption overload”).

La interrupción tiene la función de actualizar la variable del temporizador (`T`) y ejecutar la función `SevenSegTimer()` (explicada en el apartado de *Funciones*) en caso de que haya variado la misma (si se ha avanzado medio segundo). Finalmente se resetea el valor de estado de la interrupción (`runningInterrupt`) a `false`.

5.2.3 SLAVES

5.2.3.1 Slave de control de servos por I2C

Gracias a la librería VarSpeedServo explicada en el *apartado 5.2.1.1* el control de los servos es realizado simplemente por las funciones attach(), write() y detach(). Pero es necesario programar el recibo de información y el tratamiento de la misma.

Por ello se definirá un vector `myservo`, con la clase `VarSpeedServo` de dicha librería, de tamaño igual al número de servos que se quieran controlar, otro vector constante de enteros con los pins asociados llamado `servoPins` y un último vector multidimensional de bytes con orden el número de servos por 2, llamado `dataServo` que guardará la instrucción (posición y velocidad) de cada servo.

El **setup** consiste en asignar un prescalado al reloj del Trinket, mover los servos a sus posiciones 0 (a velocidad máxima), iniciar la conexión I2C y crear un evento de interrupción en caso de que se reciban datos ejecutando la función `receiveEvent()`.

El **loop** ejecutará la función `updateServos()` en caso de que sea true la variable booleana `newData`.

La función de evento `receiveEvent()` almacena la información de posición en el servo correspondiente en la variable `myservo` y activa la variable `newData`.

La función `updateServos()` realiza el proceso de `attach()`, `write()` y `detach()` para mover los servos y opcionalmente `wait()` en función de cómo se quieran ordenar las instrucciones. Finalmente, desactiva nuevamente la variable `newData` para poder repetir el proceso.

5.2.3.2 Slave de control de NeoPixeles por Serial

La librería de **Lights** está diseñada específicamente para hacer uso de los Xbee en modo API conectados por Serial al Trinket. Por lo tanto, el código de estos Trinkets será muy simple una vez se comprende la librería.

Aparte de incluir la librería hace falta inicializar una serie de variables globales:

- `Master_address`, con la dirección serial del Xbee Máster
- `Trinket_pin`, el pin al que se conecta el Xbee al Trinket
- `eHandler`, objeto de la clase con el mismo nombre predefinida en la librería Lights
- Un vector llamado `Stripes` de clase `NeoPatterns` con el número de LEDs y tiras de LEDs conectados a dicho Trinket

En el **setup** simplemente se inicia la comunicación Serial, se cambia el pin 13 a output como se explica en el *Setup* del Máster, se inicializa el `eHandler` y se realiza un `delay()` opcional por seguridad.

El **loop** solo requiere ejecutar la función `Listen()` del `eHandler`.

5.2.3.3 Slave de control de NeoPixeles por I2C

Al contrario del *apartado anterior*, el protocolo I2C no está soportado por la librería Lights, aunque se incluirá porque esta incluye a su vez la de **Adafruit_NeoPixel** y se podría querer utilizar un efecto de la anterior.

En este caso, se incluirá la librería **Wire**, se definirán las 2 direcciones para la misma y se creará un evento personalizado para la recepción de datos, `receiveEvent()`. También se crearán las funciones de inicialización y actualización de las tiras en el propio código para no hacer uso de la librería `initStripes()` y `updateStripes()`, aunque sean equivalentes a las mismas.

El evento de recepción de datos se encargará de anunciar la comunicación (con el pin 13), almacenar los bytes recibidos, correspondientes a las instrucciones de efectos, como la tira de LEDs que lo ejecuta (`stick`), el código del efecto (`effect`), el color (`Red`, `Green` y `Blue`), duración (`Interval1`, `Interval2` y `TotalSteps`) y configuración para los que lo necesiten (`Cooling` y `Sparking`) y guardar la instrucción en el objeto `Stripes` para actualizarse posteriormente en el **loop** por la función de `update()`.

5.3 CONFIGURACIÓN DE LOS XBEE

La configuración de los Xbee se realiza de forma sencilla gracias a la herramienta de XCTU mencionada en el apartado de *Programas*.

5.3.1 BÚSQUEDA DEL MODULO

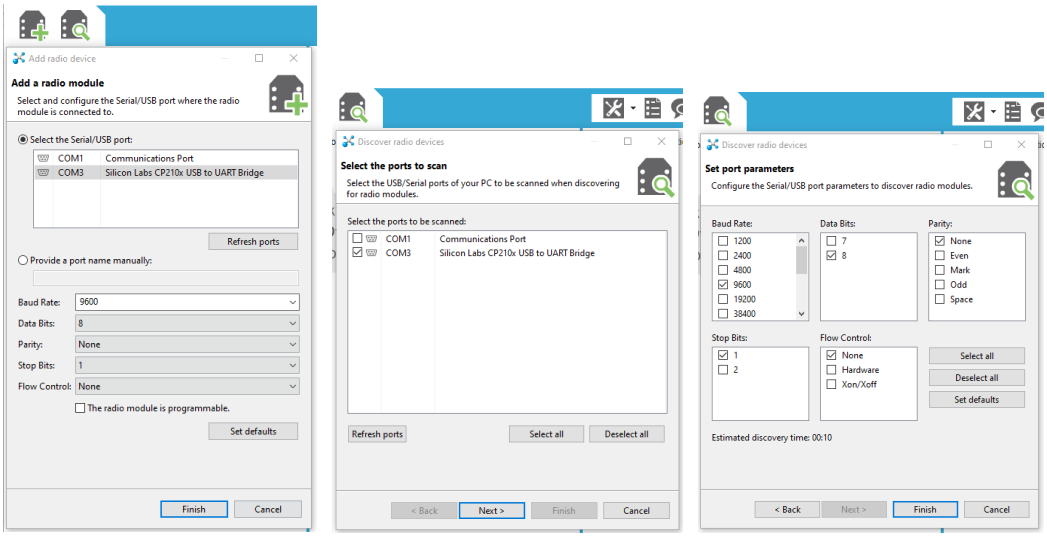


Ilustración 21 - Búsqueda del módulo de radio

Conectando el Xbee al ordenador por USB (a través del cable FTDI o habiéndolo soltado del adaptador y colocado en el programador) se podrá seleccionar el dispositivo pulsando el botón de "Add radio module" o en "Discover radio modules" y clicando el puerto (Silicon Labs CCP210x USB to UART Bridge en caso de usar el programador) y los parámetros: **Baud Rate** 9600, **Data Bits** 8, **Parity None**, **Stop Bits** 1 y **Flow Control** None.

