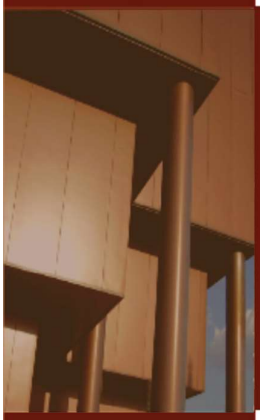
Trabajo de fin de grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Desarrollo de una aplicación de demostración sobre la plataforma Renesas Synergy

Autor: Gonzalo Alfonso Ojeda Jiménez
Tutor: Manuel Ángel Perales Esteve



Departamento de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo de Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Desarrollo de una aplicación de demostración sobre la plataforma Renesas Synergy

Autor:
Gonzalo Alfonso Ojeda Jiménez

Tutor:
Manuel Ángel Perales Esteve
Profesor Contratado Doctor

Departamento de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Autor: Gonzalo Alfonso Ojeda Jiménez

Tutor: Manuel Ángel Perales Esteve

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

*S*evilla, 2021

El Secretario del Tribunal

Agradecimientos

A mi familia, novia y amigos.

Gonzalo Alfonso Ojeda Jiménez
Alumno del Grado de Ingeniería Electrónica, Robótica y Mecatrónica
Sevilla, 2021

Resumen

En este documento se realiza un estudio del kit de iniciación Synergy SK-S7G2 de Renesas Electronics. Este, permite el desarrollo de aplicaciones sobre el entorno de desarrollo e2 Studio, basado en Eclipse.

La plataforma Synergy de Renesas, presenta una familia de microcontroladores escalable y herramientas de desarrollo que permite la portabilidad de aplicaciones entre sistemas, así como un sistema operativo de tiempo real (Azure ThreadX).

La plataforma Synergy de Renesas, presenta una familia de microcontroladores escalable y herramientas de desarrollo, incluyendo un RTOS (Azure ThreadX), APIs estandarizadas para manejar de manera sencilla los periféricos del microprocesador y herramientas de autogeneración de código, entre otros, permite la portabilidad de código entre sistemas.

Por estos motivos, sería interesante usar esta plataforma de microprocesadores en prácticas y/o trabajos de las asignaturas pertenecientes al Departamento de Electrónica.

En este trabajo de fin de grado, se desarrollará un estudio del kit SK-S7G2, así como una guía de uso del mismo, desde la instalación de los programas y drivers necesarios, hasta la demostración de uso de los periféricos principales del mismo.

Por último, se hará una conclusión, resumiendo las ventajas e inconvenientes de este producto y una comparativa con los microcontroladores de Texas Instruments que suelen usarse en asignaturas impartidas en la ETSI.

Abstract

This document performs a study of Renesas Electronics' Synergy SK-S7G2 Starter Kit. This allows the development of applications on the e2 studio development environment, based on Eclipse.

Renesas' Synergy platform presents a scalable family of microcontrollers and development tools that allows the portability of applications between systems, as well as a real-time operating system (Azure ThreadX).

Renesas' Synergy platform, presents a family of scalable microcontrollers and development tools, including an RTOS (Azure ThreadX), standardized APIs to easily manage microprocessor peripherals and code auto generation tools, among others, allows the portability of cross-system code.

For these reasons, it would be interesting to use this microprocessor platform in practices and / or assignments of the subjects belonging to the Department of Electronics.

In this final degree project, a study of the SK-S7G2 kit will be developed, as well as a guide for its use, from the installation of the necessary programs and drivers, to the demonstration of the use of its main peripherals.

Finally, a conclusion will be made, summarizing the advantages and disadvantages of this product and a comparison with the Texas Instruments micro-controllers that are usually used in ETSI.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

ARM	Advanced RISC Machine
RISC	Reduced Instruction Set Computer
DSP	Digital Signal Processor
NMI	Non-Maskable Interrupt
FPU	Float Point Unit
SIMD	Single Instruction, Multiple Data
MAC	Multiply and Accumulate
MPU	Memory Protection Unit
FFT	Fast Fourier Transform
IoT	Internet Of Things
SSP	Synergy Software Package
RTOS	Real-Time Operating System
API	Application Programming Interface
HW	Hardware
SW	Software
LQFP	Low-profile Quad Flat Package
LED	Light-Emitting Diode
LCD	Liquid-Crystal Display
USB	Universal Serial Bus
PMOD	Peripheral Module interface
QSPI	Quad Serial Peripheral Interface
SPI	Serial Peripheral Interface
I^2C	Inter-Integrated Circuit
CAN	Controller Area Network
SCI	Serial Communication Interface
IDE	Integrated Development Enviroment
CTW	Capacitive Touch Workbench
GCC	GNU Compiler Collection
BSP	Board Support Package

PLL	Phase-Locked Loop
GPIO	General Purpose Input/Output
CGC	Clock Generation Circuit
ELC	Event Link Controller
HAL	Hardware Abstraction Layer
ADC	Analog-to-Digital Converter
IRQ	Interrupt Request
PMW	Pulse Width Modulation
GPT	General PWM Timer
AGT	Asynchronous General purpose Timer
ISR	Interrupt Service Routine
DMAC	Direct Memory Access Controller
DTC	Data Transfer Controller
ELC	Event Link Controller
PGA	Programmable Gain Amplifier
DAC	Digital-to-Analog Converter
UART	Universal Asynchronous Receiver Transceiver
RTS	Ready To Send
CTS	Clear To Send
CTSU	Capacitive Touch Sensing Unit

1 INTRODUCCIÓN AL TRABAJO

Idea general del TFG

El uso de microcontroladores está presente en diversas asignaturas impartidas en la Escuela Técnica Superior de Ingeniería de Sevilla. Estas asignaturas se centran en un microcontrolador en concreto, normalmente del mismo fabricante, permitiendo a los alumnos conocer en profundidad el sistema, gracias a las prácticas y a los manuales de uso que se les proporciona.

En esta ocasión, me enfrento a un producto nuevo, de un fabricante que no está presente en ninguna asignatura o departamento de la Escuela.

En este Trabajo pues, se describe el microcontrolador elegido, sus características y el entorno de trabajo necesario para su programación, así como una guía de programación de los principales periféricos.

La guía de programación describirá los periféricos, incluyendo sus limitaciones, la configuración de los mismos, y las API necesarias para su uso, así como un ejemplo de programación de cada periférico.

Adicionalmente, se incluye una aplicación de demostración, en el cual se sintetiza todo lo descrito anteriormente.

Por último, se incluye una breve conclusión del trabajo realizado y sus posibles aplicaciones.

2 EL MICROPROCESADOR ARM CORTEX-M4

Introducción al procesador Cortex-M4

ARM es una arquitectura RISC, que engloba una gran cantidad de procesadores, diseñados por Acorn/ARM Holdings. Desde sus inicios, los procesadores ARM han estado caracterizados por su simplicidad, coste reducido y eficiencia, siendo ideales para aplicaciones que requieran poca potencia de cálculo. Así, se encuentran en la mayoría de la electrónica de consumo del mercado, como, por ejemplo, en la mayoría de los móviles, tabletas, relojes inteligentes, etc. actuales.

En la actualidad, existen 3 familias de procesadores ARM; Cortex-A, Cortex-R y Cortex M.

Los procesadores ARM Cortex-M son una familia de microprocesadores de 32 bits y conjunto de instrucciones reducido (RISC). Están optimizados para ser microprocesadores de bajo coste y energéticamente eficientes, y que pueden ser embebidos en distintos dispositivos de consumo.

2.1 Características

El procesador ARM Cortex-M4 es un procesador de 32 bits y alto rendimiento y bajo consumo, que presenta una bajo número de puertas lógicas, una baja latencia de interrupción y un depurador mejorado con capacidades de punto de rotura y seguimiento (*tracing)*.

El procesador incluye el núcleo, el controlador de interrupciones anidadas (NVIC), (fuertemente integrado con el núcleo para conseguir la menor latencia en el tratamiento de dichas interrupciones), múltiples buses de alto rendimiento, un depurador con las opciones de implementar puntos de ruptura, herramientas de seguimiento y parcheo de código entre otros.

Opcionalmente, puede incluir una unidad de protección de memoria (MPU) y una unidad de punto flotante (FPU), mejorando la robustez de la plataforma.

Adicionalmente, incluye un modo de suspensión (*sleep mode)* integrado y opcionalmente, un modo de menor consumo (*deep sleep mode*), ayudando así a mejorar la eficiencia y consumo de energía del procesador.

El procesador tiene una arquitectura Harvard (un bus para datos y otro para instrucciones) y un pipeline de 3 etapas (buscar, decodificar y ejecutar), haciendolo idóneo para sistemas embebidos.

Igualmente, este procesador es adecuado para aplicaciones de control en tiempo real, que requieren de un pipeline pequeño, un control de interrupciones determinista con la menor latencia posible y una ejecución de operaciones en el menor número de ciclos. Opcionalmente, puede incluir extensiones DSP, que contribuyen a una mayor potencia de cálculo, y al procesamiento eficiente de señales, necesitando menos ciclos para ejecutar algoritmos de control, por ejemplo.

Sus principales características se encuentran descritas en la Tabla 2-1.

Tabla 2-1 Características del microprocesador ARM Cortex-M4

Características	
Arquitectura ARM	ARMv7E-M
Arquitectura	Harvard
Pipeline	Tres etapas
Conjunto de instrucciones	RISC
Interrupciones	1 a 240 + NMI
FPU	opcional
Ejecución de instrucciones SIMD	Sí
Ejecución de instrucciones MAC en un ciclo	Sí
MPU	Opcional
Registros	Puntero a pila, *link register*, contador de programa, 13 registros de propósito general y 5 registros especiales.
Niveles de interrupción NMI	256
Latencia de interrupción	12 ciclos

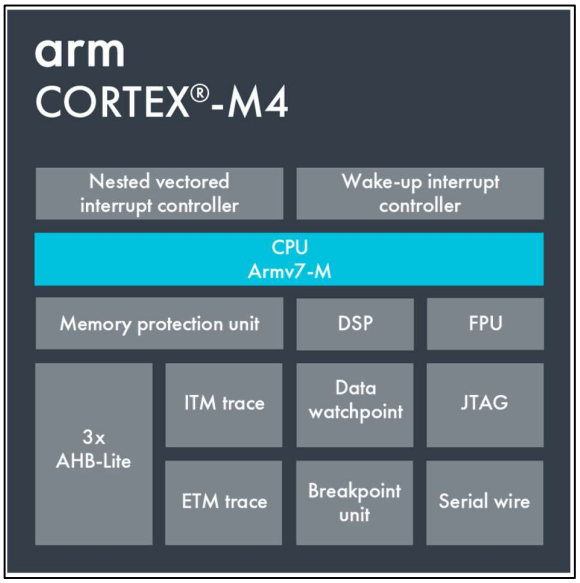


Figura 2-1 Arquitectura ARM Cortex-M4

2.2 Periféricos del procesador

Los periféricos incluidos en el procesador son los siguientes:

- NVIC (*Nested Vectored Interrupt Controller*): Es el controlador de interrupciones. Implementa hasta 240 interrupciones, con 256 prioridades. Soporta interrupciones no enmascarables (NMI) externas e interrupciones que "despiertan" al procesador cuando se encuentra en modos de bajo consumo (WIC, *Wakeup Interrupt Controller*).
- SCB (*System Control Block*): Son un conjunto de 19 registros que hacen de interfaz con el procesador, y proporcionan información, opciones de configuración y control.
- SysTick (*System Timer*): Se trata de la señal de reloj del sistema. Consiste de un temporizador de 24 bits que cuenta hacia atrás. Distintos registros se usan para controlarlo, resetearlo, etc.
- MPU (*Memory Protection Unit*): Es un periférico opcional. Se encarga de mejorar la fiabilidad de la memoria. Para ello, divide el mapa de memoria en un número de regiones, definiendo la localización, tamaño, permisos de acceso y atributos de cada una de ellas.
- FPU (*Floating Point Unit*): Es un periférico opcional. Se trata de la unidad de coma flotante, que soporta sumas, restas, multiplicaciones, divisiones, operaciones de multiplicar-acumular, y raíces cuadradas con datos con formato de simple precisión. Provee también conversiones entre formatos de coma fija y flotante.

Tabla 2-2 Periféricos del microprocesador ARM Cortex-M4

Periférico	Descripción
NVIC	Control de interrupciones integrado. Soporta el procesamiento de interrupciones con baja latencia
System Control Block	Interfaz al procesador. Provee información, configuración y control del sistema.
System Timer	El reloj del sistema. Temporizador de 24 bits. Usado como el reloj del RTOS o como simple contador.
Memory Protection Unit	Mejora la fiabilidad del sistema definiendo los atributos de memoria a diferentes regiones (hasta 8 regiones).
Floating-point Unit	Provee cálculo en punto flotante, conforme a la norma IEEE 754-2008.

2.3 Mapa de memoria

En la figura 2-2, encontramos el mapa de memoria del procesador, en el que encontramos la división de la memoria direccionable por el procesador. Las regiones de SRAM y periféricos incluye regiones de bandas de bits, que permiten manipular bits de los valores guardados en una dirección de memoria de dicha región, sin tener que ejecutar una secuencia de instrucciones de lectura, modificación y escritura para ello.

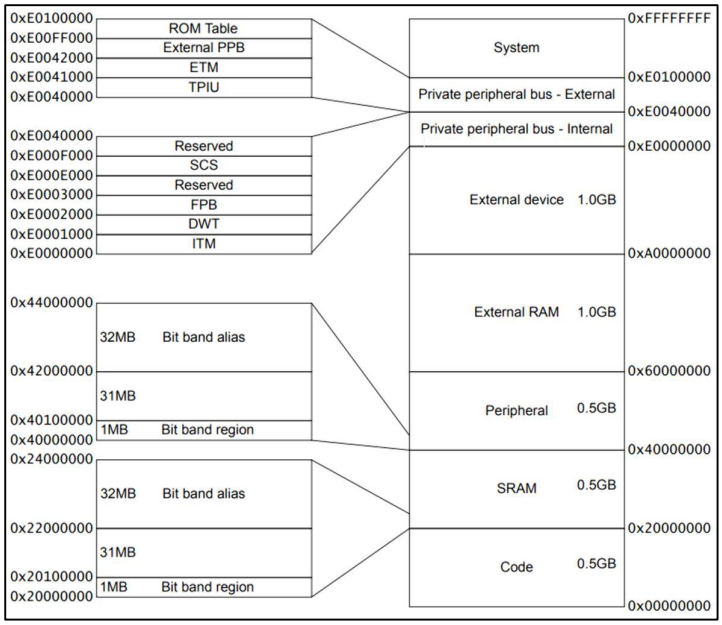


Figura 2-2 Mapa de memoria del procesador ARM Cortex-M4

2.4 Ventajas

La principal ventaja de este microprocesador es la combinación entre potencia de cálculo y eficiencia.

Por ejemplo, es más potente que su hermano pequeño, el microprocesador Cortex-M3, pero es capaz de consumir menos energía al ejecutar operaciones intensivas. Como podemos comprobar en la siguiente figura, el microprocesador Cortex-M4 es capaz de realizar la misma tarea (FFT de 512 muestras) con un consumo bastante menor.

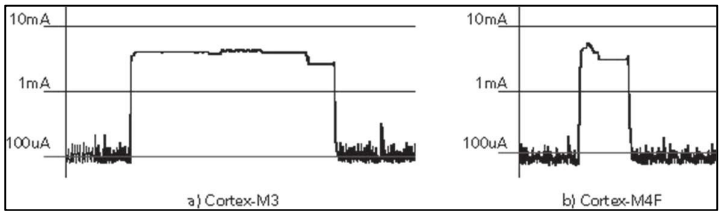


Figura 2-3 Comparación de consumos entre los microprocesadores Cortex M3 y M4 al realizar una FFT de 512 muestras.

3 RENESAS SYNERGY SK-S7G2

Introducción al kit SK-S7G2

Renesas Electronics es una compañía dedicada a soluciones en sectores como la automoción, la industria, infraestructuras y el IoT. En este sector, podemos encontrar una gran variedad de microcontroladores y microprocesadores de Renesas, tanto propietarios, como basados en ARM, dentro de los cuales encontramos los microcontroladores de la familia Synergy.

Esta familia de microcontroladores está diseñada para una gran variedad de aplicaciones, desde dispositivos móviles conectados para el IoT, hasta sistemas embebidos de alto rendimiento. Dentro, podemos encontrar distintas series de microcontroladores con distintas características y niveles de rendimiento, pero que permiten reusar códigos con facilidad y, por tanto, la escalabilidad entre los distintos microcontroladores que componen la familia.

En este capítulo, se explicarán las características del kit SK-S7G2, que ha sido utilizada durante el desarrollo de este Trabajo de Fin de Grado.

3.1 Familia Synergy

La plataforma Synergy presenta distintas series de hardware, adaptándose así a las necesidades de cada aplicación. En un extremo del espectro, encontramos la serie S1, focalizada en la eficiencia energética y aplicaciones de bajo consumo. En el polo opuesto, encontramos la serie S7, diseñada para aplicaciones que necesiten de mayor capacidad de procesamiento. Dentro de cada serie, encontraríamos distintos microcontroladores de distintas prestaciones.

3.1.1 Procesadores ARM Cortex-M

Para lograr la escalabilidad e integración que busca Renesas con su familia Synergy, necesita una gama de procesadores escalables que sean sencillos de utilizar. Para ello, han optado por incluir los procesadores ARM Cortex M, debido a su potencia, gran eficiencia energética y al uso de la misma arquitectura (ARMv7-M).

3.1.2 Synergy Software Package

Además de un amplio espectro de hardware, la familia Synergy cuenta con el Synergy Software Package, o SSP. Consiste en una suite de software optimizado e integrado junto con un Sistema Operativo en Tiempo Real (Azure RTOS ThreadX) que permiten la escalabilidad de la familia Synergy, así como la unificación, abstracción y simplificación de la programación de los mismos, mediante APIs de uso sencillo, a pesar de las diferencias a

nivel HW y SW entre dispositivos.

3.2 Características del kit SK-S7G2

3.2.1 Periféricos

El kit de iniciación SK-S7G2 incluye los siguientes componentes:

Tabla 3-1: Componentes del kit SK-S7G2

#	Componentes
1	Encapsulado LQFP con el microprocesador
2	4 conectores con acceso a todos los pines
3	3 LEDs de usuario
4	2 pulsadores conectados a pines de interrupción
5	2 botones táctiles capacitivos
6	Deslizador táctil capacitivo
7	Pantalla LCD táctil (resistiva)
8	Salida de audio de 3.5mm
9	Conector Ethernet RJ45
10	2 puertos micro USB
11	1 puerto USB tipo A
12	2 conectores PMOD

En la Figura 3-2, encontramos señalados todos los componentes descritos en la Tabla 3-1.

A estos componentes se les incluye las siguientes funcionalidades:

- USB: Puerto de comunicaciones (J5), depuración (J19), alimentación (J19) y almacenamiento.
- Memoria flash QSPI de 8MB
- Interfaces SPI, I^2C, CAN, SCI
- Interfaz compatible con Arduino Shield, permitiendo el uso de cualquier expansión de hardware diseñado para el microcontrolador Arduino UNO.

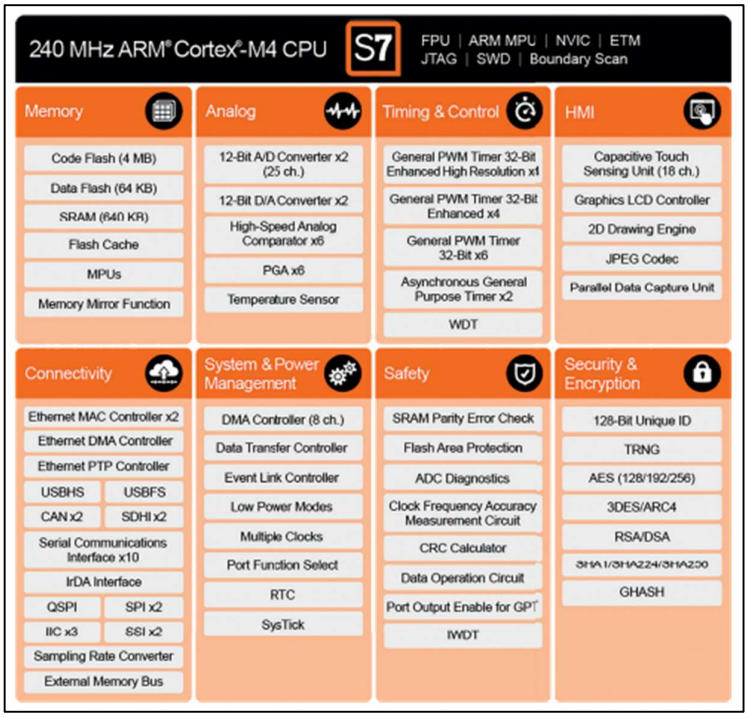


Figura 3-1. Características de la serie S7

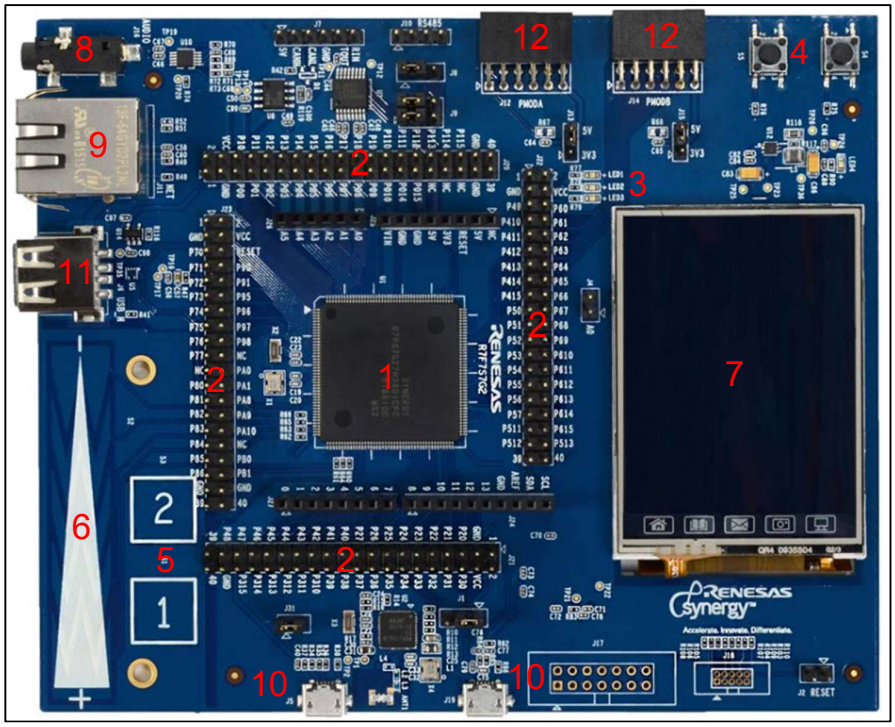


Figura 3-2. Kit SK-S7G2

4 ENTORNO DE TRABAJO

Descripción del Hardware incluido, así como el software y configuración necesarios

Al igual que para la programación de cualquier otro microcontrolador, existen ciertos requisitos necesarios para poder empezar a programarlo, desde el hardware necesario (incluido en el kit), como el software y su configuración necesarios. En este apartado se describe todo lo necesario para poder hacer uso de este kit de programación.

4.1 Hardware

El kit trae los recursos mínimos necesarios para poder programar:

- Placa SK-S7G2
- Cable USB tipo A a micro USB tipo B para alimentar y depurar
- Guía rápida de uso

4.2 Software

4.2.1 e2 Studio

e2 Studio es el entorno de desarrollo (IDE) basado en eclipse usado para programar los microcontroladores de Renesas. Con este software, cubriremos todo el proceso de desarrollo de nuestra aplicación; edición, compilación y depuración. Sus principales características son las siguientes:

- Es software propietario, pero Renesas nos permite licenciarlo de forma gratuita para uso estudiantil.
- Cubre todos los aspectos del desarrollo.
- Facilidad de uso gracias a su interfaz gráfica. El editor Eclipse CDT ofrece muchas funcionalidades para el desarrollo en C/C++.
- Capacidad de autogeneración de código.
- Configuración sencilla de periféricos.
- Compatible con distintos toolchains (Nosotros usaremos Cross ARM C Compiler)

Para el desarrollo de este Trabajo de Fin de Grado se ha usado la versión 7.5.1 de e2 Studio, que se puede encontrar en el siguiente enlace:

https://www.renesas.com/us/en/software-tool/archived-e-studio-renesas-synergy

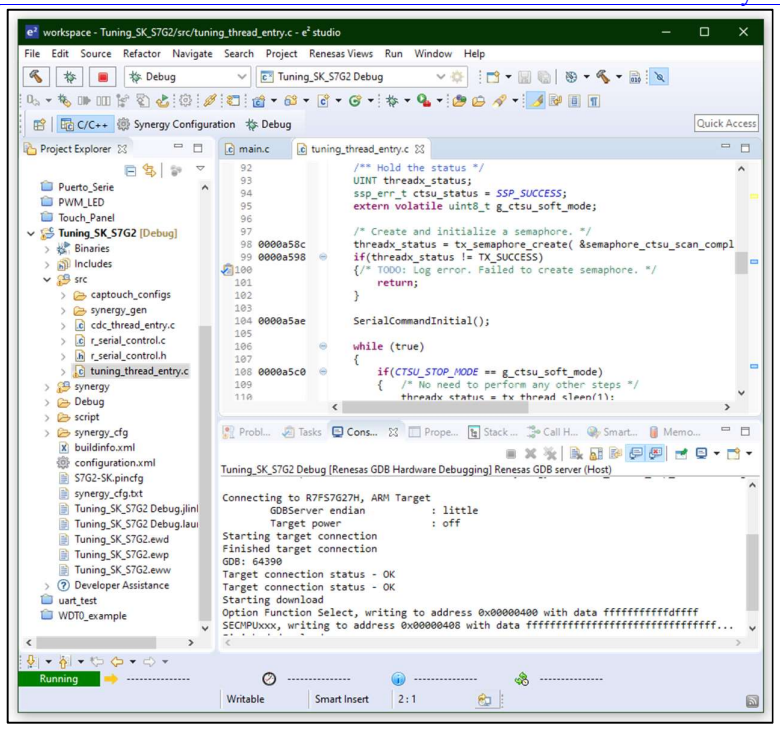


Figura 4-1. Ejemplo en e2 Studio

4.2.2 SSP (Synergy Software Package)

SSP es el software necesario para poder programar con facilidad todos los microcontroladores de la familia Synergy, incluyendo todo el middleware necesario para estandarizar la programación, siendo igual en cualquiera de ellos a pesar de la diferencia de hardware.

La versión usada para el desarrollo de este Trabajo de Fin de Grado es la v1.7.8, y está incluida en la descarga de e2 Studio enlazada anteriormente.

4.2.3 CTW for Synergy (Capacitive Touch Workbench)

CTW es un software para configurar correctamente los sensores capacitivos que se encuentren en la placa. En nuestro caso, son dos botones y un slider. Tras una sencilla configuración, se prueba su funcionamiento y el software automáticamente los calibra, estableciendo un umbral de detección independiente para cada sensor.

La versión usada para el desarrollo de este Trabajo de Fin de Grado es la v1.5.0033 y se puede encontrar en el siguiente enlace.

https://www.renesas.com/us/en/software-tool/capacitive-touch-workbench-renesas-synergy

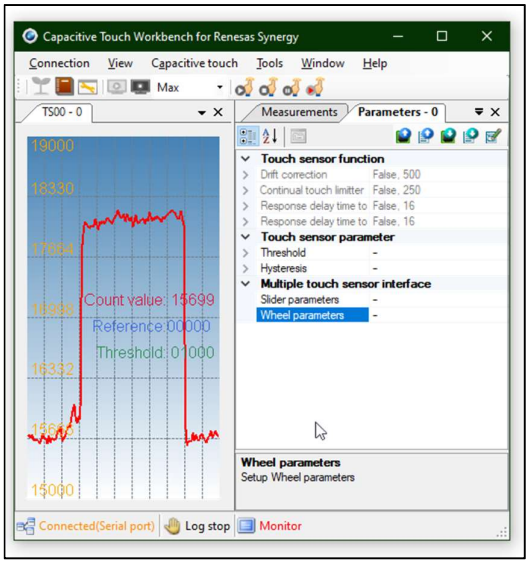


Figura 4-2. Ejemplo de *tuning* de sensor capacitivo en CTW

4.2.4 GUIX Studio

GUIX Studio es una interfaz gráfica que nos permitirá diseñar de manera visual y sencilla el layout de la pantalla a usar en nuestra aplicación. Este software generará todo el código necesario para incluir en nuestro proyecto.