Se utilizará la versión 10ee del firmware de XBEE 802.15.4 (de la familia de productos XB24) a pesar de que existe ya una versión más reciente. Como se ha mencionado anteriormente, es necesario el adaptador microUSB para cambiar a esta versión si ya se ha actualizado a una más reciente.

5.3.2 CONFIGURACIÓN DEL MÓDULO

Seleccionando el módulo y clicando en configuración se nos mostrará el siguiente menú:

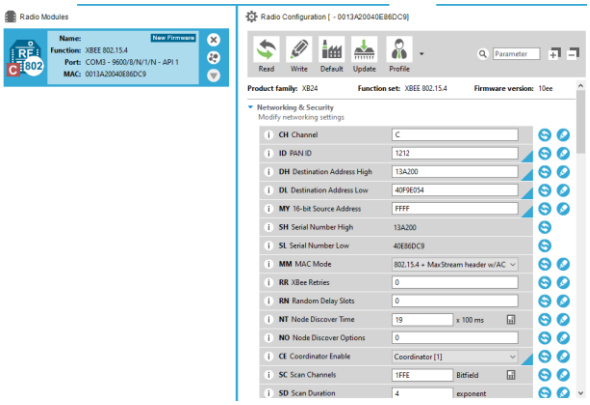


Ilustración 22 - Parámetros de configuración del módulo de radio

En la imagen anterior se muestra la configuración inicial del Xbee que actúa como Máster, aunque posteriormente se reconfigurará sobre la marcha por medio de los Trinkets gracias al *modo API*.

5.3.2.1 Configuración del Máster

Los parámetros a configurar cuentan con un código de 2 letras para su fácil distinción, los únicos que se han de modificar frente a la configuración predeterminada son:

- **CH: Canal** C de red inalámbrica 802.15.4 que se utilizará para la transmisión
- **ID: PAN ID** (Personal Area Network) 1212, alternativamente al Máster se le podría asignar el valor 0xFFFF para transmitir a todos los PANs
- **DH: Destination Address High**, primeros 32 bits de los 64 que forman la dirección a la que se quiere transmitir (usar 0 para direcciones de 16 bits), en el caso de los Xbee se trata del 13A200
- **DL: Destination Address Low** a 40F9E054, últimos 32 bits de la dirección de destino, se podrá seleccionar cualquiera de las direcciones del resto de Xbee esclavos debido a que se modificará sobre la marcha según a cuál se quiera transmitir (utilizar FFFF para transmitir a todas las direcciones del canal)
- **MY: 16-bit Source Address**, dirección del modem, para deshabilitar la recepción del mismo utilizar el valor FFFF
- **SH: Serial Number High**, primeros 32 bits del número de serie definido de fábrica e invariable que nos indica la dirección del dispositivo
- **SL: Serial Number Low**, últimos 32 bits del número de serie
- **CE: Coordinator Enable**, solo activador en caso de que sea el Máster (coordinador), desactivar en caso de que sea un slave (End Device)
- **AP: API Enable**, parámetro que activa el modo API para la configuración sobre la marcha de los Xbee. Se tendrá activado en todos los Xbee
- **DD: Device Type Identifier**, parámetro opcional para distinguir el tipo de dispositivo en la red en caso de que sea necesario, definir a 10000

5.3.2.2 Configuración de los slaves

Los slaves contarán con una configuración parecida:

- **CH: Canal** C
- **ID: PAN ID** 1212
- **DH: Destination Address High** 13A200.
- **DL: Destination Address Low** a 40E86DC9, siendo ésta la dirección de fábrica del Máster, ya que todos los slaves se comunicarán con él y nunca con el resto
- **MY: 16-bit Source Address**, a FFFF, para deshabilitar la comunicación al modem
- **CE: Coordinator Enable** a 0 en todos los slaves al ser End Points
- **AP: API Enable**, parámetro que activa el modo API para la configuración sobre la marcha de los Xbee. Se tendrá activado en todos los Xbee
- **D5 DIO5 Configuration**, configuración de la salida digital 5 (asociada al LED verde) que permitirá indicar el comienzo de la comunicación al Trinket, comenzará apagado y se encenderá cuando esté recibiendo información.
- **DD: Device Type Identifier** a 10000.

5.4 PROTOCOLOS DE COMUNICACIÓN

Existen muchas posibles formas de comunicación entre componentes electrónicos, en este Proyecto se utilizarán dos muy utilizadas en el ámbito debido a su potencial y adaptabilidad. Ambas requieren de dos cables de señal y un voltaje de referencia o tierra común (aunque la comunicación Wireless elimina parte de esta última restricción), pero hacen uso de esos dos cables y de la información transmitida de formas muy diferentes como se explica a continuación.

5.4.1 SERIAL (RX-TX)

El modelo de comunicación Serial (o serie) consiste en la transmisión secuencial bit-a-bit sobre un canal llamados BUS, en contraste con la comunicación paralela que envía varios paquetes de bytes al mismo tiempo permitiendo mayores velocidades y más versatilidad. La simplicidad de la comunican Serial permite una conexión fiable sin restricciones de distancia incluso en casos de riesgo de desincronismo. Además, la comunicación Serial requiere de una solo línea para el envío de datos (2 si se quiere que sea bidireccional), mientras que la paralela requiere tantas líneas como bits se quieran transmitir.

Por lo tanto, la comunicación Serial permite una gran fiabilidad de transmisión de datos sacrificando rendimiento y permitiendo el uso de una solo línea. Es por ello por lo que se utilizará este protocolo para la comunicación inalámbrica de los Trinkets por medio de los Xbees.

Durante la comunicación entre dos dispositivos, la UART de ambos envía la información a través de su línea TX (transmisión) y la recibe a través de su línea RX (recepción). Es por esto que los pines RX y TX de ambos dispositivos estarán conectados cruzados (RX de A TX de B y viceversa).

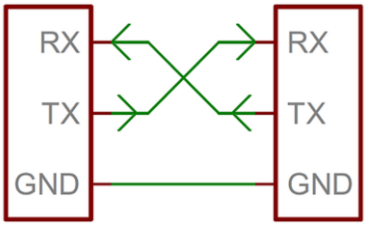


Ilustración 23 - Comunicación Serial entre 2 dispositivos

En caso de conectar varios dispositivos a una comunicación Serial se ha de conectar solo un RX a varios TX (de uno a varios) y nunca se van a conectar varios TX debido a que uno solo puede controlar la línea (riesgo de fundir las salidas si intentar comunicar "High" y "Low" a la vez). Es por esto por lo que se tendrán que desconectar los dispositivos de forma coordinada para que solo existan 2 conectados simultáneamente.