La versión utilizada es la v5.6.1.0 y se puede descargar desde el siguiente enlace:

https://www.renesas.com/us/en/software-tool/archived-guix-studio-releases-renesas-synergy

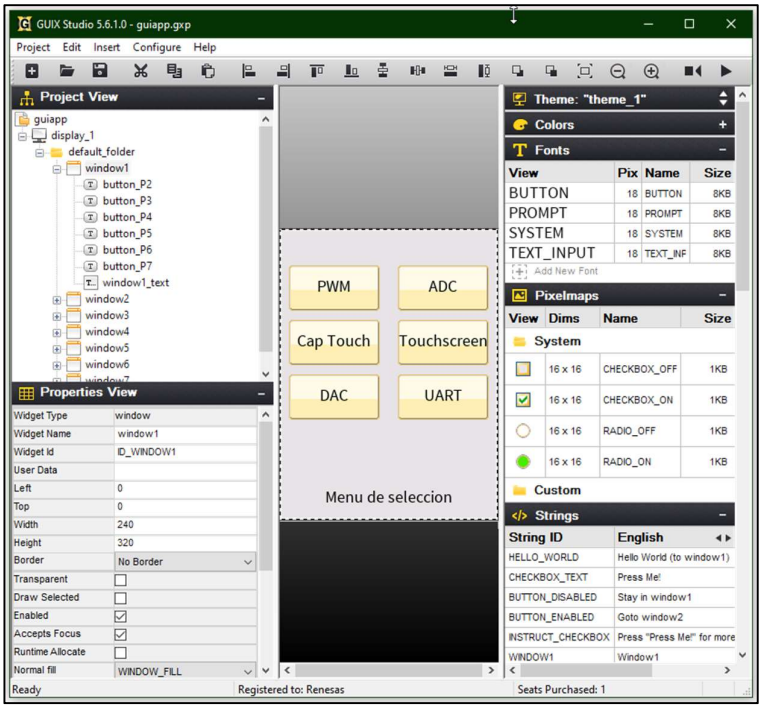


Figura 4-3. Ejemplo en GUIX Studio

4.2.5 TraceX

TraceX es una herramienta de análisis gráfico que permite visualizar eventos en sistemas de tiempo real, proporcionándonos información de ejecución, estadísticas de rendimiento, etc. De esta manera, permite monitorizar el comportamiento de los hilos (interrupciones, suspensión, reanudación, etc.) permitiendo al desarrollador depurar problemas inherentes a sistemas de tiempo real como, por ejemplo, las inversiones de prioridad.

Esta herramienta se ha testado, pero no se le ha encontrado ningún uso dentro de este proyecto.

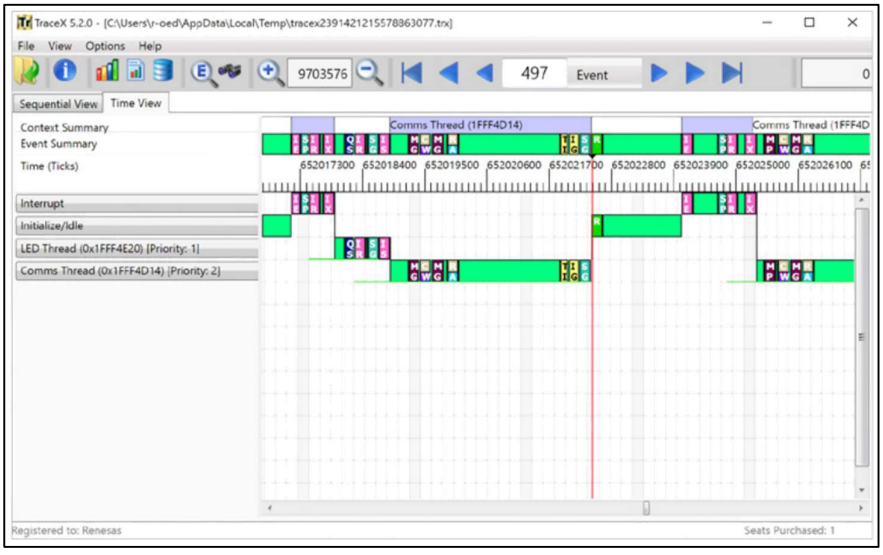


Figura 4-4. Ejemplo en TraceX

4.2.6 TeraTerm

Tera Term es un software de emulador de terminal. Lo usaremos para comunicar con la placa mediante puerto serie.

Esta herramienta se puede encontrar en el siguiente enlace:

https://osdn.net/projects/ttssh2/releases/

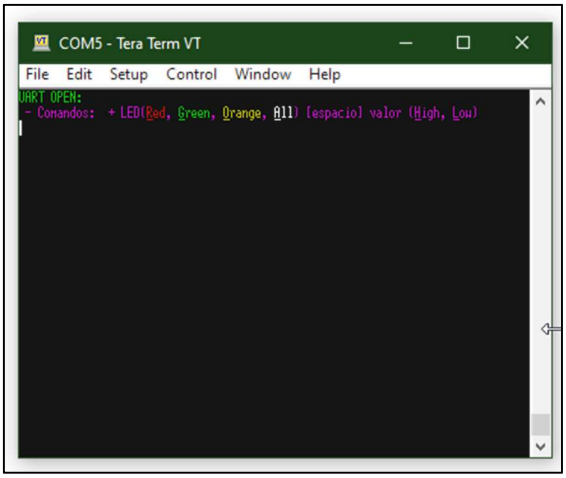


Figura 4-5. Pantalla de TeraTerm

5 CONFIGURACIÓN INICIAL Y ESTRUCTURA DEL SOFTWARE

Descripción del entorno de programación y de la estructura de las API a usar

En este apartado se describen los primeros pasos en el entorno de desarrollo, una descripción breve de las principales pantallas que se usarán, así como la estructura general de las API de programación de cada periférico.

5.1 Primeros pasos

Una vez tengamos instalado e2 Studio, encontraremos una interfaz similar a la que ofrecen otros IDE con los que ya habremos trabajado previamente en asignaturas de grado, como puede ser Code Composer Studio, usado para programar con microcontroladores de Texas Instruments.

Para crear un proyecto, iremos a *File->New->C/C++ Project* y seleccionaremos la primera opción (*Renesas Synergy C Executable Project*).

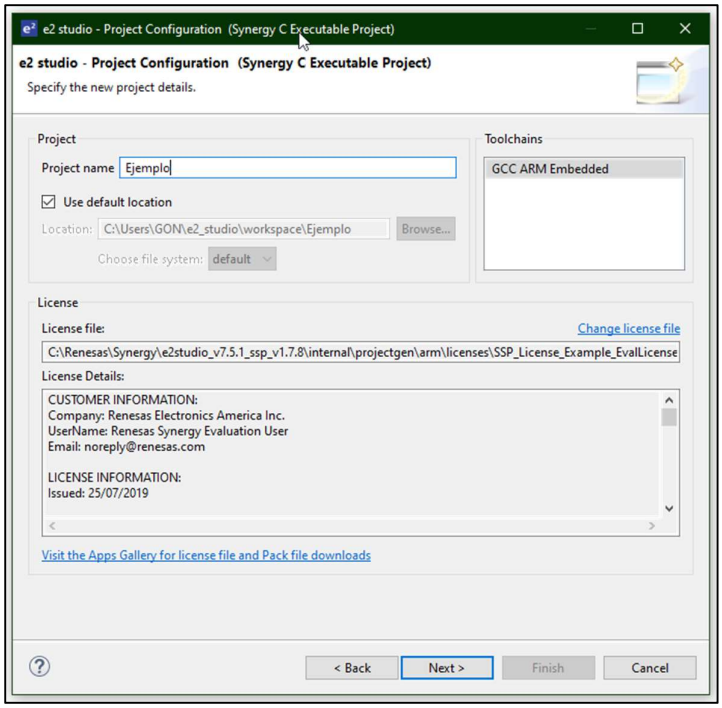


Figura 5-1. Pantalla de creación de proyecto en e2 Studio.

Nos aparecerá una ventana donde debemos indicar el nombre del proyecto, su ubicación y el toolchain a usar. Recomendamos usar la ubicación por defecto (es decir, el espacio de trabajo que hayamos elegido) y que usemos el toolchain GCC ARM Embedded. Tras esto, seleccionaremos *Next.*

Ahora, debemos seleccionar la versión del SSP que queremos usar (En nuestro caso, la 1.7.8) y nuestro kit (SK -S7G2). Adicionalmente, nos debemos seleccionar el modelo exacto. En nuestro caso, seleccionaremos ***R7FS7G27H3A01CFC*** en el campo *Device.* Por último, seleccionaremos la versión de nuestro toolchain (solo tendremos una versión instalada) y el debugger, *J-Link ARM.* Seleccionamos *Next*.

En esta última ventana, seleccionaremos *BSP* y pulsaremos en *Finish.*

BSP o *Board Support Package* se encarga de inicializar el microcontrolador tras alimentarlo o resetearlo. Es específico para cada microcontrolador y ejecuta las funciones de inicialización necesarias para llegar a la función main() de nuestra aplicación.

También tendremos la opción de elegir un ejemplo de inicio, *Blinky* y *Blinky with ThreadX.* Son el mismo ejemplo, un LED que parpadea, pero el segundo es una versión multihilo.

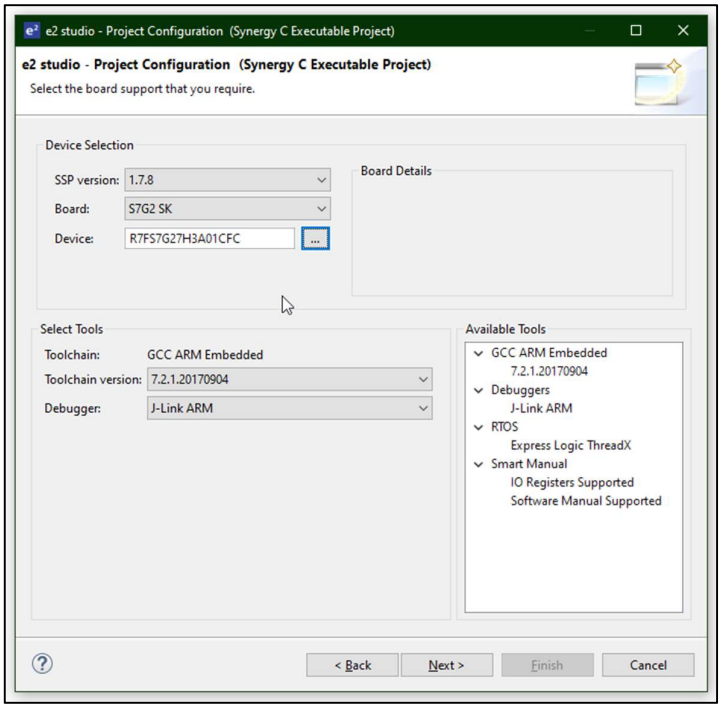


Figura 5-2. Pantalla de creación de proyecto en e2 Studio.

Tras la creación de nuestro proyecto, se abrirá automáticamente la ventana de configuración del proyecto (*Synergy Configuration*). Esta ventana contiene varias pestañas:

- *Summary:* Contiene un resumen del proyecto. Muestra las opciones elegidas en la ventana de creación del proyecto.
- *BSP:* Muestra la versión de SSP seleccionada, la placa objetivo y el modelo concreto que hemos seleccionado previamente. Podemos modificar estos campos si es necesario.
- *Clocks:* Muestra la configuración de los relojes del sistema. Aquí seleccionamos el reloj a usar en nuestro proyecto, y las distintas divisiones del mismo, las cuales serán usados por distintos periféricos. Con el uso del reloj *XTAL* de 24 MHz y un PLL, este microcontrolador puede correr a frecuencias de hasta 240MHz. Generalmente no tocaremos esta configuración, aunque algunos periféricos necesitaran de modificaciones en este apartado.
- *Pins:* En este apartado podremos cambiar la configuración de cada pin del microcontrolador. Por ejemplo, podemos configurarlo en modo *input*, para leer una señal digital, y configurar una resistencia de pull-up y habilitar la interrupción asociada a ese pin. También podemos filtrar por periférico y configurar sus pines asociados.
- *Threads:* Muestra los hilos incluidos en nuestro proyecto. Por defecto, siempre existirá el hilo *HAL/Common*, que contiene los drivers para el control de GPIO (*I/O Port*), circuito de generación de reloj (*clock generation circuit, CGC*) y el controlador de eventos (*Event Link Controller, ELC).* En esta pestaña podremos incluir nuevos hilos de programación y sus objetos (colas de mensajes, semáforos, mutex y *flags*). En el apartado *Thread Stacks*, podremos añadir los drivers de los periféricos que vayamos a usar de manera sencilla.
- *Messaging:* Los microcontroladores de Renesas incluyen un entorno de trabajo que permite enviar mensajes entre hilos de manera más eficiente que con las colas de mensaje.
- *Components:* Permite visualizar los componentes incluidos en nuestro proyecto.

Las pestañas más importantes son *Clocks*, *Pins* y *Threads*, y de ahora en adelante nos centraremos en ellos

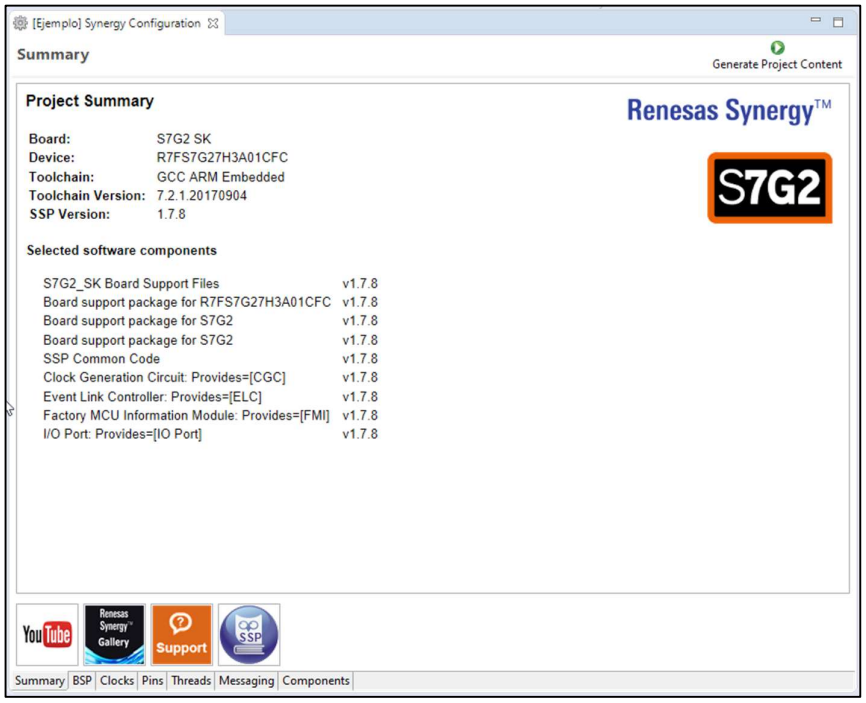


Figura 5-3. Ventana de configuración del proyecto - Resumen.

Tras haber configurado el microcontrolador acorde con las necesidades de nuestra aplicación, pulsaremos el botón *Generate Project Content* se crearán los archivos fuente necesarios en la carpeta de nuestro proyecto.

Tras haber creado y generado el proyecto, encontraremos en el explorador de proyectos una carpeta con el mismo. Dentro de la carpeta *src* encontraremos una carpeta llamada *synergy_gen* que contiene todo el código autogenerado, y un archivo llamado *hal_entry.c* que será el archivo donde debemos programar.

Es contraintuivo en un principio que no sea *main.c* donde debamos programar, pero este se encuentra entre los archivos autogenerados y ejecutará el código necesario previo para poder ejecutar nuestros hilos (en este punto, solo tendremos un hilo de ejecución), y que al final del mismo, llamará a los hilos *<nombre del hilo>_entry*.

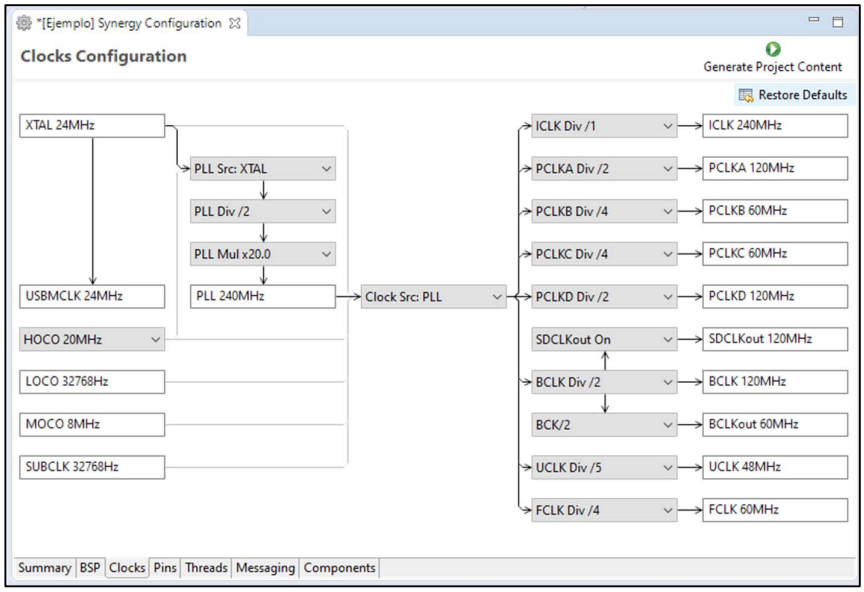


Figura 5-4. Ventana de configuración del proyecto – Relojes.

5.2 Pestaña Threads

Esta es la pestaña más importante de la ventana de configuración. En ella incluimos todos los hilos, objetos de tiempo real y los periféricos a incluir en el proyecto. También podremos configurar cada uno de ellos en esta pestaña.

5.2.1 HAL Drivers

Sobre la capa BSP, que se encarga de inicializar el micro, encontramos la capa HAL (*Hardware Abstraction Layer*) o capa de abstracción hardware. Esta capa, provee drivers para los periféricos, aislando al programador de la capa hardware.

Consiste en una colección de módulos y cada uno de ellos es un driver para cada periférico disponible. Estos módulos son independientes del RTOS y se componen de dos partes. La primera, es un driver de bajo nivel que se encarga de manipular los registros del periférico en cuestión, y un driver de alto nivel, que se encarga de enlazar la interfaz de programación con el driver de bajo nivel.

Así, se generan APIs que se pueden portar entre los microcontroladores de la familia Synergy. También permite cierta flexibilidad a la hora de elegir un driver, ya que se puede elegir otro a mitad del desarrollo, sin necesidad de cambiar el código desarrollado. Por ejemplo, se puede elegir en un principio usar el driver dedicado para el periférico hardware I2C, pero luego usar el driver para el SCI funcionando en modo I2C.

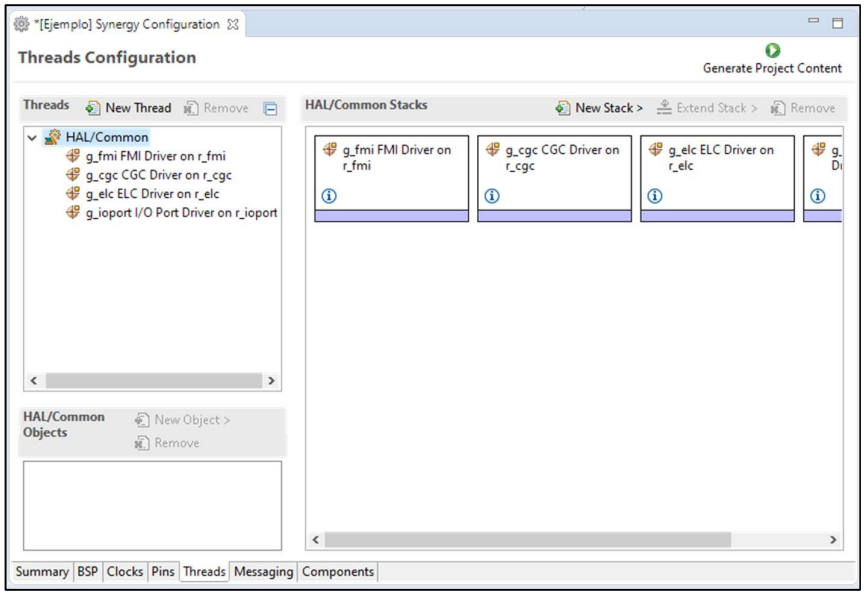


Figura 5-5. Ventana de configuración del proyecto – Hilos.

5.2.2 Application Framework

Por encima de la capa HAL, encontramos esta capa de abstracción adicional, que permite combinar distintos drivers de bajo nivel en un mismo marco y contienen drivers integrados en el RTOS. Podemos encontrar drivers HAL de un periférico y un *framework* para el mismo periférico. La diferencia radica en que el framework incluye el manejo de objetos de RTOS necesarios para evitar conflictos entre hilos que usen recursos comunes, o para la sincronización de los mismos.

Para incluir un *framework* en nuestro proyecto necesitamos crear un hilo adicional al hilo *HAL/common* que únicamente permite el uso de drivers HAL.

5.3 Pestaña Pins

En esta pantalla encontraremos la configuración de los pines de nuestro microcontrolador. La mayoría de pines están multiplexados y ofrecen distintas configuraciones.

En la zona *Pin Selection* podemos buscar el pin deseado, buscando en el árbol desplegable o filtrando por texto. También es posible realizar una búsqueda inversa, podemos buscar el periférico que deseamos programar y buscar los pines necesarios para poder hacerlo (opción recomendada).

Una vez seleccionado el pin deseado, en la zona *Pin Configuration* encontramos varios menús desplegables que ofrecen las posibles configuraciones del mismo.

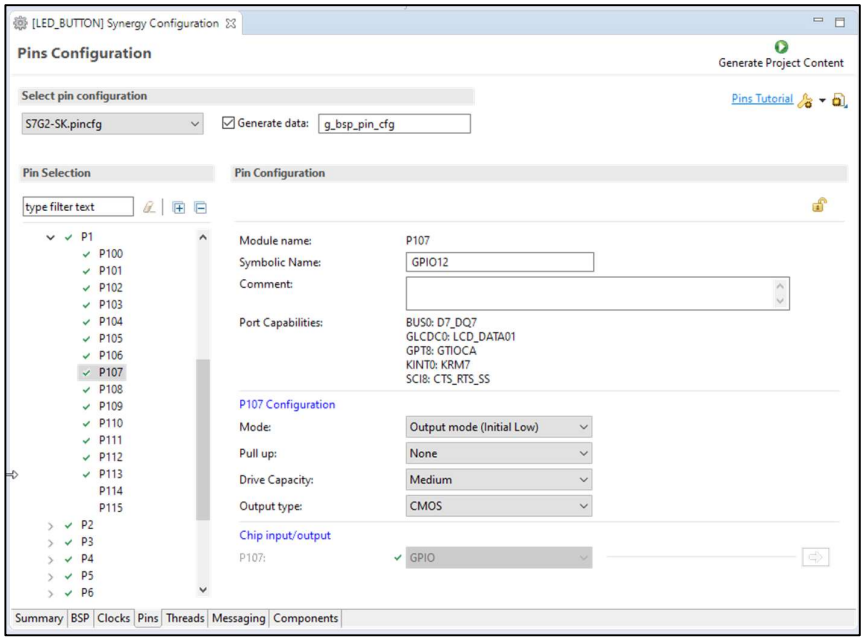


Figura 5-6. Ventana de configuración del proyecto – Pines.

5.4 Pestaña Properties

En la perspectiva por defecto del IDE, encontraremos en la zona inferior la pestaña *Properties*.

Aquí encontraremos las distintas propiedades del micro si nos encontramos en la pestaña *BSP*, algunas de las cuales se pueden cambiar, aunque no lo veremos en este proyecto, y las distintas propiedades del elemento seleccionado en la pestaña *Threads* como, por ejemplo, un módulo o un objeto de tiempo real. En el Segundo caso, podemos ver información de la API del driver seleccionado.

5.5 Pestaña Console

Aparece junto con la pestaña *Properties*. Es la consola del IDE y nos permite ver el estado de la compilación y nos muestra información importante a la hora de encontrar errores en nuestro código.

5.6 Pestaña Problems

Nos muestra información sobre nuestro código. Se dividen en tres categorías bastante autoexplicativas; *error, warning* e *info*.

5.7 Estructura de las API

Todas las API presentan una estructura común, que contiene los punteros a las funciones soportadas.

La estructura es la siguiente; *<nombre del periférico>.p_api-><nombre de la función>(<argumentos>).*

Donde:

- *<nombre del periférico>* es el nombre dado al módulo correspondiente en la pestaña *Threads* de la ventana de configuración del proyecto.

- <*nombre del periférico*> es el nombre de la función que deseamos ejecutar.
- <*argumentos*> son los argumentos admitidos por dicha función.

5.8 Objetos de RTOS

Como cualquier otro RTOS, ThreadX hace uso de varios objetos para asegurar la sincronización y la comunicación de los distintos hilos de ejecución presentes en nuestra aplicación; semáforos, *flags*, colas de mensaje y mutex.

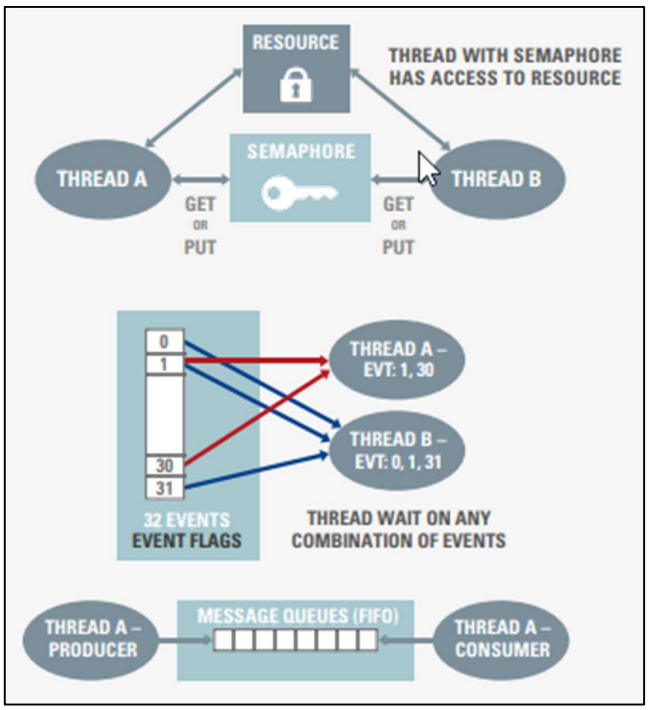


Figura 5-7 Esquema-resumen del funcionamiento de objetos RTOS.

5.8.1 Flags

Es el principal modo de sincronización entre hilos, y el más sencillo. Es una forma de informar a otro hilo que un evento concreto ha sucedido, permitiendo la ejecución de cierto código tras el evento. Es similar a una variable booleana en una aplicación con un solo hilo (con una sentencia if, ejecutaremos una porción de código si la variable booleana tiene el valor requerido).

En este caso, uno o varios hilos quedarán bloqueado esperando que un flag cambie de estado, o comprobará su estado periódicamente, ejecutando otra porción de código mientras. Por otro lado, otro hilo deberá señalizar el suceso del evento "levantando la bandera".

Los *flags* en ThreadX se agrupan en grupos de *flags*, que contienen 32 bits, permitiendo señalizar 32 eventos. Por sencillez, en los programas desarrollados en este Trabajo de Fin de Grado, se ha usado únicamente un flag por grupo.

Para incluir un grupo de *flags* en nuestra aplicación, primero debemos seleccionar un hilo (distinto al hilo *HAL/Common*), dentro de la pestaña *Threads*. Justo debajo del menú desplegable que muestra los hilos, seleccionaremos el botón *New Object* y posteriormente *Event Flags*.

Ahora nos aparecerá nuestro *flag* en la lista de objetos RTOS del hilo seleccionado. En la pestaña *Properties* podemos modificar el nombre del mismo.

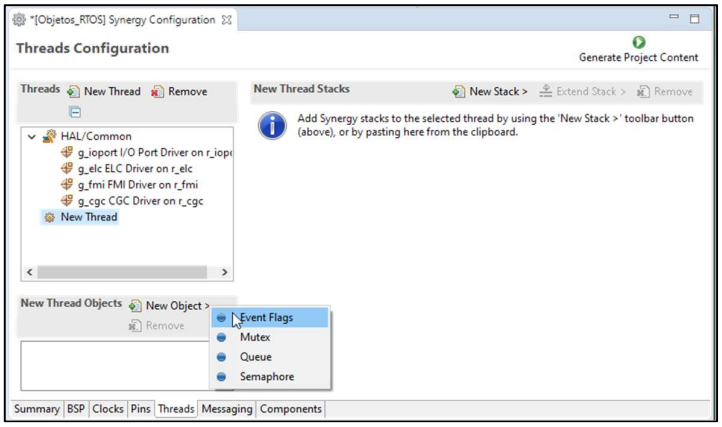


Figura 5-8 Inclusión de objetos RTOS

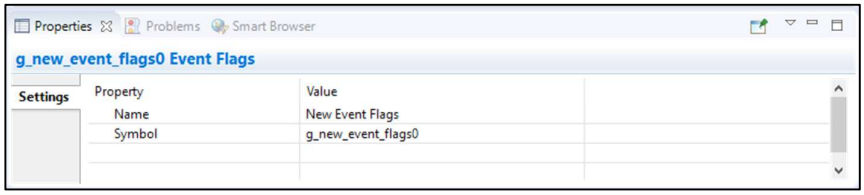


Figura 5-9 Propiedades de un *flag*

Para poder hacer uso del mismo, debemos incluir la cabecera del hilo donde hemos creado el objeto en todos los hilos que hagan uso del mismo.

Para usarlo, usaremos las siguientes funciones:

- ***tx_event_flags_set*** *(TX_EVENT_FLAGS_GROUP *group_ptr, ULONG flags_to_set, UINT set_option)*: Establece el estado del grupo de *flags group_ptr* , según los argumentos *flags_to_set* y *set_option*. El argumento *flags_to_set* contiene el valor de los 32 *flags*/bits a modificar, y el argumento *set_option* contiene la operación lógica (TX_AND o TX_OR).
- ***tx_event_flags_get*** *(TX_EVENT_FLAGS_GROUP *group_ptr, ULONG requested_flags, UINT get_option, ULONG *actual_flags_ptr, ULONG wait_option)*: Recupera el estado del grupo de *flags group_ptr*. El argumento *requested_flags* contiene el valor buscado de los 32 *flags*/bits, el argumento *get_option* contiene la operación lógica (TX_AND o TX_OR), y *wait_option* indica el tiempo de espera (TX_WAIT_FOREVER espera para siempre, bloqueando la ejecución del hilo y TX_NO_WAIT solo comprueba una vez si los *flags* solicitados están "levantados").

Adicionalmente, tras esperar un *flag*, haremos uso de una sentencia *if*, verificando que el valor guardado en *actual_flags_ptr* es el deseado. Podría darse la situación en la que un evento provoque que el hilo salga de la espera, por lo que siempre debemos comprobarlo.

5.8.2 Semáforos

En ThreadX, los semáforos son contadores. Este tipo de semáforos almacena el número de veces que se puede acceder a un recurso. El contador puede aumentar en una unidad cada vez que se ejecute una operación *put* y decrementa cada vez que una operación *get* se ejecute con éxito, además de garantizar el acceso al recurso compartido al hilo que ejecuta dicha operación.

Su uso principal es el acceso a recursos compartidos (datos, por ejemplo), aunque también se puede usar para la sincronización de hilos.

Para incluir un semáforo en nuestra aplicación, primero debemos seleccionar un hilo (distinto al hilo *HAL/Common*), dentro de la pestaña *Threads*. Justo debajo del menú desplegable que muestra los hilos,

seleccionaremos el botón *New Object* y posteriormente *Semaphore*.

Ahora nos aparecerá nuestro semáforo en la lista de objetos RTOS del hilo seleccionado. En la pestaña *Properties* podemos modificar el nombre del mismo y el valor inicial del contador.

Figura 5-10 Propiedades de un semáforo

Para poder hacer uso del mismo, debemos incluir la cabecera del hilo donde hemos creado el objeto en todos los hilos que hagan uso del mismo.

Para usarlo, usaremos las siguientes funciones:

- ***tx_semaphore_put*** *(TX_SEMAPHORE *semaphore_ptr)*: Coloca una instancia en el semáforo contador especificado, incrementando en una unidad el valor del mismo.
- ***tx_semaphore_get*** *(TX_SEMAPHORE *semaphore_ptr, ULONG wait_option)*: Recupera una instancia del semáforo contador, si el contador es mayor que cero, y decrementa en una unidad. El argumento *wait_option* indica el tiempo de espera.

5.8.3 Mutex

Es un objeto similar al semáforo. Permite el acceso a recursos compartidos o secciones críticas.

A diferencia del semáforo, el mutex es un mecanismo de bloqueo usado para acceder a un recurso, y solo un hilo puede tomar la posesión del mutex (y por tanto, el permiso para acceder al recurso compartido), mientras que el semáforo señala cuando se puede acceder a un recurso.

Cuando esté disponible, el mutex tendrá el valor 0, y su valor se incrementa por cada operación *get* exitosa, y se decrementa por cada operación *put* exitosa.

Para incluir un mutex en nuestra aplicación, primero debemos seleccionar un hilo (distinto al hilo *HAL/Common*), dentro de la pestaña *Threads*. Justo debajo del menú desplegable que muestra los hilos, seleccionaremos el botón *New Object* y posteriormente *Mutex*.

Ahora nos aparecerá nuestro mutex en la lista de objetos RTOS del hilo seleccionado. En la pestaña *Properties* podemos modificar el nombre del mismo y la opción de usar el mecanismo de herencia de prioridad (un hilo de baja prioridad adquirirá la prioridad de un hilo de mayor prioridad que esté esperando a que el primer hilo finalice el acceso al recurso compartido, minimizando el impacto de una inversión de prioridad).

Figura 5-11 Propiedades de un mutex

Para poder hacer uso del mismo, debemos incluir la cabecera del hilo donde hemos creado el objeto en todos los hilos que hagan uso del mismo.

Para usarlo, usaremos las siguientes funciones:

- ***tx_mutex_get*** *(TX_MUTEX *mutex_ptr, ULONG wait_option)*: Intenta obtener el uso exclusivo del mutex. El argumento *wait_option* indica el tiempo de espera.
- ***tx_mutex_put*** *(TX_MUTEX *mutex_ptr)*: Libera el mutex, si el hilo que llama a esta función lo posee.

5.8.4 Colas de mensaje

Las colas de mensaje son el principal forma de comunicación entre hilos. Es una cola tipo FIFO. Una cola de mensajes soporta un determinado número de mensajes de un tamaño fijo. En ThreadX, el tamaño del mensaje puede variar desde una palabra de 32 bits, hasta 16 palabras. Mensajes de mayor tamaño se pueden enviar y recibir usando punteros.

Un mensaje se copia en la cola usando una función *send* y se copia de la cola usando la función *receive*.

Para incluir una cola de mensajes en nuestra aplicación, primero debemos seleccionar un hilo (distinto al hilo *HAL/Common*), dentro de la pestaña *Threads*. Justo debajo del menú desplegable que muestra los hilos, seleccionaremos el botón *New Object* y posteriormente *Queue*.

Ahora nos aparecerá nuestra cola de mensajes en la lista de objetos RTOS del hilo seleccionado. En la pestaña *Properties* podemos modificar el nombre del mismo, así como el tamaño en palabras de 32 bits de cada mensaje y el tamaño en bytes de la cola, que debe ser igual o superior a <# mensajes deseado> * <tamaño de cada mensaje en palabras> /4 bytes/palabra.

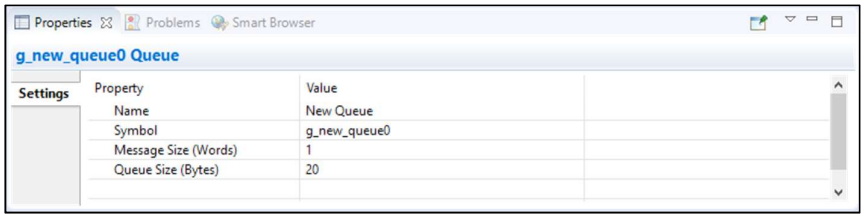


Figura 5-12 Propiedades de una cola de mensajes

Para poder hacer uso del mismo, debemos incluir la cabecera del hilo donde hemos creado el objeto en todos los hilos que hagan uso del mismo.

Para usarlo, usaremos las siguientes funciones:

- ***tx_queue_send*** *(TX_QUEUE *queue_ptr, VOID *source_ptr, ULONG wait_option)*: Envía un mensaje a la cola especificada. El mensaje enviado es copiado a la cola desde el área de memoria donde apunta el argumento *source_ptr*. El argumento *wait_option* indica el tiempo de espera en el caso de que la cola esté llena (TX_WAIT_FOREVER espera, bloqueando la ejecución del hilo, hasta que haya hueco en la cola, y TX_NO_WAIT intenta una vez escribir en la cola).
- ***tx_queue_receive*** *(TX_QUEUE *queue_ptr, VOID *destination_ptr, ULONG wait_option)*: Recupera un mensaje de la cola y lo copia en la zona de memoria donde apunta el argumento *detination_ptr*. Este area de memoria debe ser suficientemente grande para albergar el mensaje copiado. El argumento *wait_option* indica el tiempo de espera en el caso de que la cola esté llena (TX_WAIT_FOREVER espera, bloqueando la ejecución del hilo, hasta que haya un mensaje disponible en la cola, y TX_NO_WAIT intenta una vez leer una vez por la cola).

6 PROGRAMACIÓN DE PERIFÉRICOS

Configuración y programación de periféricos

Este apartado abarca el grueso del TFG. En él se describen los principales periféricos, su configuración y su programación.

6.1 Programación de GPIO

El módulo *I/O Port* implementa una API para controlar los pines de entrada/salida.

A la hora de configurar el proyecto, este periférico no ofrece ninguna opción de configuración en la pestaña *Threads*.

En la pestaña *Pins*, seleccionaremos el pin deseado y lo configuraremos en modo *input* u *output*. Si configuramos el pin en modo *output* podemos elegir el estado inicial del mismo.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.1.1 I/O Port API

Para la programación de este periférico, haremos uso de las siguientes funciones:

- ***init*** *(const ioport_cfg_t * p_cfg)*: Inicializa la configuración de pines durante la inicialización del microcontrolador conforme a lo seleccionado en la configuración del proyecto. No se debe ejecutar en

tiempo de ejecución.

- ***pinCfg*** *(ioport_port_pin_t pin, uint32_t cfg);* Configura un pin. Si en tiempo de ejecución debemos modificar la configuración de un pin, debemos usar esta función. Existe adicionalmente una función ***pinsCfg*** para cambiar la configuración de todos los pines en tiempo de ejecución, análoga a la función init.
- ***pinDirectionSet*** *(ioport_port_pin_t pin, ioport_direction_t direction)*: Cambia la configuración de un pin como entrada o salida en tiempo de ejecución.
- ***pinEventInputRead*** *(ioport_port_pin_t pin, ioport_level_t * p_pin_event)*: Lee un evento asociado al pin especificado. Devuelve su valor.
- ***pinEventOutputWrite*** *(ioport_port_pin_t pin, ioport_level_t pin_value)*: Escribe un evento asociado al pin especificado.
- ***pinEthernetModeCfg*** *(ioport_ethernet_channel_t channel, ioport_ethernet_mode_t mode)*: Configura el modo de la capa física de los canales ethernet.
- ***pinRead*** *(ioport_port_pin_t pin, ioport_level_t * p_pin_value)*: Lee el valor de un pin.
- ***pinWrite*** *(ioport_port_pin_t pin, ioport_level_t level)*: Cambia el valor de un pin al especificado.
- ***portDirectionSet*** *(ioport_port_t port, ioport_size_t direction_values, ioport_size_t mask)*: Cambia la configuración de un puerto como entrada o salida en tiempo de ejecución.
- ***portEventInputRead*** *(ioport_port_t port, ioport_size_t * p_event_data)*: Lee un evento asociado a un Puerto.
- ***portEventInputWrite*** *(ioport_port_t port, ioport_size_t event_data, ioport_size_t mask_value)*: Escribe un evento asociado al pin especificado.
- ***portRead*** *(ioport_port_t port, ioport_size_t * p_port_value)*: Lee el valor de los pines de un puerto.
- ***portWrite*** *(ioport_port_t port, ioport_size_t value, ioport_size_t mask)*: Cambia el valor de varios pines del puerto especificado.
- ***versionGet*** *(ssp_version_t * p_data)*: Devuelve la versión del driver.

6.1.2 Configuración

A diferencia de otros periféricos, este módulo siempre está presente en el hilo HAL. Además, no presenta ninguna opción de configuración en la pestaña *Threads*. En cambio, para configurar los pines, debemos irnos a la pestaña *Pins*. En ella configuraremos la configuración inicial de los pines (como hemos visto en las funciones que conforman la API, se puede modificar esta configuración en tiempo de ejecución). Esta configuración es independiente de los módulos incluidos, aunque estén relacionados. Es decir, podemos configurar por ejemplo un pin (que lo permita) en modo ADC, pero no incluir el módulo ADC en la pestaña *Threads*. Sin embargo, no tiene mucho sentido.

Como cada periférico tiene sus pines asociados, en este punto solo mostraremos la manera de configurar pines en modo entrada o salida digital. Más adelante, explicando la programación de otros periféricos, mostraremos la configuración de los pines necesarios.

Para nuestro ejemplo, necesitaremos un pin de entrada y otro de salida. Configuraremos el Pin P006, asociado al botón integrado S4, como entrada, y el Pin P601, asociado al LED verde incluido en la placa, como salida.

El pin P006, contempla una opción adicional, IRQ. Implica configurar una interrupción y necesita de su propio módulo en la pestaña *Threads*.

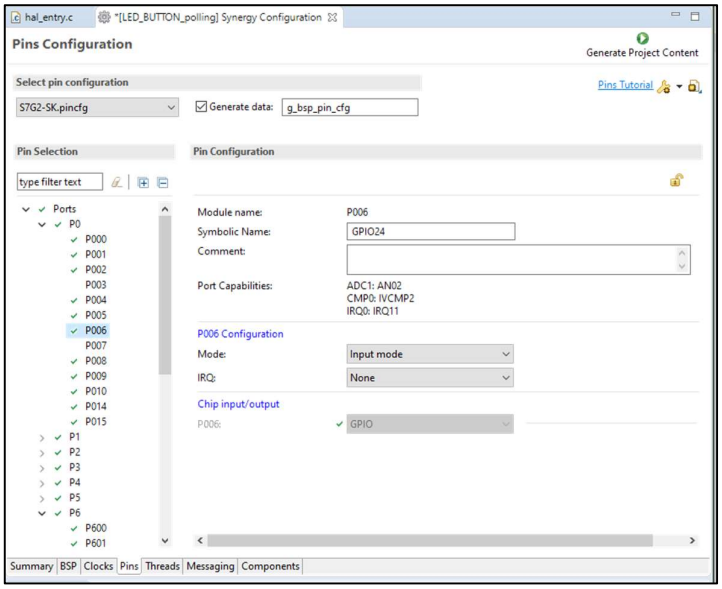


Figura 6-1 Configuración de pin como entrada digital.

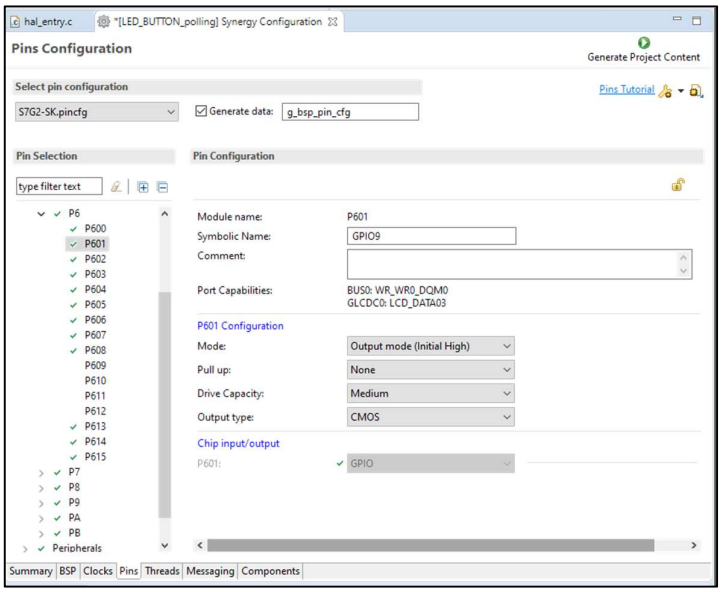


Figura 6-2 Configuración de pin como salida digital.

Los pines de salida los hemos configurado en *Output Mode (Initial High)*, forzando una salida a nivel alto. Esto se debe a que los LEDs integrados en la placa son activos a nivel bajo, y buscamos que estén apagados al ejecutar la aplicación.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.1.3 Ejemplo de programación

Leer el valor del botón S4 y encender el LED 2 cuando el primero esté pulsado.

6.1.3.1 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
void hal_entry(void)
{
    /* TODO: add your own code here */
    ioport_level_t valor_S4;

    while (1){
        g_ioport.p_api->pinRead(IOPORT_PORT_00_PIN_06, &valor_S4);
        if (valor_S4 == IOPORT_LEVEL_LOW){
            g_ioport.p_api->pinWrite(IOPORT_PORT_06_PIN_00,
IOPORT_LEVEL_LOW);
        }
        else g_ioport.p_api->pinWrite(IOPORT_PORT_06_PIN_00,
IOPORT_LEVEL_HIGH);
    }

}
```

6.2 Programación de External IRQ

El módulo *External IRQ* (*interrupt request*) implementa una API para configurar y usar los pines habilitados para la interrupción hardware. Permite configurar múltiples opciones, como el flanco de detección, habilitar un filtro, etc.

A la hora de configurar el proyecto, este periférico debe ser configurado en la pestaña *Threads*, y los pines correspondientes, en la pestaña *Pins*.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.2.1 External IRQ API

Para la programación de este periférico, haremos uso de las siguientes funciones:

- ***open*** *(external_irq_ctrl_t * const p_ctrl, irq_cfg_t const * const p_cfg)*: Abre una instancia del driver en nuestra aplicación, y aplica la configuración elegida en la pestaña *Threads*.
- ***enable*** *(external_irq_ctrl_t * const p_ctrl)*: Habilita la función *callback* de la interrupción asociada, que se llama desde la rutina de servicio de interrupción.
- ***disable*** *(external_irq_ctrl_t * const p_ctrl)*: Deshabilita la función *callback*.
- ***triggerSet*** *(external_irq_ctrl_t * const p_ctrl, external_irq_trigger_t const trigger)*: Establece el evento que genera la interrupción (flanco de subida, flanco de bajada, ambos o bajo nivel).
- ***filterEnable*** *(external_irq_ctrl_t * const p_ctrl)*: Habilita el filtrado.
- ***filterDisable*** *(external_irq_ctrl_t * const p_ctrl)*: Deshabilita el filtrado.
- ***close*** *(external_irq_ctrl_t * const p_ctrl)*: Cierra la instancia del driver.
- ***versionGet*** *(ssp_version_t * p_version)*: Devuelve la versión del driver.

6.2.2 Configuración

Este driver no se encuentra en el hilo común o HAL. Debemos incluirlo en el mismo, o en el hilo en el que vayamos a necesitar esa interrupción hardware.

Para seleccionarlo, debemos ir a la pestaña *Threads* y buscar el driver en el menú que aparece al pulsar el botón *New Stack*. Debemos ir a *Driver>Input>External IRQ Driver on r_icu.*

Una vez hayamos elegido la instancia del driver, debemos pulsar en el módulo, y nos vamos a la pestaña *Properties*. En ella encontraremos las opciones de configuración disponibles.

Las propiedades que debemos modificar son el canal (hay un canal asociado a cada uno de los 16 pines habilitados con interrupciones hardware), la prioridad de la interrupción, el evento que genera la interrupción, el nombre de la función *callback.*

Opcionalmente, podemos habilitar el filtrado digital y la frecuencia de muestreo del mismo. También podemos seleccionar la opción que habilita el módulo directamente tras la inicialización del mismo, quitando la necesidad de tener que hacerlo nosotros al inicio del hilo, si pensamos tener la interrupción habilitada desde el inicio de la ejecución.

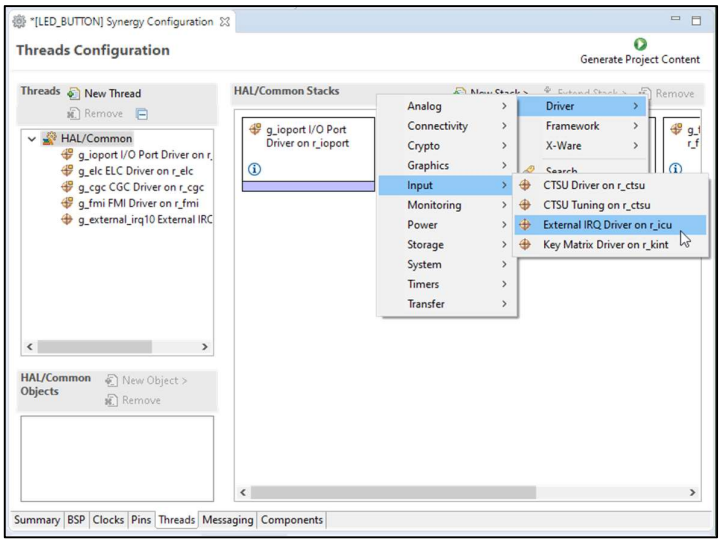


Figura 6-3 Selección del driver en la pestaña *Threads.*

Una vez tengamos configurado el driver, debemos ir a la pestaña *Pins* y habilitar los pines necesarios.

Para buscar los pines asociados a un IRQ, podemos buscar el periférico (IRQ) en el árbol desplegable.

A la hora de habilitar los pines que queremos, encontraremos un símbolo de error al seleccionar los pines con función de *output*. Tal y como indica en la descripción, debemos ir al puerto y seleccionar desde allí el modo IRQ. En la Figura 6-5 mostramos este error.

Una vez estemos en la configuración del pin correspondiente, habilitamos el modo IRQ seleccionando IRQXX en el menú desplegable.

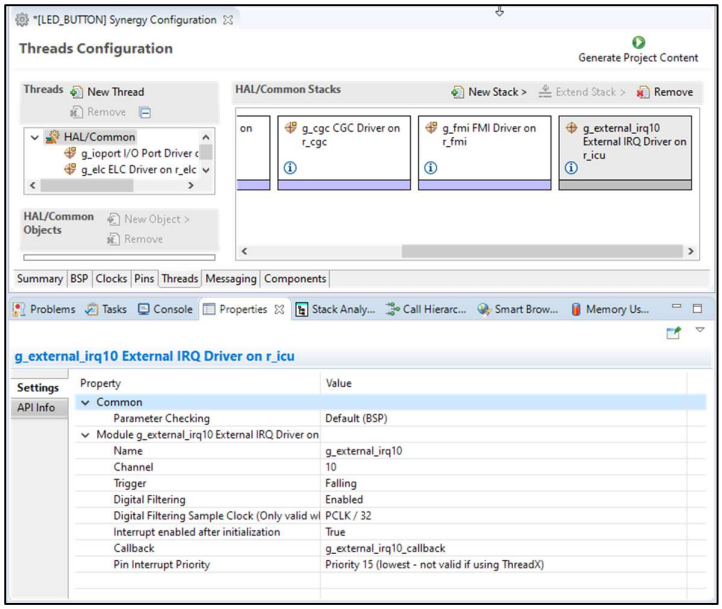


Figura 6-4 Propiedades del driver *External IRQ*

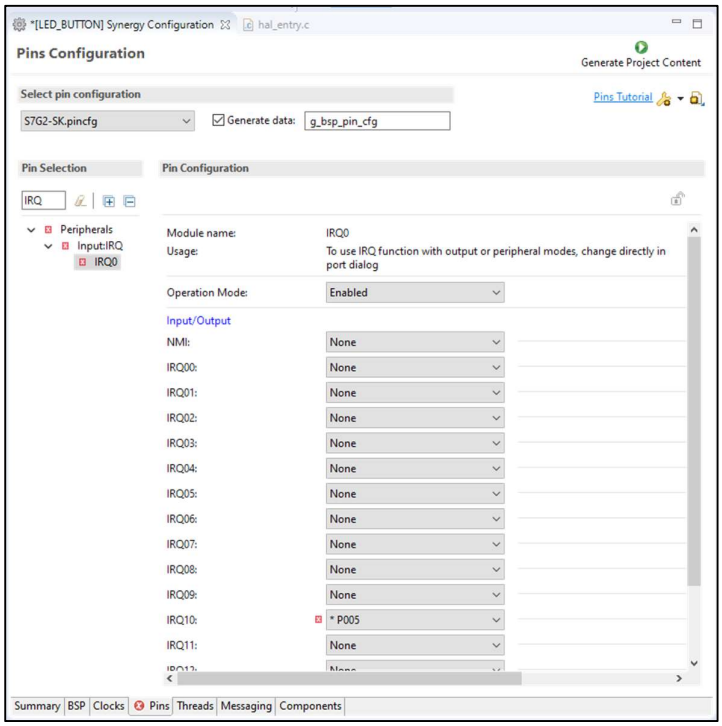


Figura 6-5 Error de configuración del canal 10 del módulo IRQ.

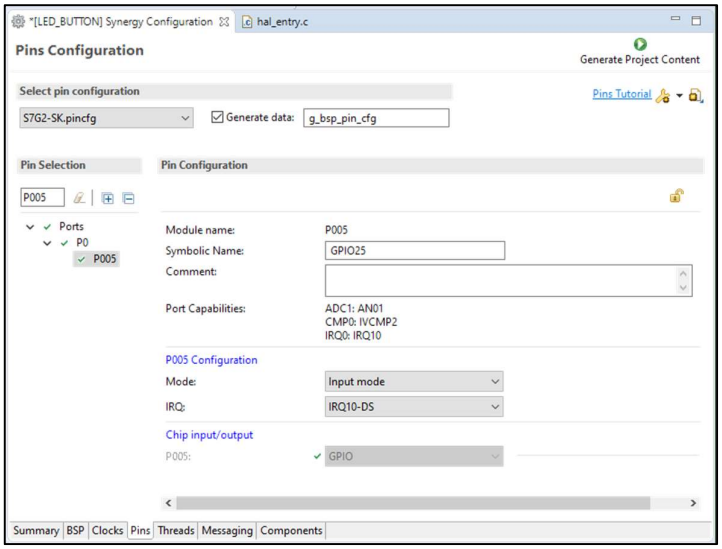


Figura 6-6 Configuración correcta del canal 10 del módulo IRQ.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.2.3 Ejemplo de programación

Leer el valor del botón S4 y encender el LED 2 cuando el primero esté pulsado, usando interrupciones hardware.

6.2.3.1 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
#include <stdbool.h>

volatile bool flag_S4 = false;

void hal_entry(void)
{
    /* TODO: add your own code here */

    ioport_level_t valor_LED=0;
    g_external_irq11.p_api->open(g_external_irq11.p_ctrl,
g_external_irq11.p_cfg); //inicializamos la interrupción HW.
    g_external_irq11.p_api->enable(g_external_irq11.p_ctrl);//Podemos
eliminar esta línea de código si hemos seleccionado la habilitación
automática tras la inicialización del microcontrolador.

    while (1){
        //esperamos interrupción HW
        if (flag_S4){
            flag_S4 = false;
            g_ioport.p_api->pinRead(IOPORT_PORT_06_PIN_00, &valor_LED);
            if (valor_LED == IOPORT_LEVEL_HIGH){    //si el LED estaba
encendido, lo apagamos y viceversa
                g_ioport.p_api->pinWrite(IOPORT_PORT_06_PIN_00,
IOPORT_LEVEL_LOW);
```

```
            }
            else g_ioport.p_api->pinWrite(IOPORT_PORT_06_PIN_00,
IOPORT_LEVEL_HIGH);
        }
    }
}
void g_external_irq11_callback (external_irq_callback_args_t *p_args){
//función llamada por el ISR tras recibir interrupción asociada al botón S4
    flag_S4 = true;
}
```

Como la función *callback* se llama desde una rutina de interrupción, debe ser lo más escueta posible para no afectar el rendimiento del sistema. Al ser un programa simple, la merma de rendimiento sería nula, pero es una buena práctica realizar lo justo y necesario en las interrupciones.