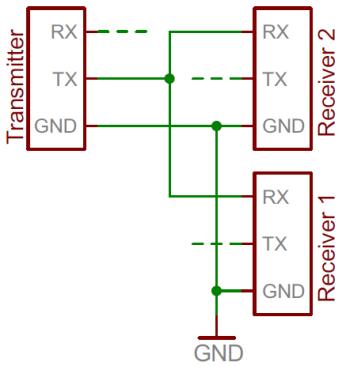


Ilustración 24 - Comunicación Serial entre varios dispositivos

Esto último se consigue gracias a la reprogramación de los Xbee mediante el modo API (véase el *apartado 5.5*).

5.4.2 I2C (SDA-SCL)

El protocolo I^2C (Inter-Integrated Circuit) es un modelo de transmisión síncrona de paquetes que permite la conexión de un sistema con varios masters y slaves por un mismo bus.

La comunicación I^2C requiere un mecanismo de sincronización para evitar la pérdida de paquetes. Debido a esto no existe ninguna posible implementación inalámbrica de I^2C.

Este protocolo requiere de dos líneas entre los dispositivos, la línea de transmisión de los datos (SDA) y la que contiene la señal de reloj (SCL). A ambas líneas se pueden conectar tantos dispositivos como se requiera, a cada uno de ellos se le asignará una dirección de un byte (de 7 a 10 bits) con la que se detecta si el paquete que está siendo transmitido ha de ser leído por el dispositivo.

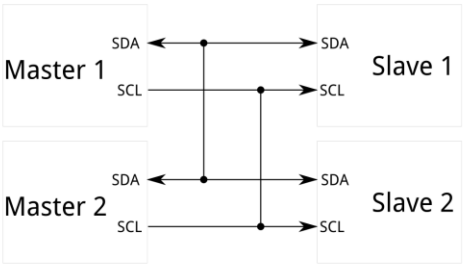


Ilustración 25 – Conexionado del modelo I2C

Este sistema de direccionamiento se logra gracias a que, al contrario que la comunicación Serial, el envío de información es a través de paquetes (o mensajes) formados por: inicio de la transmisión con una bajada de la señal SDA antes de la bajada del reloj, seguido de la dirección de envío y de la información ("data frame") y finalizando con una subida del voltaje de SDA después de la subida del reloj. El "data frame" también contiene un par de bits (ACK/NACK Bit) que avisan al transmisor del mensaje que el paquete ha sido recibido correctamente.

La gran versatilidad que supone este modelo de comunicación lo hace uno de los más utilizados en el ámbito de los microcontroladores debido a que permite el control de múltiples componentes utilizando únicamente dos pines digitales GPIO.

5.5 MODO API

Los módulos Xbee están configurados por defecto para trabajar en modo transparente, es decir toda la información recibida por un Xbee es transmitida al siguiente.

Esto supone una serie de limitaciones para ser reconfigurados por dicha línea de información porque ha de leer todos los mensajes para comprobar que no se le esté pidiendo entrar en modo "Command" (modo en el que se puede utilizar la información transmitida para variar su configuración). Es por ello por lo que se requiere un modo ordenado y estructurado de la transmisión de paquetes, el modo Application Programming Interface.

Esto protocolo permite reconfigurar el conjunto de módulos Xbee de la red de modo local y remoto (desde el mismo Xbee o enviando la información desde otro). Además, permite obtener más información, como la intensidad de la señal, el correcto envío de paquetes o la dirección de envío.

La estructura de paquetes del modo API está compuesta por:

Un delimitador de **inicio** (0x7E en hexadecimal), la **longitud** del paquete (compuesto por los valores del MSB y LSB, "Most/Least Significant Byte"), el "**frame data**" con la información trasmitida/recibida (que a su vez contiene el tipo de información), y un valor de comprobación de la suma de bytes del paquete, "**checksum**".

Esto permite enviar, por ejemplo, un paquete con el delimitador inicial, con un data frame que comience con 0x08 si se quieren mandar instrucciones de configuración al Xbee, 0x10 para enviar a un Xbee remoto información, o 0x17 para configurar un Xbee remoto.

La librería *Lights* hace uso de todos estos parámetros para configurar los Xbees durante la ejecución del programa.

Para más información al respecto a la estructura de paquetes del modo API se puede consultar el siguiente enlace de la compañía desarrolladora de los módulos [26].

5.6 ANÁLISIS DEL SISTEMA

El sistema final es el resultado de la unión de todos los componentes y herramientas explicados hasta ahora, conectados en una protoboard. El objetivo de la comunicación por radio del diseño es permitir el distanciamiento de los componentes para aportar versatilidad al sistema, pero al ser una prueba de concepto es preferible conectar todos ellos en una sola protoboard dando a entender que no existe ninguna necesidad de que los componentes comunicados por radio estén contiguos.

Asimismo, la conexión I2C también permite el cableado en distancias medias-largas a pesar de que se suele utilizar en electrónica con componentes cercanos. La idea es que el Máster pueda actuar de forma dependiente de los slaves para mantener el sincronismo pero con una serie de instrucciones organizadas para cada slave independientes del resto.

En el código utilizado para el Máster, las instrucciones están regidas únicamente en función del tiempo pero también se pueden añadir fácilmente condicionales físicos como botones conectados a cualquier entrada digital del Trinket Máster que puedan variar los efectos en función de cual se pulse. Ej.: `if(digitalRead(inPin)) eManager.Change(...)`.

Es posible cualquier combinación de condiciones y temporizadores para coordinar la coreografía. Facilitar los cambios en el código del bucle del Máster es el objetivo final de toda la programación, requiriendo únicamente variar el orden, condiciones y payloads de las funciones `eManager.Change()` y `sendWireData()` en el **loop** del Máster sin modificar la programación de ninguna librería o slave.

5.7 DISEÑO E IMPLEMENTACIÓN

A nivel de conexionado, el sistema en su conjunto no es muy complicado como se mencionó brevemente en el apartado del *Circuito*, pero eso se debe a que no se trata con componentes electrónicos de bajo nivel como puertas lógicas o microchips sino con controladores capaces de controlar por señales analógicas y digitales varios componentes, a la vez que se realizan interrupciones para la comunicación.

Los cables de comunicación no envían bytes de forma lineal y directa sino que forman paquetes con frames que incluyen el destinatario (en el caso de I2C) o instrucciones de configuración (en el caso de Serial).

A pesar de utilizar componentes de alto nivel, la programación es de medio nivel, C++ es un lenguaje muy básico en comparación con JavaScript, Python… y eso complica tareas sencillas como el análisis de bytes recibidos teniendo que tratarlos como vectores en vez de las cadenas de bytes que son.