6.3 Programación de Timer (GPT/AGT)

Nuestro microcontrolador distingue entre dos tipos de temporizadores, *GPT* (*General PWM Timer*) y AGT (*Asynchronous General purpose Timer*).

Ambos módulos implementan una API para configurar y usar temporizadores en nuestra aplicación. Se puede configurar también una interrupción para responder a los eventos generados por los temporizadores.

Las funcionalidades de ambos temporizadores son también las mismas. Cuando se configura un temporizador con un determinado periodo, pueden ocurrir varias cosas:

1. Que se genere una interrupción y el ISR llame a la función *callback*, si se ha configurado de tal manera.
2. Que el driver conmute un pin.
3. Que se genere una transferencia de datos usando el driver DMAC o DCT (a configurar en una instancia de dichos drivers).
4. Que provoque el inicio de otro periférico (a configurar en el módulo ELC).

Las diferencias entre ambos temporizadores son las siguientes:

1. Los temporizadores GPT usan el reloj PCLKD, mientras que los AGT pueden hacer uso de varios relojes (PCLKB, LOCO y Fsub).
2. Cada temporizador GPT contiene dos canales, asociados a dos pines distintos.
3. Los temporizadores GPT tienen una resolución de 32 bits, mientras que los AGT tienen una resolución de 16 bits.
4. Existen 14 temporizadores GPT (28 contando con los dos canales de cada uno), mientras que los AGT tienen 2.
5. Los dos temporizadores AGT se pueden configurar en cascada, generando un único temporizador de 32 bits.
6. Los temporizadores AGT pueden despertar la CPU cuando se encuentre en modos de bajo consumo, si usa los relojes LOCO o Fsub.

Como norma general, se usarán temporizadores GPT.

A la hora de configurar el proyecto, este periférico debe ser configurado en la pestaña *Threads*, y los pines correspondientes, en la pestaña *Pins*. En este caso, debemos adicionalmente configurar los relojes en la pestaña *Clocks*.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

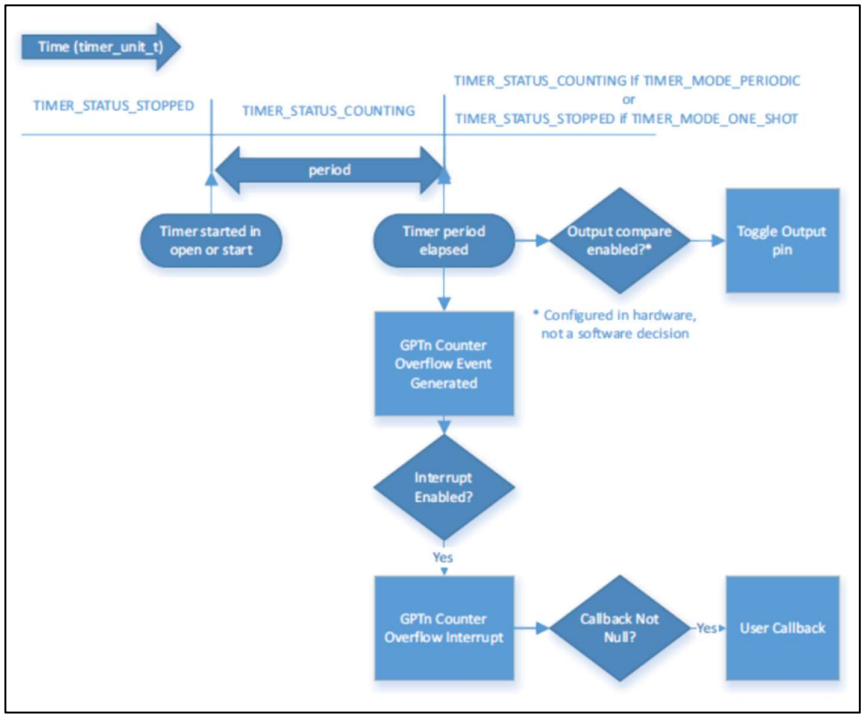


Figura 6-7 Esquema de funcionamiento del temporizador.

6.3.1 Timer API

Ambos tipos de temporizador comparten API. Para la programación de este periférico, haremos uso de las siguientes funciones:

- ***open*** *(timer_ctrl_t * const p_ctrl, timer_cfg_t const * const p_cfg)*: Abre una instancia del driver en nuestra aplicación, y aplica la configuración elegida en la pestaña *Threads*.
- ***stop*** *(timer_ctrl_t * const p_ctrl)*: Para de contar.
- ***start*** *(timer_ctrl_t * const p_ctrl)*: Empieza a contar.
- ***reset*** *(timer_ctrl_t * const p_ctrl)*: Resetea el contador.
- ***counterGet*** *(timer_ctrl_t * const p_ctrl, timer_size_t * const p_value)*: Guarda el valor actual del contador en *p_value*.
- ***periodSet*** *(timer_ctrl_t * const p_ctrl, timer_size_t const period, timer_unit_t const unit)*: Establece el tiempo hasta que el temporizador expira.
- ***dutyCycleSet*** *(timer_ctrl_t * const p_ctrl, timer_size_t const duty_cycle, timer_pwm_unit_t const duty_cycle_unit, uint8_t const pin)*: Establece el duty cycle.
- ***infoGet*** *(timer_ctrl_t * const p_ctrl, timer_info_t * const p_info)*: Guarda información del temporizador en el puntero a estructura p_info. Como información más importante, guarda el tiempo restante para que el temporizador expire (en forma de "cuentas de reloj").
- ***close*** *(timer_ctrl_t * const p_ctrl)*: Cierra la instancia del driver y permite ser reconfigurado.

- ***versionGet*** *(ssp_version_t * p_version)*: Devuelve la versión del driver.

6.3.2 Configuración

Este driver no se encuentra en el hilo común o HAL. Debemos incluirlo en el mismo, o en el hilo en el que vayamos a necesitar un temporizador.

Para seleccionarlo, debemos ir a la pestaña *Threads* y buscar el driver en el menú que aparece al pulsar el botón *New Stack*. Debemos ir a *Driver>Timers>Timer Driver on r_gpt*.

Una vez hayamos elegido la instancia del driver, debemos pulsar en el módulo, y nos vamos a la pestaña *Properties*. En ella encontraremos las opciones de configuración disponibles.

Las principales propiedades que debemos modificar son el canal (hay un canal asociado a cada uno de los 14 temporizadores GPT), el modo de funcionamiento (un disparo, periódico o PWM), el periodo del timer y el valor del duty cycle si hemos elegido el modo PWM.

Adicionalmente, si queremos obtener la señal del temporizador por uno de los pines asociados al mismo (por ejemplo, una señal PWM), podemos habilitar esta opción, para cualquiera de los dos pines asociados al mismo temporizador.

Por último, podemos también configurar una interrupción cuando el registro del temporizador desborde.

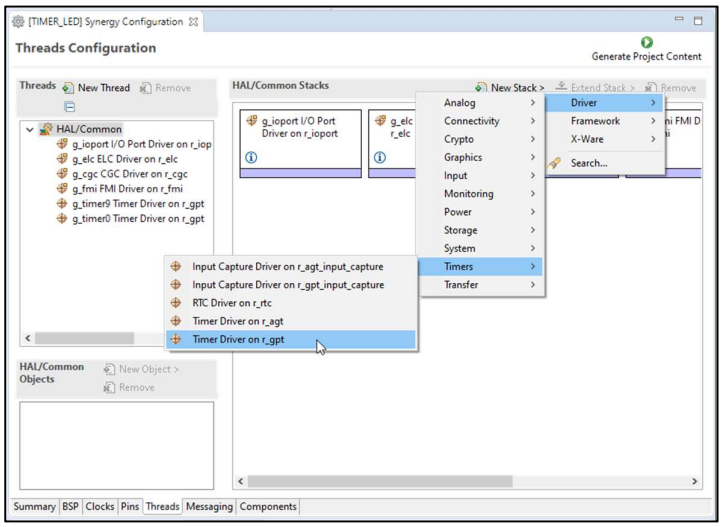


Figura 6-8 Selección del driver en la pestaña *Threads.*

Este driver ofrece una gran sencillez a la hora de configurar un timer, debido a que permite configurar rápidamente un temporizador del periodo que queramos indicando únicamente el valor del mismo, y sus unidades, con una precisión de nanosegundos.

Respecto al funcionamiento en modo PWM, podemos seleccionar de igual manera el valor del periodo, y seleccionar el duty cycle como un porcentaje, con 3 decimales de precisión (usando la opción *Unit Percent x 1000*).

En ambos casos se permite configurar el periodo o el duty cycle como el número de *ticks* de reloj que debe contar el driver. En ambos casos, pero especialmente en este, se debe conocer de antemano la frecuencia del reloj PCLKD en la pestaña *Clocks* para poder calcular dichos valores.

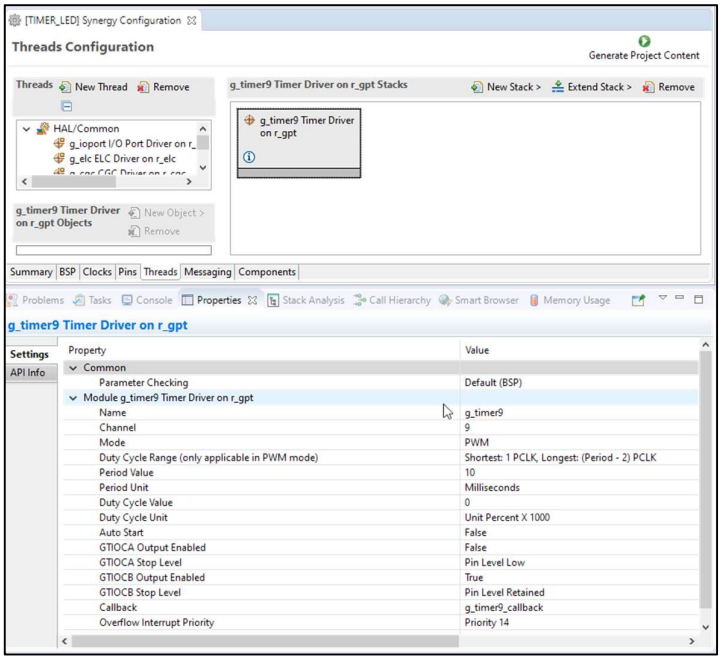


Figura 6-9 Propiedades del driver *Timer GPT*

Una vez tengamos configurado el driver, debemos ir a la pestaña *Pins* y habilitar los pines que sean necesarios.

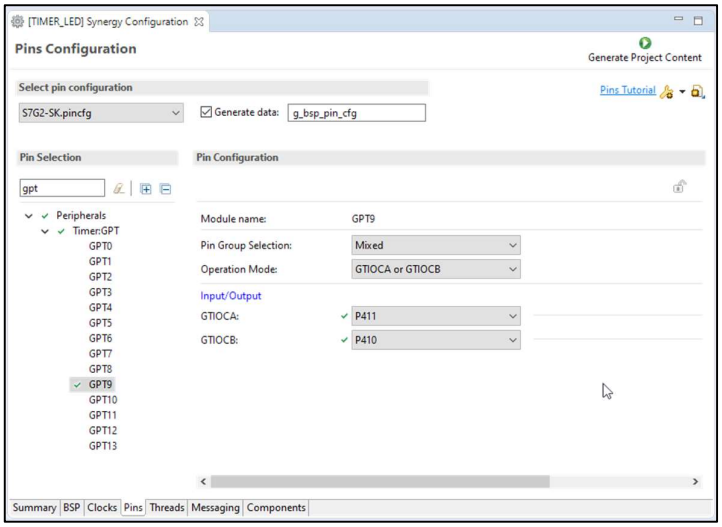


Figura 6-10 Configuración de la salida del temporizador GPT9.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.3.3 Ejemplo de programación

Establecer otro temporizador en modo PWM (asociado a un pin que lo permita) y que genere una señal senoidal, con un periodo de T segundos.

6.3.3.1 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
#include <math.h>
#include <stdbool.h>

#define pi 3.14159265359

volatile bool flag_PWM = false;
volatile bool flag_timer = false;

timer_size_t duty_cycle[100];

void hal_entry(void)
{
    /* TODO: add your own code here */
    int j=0;
    int T=4;
    double w = 2*pi/T;

    ioport_level_t valor_LED=0;

    for (j = 0;j < 100;j++){
        duty_cycle[j] = (timer_size_t) 50000.0 * cos(w*T/100*j) + 50000;
    }
    j = 0;

    g_timer9.p_api->open(g_timer9.p_ctrl, g_timer9.p_cfg);
    g_timer0.p_api->open(g_timer0.p_ctrl, g_timer0.p_cfg);
    g_timer0.p_api->periodSet(g_timer0.p_ctrl, (timer_size_t) T/2.0*1000,
TIMER_UNIT_PERIOD_MSEC);

    g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl, 0,
TIMER_PWM_UNIT_PERCENT_X_1000, 1); //El último argumento indica el pin de
salida, 1 para GTIOCB
    g_timer9.p_api->start(g_timer9.p_ctrl); //nos podemos ahorrar esta linea
en la configuración del driver
    g_timer0.p_api->start(g_timer0.p_ctrl); //nos podemos ahorrar esta linea
en la configuración del driver

    while(1){
        if(flag_PWM){
            flag_PWM = false;
            j++; //contamos cada 10ms
            if (j > (100*T-1)) {

                j=0; //si hemos completado un periodo de nuestra senoide,
reseteamos el contador. como actualizamos 100 veces en todo el periodo de la
senoide, 100 (veces contamos 10ms en un segundo), por el periodo T en
segundos.
            }
            if (j%T){ //si j es divisible entre T, toca actualizar el duty
cycle
                g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl, (timer_size_t)
duty_cycle[j/T] , TIMER_PWM_UNIT_PERCENT_X_1000, 1);
            }
        }

        if (flag_timer){ //cada segundo, conmutamos los leds
            flag_timer = false;
            g_ioport.p_api->pinRead(IOPORT_PORT_06_PIN_01, &valor_LED);
```

```
            if (valor_LED == IOPORT_LEVEL_HIGH){    //si el LED estaba
encendido, lo apagamos y viceversa
                g_ioport.p_api->pinWrite(IOPORT_PORT_06_PIN_01,
IOPORT_LEVEL_LOW);
            }
            else g_ioport.p_api->pinWrite(IOPORT_PORT_06_PIN_01,
IOPORT_LEVEL_HIGH);
        }
    }
}

void g_timer9_callback(timer_callback_args_t *p_args){
    flag_PWM = true;
}

void g_timer0_callback(timer_callback_args_t *p_args){
    flag_timer = true;
}
```

6.4 Programación de ADC

El módulo *ADC* implementa una API para la conversión de señales analógicas a digitales. El driver soporta ADCs de hasta 16 bits de resolución (los incluidos en nuestro kit tienen una resolución de 12 bits).

En total, el kit SK-S7G2 trae dos conversores, con 13 y 12 canales cada uno.

El driver permite varios modos de operación (escaneo simple, escaneo continuo y escaneo grupal). Permite también programar un amplificador de ganancia (PGA) para la señal de entrada.

En modo simple, uno o más canales son escaneados por cada disparo (HW o SW). Se utiliza una máscara en la configuración para indicar los canales leídos. Se genera una interrupción cuando se ha completado la conversión de todos los canales (se debe configurar la función *callback*).

En el modo continuo, el driver convierte secuencialmente las señales analógicas de los canales seleccionados, de manera continua (en orden ascendente según el número del canal). Requiere un único disparo para empezar a escanear. No usa ninguna interrupción. Es aconsejable usar la función *scanStatusGet* para determinar cuándo hay datos disponibles.

A la hora de configurar el proyecto, este periférico debe ser configurado en la pestaña *Threads*, y los pines correspondientes, en la pestaña *Pins*.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.4.1 ADC API

Para la programación de este periférico, haremos uso de las siguientes funciones:

- ***open*** *(adc_ctrl_t* const p_ctrl, adc_cfg_t const * const p_cfg)*: Abre una instancia del driver en nuestra aplicación, y aplica la configuración elegida en la pestaña *Threads*.
- ***scanCfg*** *(adc_ctrl_t* const p_ctrl, adc_channel_cfg_t const * const p_channel_cfg)*: Configura el periférico (canales, grupos, modo de escaneo, disparo del escaneo (HW o SW), etc.) para la unidad

inicializada en *open*.

- ***scanStart*** *(adc_ctrl_t* const p_ctrl)*: Comienza el escaneo (en caso de disparo por SW) o habilita el disparo por HW.
- ***scanStop*** *(adc_ctrl_t* const p_ctrl)*: Termina el escaneo (en caso de disparo por SW) o deshabilita el disparo por HW.
- ***scanStatusGet*** *(adc_ctrl_t* const p_ctrl)*: Comprueba el estado del escaneo.
- ***read*** *(adc_ctrl_t* const p_ctrl, adc_register_t const reg_id, uint16_t * const p_data)*: Guarda el resultado de la conversión en el puntero *p_data*.
- ***read32*** *(adc_ctrl_t* const p_ctrl, adc_register_t const reg_id, uint32_t * const p_data)*: Igual que *read*, pero lo guarda en una variable de 32 bits.
- ***calibrate*** *(adc_ctrl_t* const p_ctrl, void * const p_extend)*: Calibra el ADC o el PGA asociado. El driver puede requerir argumentos específicos a través del puntero *p_extend*.
- ***offsetSet*** *(adc_ctrl_t* const p_ctrl, adc_register_t const reg_id, int32_t const offset)*: Establece el offset del PGA si está configurado en modo diferencial.
- ***close*** *(adc_ctrl_t * const p_ctrl)*: Cierra la instancia del driver.
- ***infoGet*** *(adc_ctrl_t * const p_ctrl, adc_info_t * const p_adc_info)*: Devuelve las direcciones del registro de datos del primer canal y el número total de bytes.
- ***versionGet*** *(ssp_version_t * p_version)*: Devuelve la versión del driver.

6.4.2 Configuración

Este driver no se encuentra en el hilo común o HAL. Debemos incluirlo en el mismo, o en el hilo en el que vayamos a necesitar un temporizador.

Para seleccionarlo, debemos ir a la pestaña *Threads* y buscar el driver en el menú que aparece al pulsar el botón *New Stack*. Debemos ir a *Driver>Analog>ADC Driver on r_adc*.

Una vez hayamos elegido la instancia del driver, debemos pulsar en el módulo, y nos vamos a la pestaña *Properties*. En ella encontraremos las opciones de configuración disponibles.

Las principales propiedades a configurar son la unidad del ADC a usar, la resolución (hasta 12 bits), el modo de funcionamiento y la máscara de canales a leer. Si seleccionamos el modo simple, debemos configurar la función callback que será llamada tras la conversión por el ISR.

Opcionalmente, podemos habilitar la calibración del ADC al ejecutar la función *open* (recomendado) y configurar el amplificador de ganancia programable.

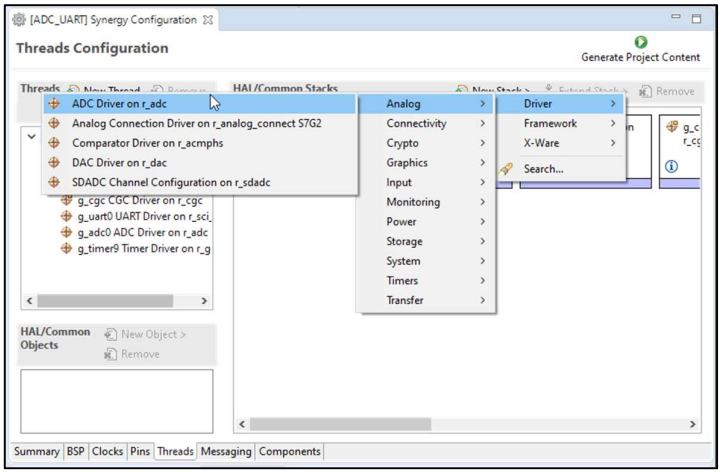


Figura 6-11 Selección del driver en la pestaña *Threads*.

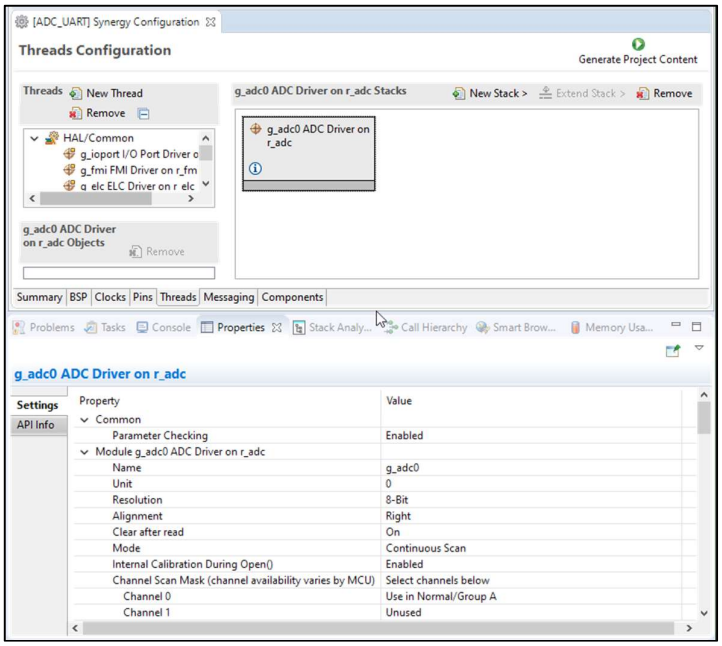


Figura 6-12 Propiedades del driver *ADC*.

Una vez tengamos configurado el driver, debemos ir a la pestaña *Pins* y habilitar los pines necesarios.

Para buscar los pines asociados al ADC, podemos buscar el periférico (ADC) en el árbol desplegable.

Una vez estemos en la configuración de la unidad correspondiente (ADC0 o ADC1), habilitamos el modo de operación seleccionando *Custom* en el menú desplegable, y seleccionamos los pines que hemos configurado en la pestaña *Threads*.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

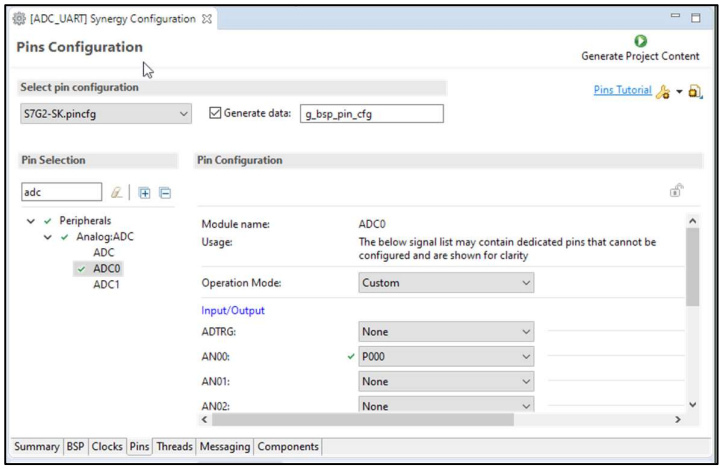


Figura 6-13 Configuración del canal 0 de la unidad ADC0.

6.4.3 Ejemplo de programación

Conectar un potenciómetro al canal 0 de la unidad 0 del ADC y generar una señal PWM proporcional al valor leído para iluminar un LED conectado al pin P410.

6.4.3.1 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"

void hal_entry(void)
{
    /* TODO: add your own code here */
    volatile ssp_err_t scan_status;
    uint16_t dato_convertido;

    g_timer9.p_api->open(g_timer9.p_ctrl, g_timer9.p_cfg);
//inicializamos el timer que usaremos para el PWM.
    g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl, 0, TIMER_PWM_UNIT_PERCENT,
1);    //Forzamos duty cycle nulo

    g_adc0.p_api->open(g_adc0.p_ctrl, g_adc0.p_cfg);
//abrimos ADC0 (8-bit)
    g_adc0.p_api->scanCfg(g_adc0.p_ctrl, g_adc0.p_channel_cfg);
//configuramos el modo de escaneo, establecido previamente en el BSP. Modo
continuo en este caso
    g_adc0.p_api->scanStart(g_adc0.p_ctrl);
//escaneamos (SW trigger)

    while(1){

        g_adc0.p_api->read(g_adc0.p_ctrl, ADC_REG_CHANNEL_0,
&dato_convertido);        //leemos y volcamos dato a una variable
        g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl, (timer_size_t)
(dato_convertido*100/255), TIMER_PWM_UNIT_PERCENT, 1); //modificamos el duty
cycle según el valor leído.

    }
}
```

6.5 Programación de DAC

El módulo *DAC* implementa una API para la conversión de señales digitales a analógicas.

En total, el kit SK-S7G2 trae dos conversores DAC, con una resolución de 12 bits, por lo que podremos generar señales con un 4096 valores entre 0V y Vcc (3,3V).

El driver permite configurar un amplificador a la salida del conversor y un multiplicador de tensión.

A la hora de configurar el proyecto, este periférico debe ser configurado en la pestaña *Threads*, y los pines correspondientes, en la pestaña *Pins*.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.5.1 DAC API

Para la programación de este periférico, haremos uso de las siguientes funciones:

- ***open*** *(dac_ctrl_t* const p_ctrl, dac_cfg_t const * const p_cfg)*: Abre una instancia del driver en nuestra aplicación, y aplica la configuración elegida en la pestaña *Threads*.
- ***start*** *(dac_ctrl_t* const p_ctrl)*: Activa el conversor si no está activo.
- ***stop*** *(dac_ctrl_t* const p_ctrl)*: Para el conversor si está activo.
- ***write*** *(dac_ctrl_t* const p_ctrl, dac_size_t value)*: Escribe un valor de tensión en el pin de salida.
- ***close*** *(dac_ctrl_t * const p_ctrl)*: Cierra la instancia del driver.
- ***infoGet*** *(dac_info_t * const p_info)*: Guarda la información sobre la resolución del conversor en el puntero proporcionado.
- ***versionGet*** *(ssp_version_t * p_version)*: Devuelve la versión del driver.

6.5.2 Configuración

Este driver no se encuentra en el hilo común o HAL. Debemos incluirlo en el mismo, o en el hilo en el que vayamos a necesitar un temporizador.

Para seleccionarlo, debemos ir a la pestaña *Threads* y buscar el driver en el menú que aparece al pulsar el botón *New Stack*. Debemos ir a *Driver>Analog>DAC Driver on r_dac*.

Una vez hayamos elegido la instancia del driver, debemos pulsar en el módulo, y nos vamos a la pestaña *Properties*. En ella encontraremos las opciones de configuración disponibles.

Este driver proporciona pocas opciones de configuración. Debemos elegir el canal a usar (de los dos posibles) y elegir la sincronización con el ADC para evitar interferencias (si nuestra aplicación va a hacer funcionar ambos conversores) y seleccionar el amplificador o el multiplicador de tensión.

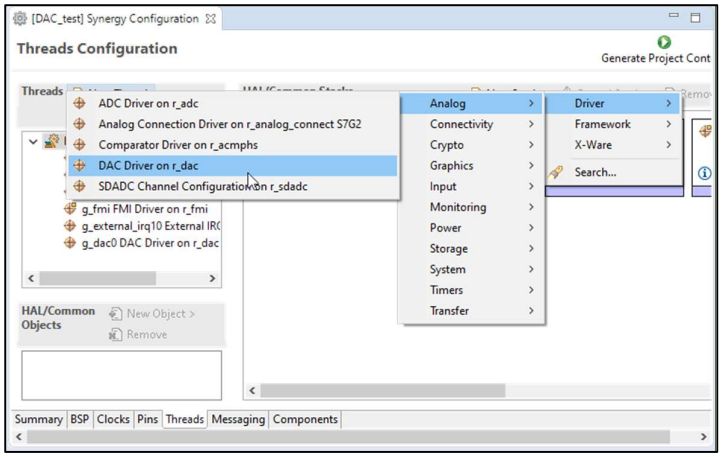


Figura 6-14 Selección del driver en la pestaña *Threads*.