Es por esto que la librería *Lights* es tan compleja y es la que más tiempo ha requerido para hacerla funcionar. Al ser tan dependiente de la configuración de los Xbee, ya que es la que se encarga de su reconfiguración, es complicado encontrar un error en caso de fallar algo en alguna de las dos. Los módulos de radio tienen que compartir una misma versión de firmware y configuración. Además, toda la librería requiere la correcta conexión de los Trinkets con los Xbees y el uso correcto de las funciones de la misma en los códigos de Máster y slaves.

Se puede observar en las versiones iniciales de los códigos de prueba como se comienza con una prueba de funcionamiento que simplemente enciende un LED del Trinket para testear la comunicación, posteriormente se aumenta la información enviada y mejora el análisis de la misma. Y, finalmente, se añaden componentes ya testeados por separado combinando el código de la comunicación con el control de dichos componentes.

Capítulo 6. ANÁLISIS DE RESULTADOS

El resultado más favorable del Proyecto ha sido el diseño de un conjunto documentado de librerías, funciones y esquemas de conexión que permitirían un diseño igual al prototipado, pero a gran escala y completamente funcional.

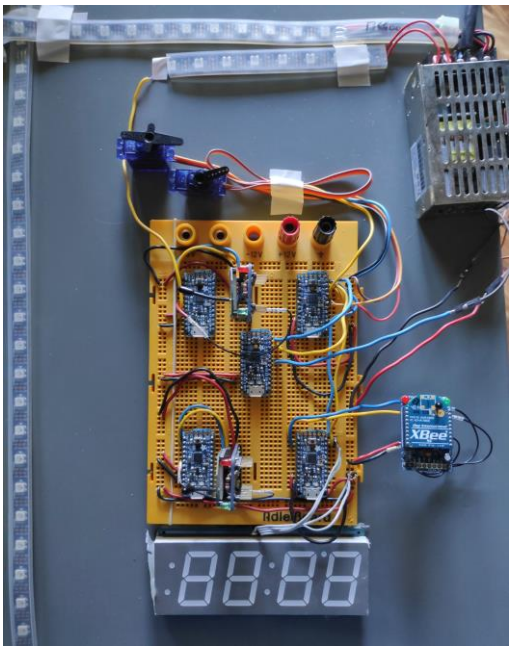


Ilustración 26 – Diseño con 5 Trinkets, 3 Xbees, 2 tiras de LEDs y 2 servos

Las tiras de **LEDs** se pueden encadenar para abarcar grandes distancias y se pueden conectar hasta 17 por Trinket (6 salidas analógicas y 14 digitales menos 2 para las conexiones RX/TX o SDA/SCL) alejando el slave del Máster por cable grandes distancias o por Wireless ahorrando así en instalación.

Los efectos programados para las tiras de LEDs son bastante diversos y permiten bastante variación en términos de velocidad, intensidad, colores (en RGB 255^3 colores) con diferentes intervalos y algunos con otros parámetros de configuración. Pero, de ser necesario, es bastante sencillo ampliar la librería *Lights*, ya que es bastante modular a pesar de su complejidad.

Con el código que se ha descrito anteriormente (*apartado 5.2.3.1*), los **servomotores** solo se pueden controlar por un slave conectado por I2C, aunque también se ha preparado un código que lo permite a través de Serial. El problema es que requiere una configuración distinta de los Xbee que no es compatible con la librería *Lights* y por ello se ha descartado.

Además, tener los servos conectados por cable es lo más lógico teniendo en cuenta que requieren una instalación fija para el movimiento de los láseres. El soporte de los láseres para los servos no entraba dentro de los objetivos del Proyecto, pero el hecho de tener una programación que controla de forma sencilla y eficiente la posición y velocidad de los mismos permite adaptar el uso de los servos a cualquier tipo de funcionalidad, no solo la de control de láseres sino también de cualquier otro componente.

La librería utilizada para el **display** de 7 segmentos también permite usarlo para mostrar una cuenta regresiva en caso de querer usarse para un espectáculo con tiempo limitado o que requiera una cuenta atrás.

En caso de querer añadir una gran cantidad de tiras de LEDs se necesitaría ampliar la potencia de la **batería** o añadir más. Lo último también es necesario en caso de alejar las tiras por conexión inalámbrica.

En **resumen**, el sistema es ampliamente versátil y modular con una instalación sencilla independientemente de la escala. En cualquiera de los casos mencionados y de la escala del sistema lo único que hay que modificar son los payloads y/o las instrucción enviadas en el *loop del Máster* por medio de Arduino IDE y no hace falta modificar ninguna ningún otro componente. Solamente se requeriría XCTU para configurar por primera vez más módulos de radio en caso de que se necesiten más de tres.

Capítulo 7. CONCLUSIONES Y TRABAJOS FUTUROS

A pesar de que los objetivos del Proyecto no abarcaban la instalación completa del sistema, sería una buena continuación del trabajo probar la implantación del sistema desarrollado a mayor escala y con un objetivo más estético y espectacular, con los soportes para controlar los láseres, las tiras de LEDs en diferentes formas cubriendo una parte de una sala o fachada y una coreografía más compleja.

El sistema en su conjunto también deja abierta la posibilidad de incluir una coreografía de efectos variables al ritmo de una canción o de un instrumento. Lo único que requeriría es un micrófono electrostático conectado a un par de entradas, una analógica y otra digital del Máster y convertir la señal analógica de volumen en intensidad lumínica (de 0 a 255) y la entrada de tono en colores.

El Proyecto, por lo tanto, permite una continuación a nivel funcional y de ampliación de hardware que no requeriría apenas de programación, pero sí de impresoras 3D, más componentes y un diseño para encapsular los componentes como una caja incluso con amplificadores para las antenas de los módulos de radio.

Una vez finalizado el desarrollo del sistema, se puede concluir que el valor asociado al coste de los componentes es superior en el caso de componentes más **económicos**.

Solo deberían ser consideradas alternativas en proyectos a mayor escala de una duración muy larga. No porque haya riesgo de fallo o deterioro del sistema que se ha desarrollado sino porque la alternativa de instalación más cara puede contar con elementos más eficientes energéticamente y, a largo plazo, esto puede aportar algo de rentabilidad. No obstante, el coste de instalaciones masivas es muy superior cuando no se utilizan componentes como los del Proyecto, sino paneles de LEDs o pantallas LCD de gran tamaño. En este caso, el margen de beneficio en el apartado energético no es comparable al costo inicial.