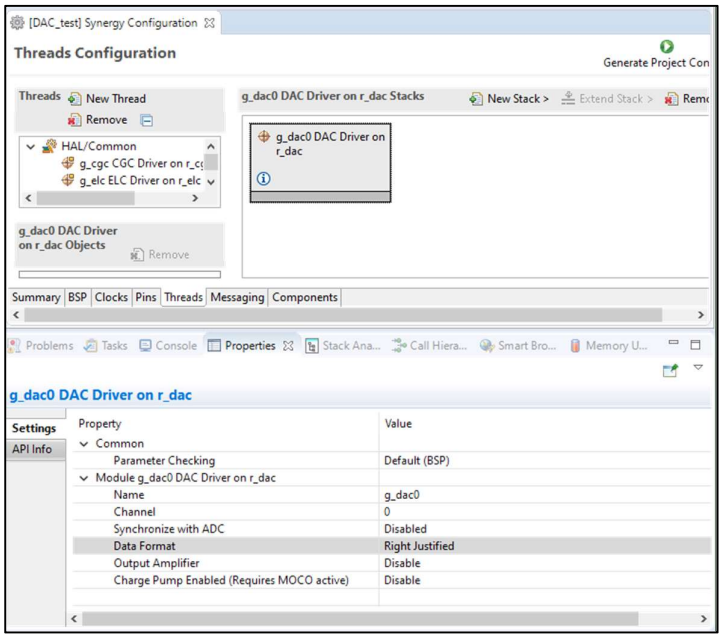


Figura 6-15 Propiedades del driver *DAC*.

Una vez tengamos configurado el driver, debemos ir a la pestaña *Pins* y habilitar los pines necesarios.

Para buscar los pines asociados al DAC, podemos buscar el periférico (DAC) en el árbol desplegable.

Una vez estemos en la configuración de la unidad correspondiente (DAC120 o DAC121), habilitamos el modo de operación seleccionando *Enabled* en el menú desplegable, y seleccionamos el pin asociado a ese conversor.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

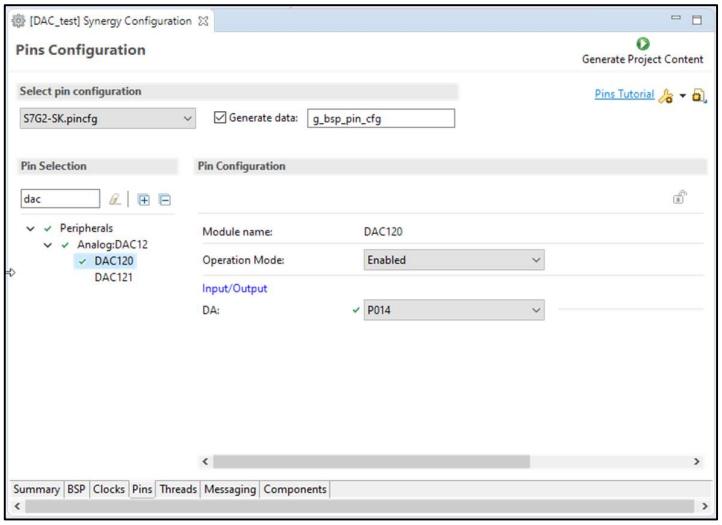


Figura 6-16 Configuración del canal 0 de la unidad DAC120.

6.5.3 Ejemplo de programación

Escribir un programa que modifique el valor de tensión en el pin P014 cuando se pulse el pulsador S5.

6.5.3.1 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
#include "stdbool.h"
dac_size_t valor_dac[]={0, 410, 819, 1229, 1638, 2048, 2458, 2867, 3277,
3686, 4095}; //12 bits

bool flag;
ssp_err_t status_dac;
int i=0;
void hal_entry(void)
{
    /* TODO: add your own code here */
    flag=false;
    g_external_irq10.p_api->open(g_external_irq10.p_ctrl,
g_external_irq10.p_cfg);  //abrimos modulo interrupción HW para botón S5
    status_dac=g_dac0.p_api->open(g_dac0.p_ctrl, g_dac0.p_cfg); //abrimos
modulo DAC para pin P014
    status_dac=g_dac0.p_api->write(g_dac0.p_ctrl, valor_dac[i]);
    status_dac=g_dac0.p_api->start(g_dac0.p_ctrl);
    while(1){
        R_BSP_SoftwareDelay(10, BSP_DELAY_UNITS_MILLISECONDS); //espera de
0.01 segundos
        if (flag==true){
            flag=false;
            status_dac=g_dac0.p_api->write(g_dac0.p_ctrl, valor_dac[i]);
        }
    }

}
void g_external_irq10_callback(external_irq_callback_args_t *p_args){
    flag=true;
    i++;
    if (i>10){
```

```
            i=0;
        }
}
```

6.6 Programación de UART

El módulo *UART* implementa una API para configurar los 10 periféricos *SCI (Serial Communications Interface)* que se encuentran en nuestro kit. Se puede definir una función *callback* desde la cual podemos manejar el *handshake* y la operación de datos.

El módulo soporta el protocolo UART estándar y sus principales características son las siguientes:

- Comunicación full-dúplex.
- Comunicación simultánea por distintos canales.
- Transmisión y recepción de datos impulsados mediante interrupciones.
- Posibilidad de modificar el baudrate en tiempo de operación.
- Control del flujo de datos mediante pines RTS (*Ready To Send*) / CTS (*Clear To Send*). *(*)*
- Posibilidad de abortar operaciones de lectura y escritura.

A la hora de configurar el proyecto, este periférico debe ser configurado en la pestaña *Threads*, y los pines correspondientes, en la pestaña *Pins*.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

(*) El driver solo puede controlar una de las dos señales, al estar multiplexadas en el mismo pin. Para solucionar este inconveniente, existe la posibilidad de configurar un pin GPIO como pin RTS, y manejar la operación del mismo mediante interrupciones.

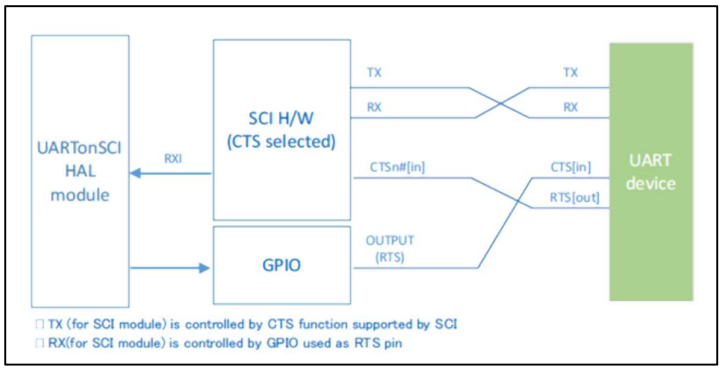


Figura 6-17 Control de flujo de datos con un pin GPIO.

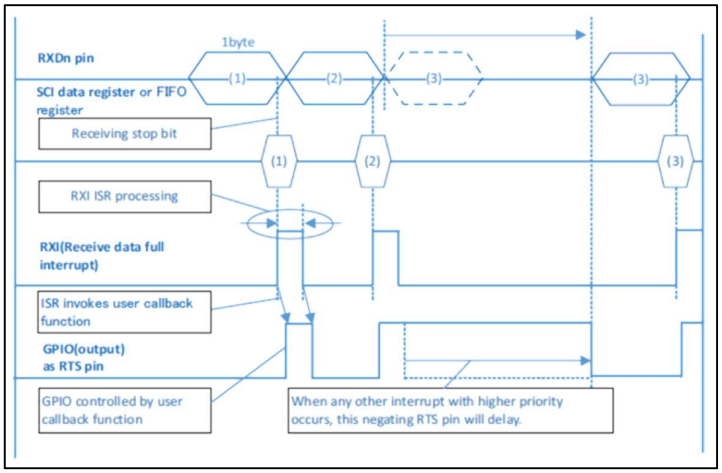


Figura 6-18 Control de flujo de datos con un pin GPIO.

6.6.1 UART API

Para la programación de este periférico, haremos uso de las siguientes funciones:

- ***open*** *(uart_ctrl_t* const p_ctrl, uart_cfg_t const * const p_cfg)*: Abre una instancia del driver en nuestra aplicación, y aplica la configuración elegida en la pestaña *Threads*.
- ***read*** *(uart_ctrl_t * const p_ctrl, uint8_t const * const p_dest, uint32_t const bytes)*: Lee los datos recibidos por la UART. Los bytes recibidos se almacenan en el buffer de lectura. Cuando se lee el número de caracteres esperado, se genera una interrupción con el evento *UART_EVENT_RX_COMPLETE*. Si no se usa *read*, los datos se reciben mediante interrupción, tratada en la función *callback*, con el evento *UART_EVENT_RX_CHAR*, para cada byte recibido.
- ***write*** *(uart_ctrl_t * const p_ctrl, uint8_t const * const p_src, uint32_t const bytes)*: Escribe por la UART. El buffer de escritura se usa hasta terminar la transmisión. Cuando ha terminado (todos los bytes se han transmitido), se genera una interrupción con el evento *UART_EVENT_TX_COMPLETE*.
- ***baudSet*** *(uart_ctrl_t * const p_ctrl, uint32 _t const baudrate)*: Establece la tasa de baudios.
- ***infoGet*** *(uart _ctrl_t * const p_ctrl, uart _info_t * const p_info)*:
- ***close*** *(uart_ctrl_t * const p_ctrl)*: Cierra la instancia del driver.
- ***versionGet*** *(ssp_version_t * p_version)*: Devuelve la versión del driver.

6.6.2 Configuración

Este driver no se encuentra en el hilo común o HAL. Debemos incluirlo en el mismo, o en el hilo en el que vayamos a necesitar este periférico.

Para seleccionarlo, debemos ir a la pestaña *Threads* y buscar el driver en el menú que aparece al pulsar el botón *New Stack*. Debemos ir a *Driver>Connectivity>UART Driver on r_sci*.

Una vez hayamos elegido la instancia del driver, debemos pulsar en el módulo, y nos vamos a la pestaña *Properties*. En ella encontraremos las opciones de configuración disponibles.

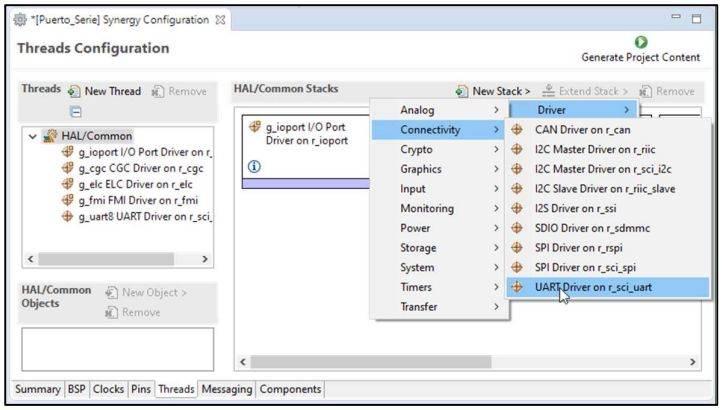


Figura 6-19 Selección del driver en la pestaña *Threads*

Al seleccionar el driver, observamos que este, hace uso de otros drivers para la transmisión y recepción de datos. Por defecto, solo incluye el driver para la transmisión. Para añadir el driver para la recepción de datos, pulsamos en *Add DTC Driver for Reception* y seleccionamos la única opción disponible.

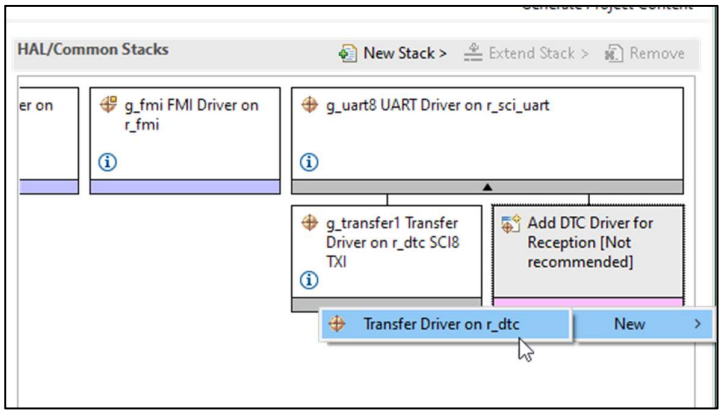


Figura 6-20 Selección del driver para la recepción de datos.

Las principales propiedades para poder usar este periférico son el canal y el *baudrate*. El resto de opciones pueden dejarse por defecto.

Si queremos controlar el flujo de datos debemos habilitar la operación RTS, seleccionar la opción CTS y definir la función *callback* a través de la cual debemos controlar su operación usando un pin externo.

Una vez tengamos configurado el driver, debemos ir a la pestaña *Pins* y habilitar los pines necesarios.

Para buscar los pines asociados a un canal de la UART, podemos buscar el periférico (SCI) en el árbol desplegable.

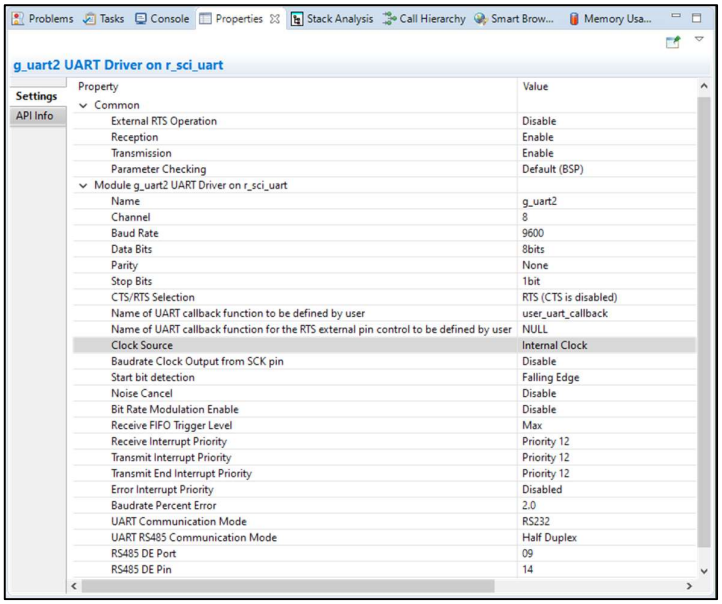


Figura 6-21 Propiedades del driver *UART*

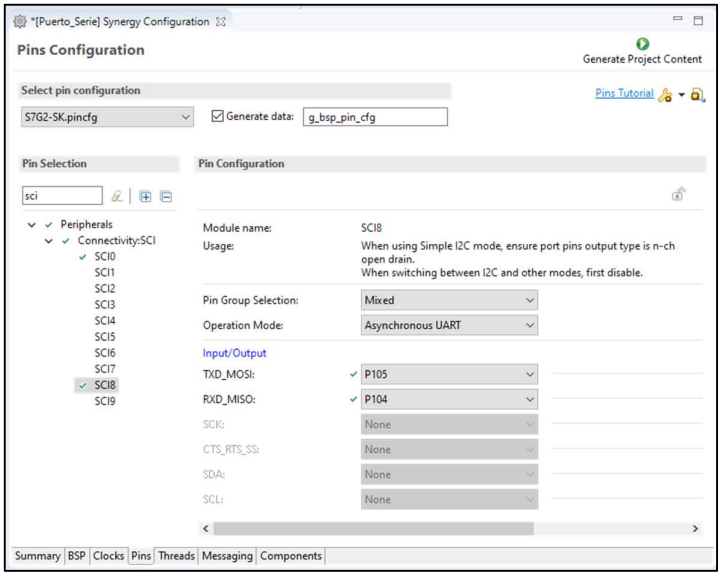


Figura 6-22 Configuración del módulo UART

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.6.3 Ejemplo de programación

Escribir un programa que reciba datos por UART y transmita los caracteres leídos. (necesitaremos de un conversor de puerto serie a USB, como un FT232RL, para conectarlo al ordenador)

6.6.3.1 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
#include "string.h"
#include <stdbool.h>

bool flag_rx_complete = false;
char RX_Buffer[64];
char TX_Buffer[] = "Recibido: ";
int RX_Buffer_index = 0;

void hal_entry(void)
{
    /* TODO: add your own code here */

    g_uart8.p_api->open(g_uart8.p_ctrl, g_uart8.p_cfg);
    g_uart8.p_api->baudSet(g_uart8.p_ctrl, (uint32_t)9600);
    g_uart8.p_api->write(g_uart8.p_ctrl,(const uint8_t *)"\033[0;32mUART8
OPEN\r\n\033[0;39m", strlen("\033[0;32mUART8 OPEN\r\n\033[0;39m"));

    while(1){
        if(flag_rx_complete){
            strcpy(TX_Buffer, "Recibido: ");
            strcat(TX_Buffer, RX_Buffer);
            strcat(TX_Buffer, "\r\n");
            g_uart8.p_api->write(g_uart8.p_ctrl,(const uint8_t *)TX_Buffer,
strlen(TX_Buffer));

            flag_rx_complete = false;
        }

        R_BSP_SoftwareDelay(50, BSP_DELAY_UNITS_MILLISECONDS); //bsp_delay
    }
}


void user_uart_callback(uart_callback_args_t *p_args){ //como no tenemos un
límite de caracteres a leer, usamos el evento RX_CHAR para leer los datos
recibidos.
    if (p_args->channel == 8){ //si pasa algo en nuestra UART
        switch (p_args->event){
            case UART_EVENT_RX_CHAR: //y recibimos un carácter
                if (p_args->data != 10 && p_args->data != 8) { //si el dato
recibido no es un line-feed o backspace
                    RX_Buffer[RX_Buffer_index]=p_args->data; //lo guardamos
en el buffer

                    if (RX_Buffer[RX_Buffer_index] == 13){ //si el dato es un
retorno de carro
                        RX_Buffer[RX_Buffer_index] = NULL; //cerramos la
cadena
                        RX_Buffer_index=0;                  //reseteamos el
índice
                        flag_rx_complete=true;
                    }
                    else {
                        RX_Buffer_index++;      //incrementamos el índice
                    }
                }
                else if (p_args->data == 8) {    //si recibimos un backspace
                    RX_Buffer_index--;          //decrementamos el índice
```

```
                    if (RX_Buffer_index<0) {    //saturamos
                        RX_Buffer_index=0;
                    }
                    RX_Buffer[RX_Buffer_index] = NULL;  //borramos el
carácter anterior
                }
                break;
            default:
                break;
        }
    }
}
```

6.7 Programación de Communications Framework

Este *framework* implementa una API para configurar una conexión serie sobre el puerto USB (J5).

Las características del driver son las siguientes:

- Las funciones de este *framework* son compatibles con otras implementaciones, como la UART, por lo que es sencillo pasar de una implementación a otra sin modificar demasiado el código.
- Permite el bloqueo del canal de comunicaciones para tener el acceso exclusivo del mismo.
- Soporta dos velocidades USB, *high speed* (HS) y *full speed* (FS).

A la hora de configurar el proyecto, este periférico debe ser configurado en la pestaña *Threads*, y los pines correspondientes, en la pestaña *Pins*.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.7.1 Communications Framework API

Para la programación de este periférico, haremos uso de las siguientes funciones:

- ***open*** *(sf_comms_ctrl_t * const p_ctrl, sf_comms_cfg_t const * const p_cfg)*: Abre una instancia del driver en nuestra aplicación, y aplica la configuración elegida en la pestaña *Threads*.
- ***read*** *(sf_comms_ctrl_t * const p_ctrl, uint8_t * const p_dest, uint32_t const bytes, UINT const timeout)*: Lee los datos recibidos. La función vuelve cuando ha leído los bytes solicitados o tras el *timeout*.
- ***write*** *(sf_comms_ctrl_t * const p_ctrl, uint8_t * const p_src, uint32_t const bytes, UINT const timeout)*: Transmite los datos. La función vuelve tras enviar el número de bytes indicados o tras el *timeout*.
- ***lock*** *(sf_comms_ctrl_t * const p_ctrl, sf _comms_lock_t lock_type, UINT timeout)*: Bloquea el driver de comunicaciones, reservando el uso exclusivo del mismo.
- ***unlock*** *(sf_comms_ctrl_t * const p_ctrl, sf _comms_lock_t lock_type)*: Desbloquea el driver de comunicaciones.
- ***close*** *(sf_comms_ctrl_t * const p_ctrl)*: Cierra la instancia del driver.
- ***versionGet*** *(ssp_version_t * p_version)*: Devuelve la version del driver.

6.7.2 Configuración

Este driver no se encuentra en el hilo común o HAL. Debemos incluirlo en el hilo en el que vayamos a necesitar este periférico.

Para seleccionarlo, debemos ir a la pestaña *Threads*, **crear un hilo nuevo** y buscar el driver en el menú que aparece al pulsar el botón *New Stack*. Debemos ir a *Framework>Connectivity>Communications Framework on sf_el_ux_comms_v2*.

Una vez hayamos elegido la instancia del driver, debemos pulsar en el módulo, y nos vamos a la pestaña *Properties*. En ella encontraremos las opciones de configuración disponibles.

También encontramos que hace uso de otros drivers, por lo que deberemos configurarlos también.

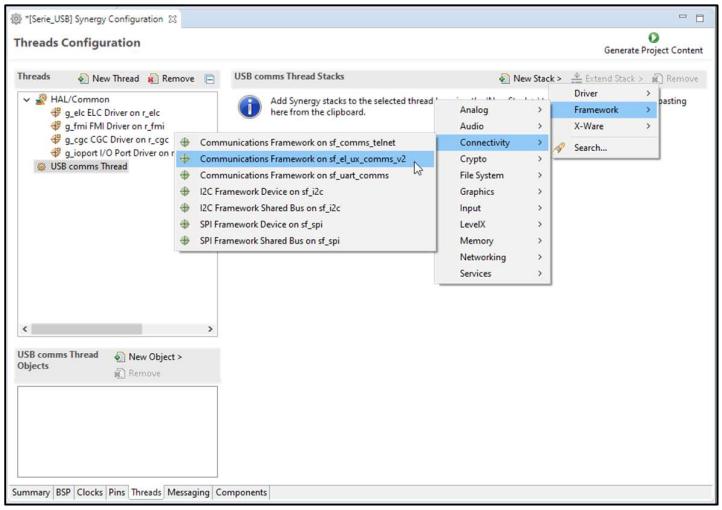


Figura 6-23 Selección del driver en la pestaña *Threads*

Al seleccionar el driver, observamos que este, hace uso de otros drivers, y que debemos incluir algunos para poder usarlo. Debemos añadir los módulos que están señalados en la Figura 6-24.

Una vez que hemos seleccionado esos módulos, debemos buscar en sus propiedades la opción *Show linkage warning* y cambiar el valor a *Disabled* para evitar que nos marque un error. En el caso del módulo *USBX Port CDC*, debemos cambiar el valor de la opción *Full Speed Interrupt Priority* a cualquier valor entre 0 y 14.

En el módulo principal, podemos modificar el tamaño del buffer de lectura, si lo necesitamos.

El resto de módulos podemos dejarlos por defecto.

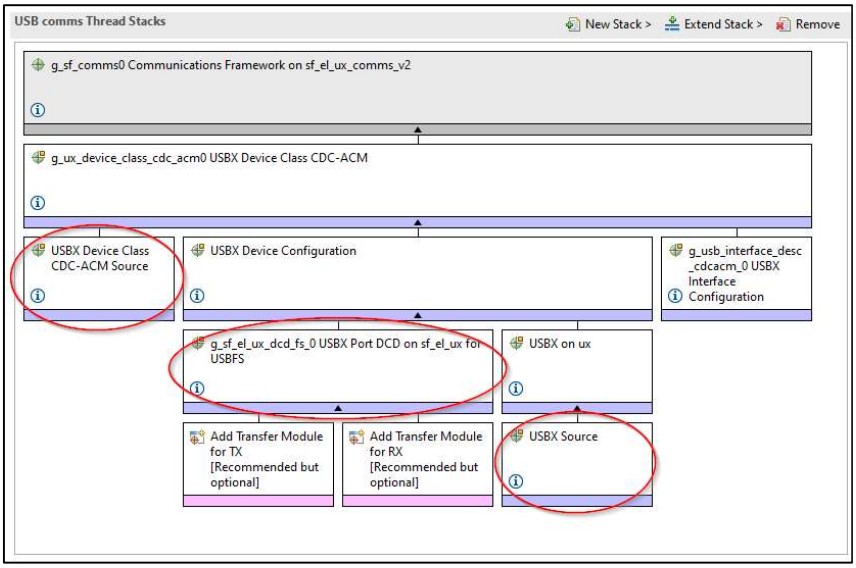


Figura 6-24 Selección de drivers adicionales

Una vez tengamos configurado el driver, debemos ir a la pestaña *Pins* y habilitar los pines necesarios.

Para buscar los pines asociados al puerto USB, podemos buscar el periférico (USB) en el árbol desplegable.

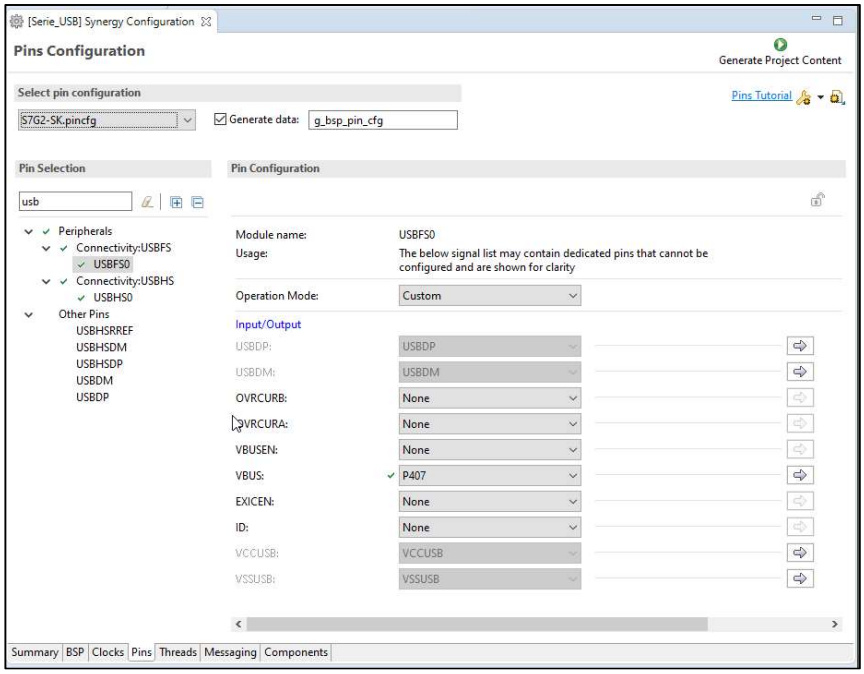


Figura 6-25 Configuración de los pines USB

La configuración por defecto es la que debemos usar.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.7.3 Ejemplo de programación

Escribir un programa en el que se transmita algún dato por USB cuando se pulse un botón integrado en la placa.

6.7.3.1 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
void hal_entry(void)
{
    /* TODO: add your own code here */
}
```

6.7.3.2 LED_thread_entry.c

```
#include "LED_thread.h"
#include "USB_comms_thread.h"
#include <string.h>

#define RED_LED_PIN IOPORT_PORT_06_PIN_01
#define GREEN_LED_PIN IOPORT_PORT_06_PIN_00
#define ORANGE_LED_PIN IOPORT_PORT_06_PIN_02
#define ON IOPORT_LEVEL_LOW
#define OFF IOPORT_LEVEL_HIGH

static int flag = 0;
uint8_t tx_string[32];

/* LED Thread entry function */
void LED_thread_entry(void)
{
    /* TODO: add your own code here */
    ioport_level_t valor_LED=0;

    g_external_irq11.p_api->open(g_external_irq11.p_ctrl,
g_external_irq11.p_cfg);
    g_external_irq11.p_api->enable(g_external_irq11.p_ctrl);

    while (1)
    {
        if (flag){
            g_ioport.p_api->pinRead(GREEN_LED_PIN, &valor_LED);
            if (valor_LED == IOPORT_LEVEL_HIGH){    //si el LED estaba
encendido, lo apagamos y viceversa
                g_ioport.p_api->pinWrite(GREEN_LED_PIN, IOPORT_LEVEL_LOW);
                strcpy(tx_string, "LED on\n\r");
                tx_queue_send(&USB_queue, tx_string, TX_WAIT_FOREVER);
//enviamos por cola el estado del LED
            }
            else {
                g_ioport.p_api->pinWrite(GREEN_LED_PIN, IOPORT_LEVEL_HIGH);
                strcpy(tx_string, "LED off\n\r");
                tx_queue_send(&USB_queue, tx_string, TX_WAIT_FOREVER);
//enviamos por cola el estado del LED
            }
            flag=0;
        }
        tx_thread_sleep (1);
    }
}
```

```
void g_external_irq11_callback(external_irq_callback_args_t *p_args){
    flag = 1;
}
```

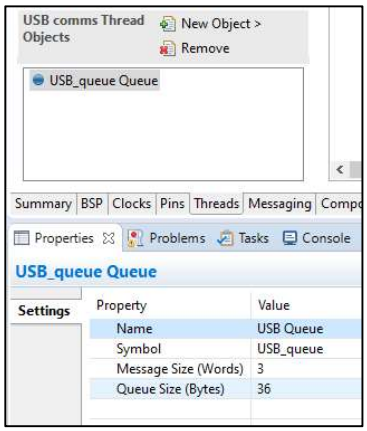


Figura 6-26 Propiedades de la cola incluida en el programa de ejemplo.

6.7.3.3 USB_comms_thread_entry.c

```
#include "USB_comms_thread.h"
#include <string.h>

uint8_t rx_string[32];

/* USB comms Thread entry function */
void USB_comms_thread_entry(void)
{
    /* TODO: add your own code here */
    g_sf_comms0.p_api->open(g_sf_comms0.p_ctrl, g_sf_comms0.p_cfg);
    while (1)
    {
        tx_queue_receive(&USB_queue, rx_string, TX_WAIT_FOREVER); //recibimos
por cola el estado del LED
        g_sf_comms0.p_api->write(g_sf_comms0.p_ctrl, rx_string,
strlen(rx_string), TX_WAIT_FOREVER); //enviamos por USB el estado del LED
        tx_thread_sleep (1);
    }
}
```

6.8 Programación de Capacitive Touch Button Framework

Este *framework* implementa una API para usar los botones capacitivos incluidos en nuestro kit.

Las características del driver son las siguientes:

- Funciona conjuntamente con el software *Capacitive Touch Workbench for Renesas Synergy*. Es un software que genera los datos de configuración de las interfaces táctiles incluidas en nuestra placa.
- Provee una función *callback* que responde a los eventos generados:
    - Realiza el *de-bouncing*.
    - Soporta múltiples tipos de eventos, como pulsar, soltar y pulsaciones largas.

    - Llama a la función *callback* para cada botón en el orden dado en la tabla de configuración.

- Requiere del módulo *Capacitive Touch Framework* (La configuración por defecto es válida para el uso que le vamos a dar, y aparece automáticamente).

A la hora de configurar el proyecto, este periférico debe ser configurado en la pestaña *Threads*, y los pines correspondientes, en la pestaña *Pins*.

Una vez configurados los pines deseados, pulsamos el botón Generate Project Content.

6.8.1 Capacitive Touch Button Framework API

Para la programación de este periférico, haremos uso de las siguientes funciones:

- ***open*** *(sf_touch_ctsu_button_ctrl_t * const p_ctrl, sf_touch_ctsu_button_cfg_t const * const p_cfg)*: Abre una instancia del driver en nuestra aplicación, y aplica la configuración elegida en la pestaña *Threads*.
- ***enable*** *(sf_touch_ctsu_button_ctrl_t * const p_ctrl, sf_touch_ctsu_button_id const button_id)*: Habilita la notificación a la función *callback*, para poder responder a los eventos táctiles.
- ***disable*** *(sf_touch_ctsu_button_ctrl_t * const p_ctrl, sf_touch_ctsu_button_id const button_id)*: Deshabilita la notificación a la función *callback*.
- ***close*** *(sf_touch_ctsu_button_ctrl_t * const p_ctrl)*: Cierra la instancia del driver.
- ***versionGet*** *(ssp_version_t * p_version)*: Devuelve la version del driver.

6.8.2 Configuración

Este driver no se encuentra en el hilo común o HAL. Debemos incluirlo en el hilo en el que vayamos a necesitar este periférico.

Para seleccionarlo, debemos ir a la pestaña *Threads*, **crear un hilo nuevo** y buscar el driver en el menú que aparece al pulsar el botón *New Stack*. Debemos ir a *Framework>Input>Cap Touch Button Framework on sf_touch_ctsu_button*.

Una vez hayamos elegido la instancia del driver, debemos pulsar en el módulo, y nos vamos a la pestaña *Properties*. En ella encontraremos las opciones de configuración disponibles.

Al seleccionar el driver, observamos que este, hace uso de otros drivers, pero que estos ya están incluidos.

Las propiedades que **debemos** cambiar son las siguientes:

- *Cap Touch Button Framework*:
    - Número de botones: Seleccionamos 2, que son los que tenemos en nuestro kit.
    - *Debounce multipliers*: Podemos dejar la configuración por defecto. Si tenemos problemas con el debouncing podemos incrementarlos.
- *Cap Touch Framework*:
    - Prioridad del hilo: Podemos cambiarla a la prioridad que deseemos. Con la configuración por defecto no debe dar problemas.
    - Frecuencia de actualización: Por defecto está en 50 Hz. A mayores frecuencias los botones responderán más rápido, pero también serán más inestables.

- *CTSU Driver*:
    - Número máximo de canales activos: En este caso, tenemos dos botones, por lo que debemos seleccionar 2.

El resto de opciones de configuración las mantendremos en los valores por defecto.

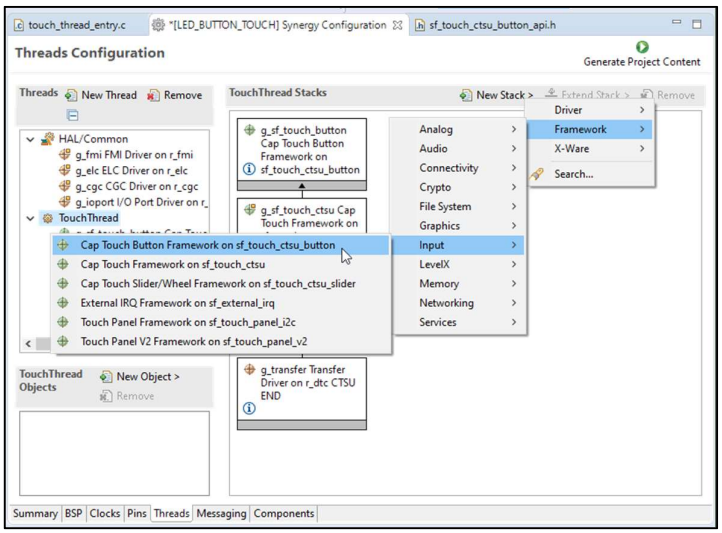


Figura 6-27 Selección del driver en la pestaña *Threads*

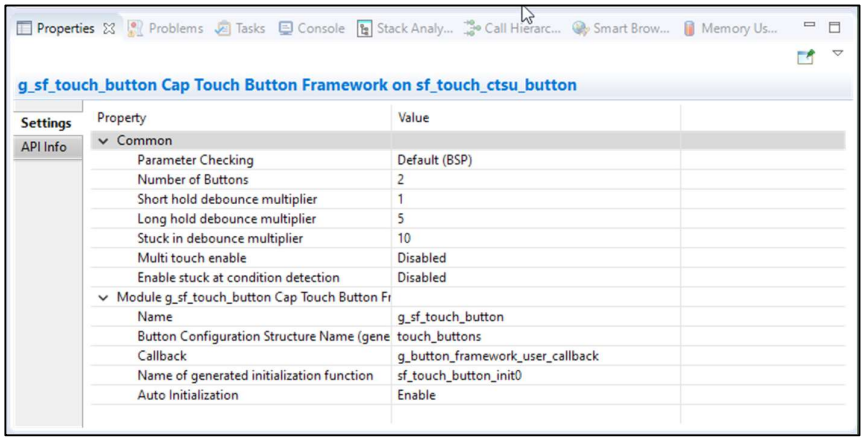


Figura 6-28 Propiedades del driver.

Una vez tengamos configurado el driver, debemos ir a la pestaña *Pins* y habilitar los pines necesarios.

Para buscar los pines asociados a la interfaz capacitiva, podemos buscar el periférico (CTSU) en el árbol desplegable.

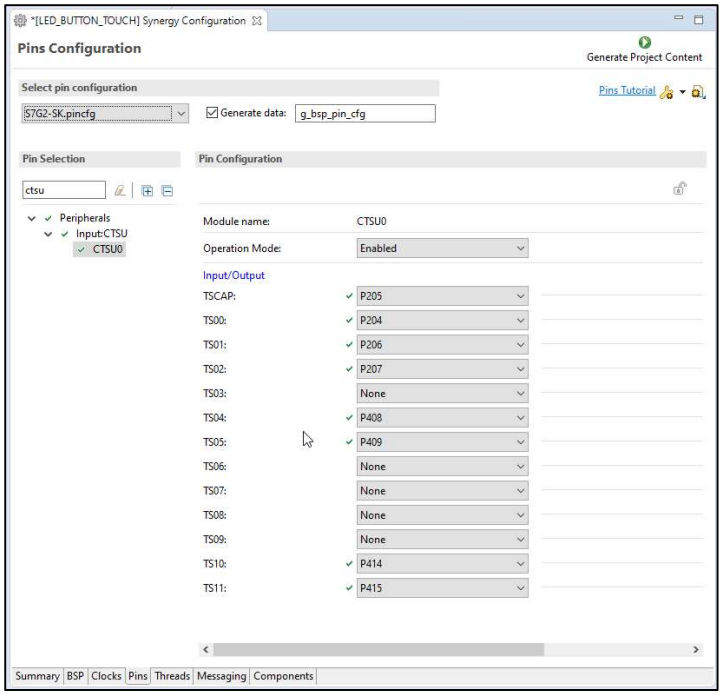


Figura 6-29 Configuración de los pines CTSU

La configuración por defecto es la que debemos usar.

Por último, la interfaz táctil hace uso del reloj PCLKB, y este no debe superar los 24MHz de frecuencia.

También debe ser un múltiplo de 2 MHz, por lo que bajaremos la frecuencia del PLL a 192MHz usando un multiplicador x16 y usaremos un divisor /8 para dejar la frecuencia del reloj PCLKB en los 24MHz.

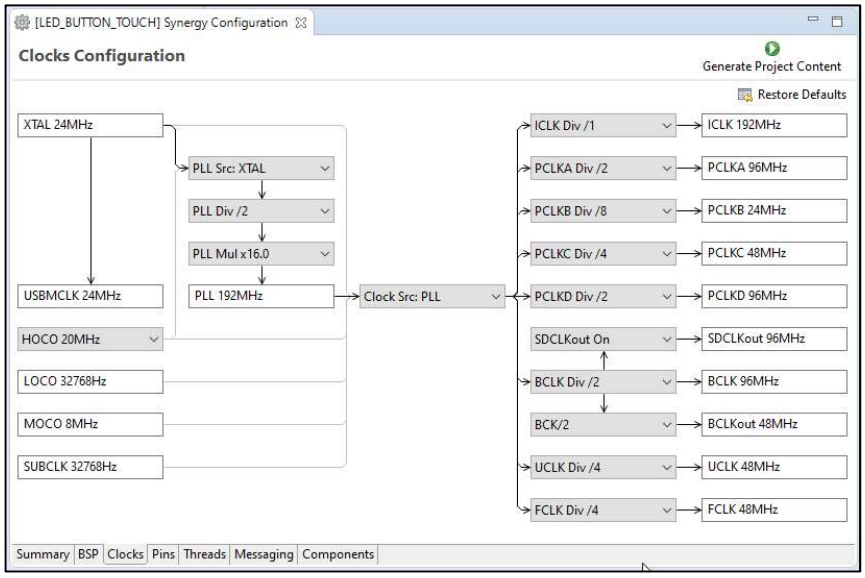


Figura 6-30 Configuración de reloj.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

Adicionalmente, este módulo necesita del software CTW para su calibración. Este software nos permitirá generar los datos de configuración de las interfaces táctiles incluidas en nuestra placa. En el capítulo 6 de esta memoria se desarrollan los pasos a seguir.

6.8.3 Ejemplo de programación

Escribir un programa que detecte las pulsaciones en los botones táctiles y cambie el LED integrado encendido.

6.8.3.1 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
void hal_entry(void)
{
    /* TODO: add your own code here */
}
```

6.8.3.2 touch_thread_entry.c

```
/** Programa que cambia el led encendido según se pulsan los botones
capacitivos **/
#include <touch_thread.h>

#define RED_LED_PIN IOPORT_PORT_06_PIN_01
#define GREEN_LED_PIN IOPORT_PORT_06_PIN_00
#define ORANGE_LED_PIN IOPORT_PORT_06_PIN_02
#define ON IOPORT_LEVEL_LOW
#define OFF IOPORT_LEVEL_HIGH
static int LED_index = 0;
static int flag = 0;
/* New Thread entry function */
void touch_thread_entry(void)
{
    /* TODO: add your own code here */
    g_ioport.p_api->pinWrite(RED_LED_PIN, OFF);
    g_ioport.p_api->pinWrite(GREEN_LED_PIN, OFF);
    g_ioport.p_api->pinWrite(ORANGE_LED_PIN, OFF);

    while (1)
    {
        switch (LED_index){
            case 0: //Encendemos el LED verde
                g_ioport.p_api->pinWrite(RED_LED_PIN, OFF);
                g_ioport.p_api->pinWrite(GREEN_LED_PIN, ON);
                g_ioport.p_api->pinWrite(ORANGE_LED_PIN, OFF);
                break;
            case 1: //Encendemos el LED rojo
                g_ioport.p_api->pinWrite(RED_LED_PIN, ON);
                g_ioport.p_api->pinWrite(GREEN_LED_PIN, OFF);
                g_ioport.p_api->pinWrite(ORANGE_LED_PIN, OFF);
                break;
            case 2: //Encendemos el LED naranja
                g_ioport.p_api->pinWrite(RED_LED_PIN, OFF);
                g_ioport.p_api->pinWrite(GREEN_LED_PIN, OFF);
                g_ioport.p_api->pinWrite(ORANGE_LED_PIN, ON);
                break;
            default:
                break;
        }
        tx_thread_sleep (10); //10 clock ticks
    }
```

```
}
void g_button_framework_user_callback(sf_touch_ctsu_button_callback_args_t
*p_args){
    switch (p_args->event){
            case TOUCH_BUTTON_STATE_RELEASED:
                if(p_args->id == 0) LED_index++;
                else if (p_args->id == 1) LED_index--;

                if (LED_index>2) LED_index = 2; //saturamos
                else if (LED_index<0) LED_index = 0;

                flag = 1;
                break;
            default:
                break;
        }
}
```

6.9 Programación de Capacitive Touch Slider Framework

Este *framework* implementa una API para usar el slider capacitivo incluido en nuestro kit. Este framework está diseñado también (como el *framework* de los botones capacitivos) para trabajar con los datos de configuración generados en la herramienta *Capacitive Touch Workbench for Renesas Synergy*.

Las características del driver son las siguientes:

- Soporta múltiples instancias de sliders.
- Provee una función *callback* que responde a los eventos generados, simplificando el procesamiento:
    - Genera una interrupción cuando cambia el estado del slider.
    - Cada función *callback* está asociada a cada slider, e incluye el evento y la posición.
- Soporta la detección de múltiples pulsaciones.

A la hora de configurar el proyecto, este periférico debe ser configurado en la pestaña *Threads*, y los pines correspondientes, en la pestaña *Pins*.

Una vez configurados los pines deseados, pulsamos el botón Generate Project Content.

6.9.1 Capacitive Touch Slider Framework API

Para la programación de este periférico, haremos uso de las siguientes funciones:

- ***open*** *(sf_touch_ctsu_slider_ctrl_t * const p_ctrl, sf_touch_ctsu_slider_cfg_t const * const p_cfg)*: Abre una instancia del driver en nuestra aplicación, y aplica la configuración elegida en la pestaña *Threads*.
- ***enable*** *(sf_touch_ctsu_slider_ctrl_t * const p_ctrl, sf_touch_ctsu_slider_id_t slider_id)*: Habilita la notificación a la función *callback*, para poder responder a los eventos táctiles.
- ***disable*** *(sf_touch_ctsu_slider_ctrl_t * const p_ctrl, sf_touch_ctsu_slider_id_t slider_id)*: Deshabilita la notificación a la función *callback*.
- ***close*** *(sf_touch_ctsu_slider_ctrl_t * const p_ctrl)*: Cierra la instancia del driver.

- ***versionGet*** *(ssp_version_t * p_version)*: Devuelve la version del driver.

6.9.2 Configuración

Este driver no se encuentra en el hilo común o HAL. Debemos incluirlo en el hilo en el que vayamos a necesitar este periférico.

Para seleccionarlo, debemos ir a la pestaña *Threads*, **crear un hilo nuevo** y buscar el driver en el menú que aparece al pulsar el botón *New Stack*. Debemos ir a *Framework>Input>Cap Touch Slider Framework on sf_touch_ctsu_slider*.

Una vez hayamos elegido la instancia del driver, debemos pulsar en el módulo, y nos vamos a la pestaña *Properties*. En ella encontraremos las opciones de configuración disponibles.

Al seleccionar el driver, observamos que este, hace uso de otros drivers, pero que estos ya están incluidos.

Las propiedades que **debemos** cambiar son las siguientes:

- *Cap Touch Sliders Framework*:
    - Número de *sliders*: Seleccionamos 1, que son los que tenemos en nuestro kit.
- *Cap Touch Framework*:
    - Prioridad del hilo: Podemos cambiarla a la prioridad que deseemos. Con la configuración por defecto no debe dar problemas.
    - Frecuencia de actualización: Por defecto está en 50 Hz. A mayores frecuencias los botones responderán más rápido, pero también serán más inestables.
- *CTSU Driver*:
    - *Offset Adjustment*: **Lo cambiaremos a *Disabled***. Al usar esta opción se logra una lectura del slider con menor ruido. De lo contrario, obtendremos datos espurios.
    - Número máximo de canales activos: En este caso, el slider se compone de 5 botones/sensores capacitivos, por lo que seleccionaremos 5. Si en nuestro proyecto usamos conjuntamente el slider y los botones capacitivos, seleccionaremos 7. *(*)*

(*): Los drivers de bajo nivel se pueden compartir entre el slider y los botones capacitivos, por lo que es recomendable reutilizarlos. A la hora de incluir este *framework* en nuestro proyecto, nos pedirá elegir una instancia nueva de los drivers de nivel inferior, o reutilizar los que ya se estuvieran utilizando para los botones.

El resto de opciones de configuración las mantendremos en los valores por defecto.

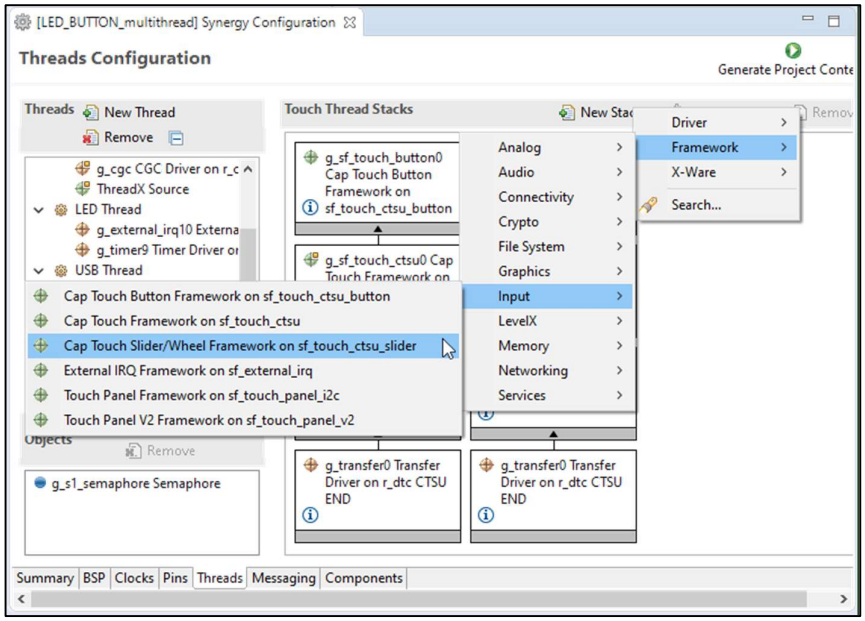


Figura 6-31 Selección del driver en la pestaña *Threads*

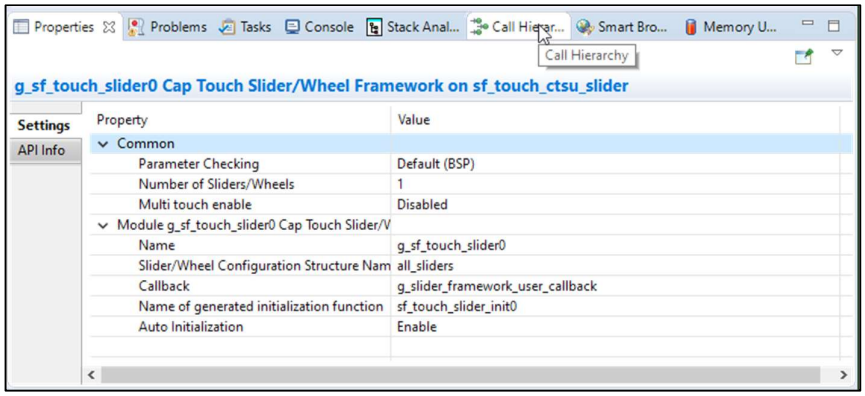


Figura 6-32 Propiedades del driver.

Una vez tengamos configurado el driver, debemos ir a la pestaña *Pins* y habilitar los pines necesarios.

Para buscar los pines asociados a la interfaz capacitiva, podemos buscar el periférico (CTSU) en el árbol desplegable.

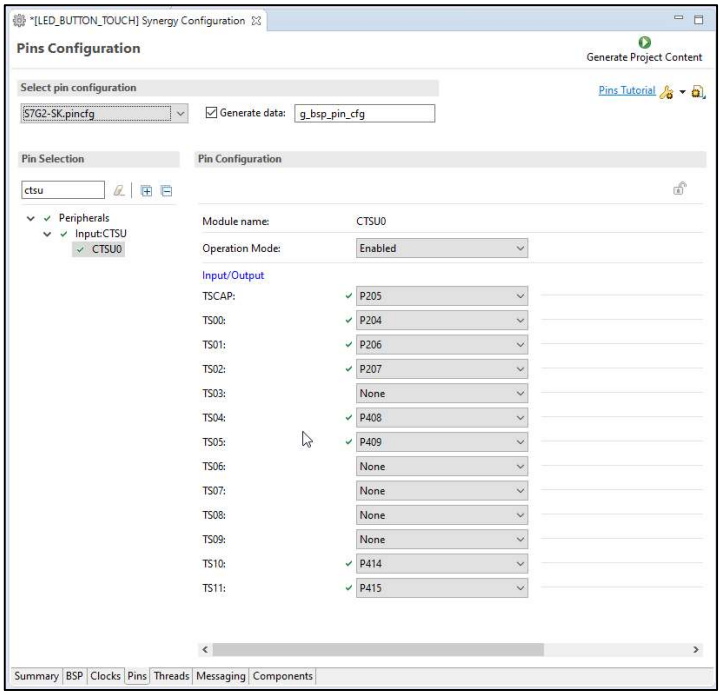


Figura 6-33 Configuración de los pines CTSU

La configuración por defecto es la que debemos usar.

Por último, la interfaz táctil hace uso del reloj PCLKB, y este no debe superar los 24MHz de frecuencia.

También debe ser un múltiplo de 2 MHz, por lo que bajaremos la frecuencia del PLL a 192MHz usando un multiplicador x16 y usaremos un divisor /8 para dejar la frecuencia del reloj PCLKB en los 24MHz.

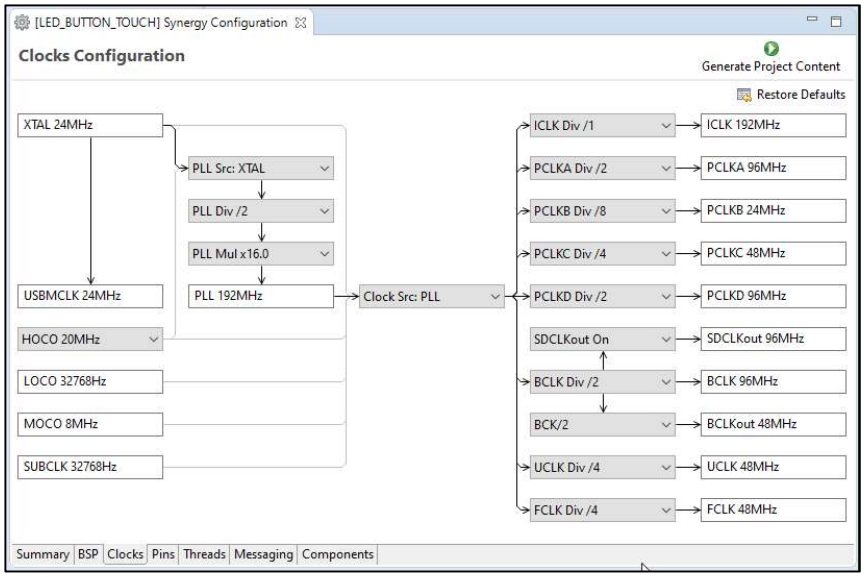


Figura 6-34 Configuración de reloj.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

Adicionalmente, este módulo necesita del software CTW para su calibración. Este software nos permitirá generar los datos de configuración de las interfaces táctiles incluidas en nuestra placa. En el capítulo 6 de esta memoria se desarrollan los pasos a seguir.

6.9.3 Ejemplo de programación

Escribir un programa que detecte la posición del slider y genere una salida PWM acorde a la misma.