Capítulo 8. REFERENCIAS

[1] Láser performance for ElecTRONica show in Disney California, Junio 2011
https://www.youtube.com/watch?v=mgIZrSlPuko

[2] Shenzhen Light Show, Septiembre 2019
https://www.youtube.com/watch?v=nPhGN0T9kMc&t=58s

[3] Página web official de Adafruit Industries
https://www.adafruit.com/

[4] Adafruit Pro Trinket - 5V 16MHz
https://www.adafruit.com/product/2000

[5] Arduino Nano
https://store.arduino.cc/arduino-nano

[6] Depuración de Código o "debugging" en Wikipedia
https://en.wikipedia.org/wiki/Debugging

[7] Adafruit NeoPixel Digital RGB LED Strip - White 30 LED – WHITE
https://www.adafruit.com/product/1376?length=1

[8] XBee Module - ZB Series S2C - 2mW with Wire Antenna - XB24CZ7WIT-004
https://www.adafruit.com/product/968

[9] XBee Adapter kit - v1.1
https://www.adafruit.com/product/126

[10] Waveshare Xbee USB Adapter ("programmer")
https://www.waveshare.com/xbee-usb-adapter.htm

[11] Micro servo – Tower Pro SG92R
https://www.adafruit.com/product/169

[12] Red 7-segment clock display - 1.2" digit height
https://www.adafruit.com/product/1264

[13] Sunpower SPS-025-05
https://www.sunpower-uk.com/product/25w-single-output-switching-power-supply/25w-5v-5a-enclosed-switching-power-supply/

[14] IC Socket - for 28-pin 0.6" Chips
https://www.adafruit.com/product/2206

[15] Proceso de soldadura de los Xbee
https://learn.adafruit.com/xbee-radios/solder-it

[16] Arduino IDE
https://www.arduino.cc/en/main/software

[17] XCTU
https://www.digi.com/products/embedded-systems/digi-xbee/digi-xbee-tools/xctu

[18] Autodek TinkerCAD
https://www.tinkercad.com/

[19] Fritzing
https://fritzing.org/home/

[20] HyperTerminal
https://www.hilgraeve.com/hyperterminal/

[21] VarSpeedServo by NETLab Toolkit Group
https://github.com/netlabtoolkit/VarSpeedServo

[22] Adafruit_LED_Backpack by Adafruit
https://github.com/adafruit/Adafruit_LED_Backpack

[23] Arduino library CountUpDownTimer by Andrew Mascolo
https://github.com/AndrewMascolo/CountUpDownTimer

[24] Arduino library for Serial communication
https://www.arduino.cc/en/pmwiki.php?n=Reference/serial

[25] Arduino library for I2C communication
https://www.arduino.cc/en/reference/wire

[26] Xbee API frame structure
http://cms.digi.com/resources/documentation/Digidocs/90001942-13/

[27] Google Search Images for Akihabara
https://www.digi.com/resources/documentation/Digidocs/90001942-13/concepts/c_api_frame_structure.htm

[28] Página web official de Light Balance
https://lightbalance.net/about/

[29] Shows de Light Balance en America's Got Talent 2017
https://www.youtube.com/watch?v=E8Ecz_sntDo

[30] Librería Lights
https://github.com/arenillas/Arduino

ANEXO I: SDG

A nivel de objetivos, dentro de los posibles SDGs (Sustainable Development Goals), el Proyecto se ha enfocado en el apartado **económico**, teniendo en cuenta que el mundo del espectáculo unido a la tecnología está en pleno desarrollo. Cada vez se pueden encontrar más paneles LED e iluminación variable en las fachadas de los edificios, siendo conocidos por ello distritos de Tokyo como Akihabara [27].

La iluminación siempre ha sido uno de los factores más llamativos en el ámbito de la **publicidad** dada la atención que atrae. Es por ello que se utiliza de forma masiva en los grandes focos de población mundial. Con la reducción de los precios de la electrónica gracias al crecimiento exponencial de la tecnología en este siglo el futuro nos depara unas ciudades llenas de publicidad en forma de paneles LED o tiras de LEDs iluminando carteles.

La publicidad invita al consumo y el consumo al desarrollo económico, pero este no es el principal objetivo de este Proyecto, sino el coste de preparación y montaje del sistema.

El mayor gasto de tiempo en la implantación de sistemas de iluminación complejos no es la de instalación del hardware sino la organización y **diseño** del conjunto a nivel de componentes, estudio del mercado, presupuesto, programación y testeo. El coste de mano de obra para la instalación es mucho menor y menos cualificado que el de los encargados de hacer todo lo mencionado anteriormente como los ingenieros y programadores.

Contar con un diseño ampliamente escalable y configurable, con unos requerimientos mínimos a nivel económico y de conocimientos de la materia para su instalación, implicaría un desarrollo mucho más rápido de la iluminación en cualquier ámbito publicitario o decorativo.

El mundo del **espectáculo** está directamente conectado a este uso de la publicidad, siendo uno de los sectores que más dinero mueven. Esto es precisamente por el tamaño de la audiencia, y no el precio que pagan por su visualización, sino el valor que aporta que todas esas personas vean la publicidad que contiene el espectáculo.

Ya se han mencionado anteriormente ejemplos de espectáculos lumínicos, pero no se ha mostrado la importancia que tiene la preparación de los mismos. Un caso más claro para demostrar dicha importancia es la del show preparado por el equipo de **Ligth Balance** [29], ganadores de varios premios como America's y Britain's Got Talent con actuaciones como las del enlace [30] que demuestran que la programación y sincronización de todo el equipo es un apartado de vital importancia en espectáculos tan elaborados.

Es imposible realizar una estimación del coste medio de cualquier espectáculo o instalación debido a que es necesario incluir salarios, componentes personalizados…, pero si se aplicase el diseño preparado en este Proyecto se podría eliminar por completo el coste de programación y centrarse en la fácil instalación de los componentes de bajo coste que se pueden comprar en grandes cantidades a un precio comedido.

La creación de un diseño universal adaptable a cualquier tipo de espectáculo no es realista, pero el desarrollo de un sistema versátil y de bajo coste permitiría la expansión del sector, promoviendo su uso y su desarrollo a nivel económico y de innovación.

ANEXO II: CÓDIGO

Este anexo recopila la programación en Arduino de todos los Trinkets tanto a nivel de librerías como de código principal. **No se incluyen** en este anexo librerías públicas no modificadas para el Proyecto siendo estas de fácil acceso a través de los enlaces incluidos en las *Referencias*, ni librerías integradas en *Arduino IDE*. La librería *Lights* se ha publicado en GitHub [30] permitiendo su acceso para la posible réplica del diseño o análisis del código.

Las librerías y la programación de los Trinkets se explican en detalle en el apartado de *Programación*, pero el nombre de variables y funciones es autoexplicativo, cuentan con comentarios en las ocasiones en caso de que sea necesario para su lectura comprensiva.