6.9.3.1 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
void hal_entry(void)
{
    /* TODO: add your own code here */
}
```

6.9.3.2 slider_thread_entry.c

```
#include "slider_thread.h"

volatile
volatile sf_touch_ctsu_slider_callback_args_t *p_args_aux;
volatile uint32_t pos;

/* Slider Thread entry function */
void slider_thread_entry(void)
{
    /* TODO: add your own code here */
    g_timer9.p_api->open(g_timer9.p_ctrl, g_timer9.p_cfg);
    g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl, 0, TIMER_PWM_UNIT_PERCENT,
1);
    while (1){
        tx_semaphore_get(&new_slider_input_semaphore, TX_WAIT_FOREVER);
//esperamos a la interrupción generada por eventos capacitivos
        if (p_args_aux->id == 1 && (p_args_aux->event ==
SF_TOUCH_CTSU_SLIDER_STATE_TOUCHED || p_args_aux->event ==
SF_TOUCH_CTSU_SLIDER_STATE_HELD)){ //si detectamos pulsaciones, actualizamos
el duty cycle del PWM según el valor de la posición leído
            pos = p_args_aux->current_position/5;
            g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl, (timer_size_t) pos,
TIMER_PWM_UNIT_PERCENT, 1);
        }
        tx_thread_sleep (1);
    }
}
void g_slider_framework_user_callback(sf_touch_ctsu_slider_callback_args_t
*p_args){
    p_args_aux = p_args;
    tx_semaphore_put(&new_slider_input_semaphore); //usando un semáforo,
desbloqueamos la espera. Podríamos usar una simple variable bool, pero en
programación multihilo recomendamos usar objetos de tiempo real (podríamos
haber usado también un event flag).
}
```

6.10 Programación de Touch Panel Framework

Este *framework* implementa una API para usar el sensor táctil de la pantalla incluida en nuestro kit. El driver leerá.

Las características del driver son las siguientes:

- Lee los datos generados por el controlador de bajo nivel del sensor táctil y publica mensajes en cualquier cola suscrita a esos eventos en el *Messaging Framework*.
- Provee datos de posición (coordenadas X e Y).
- Provee un tipo de evento táctil (pulsación, pulsación mantenida, soltar e inválido).

A la hora de configurar el proyecto, este periférico debe ser configurado en la pestaña *Threads*, y los pines correspondientes, en la pestaña *Pins*.

Una vez configurados los pines deseados, pulsamos el botón Generate Project Content.

6.10.1 Touch Panel Framework API

Para la programación de este periférico, haremos uso de las siguientes funciones:

- ***open*** *(sf_touch_panel_ctrl_t * const p_ctrl, sf_touch_panel_cfg_t const * const p_cfg)*: Abre una instancia del driver en nuestra aplicación, y aplica la configuración elegida en la pestaña *Threads*. Crea los objetos de tiempo real requeridos e inicializa los módulos de menor nivel.
- ***calibrate*** *(sf_touch_panel_ctrl_t * const p_ctrl, sf_touch_panel_calibrate_t const * const p_expected, sf_touch_panel_payload_t const * const p_actual, ULONG timeout)*:
- ***start*** *(sf_touch_panel_ctrl_t * const p_ctrl)*: Empieza a escanear eventos táctiles.
- ***stop*** *(sf_touch_panel_ctrl_t * const p_ctrl)*: Para de escanear.
- ***reset*** *(sf_touch_panel_ctrl_t * const p_ctrl)*: Resetea el controlador de bajo nivel, si se ha provisto del pin reset en la configuración inicial.
- ***close*** *(sf_touch_panel_ctrl_t * const p_ctrl)*: Cierra la instancia del driver.
- ***versionGet*** *(ssp_version_t * p_version)*: Devuelve la version del driver.

6.10.2 Configuración

Este driver no se encuentra en el hilo común o HAL. Debemos incluirlo en el hilo en el que vayamos a necesitar este periférico.

Para seleccionarlo, debemos ir a la pestaña *Threads*, **crear un hilo nuevo** y buscar el driver en el menú que aparece al pulsar el botón *New Stack*. Debemos ir a *Framework>Input> Touch Panel Framework on sf_touch_panel_i2c*.

Una vez hayamos elegido la instancia del driver, debemos pulsar en el módulo, y nos vamos a la pestaña *Properties*. En ella encontraremos las opciones de configuración disponibles.

Al seleccionar el driver, observamos que este, hace uso de otros módulos, y que no están seleccionados.

En la Figura 6-36, encontraremos los módulos a seleccionar.

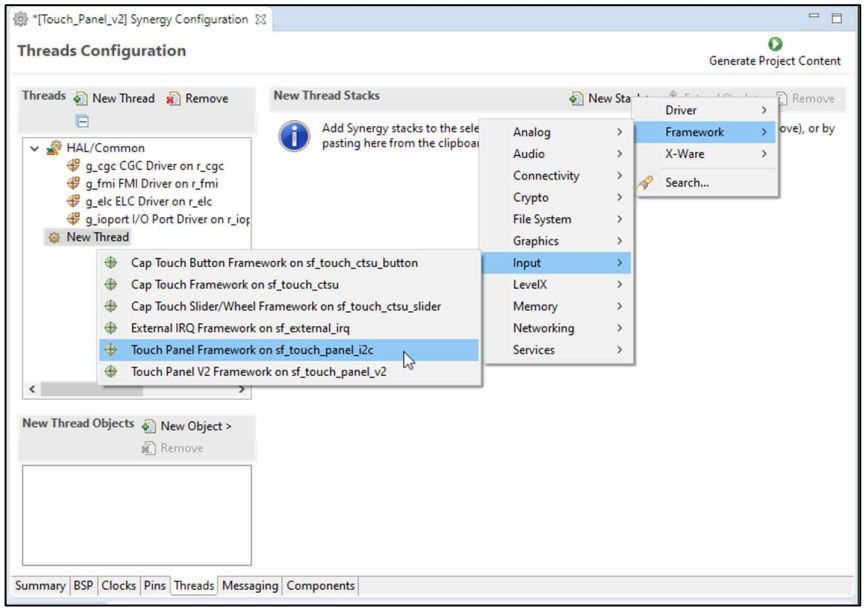


Figura 6-35 Selección del driver en la pestaña *Threads*

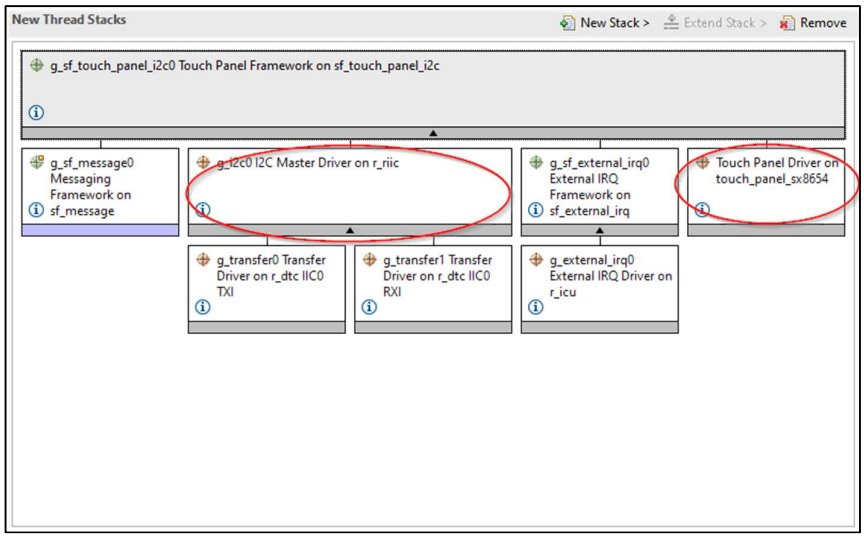


Figura 6-36 Pila de módulos.

Las propiedades que **debemos** cambiar son las siguientes:

- *Touch Panel Framework*:
    - Anchura, en píxeles: Seleccionamos 240.
    - Altura, en píxeles: Seleccionamos 320.
    - Pin para el reseteo: Lo cambiamos a IOPORT_PORT_06_PIN_09.
- *I2C Master Driver*:
    - Canal: Seleccionamos el 2.
    - Velocidad: Podemos seleccionar el modo estándar o el *Fast Mode*.

- Dirección del esclavo: La cambiaremos a 0x48.
- Modo de direccionamiento: Lo dejaremos en 7 bits.

- *External IRQ Driver*:
  - Canal: Usaremos el canal 9.
  - Disparo: Seleccionamos el flanco de bajada.
  - Filtrado: Habilitado.

El resto de opciones de configuración las mantendremos en los valores por defecto.

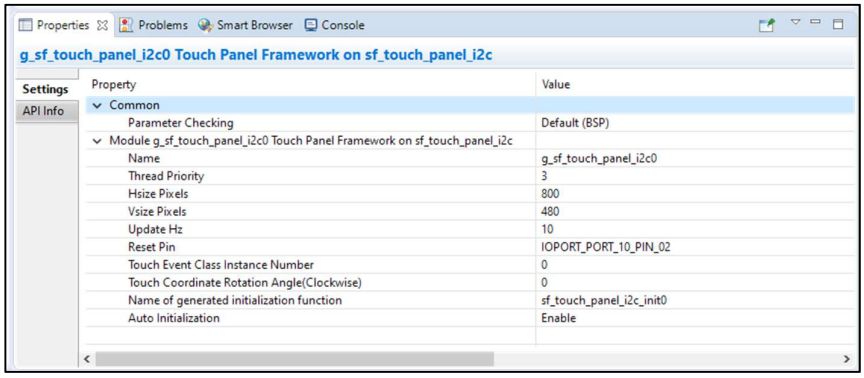


Figura 6-37 Propiedades del driver.

Una vez tengamos configurado el driver, debemos ir a la pestaña *Pins* y habilitar los pines necesarios.

Para buscar los pines asociados a la interfaz capacitiva, podemos buscar el periférico (SPI) en el árbol desplegable.

Seleccionaremos el modo de operación *Simple I2C* y mantendremos la configuración de pines que aparece.

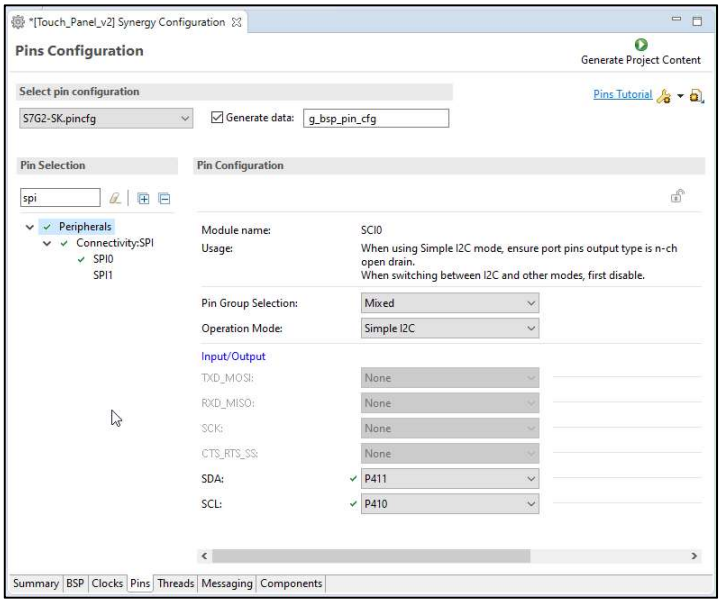


Figura 6-38 Configuración de los pines CTSU

Por último, la pantalla táctil hace uso del reloj PCLKB, pero el driver no presenta ninguna restricción o límite en la frecuencia del mismo, por lo que no debemos preocuparnos.

El único error posible sería al seleccionar una frecuencia de reloj con la cual e2 Studio no sea capaz de configurar correctamente la velocidad del bus I2C.

Por último, cabe comentar que al hacer uso del *Messaging Framework* (un *framework* que extiende la funcionalidad de las comunicaciones entre hilos del RTOS), encontraremos algunos elementos en la pestaña *Messaging*. Esta pestaña estará configurada automáticamente para el uso que le vamos a dar con el driver del sensor táctil de la pantalla.

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

6.10.3 Ejemplo de programación

Escribir un programa que escriba por puerto serie la coordenada de la pulsación en la pantalla táctil.

6.10.3.1 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
void hal_entry(void)
{
    /* TODO: add your own code here */
}
```

6.10.3.2 touch_panel_thread_entry.c

```
#include "touch_panel_thread.h"
#include "USB_thread.h"
//necesario para poder usar la cola de mensajes
#include "sf_message_port.h"
#include <stdio.h>

uint8_t tx_string[128];
uint16_t touch_x, touch_y;

/* Touch Panel Thread entry function */
void touch_panel_thread_entry(void)
{
    /* TODO: add your own code here */
    sf_message_header_t * p_message = NULL;
    g_sf_touch_panel_i2c0.p_api->open(g_sf_touch_panel_i2c0.p_ctrl,
g_sf_touch_panel_i2c0.p_cfg);
    g_sf_touch_panel_i2c0.p_api->start(g_sf_touch_panel_i2c0.p_ctrl);
    while (1)
    {
        p_message = NULL;
        strcpy(tx_string, ""); //limpia la cadena de caracteres
        g_sf_message0.p_api->pend(g_sf_message0.p_ctrl,
&touch_panel_thread_message_queue, (sf_message_header_t **) &p_message,
TX_WAIT_FOREVER);
        switch(p_message->event_b.class_code){
            case SF_MESSAGE_EVENT_CLASS_TOUCH:
                if (p_message->event_b.code == SF_MESSAGE_EVENT_NEW_DATA){
                    sf_touch_panel_payload_t * touch_panel_message =
(sf_touch_panel_payload_t *) p_message;
                    switch(touch_panel_message->event_type){
                        case SF_TOUCH_PANEL_EVENT_DOWN:
```

```
                            touch_x = (unsigned int)touch_panel_message->x;
                            touch_y = (unsigned int)320-touch_panel_message-
>y; //la pantalla está girada
                            sprintf(tx_string, "\rt_press   x=%d - y=%d  \r",
(unsigned int)touch_x, (unsigned int)touch_y);
                            tx_queue_send(&g_cdc_queue, tx_string,
TX_WAIT_FOREVER);
                            break;
                        case SF_TOUCH_PANEL_EVENT_UP:
                            sprintf(tx_string, "\rt_release x=000 -
y=000\r");
                            tx_queue_send(&g_cdc_queue, tx_string,
TX_WAIT_FOREVER);
                            break;
                        case SF_TOUCH_PANEL_EVENT_INVALID:
                        case SF_TOUCH_PANEL_EVENT_HOLD:
                        case SF_TOUCH_PANEL_EVENT_MOVE:
                        case SF_TOUCH_PANEL_EVENT_NONE:
                            break;
                    }
                }
                break;
        }
        g_sf_message0.p_api->bufferRelease(g_sf_message0.p_ctrl,
(sf_message_header_t *) p_message, SF_MESSAGE_RELEASE_OPTION_NONE);
        tx_thread_sleep (1);
    }
}
```

6.10.3.3 USB_thread_entry.c

```
#include "USB_thread.h"
uint8_t rx_string[128];

/* USB Thread entry function */
void USB_thread_entry(void)
{
    g_sf_comms0.p_api->open(g_sf_comms0.p_ctrl,g_sf_comms0.p_cfg);
//abrimos conexión serie sobre USB

    /* TODO: add your own code here */
    while (1)
    {
        tx_queue_receive(&g_cdc_queue, rx_string, TX_WAIT_FOREVER);
//espera bloqueante hasta que recibimos algo por la cola. Se guarda en
rx_string
        g_sf_comms0.p_api->write(g_sf_comms0.p_ctrl, rx_string, 128,
TX_WAIT_FOREVER);           //escribimos el dato recibido
        tx_thread_sleep (1);
    }
}
```

6.11 Programación de GUIX

Renesas ofrece una manera sencilla y cómoda para programar la pantalla incluida en nuestro kit. Se trata de usar la herramienta GUIX Studio para generar los recursos necesarios a usar en nuestra aplicación.

Para hacer uso de ella, necesitaremos incluir dos módulos en nuestro proyecto; el módulo *GUIX on gx*, que implementa el motor gráfico y el driver de decodificación JPEG entre otros, y el driver SPI para comunicar con la pantalla.

Adicionalmente, deberemos incluir el *Touch Panel Framework* si queremos obtener información de los eventos táctiles de la pantalla.

Por último, en el siguiente enlace (https://www.renesas.com/us/en/document/scd/1221281) encontraremos un programa de ejemplo, el cual contiene un par de ficheros con la configuración de la pantalla, que deberemos copiar en nuestro proyecto.

Si el enlace anterior no funciona, podemos buscar "GUIX SK-S7G2 sample code" en la página de búsqueda de Renesas. https://www.renesas.com/us/en/search .

6.11.1 Configuración

El módulo *GUIX on gx* no se encuentra en el hilo común o HAL. Debemos incluirlo en el hilo en el que vayamos a necesitar este periférico.

Para seleccionarlo, debemos ir a la pestaña *Threads*, **crear un hilo nuevo** y buscar el driver en el menú que aparece al pulsar el botón *New Stack*. Debemos ir a *X-WARE > GUIX > GUIX on gx*.

El driver SPI lo incluiremos en el mismo hilo.

Para seleccionarlo, Debemos ir a *Driver > Connectivity > SPI Driver on r_sci_spi*.

El hilo donde vayamos a incluir estos módulos, debe tener las siguientes propiedades:

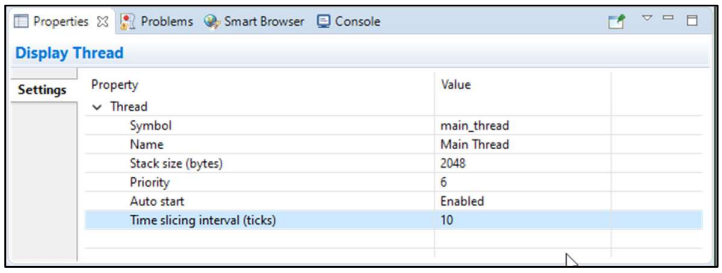


Figura 6-39 Propiedades del hilo

Al seleccionar el módulo *GUIX on gx*, observamos que este, hace uso de otros módulos y drivers.

En la Figura 6-40, encontraremos los módulos que requieren de algún cambio en sus propiedades.

En las figuras 6-41 a 6-47, encontraremos la configuración adecuada para cada módulo.

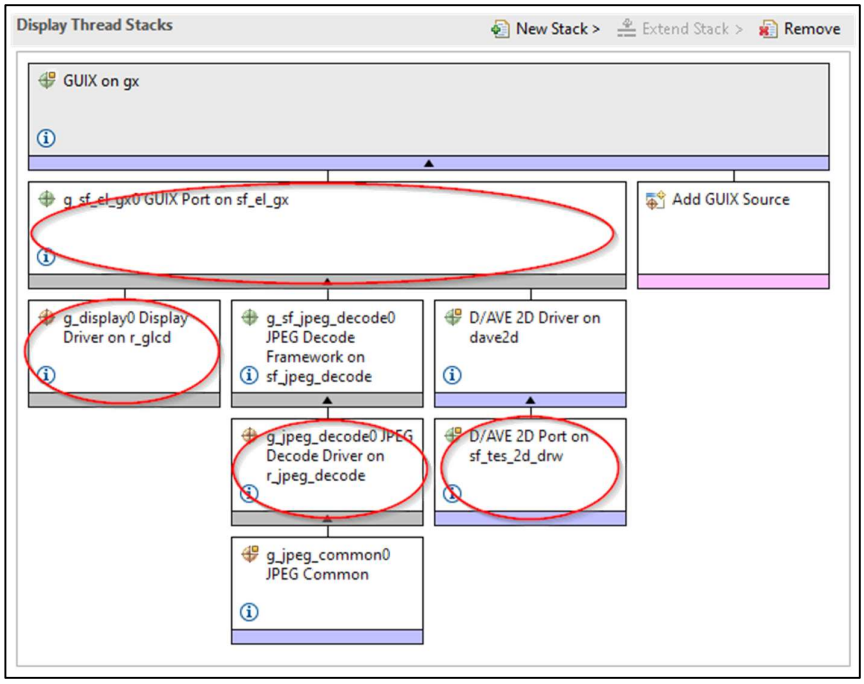


Figura 6-40 Pila de módulos.

Las propiedades de los módulos deben quedar así:

- *GUIX on gx:*

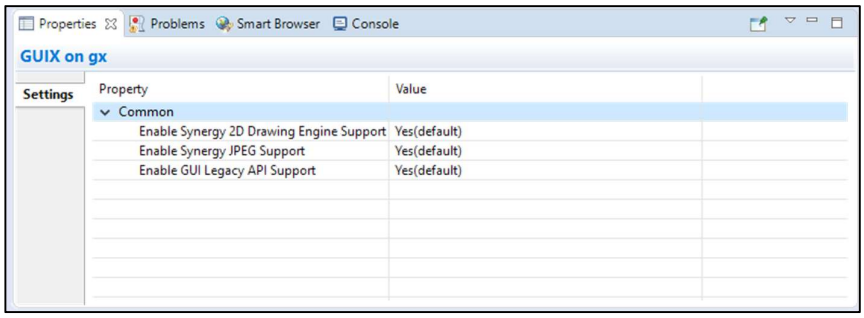


Figura 6-41 Propiedades de *GUIX on gx*

- *GUIX Port on sf_el_gx*:

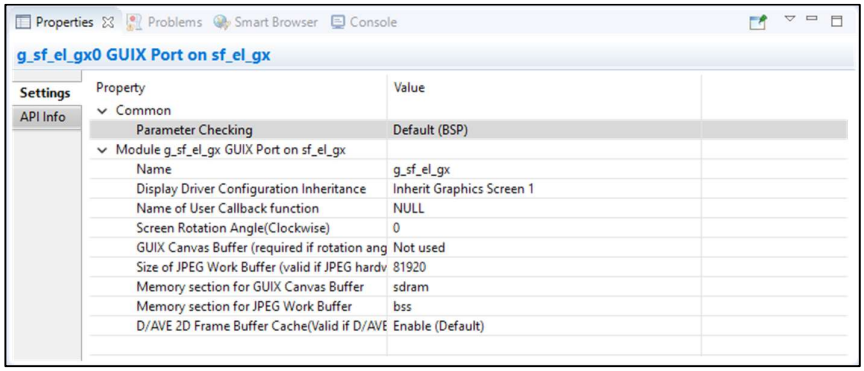


Figura 6-42 Propiedades de *GUIX Port on sf_el_gx*

- *Display Driver on r_glcd*:
  - *Graphics Screen 1*:

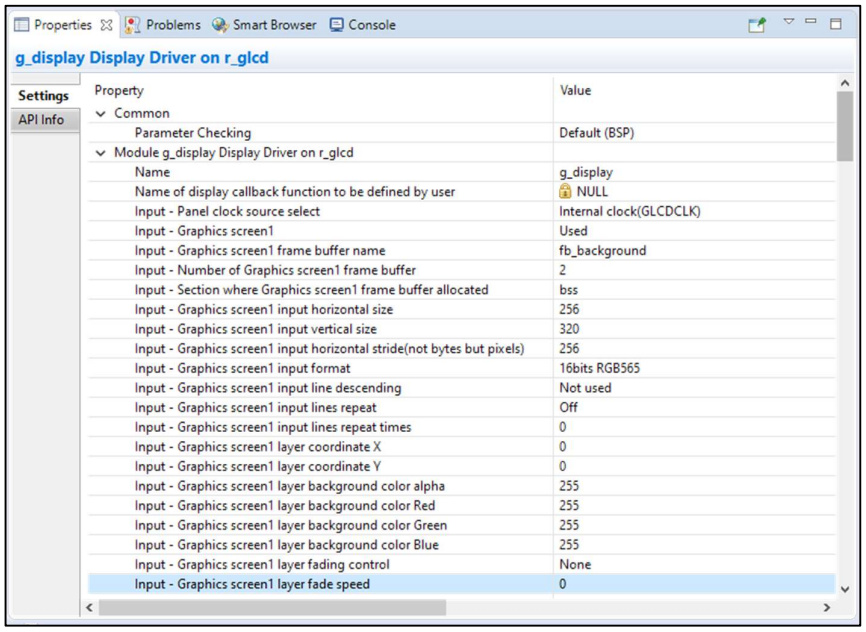


Figura 6-43 Propiedades de *Display Driver on r_glcd* (1)

- *Output* y *TCON*:

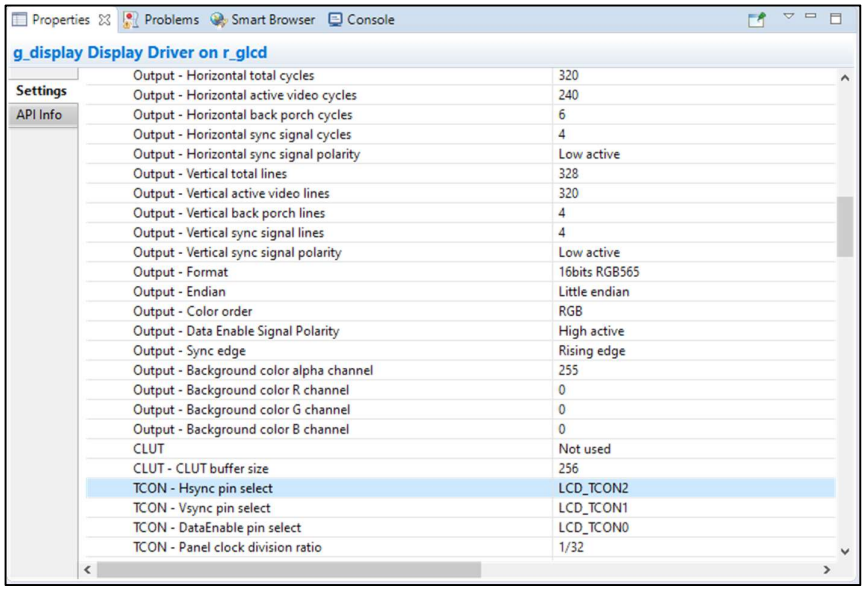


Figura 6-44 Propiedades de *Display Driver on r_glcd* (2)

- Interrupciones:

Figura 6-45 Propiedades de *Display Driver on r_glcd* (2)

- *JPEG Decode Driver on r_jpeg_decode*:

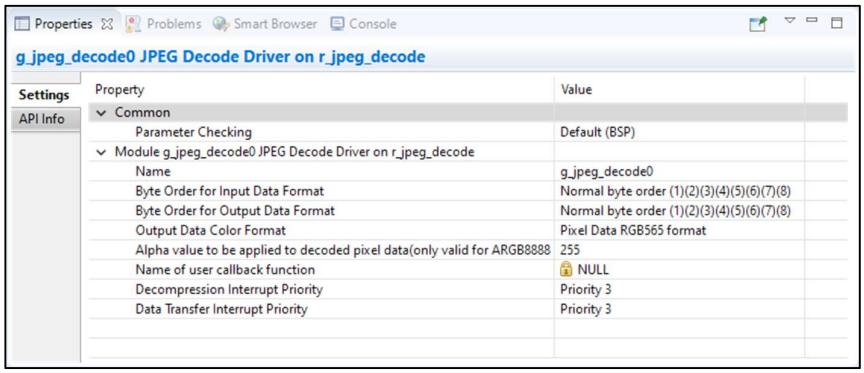


Figura 6-46 Propiedades de *JPEG Decode Driver on r_jpeg_decode*

- *D/AVE 2D Port on sf_tes_2d_drw*:

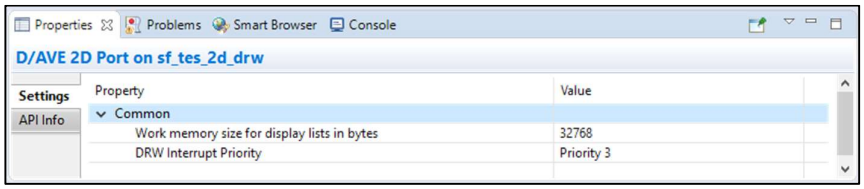


Figura 6-47 Propiedades de *D/AVE 2D Port on sf_tes_2d_drw*

Al seleccionar el módulo *SPI Driver on r_sci_spi*, observamos que este, hace uso de otros módulos y drivers.

En las figuras 6-48 a 6-54, encontraremos la configuración adecuada para cada módulo.

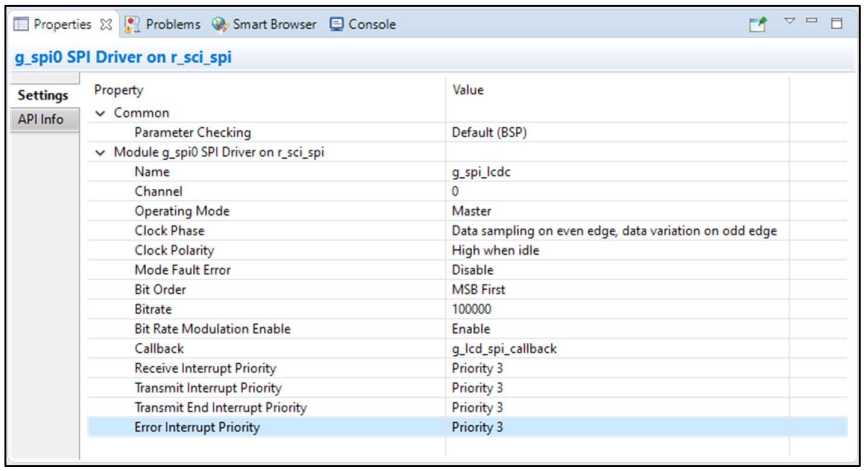


Figura 6-48 Propiedades de *SPI Driver on r_sci_spi*

Una vez tengamos configurado el driver, debemos ir a la pestaña *Pins* y habilitar los pines necesarios.

Para buscar los pines asociados al driver SPI sobre SCI, podemos buscar el periférico (SCI0) en el árbol desplegable.

Seleccionaremos el modo de operación *Simple SPI* y seleccionaremos los pines que aparecen en la Figura 6-49.