LIBRERÍAS

Las librerías externas en Arduino pueden contar con dos documentos como en C++, el archivo ".h" enumera las funciones y clases y es obligatorio, el archivo ".cpp" incluye el código fuente de las funciones y clases. En caso de ser una librería pequeña se pueden inicializar las funciones y clases así como definir su código interno en el archivo ".h" evitando así la necesidad del ".cpp", este es el caso de la siguiente librería modificada:

CountUpdown.h

```
/* Count Up/Down Timer */ #
ifndef CountUpDownTimer_h# define CountUpDownTimer_h

#include<Arduino.h>

# define UP 1# define DOWN 0

class CountUpDownTimer {
  public:
    CountUpDownTimer(bool type): _type(type) {}

  boolean Timer() {
    static unsigned long duration = 500000; // 0.5 second
    timeFlag = false;
```

```
    if (!Stop && !Paused) // if not Stopped or Paused, run timer
    {
      if ((_micro = micros()) - time > duration) // check the time difference and
see if 1 second has elapsed
      {
        _type == UP ? Clock++ : Clock--;

        timeFlag = true;

        if (_type == DOWN && Clock == 0) // check to see if the clock is 0
          Stop = true; // If so, stop the timer

        _micro < time ? time = _micro : time += duration; // check to see if
micros() has rolled over, if not, then increment "time" by duration
      }
    }
    return !Stop; // return the state of the timer
  }

  void StartTimer() {
    Watch = micros();
    Stop = false;
    Paused = false;
    if (_type == UP)
      Clock = 0;
  }

  void StopTimer() {
    Stop = true;
  }

  void StopTimerAt(unsigned int hours, unsigned int minutes, unsigned int
seconds) {
    if (TimeCheck(hours, minutes, seconds))
      Stop = true;
  }

  void PauseTimer() {
    Paused = true;
  }

  void ResumeTimer() // You can resume the timer if you ever stop it.
  {
    Paused = false;
  }

  void SetTimer(unsigned int hours, unsigned int minutes, unsigned int seconds) {
    // This handles invalid time overflow ie 1(H), 0(M), 120(S) -> 1, 2, 0
    unsigned int _S = (seconds / 60), _M = (minutes / 60);
    if (_S) minutes += _S;
    if (_M) hours += _M;

    Clock = (hours * 3600) + (minutes * 60) + (seconds % 60);
```

```
    R_clock = Clock;
    Stop = false;
  }

  void SetTimer(unsigned int seconds) {
    // StartTimer(seconds / 3600, (seconds / 3600) / 60, seconds % 60);
    Clock = seconds;
    R_clock = Clock;
    Stop = false;
  }

  int ShowHours() {
    return Clock / 2 / 3600;
  }

  int ShowMinutes() {
    return (Clock / 2 / 60) % 60;
  }

  int ShowSeconds() {
    return Clock / 2 % 60;
  }
  int ShowHalfSeconds() {
    return Clock % 2;
  }

  unsigned long ShowMilliSeconds() {
    return (_micro - Watch) / 1000.0;
  }

  boolean TimeHasChanged() {
    return timeFlag;
  }

  boolean TimeCheck(unsigned int hours, unsigned int minutes, unsigned int
seconds) // output true if timer equals requested time
  {
    return (hours == ShowHours() && minutes == ShowMinutes() && seconds ==
ShowSeconds()); //&& ShowHalfSeconds...
  }

  private:
    unsigned long Watch, _micro, time = micros();
  unsigned int Clock = 0, R_clock;
  boolean Reset = false, Stop = false, Paused = false;
  volatile boolean timeFlag = false;
  bool _type;
};

#
endif
```

TRINKETS

Como se ha mencionado durante el *Capítulo 5*, se diferenciará el código de 4 tipos de Trinkets según su función y componentes:

Máster

```
#include <Lights.h>
#include <Wire.h>
#include <CountUpDownTimer.h>
#include "Adafruit_LEDBackpack.h"

int timer1_counter;

//-----------------------------------NEOPIXELES---------------------------------
---
// Definition of Slave Addresses
uint64_t Slave_address2 = 0x0013A20040F9E054;
uint64_t Slave_address1 = 0x0013A20040E86DFC;  //Xbee nuevo
uint8_t WireMasterAddres = 1;
uint8_t WireSlaveAddressServo = 2;
uint8_t WireSlaveAddressNeo = 9;

// Effects Manager Creation
EffectsManager eManager = EffectsManager();

// Payload of TX Message
// Use this section to compose payload_messages easily
uint8_t StickA = 1;
uint8_t EffectA = FULL_COLOR;
uint8_t RedA = 255;
uint8_t GreenA = 0;
uint8_t BlueA = 0;
uint8_t Interval1A = 0;
uint8_t Interval2A = 0;
uint8_t TotalStepsA = 0;
uint8_t CoolingA = 0;
uint8_t SparkingA = 0;
uint8_t StickB = 2;
uint8_t EffectB = FULL_COLOR;
uint8_t RedB = 25;
uint8_t GreenB = 0;
uint8_t BlueB = 0;
uint8_t Interval1B = 0;
uint8_t Interval2B = 0;
uint8_t TotalStepsB = 0;
uint8_t CoolingB = 0;
uint8_t SparkingB = 0;
//uint8_t payload[] = {StickA, EffectA, RedA, GreenA, BlueA, Interval1A,
Interval2A, TotalStepsA, CoolingA, SparkingA};
```

```
uint8_t payload1[] = {StickA, FULL_COLOR, 20, 0, 0, 0, 0, 0, 0, 0, StickB,
FULL_COLOR, 0, 100, 0, 0, 0, 0, 0, 0};
uint8_t payload1a[] = {StickA, FULL_COLOR, 0, 0, 0, 0, 0, 0, 0, 0, StickB,
FULL_COLOR, 0, 0, 0, 0, 0, 0, 0, 0};
uint8_t payload2[] = {StickA, DROP, 0, 0, 100, 50, 0, 0, 0, 0, StickB, DROP, 0,
0, 100, 100, 0, 0, 0, 0};
uint8_t payload3[] = {StickA, DOUBLE_DROP, 100, 0, 0, 100, 0, 0, 0, 0, StickB,
DOUBLE_DROP, 0, 100, 0, 100, 0, 0, 0, 0};
uint8_t payload5[] = {StickA, SPARKLE, 0, 255, 100, 1, 0, 0, 0, 0, StickB,
SPARKLE, 0, 255, 0, 1, 0, 0, 0, 0};
uint8_t payload6[] = {StickA, STROBE, 0, 0, 20, 255, 255, 255, 0, 0, StickB,
STROBE, 0, 100, 0, 255, 255, 255, 0, 0};
uint8_t payload7[] = {StickA, FADE, 0, 0, 100, 50, 50, 100, 0, 0, StickB, FADE,
100, 0, 0, 50, 50, 100, 0, 0};
uint8_t payload8[] = {StickA, RAINBOW, 0, 0, 20, 2, 0, 40, 0, 0, StickB, RAINBOW,
20, 0, 0, 50, 0, 20, 0, 0};
uint8_t payload9[] = {StickA, SWEEP, 100, 0, 0, 60, 50, 100, 0, 0, StickB, SWEEP,
0, 100, 0, 60, 50, 100, 0, 0};
uint8_t payload10[] = {StickA, FIRE_H, 100, 0, 0, 100, 50, 100, 0, 0, StickB,
FIRE_H, 0, 100, 0, 100, 50, 100, 0, 0};
uint8_t payload11[] = {StickA, FIRE_V, 100, 0, 0, 100, 5, 100, 55, 120, StickB,
FIRE_V, 0, 100, 0, 100, 2, 100, 55, 120};
uint8_t payload12[] = {StickA, BREATHE, 0, 0, 100, 20, 0, 100, 0, 0, StickB,
BREATHE, 100, 0, 0, 20, 0, 100, 0, 0};


//-------------------------------------SERVOS---------------------------------------

//uint8_t servoData1[3] = {servo,pos,vel};
uint8_t servoData1[3] = {0,180,10};
uint8_t servoData2[3] = {1,135,100};
uint8_t servoData3[3] = {0,0,200};
uint8_t servoData4[3] = {1,30,200};

//-------------------------------------TIMER----------------------------------------

CountUpDownTimer T(UP);
Adafruit_7segment matrix = Adafruit_7segment();

int tiempo;
//int tiempom;
boolean drawDots = false;

bool runningInterrupt = false;

//-------------------------------------SETUP----------------------------------------

void setup(){
  noInterrupts();           // disable all interrupts
  TCCR1A = 0;
  TCCR1B = 0;
  timer1_counter = 34286;   // preload timer 65536-16MHz/256/2Hz
  //timer1_counter = 3036;  // preload timer 65536-16MHz/256/1Hz
```

```
  TCNT1 = timer1_counter;   // preload timer
  TCCR1B |= (1 << CS12);    // 256 prescaler
  TIMSK1 |= (1 << TOIE1);   // enable timer overflow interrupt
  interrupts();             // enable all interrupts

  Serial.begin(9600);
  Wire.begin(WireMasterAddres);

  // Pin 13 is the LED. Is used only for debugging inside Lights.cpp
  // It turns on when Trinket tries to stablish communication
  // It turns off when Slave Trinket changes the effect
  pinMode(13, OUTPUT);
  digitalWrite(13, LOW);

  T.StartTimer();
  matrix.begin(0x70);
}

//-------------------------------------LOOP-------------------------------------

// Main loop
void loop(){
  switch (T.ShowSeconds() % 60) {
    case 5:
      eManager.Change(Slave_address1, payload1, 20);
      sendWireData(WireSlaveAddressNeo, payload6, sizeof(payload6));
      break;
    case 10:
      sendWireData(WireSlaveAddressServo, servoData1, sizeof(servoData1));
      sendWireData(WireSlaveAddressServo, servoData2, sizeof(servoData2));
      break;
    case 15:
      eManager.Change(Slave_address1, payload5, 20);
      sendWireData(WireSlaveAddressNeo, payload8, sizeof(payload5));
      break;
    case 20:
      sendWireData(WireSlaveAddressServo, servoData3, sizeof(servoData3));
      sendWireData(WireSlaveAddressServo, servoData4, sizeof(servoData4));
      break;
    case 25:
      //eManager.Change(Slave_address2, payload6, 20);
      sendWireData(WireSlaveAddressNeo, payload1, sizeof(payload1));
      break;
    case 30:
      //T.StopTimer();
      //T.ResumeTimer();
      //T.ResetTimer();
      //T.StartTimer();
      break;
  }
  delay(100);
}
```

```
//-------------------------------------FUNCIONES-----------------------------------
--

void sendWireData(uint8_t WireSlaveAddress, uint8_t data[], int n) {
  digitalWrite(13, !digitalRead(13));  //change_pin()
  Wire.beginTransmission(WireSlaveAddress);
  Wire.write(data,n);
  Wire.endTransmission();
  digitalWrite(13, !digitalRead(13));  //change_pin()
}

void SevenSegTimer() {
    drawDots = ~drawDots;
    matrix.drawColon(drawDots);
    matrix.writeDisplay();

    if (T.ShowHalfSeconds()==0){

      tiempo = T.ShowMinutes()*100 + T.ShowSeconds();

      matrix.print(tiempo, DEC);
      matrix.writeDisplay();
    }
}

//-------------------------------------INTERRUPTS----------------------------------
---

ISR(TIMER1_OVF_vect)        // interrupt service routine
{
  TCNT1 = timer1_counter;   // preload timer

  if(runningInterrupt) return; //bail out if we interrupted ourself
  runningInterrupt = true;
  interrupts();

  T.Timer();
  if (T.TimeHasChanged())  SevenSegTimer();

  runningInterrupt = false;
}
```

Slave de control de servos por I2C

```
#include <VarSpeedServo.h>
#include <Wire.h>
#ifdef __AVR__
#include <avr/power.h> // Required for 16 MHz Adafruit Trinket
#endif

VarSpeedServo myservo[3];
const int servoPins[3] = {14, 15, 16};

uint8_t dataServo[3][2];
boolean newData = false;

void setup()  {
#if defined(__AVR_ATtiny85__) && (F_CPU == 16000000)
  clock_prescale_set(clock_div_1);                    //Trinket Pro
#endif

  myservo[0].attach(servoPins[0]);
  myservo[1].attach(servoPins[1]);
  //myservo[2].attach(servoPins[2]);

  myservo[0].write(0, 255, true);  // degrees, speed, wait until done
  myservo[1].write(0, 255, true);
  //myservo[2].write(0,255,true);

  myservo[0].detach();
  myservo[1].detach();
  //myservo[2].detach();

  delay(100);

  Wire.begin(2);
  Wire.onReceive(receiveEvent);
}

void loop()  {
  if (newData) updateServos();
  delay(100);
}

void receiveEvent(int howMany) {
  while (Wire.available() > 0) {
    uint8_t m = Wire.read();
    dataServo[m][0] = Wire.read();
    dataServo[m][1] = Wire.read();
    newData = true;
  }
}

void updateServos() {
  myservo[0].attach(servoPins[0]);
```

```
  myservo[1].attach(servoPins[1]);
  //myservo[2].attach(servoPins[2]);

  myservo[0].write(dataServo[0][0], dataServo[0][1], false);
  myservo[1].write(dataServo[1][0], dataServo[1][1], true);
  //myservo[2].write(dataServo[2][0], dataServo[2][1], true); //todos a false
menos ultimo??

  myservo[0].wait();
  myservo[1].wait();
  //myservo[2].wait();      //Necesario?? Queremos que termine?

  myservo[0].detach();
  myservo[1].detach();
  //myservo[2].detach();

  newData = false;
}
```

Slave de control de NeoPixeles por Serial

```
#include <Lights.h>

// Definition of Master Addresses
uint64_t Master_address = 0x0013A20040E86DC9;

// Trinket Pin which is connected to XBee pin 15
uint8_t Trinket_pin = 5;

// Effects Handler Creation
EffectsHandler eHandler = EffectsHandler(Master_address, Trinket_pin);

// Neopixel Creation
#define s_number 2 //Number of Neopixel Strips
// Number of leds in each Neopixel Strip
#define Stick1_pixels 52 //Tira de 52 leds
#define Stick2_pixels 8 //Tira de 08 leds
// Trinket pins used for Neopixel Strips control
#define Stick1_pin 14 //Tira de 52 leds
#define Stick2_pin 12 //Tira de 08 leds
NeoPatterns Stripes[s_number] = {NeoPatterns(Stick1_pixels, Stick1_pin, NEO_GRB +
NEO_KHZ800), NeoPatterns(Stick2_pixels, Stick2_pin, NEO_GRB + NEO_KHZ800)};

void setup(){
  Serial.begin(9600);
  // Pin 13 is the LED. Is used only for debugging inside Lights.cpp
  // It turns on when Trinket tries to stablish communication
  // It turns off when Slave Trinket changes the effect
  pinMode(13, OUTPUT);
  pinMode(Trinket_pin, INPUT);
  eHandler.Init(Stripes, s_number);
  delay(1000);
}

// Main loop
void loop(){
  eHandler.Listen(Stripes, s_number);
}
```

Slave de control de NeoPixeles por I2C

```
#include <Lights.h>
#include <Wire.h>

// Definition of Master Addresses
uint8_t WireMasterAddres = 1;
uint8_t WireSlaveAddressNeo = 9;

// Neopixel Creation
#define s_number 2 //Number of Neopixel Strips
// Number of leds in each Neopixel Strip
#define Stick1_pixels 52 //Tira de 52 leds
#define Stick2_pixels 8 //Tira de 08 leds
// Trinket pins used for Neopixel Strips control
#define Stick1_pin 14 //Tira de 52 leds
#define Stick2_pin 12 //Tira de 08 leds
NeoPatterns Stripes[s_number] = {NeoPatterns(Stick1_pixels, Stick1_pin, NEO_GRB +
NEO_KHZ800), NeoPatterns(Stick2_pixels, Stick2_pin, NEO_GRB + NEO_KHZ800)};

void setup(){
  pinMode(13, OUTPUT);
  initStripes(Stripes, s_number);

  Wire.begin(WireSlaveAddressNeo);
  Wire.onReceive(receiveEvent);

  delay(1000);
}

void loop(){
  updateStripes(Stripes, s_number);
}

void initStripes(NeoPatterns Stripes[], uint8_t length) {
  for (int i=1; i <= length; i++){
    Stripes[i-1].begin();
    Stripes[i-1].ColorSet(Stripes[i-1].Color(0, 0, 0));
  }
}

void updateStripes(NeoPatterns Stripes[], uint8_t length) {
  for (int i=1; i <= length; i++){
    Stripes[i-1].Update();
  }
}

void receiveEvent(int howMany) {
  uint8_t stick = 0;
  uint8_t effect = 0;
  uint8_t Red = 0;
  uint8_t Green = 0;
  uint8_t Blue = 0;
```

```
uint8_t Interval1 = 0;
uint8_t Interval2 = 0;
uint8_t TotalSteps = 0;
uint8_t Cooling = 0;
uint8_t Sparking = 0;

digitalWrite(13, !digitalRead(13));  //change_pin()

while(Wire.available()) {
  //RECEIVING
  for (int j = 0; j < 10; j++) {
    //while(Wire.available() == 0);   //Bloquea el código del Máster
    switch (j){
      case 0:
         stick = Wire.read();
         break;
      case 1:
         effect = Wire.read();
         break;
      case 2:
         Red = Wire.read();
         break;
      case 3:
         Green = Wire.read();
         break;
      case 4:
         Blue = Wire.read();
         break;
      case 5:
         Interval1 = Wire.read();
         break;
      case 6:
         Interval2 = Wire.read();
         break;
      case 7:
         TotalSteps = Wire.read();
         break;
      case 8:
         Cooling = Wire.read();
         break;
      case 9:
         Sparking = Wire.read();
         break;
     }
  }

  //SWITCHING EFFECT
  switch (effect){
    case 1:
      Stripes[stick-1].FullColor(Stripes[stick-1].Color(Red, Green, Blue));
      break;
    case 2:
```

```
        Stripes[stick-1].Drop(Stripes[stick-1].Color(Red, Green, Blue),
Interval1);
        break;
      case 3:
        Stripes[stick-1].DoubleDrop(Stripes[stick-1].Color(Red, Green, Blue),
Interval1);
        break;
      case 4:
        Stripes[stick-1].Sparkle(Stripes[stick-1].Color(Red, Green, Blue),
Interval1);
        break;
      case 5:
        Stripes[stick-1].Strobe(Stripes[stick-1].Color(Red, Green, Blue),
Interval1, Interval2, TotalSteps);
        break;
      case 6:
        Stripes[stick-1].Fade(Stripes[stick-1].Color(Red, Green, Blue),
Interval1, TotalSteps);
        break;
      case 7:
        Stripes[stick-1].Rainbow(Interval1, TotalSteps);
        break;
      case 8:
        Stripes[stick-1].Sweep(Stripes[stick-1].Color(Red, Green,
Blue),Interval1);
        break;
      case 9:
        Stripes[stick-1].Fire_h(Stripes[stick-1].Color(Red, Green, Blue));
        break;
      case 10:
        Stripes[stick-1].Fire_v(Interval1, Interval2, Cooling, Sparking);
        // Valores que funcionan bien
        // Stripes[stick-1].Fire_v(50, Interval2, 55, 120);
        break;
      case 11:
        Stripes[stick-1].Breathe(Stripes[stick-1].Color(Red, Green, Blue),
Interval1, TotalSteps);
        break;
    }
  }

  digitalWrite(13, !digitalRead(13));  //change_pin()
}
```