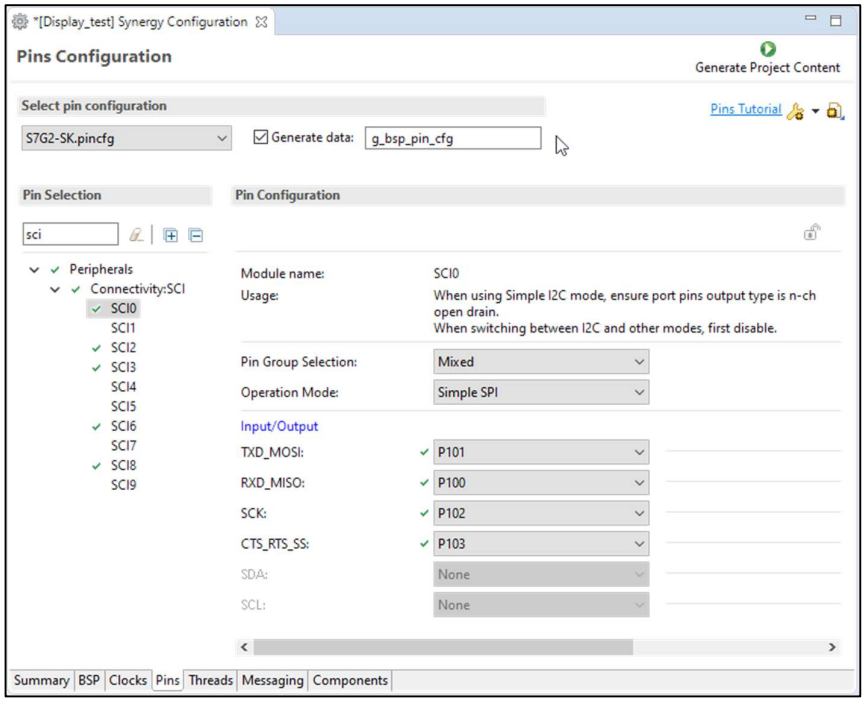


Figura 6-49 Configuración de pines del periférico SCI0

Para buscar los pines asociados al driver I2C, podemos buscar el periférico (IIC2) en el árbol desplegable. Seleccionaremos el modo de operación *Enabled* y seleccionaremos los pines que aparecen en la Figura 6-50.

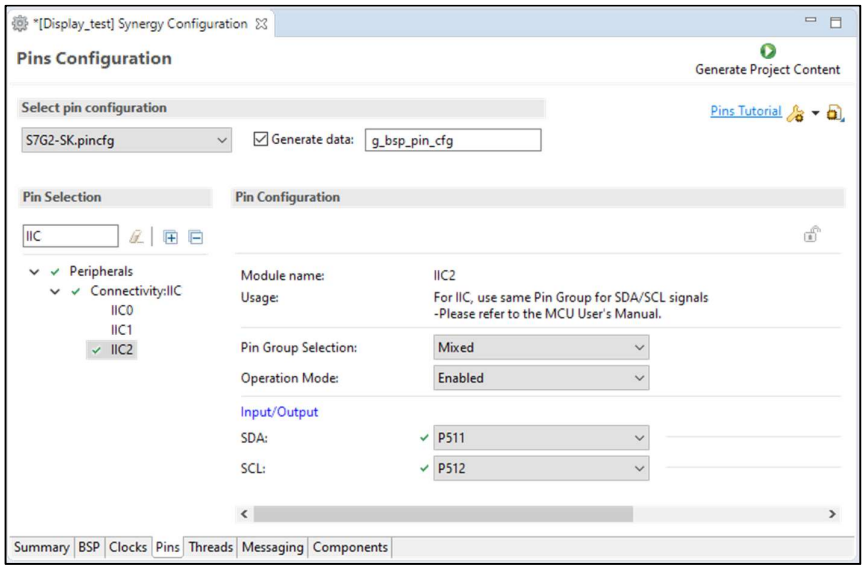


Figura 6-50 Configuración de pines del periférico IIC2

Ahora, debemos buscar algunos pines que necesitamos configurar para el correcto funcionamiento de la pantalla. El primero, es el pin P115 que debe quedar configurado como muestra la siguiente figura.

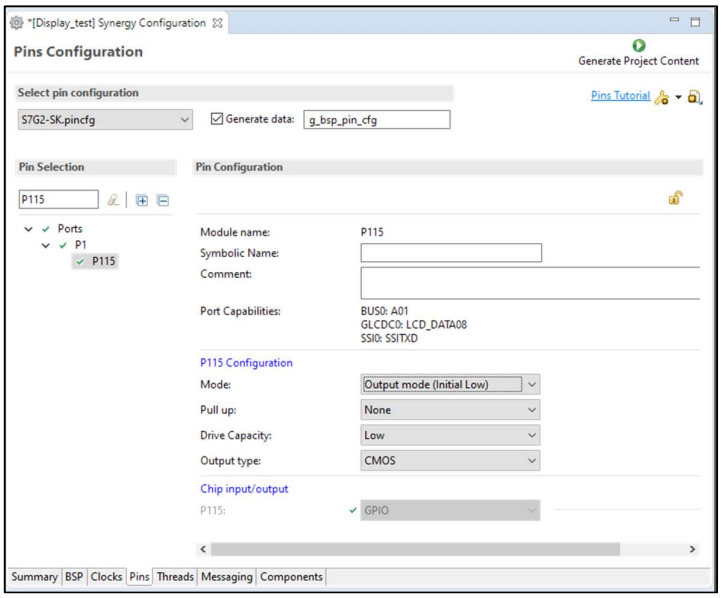


Figura 6-51 Configuración del pin P115

Los siguientes pines a configurar, son los pines P609, P610 y P611, con la misma configuración, como muestra la siguiente figura.

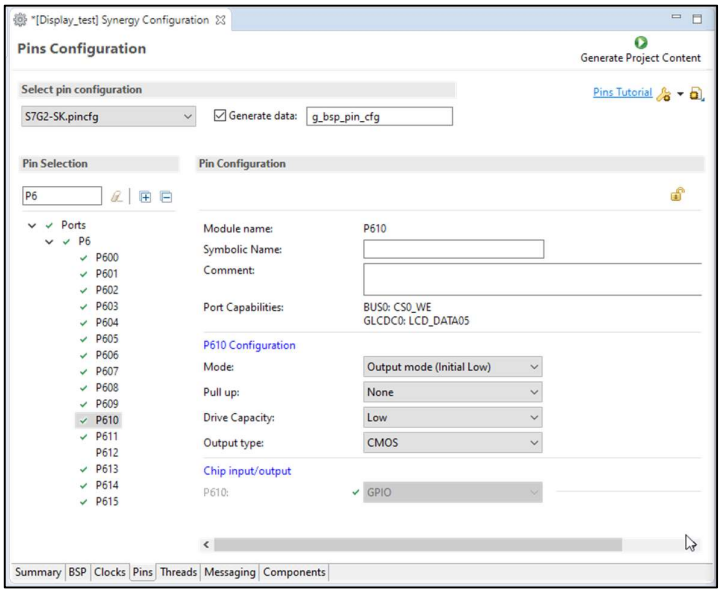


Figura 6-52 Configuración de los pines P609, P610 y P611

Lo siguiente, es verificar que los pines del periférico GLCDC0 son los correctos. Después, debemos comprobar que en *Drive Capacity* está seleccionada la opción *High*. Podremos hacer uso de la flecha que se encuentra a la

derecha de cada pin para entrar en su configuración.

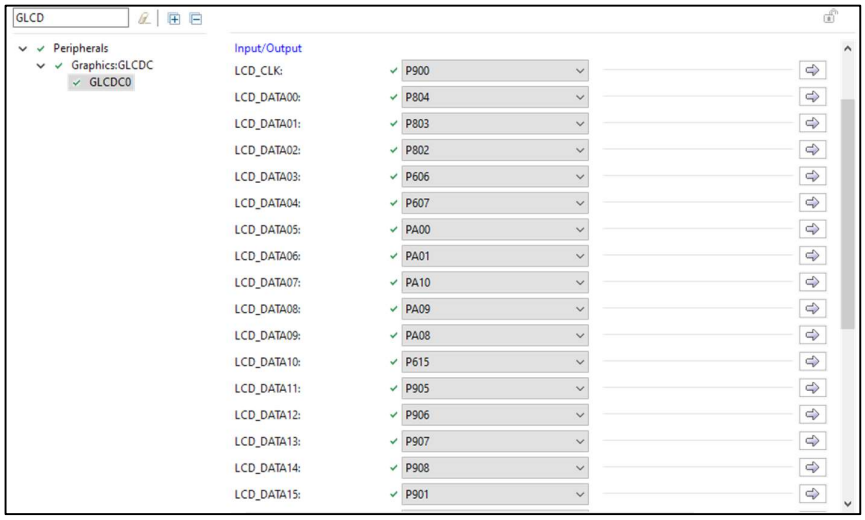


Figura 6-53 Configuración de pines del periférico GLCD0

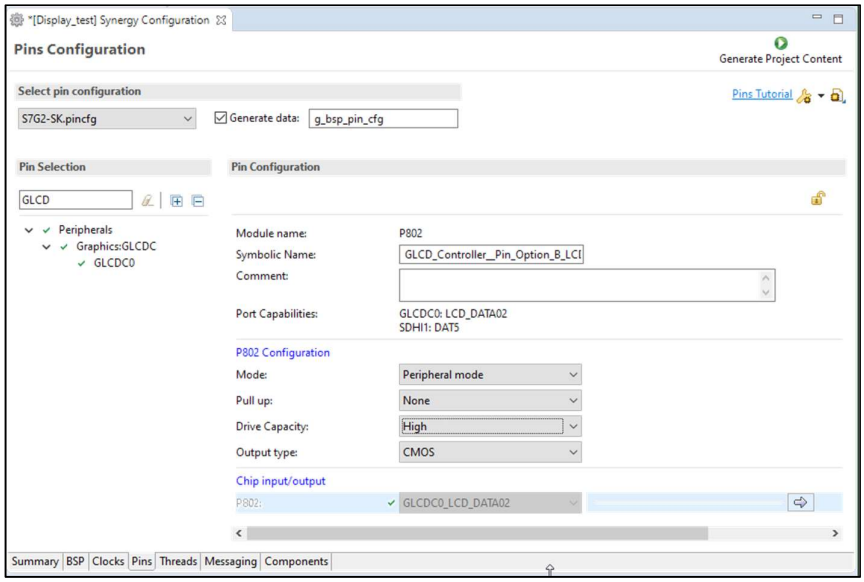


Figura 6-54 Configuración de pines del periférico GLCD0 en modo *High Drive Capacity*

Una vez configurados los pines deseados, pulsamos el botón *Generate Project Content*.

Tras haber generado el proyecto, necesitaremos los ficheros fuente que se encuentran en el archivo .zip descargado al principio.

Estos se encuentran en el archivo *Source Files.zip*.

Los ficheros a incluir son:

- lcd.h y lcd_setup.c : Contienen los comandos para controlar el panel LCD, así como la secuencia de inicialización de la misma. Esos archivos irán en la carpeta */src/hardware/* de nuestro proyecto. Si la carpeta *hardware* no está creada, la crearemos.

- main_thread_entry.c : Contiene el código necesario para inicializar la aplicación, así como para leer los eventos de la cola y transformarlos en acciones en GUIX. Este archivo sustituirá al archivo autogenerado por e2Studio, en la carpeta */src/*.

- guiapp_event_handler.c : Contiene las funciones para manejar los eventos de los elementos creados en GUIX Studio para el ejemplo que Renesas incluye en el mismo fichero .zip. Nos servirá de apoyo para crear nuestras propias funciones para manejar eventos. Al igual que el fichero anterior, lo copiaremos en la carpeta */src/*.

Estos archivos contienen referencias al proyecto de ejemplo que Renesas incluye en el fichero .zip que hemos descargado. Si nuestro proyecto no se llama “guiapp”, deberemos cambiar las referencias a “guiapp” por el nombre de nuestro proyecto.

El último paso, será crear un proyecto en GUIX Studio.

En el capítulo 7 podemos ver cómo hacerlo.

7 CALIBRACIÓN DE LA INTERFAZ CAPACITIVA

Descripción del software CTW y calibración de la interfaz capacitiva

Este apartado se centra en la calibración de la interfaz capacitiva incluido en el kit de programación. Se describen los pasos necesarios para incluir dicha interfaz en nuestra aplicación.

7.1 Capacitive Touch Workbench for Renesas Synergy

Capacitive Touch Workbench for Renesas Synergy o CTW es una herramienta que nos ayudará a generar algunos datos de configuración de la interfaz táctil presente en el hardware de la familia Synergy, así como calibrar sus parámetros, buscando conseguir la detección de eventos táctiles apropiada.

El propio programa guía al usuario de lo que debe hacer para calibrar la interfaz capacitiva y muestra los resultados.

Para hacer uso de esta herramienta, necesitaremos descargar la herramienta desde el enlace incluido en el punto 4.2.3.

Adicionalmente, debemos un proyecto "base" del cual el programa toma ciertos datos. Este proyecto se puede descargar buscando "r20an0448eu0108" o "Tuning the Capacitive Touch Solution Sample Code" en la página https://www.renesas.com/us/en/search. Descargaremos el archivo .zip.

Tras descomprimir el archivo, encontraremos un manual para usar la herramienta CTW y los proyectos "base" para distintos kits de desarrollo, entre otros.

7.2 Importar el proyecto base

Lo primero que debemos hacer es importar el proyecto base, que usaremos para calibrar la interfaz capacitiva

en nuestro proyecto.

Descomprimiremos el archivo .zip en la ruta que deseemos y luego nos abriremos e2 Studio.

Desde e2 Studio tendremos que ir a *File>Import*.

Se abrirá una ventana en la que seleccionaremos la opción *Existing Projects into Workspace* y pulsaremos *Next*.

En la siguiente pantalla, podremos buscar el directorio donde hemos descomprimido el proyecto.

Una vez hayamos seleccionado nuestro proyecto, pulsaremos en *Finish* y se importará el proyecto.

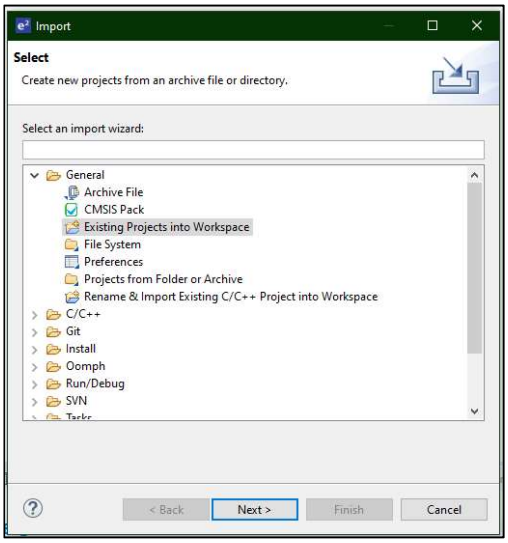


Figura 7-1Ventana de importación de proyectos

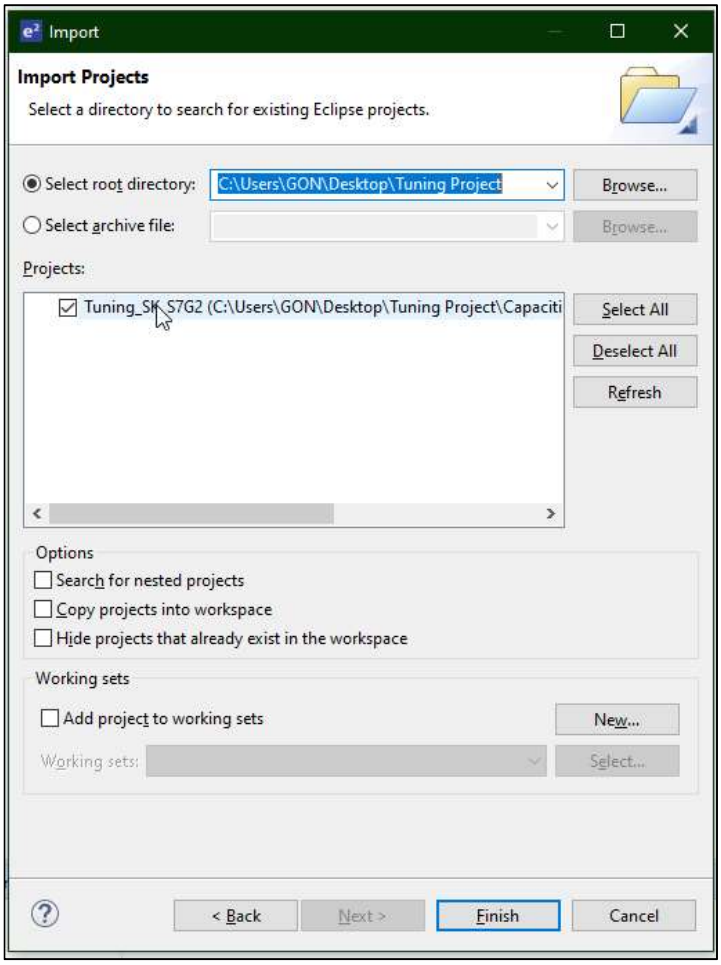


Figura 7-2 Selección del proyecto a importar.

7.3 Calibración

Para configurar la interfaz táctil de nuestro proyecto, abriremos CTW.

Una vez abierto, pulsaremos en el botón *START*.

Figura 7-3 Pantalla principal de CTW

Se nos abrirá una ventana que nos guiará durante el proceso.

Los pasos son los siguientes:

1. Seleccionar el directorio raíz de nuestro proyecto (*Application Project Directory*) y del proyecto base (*Tuning Project Directory*). Pulsamos *Next*.

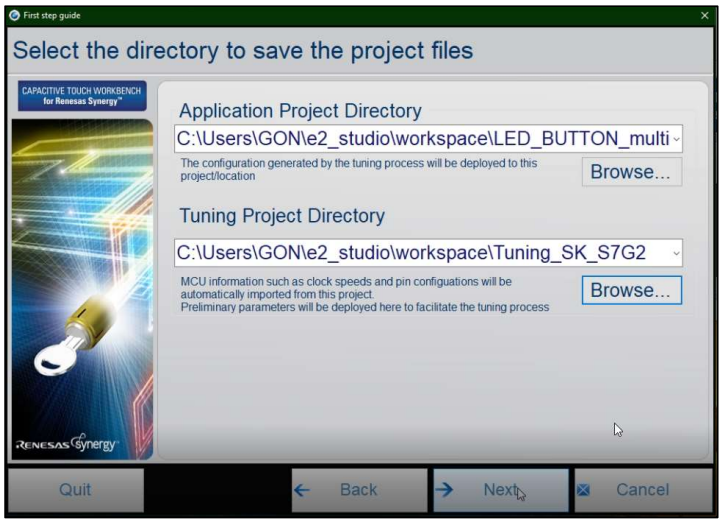


Figura 7-4 Selección de los proyectos.

2. Seleccionar *Self capacitance method*. Pulsamos *Next*.
3. Crear la disposición de los botones/sliders. Según nuestra aplicación, pondremos uno o dos botones, el slider, o cualquier combinación posible con el HW de nuestro kit. A cada botón y slider debemos asignarle su canal. Esta información la podremos encontrar en el manual incluido en el fichero .zip que descargamos al principio. Pulsamos *Next*.

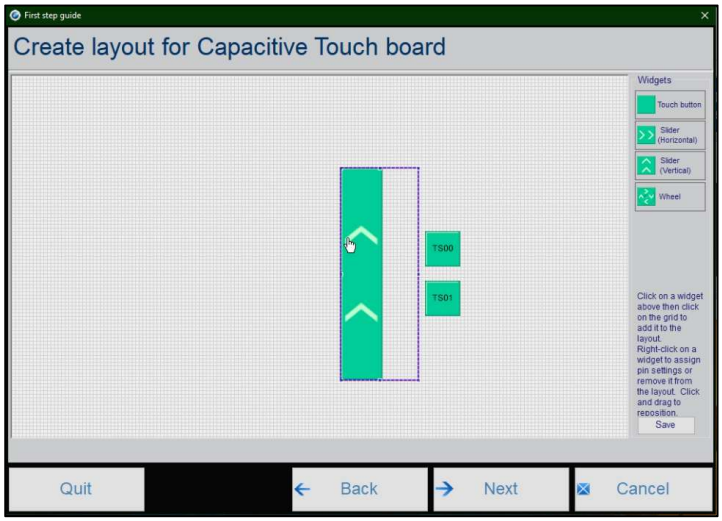


Figura 7-5 Disposición de la interfaz capacitiva.

a. Para los botones, debemos seleccionar los canales TS00 y TS01
b. Para los sliders, debemos seleccionar primero los 5 canales que tiene nuestra placa, y luego, los canales TS02, TS04, TS05, TS10 y TS11. *(*)*

Figura 7-6 Selección de canales del slider.

4. Mantendremos las resistencias en los valores por defecto. Pulsamos *Next*.
5. Compilaremos y ejecutaremos el proyecto base en e2 Studio.
6. Conectamos un cable USB al puerto J5 de nuestra placa, para que pueda comunicar por puerto serie.
7. Pulsamos *Next* en CTW. Aparecerá una ventana para seleccionar el puerto virtual sobre USB y la tasa de baudios. Seleccionaremos **38400 bps.**

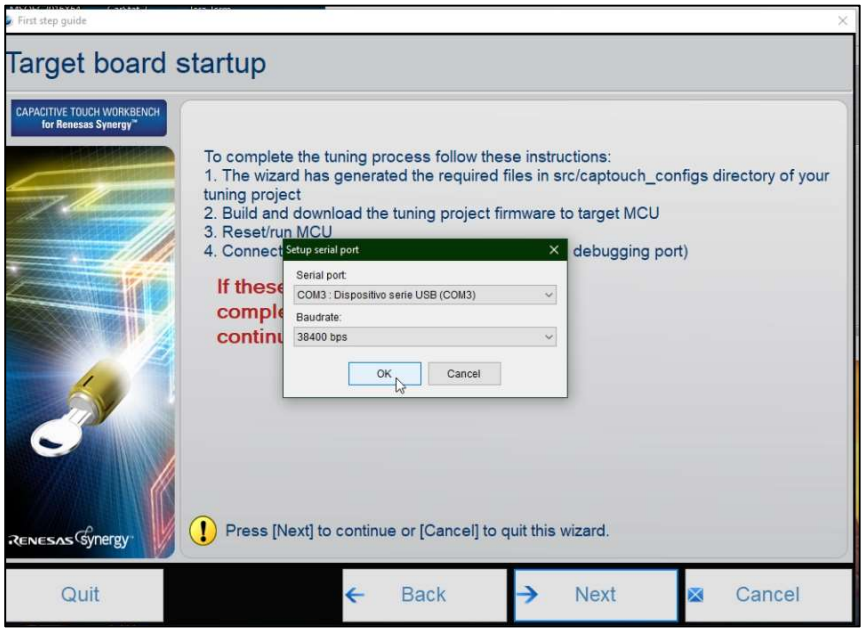


Figura 7-7 Configuración del puerto serie.

8. Pulsamos *Next* y esperamos a que mida la capacitancia parásita de cada sensor. Pulsamos *Next*.
9. Pulsamos *Next* y esperamos a que el programa muestre la frecuencia para cada sensor. Pulsamos *Next*.

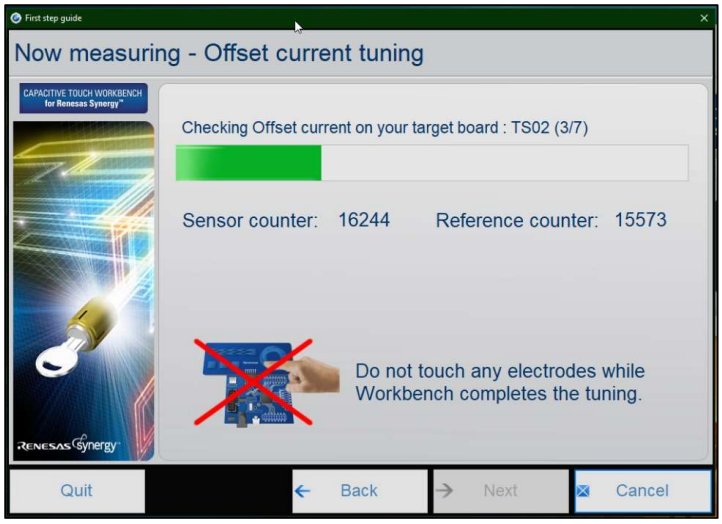


Figura 7-8 Medición automática.

10. El programa nos pedirá ahora que toquemos los botones y slider, para tomar muestras y determinar el umbral de detección de cada sensor. El proceso consistirá en tocar varias veces cada botón (según nos solicite el programa) y en deslizar un dedo sobre el slider varias veces. Al terminar, mostrará los resultados para los botones, pero no para el slider.

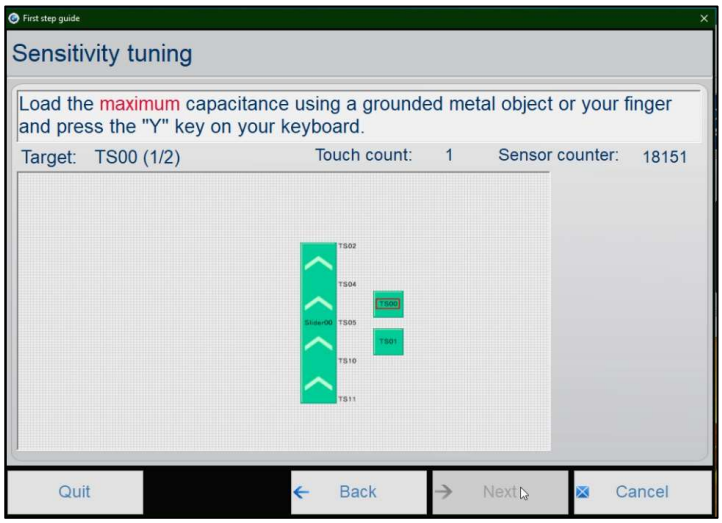


Figura 7-9 Calibración de sensibilidad

11. Finalmente nos indicará que ha guardado la configuración en el directorio de nuestra aplicación (que seleccionamos previamente en el paso 1).

(*): Si invertimos este orden, obtendremos valores contrarios a los esperados en nuestra aplicación. Podremos solucionarlo fácilmente con una linea de código al guardar el valor de la posición leído. En nuestro caso, el valor de la posición varía entre 0 y 500, por lo que, al guardar el valor leído, guardaremos 500 - <valor leído>.

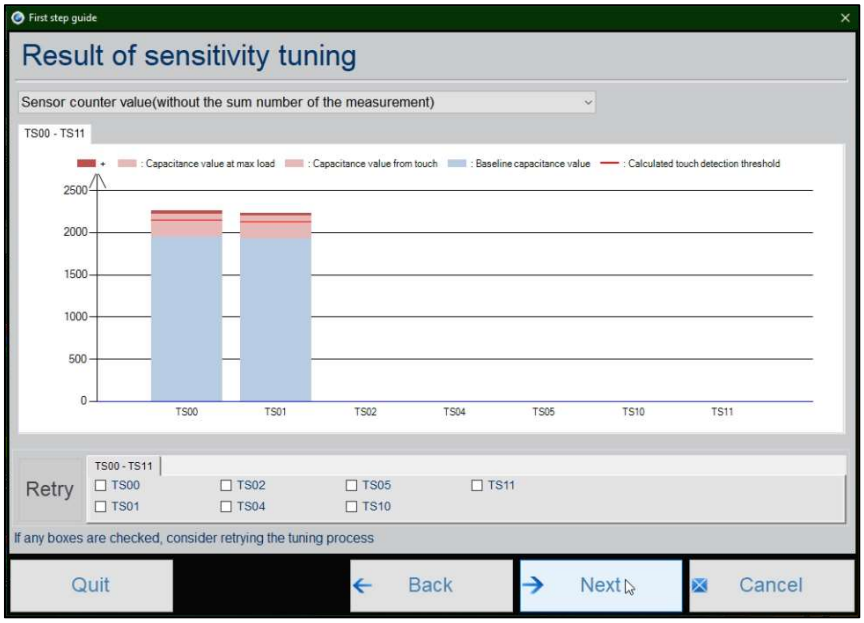


Figura 7-10 Resultado de la calibración.

8 CREACIÓN DE INTERFACES CON GUIX STUDIO

Descripción del software GUIX Studio y desarrollo de una interfaz gráfica

GUIX Studio es un software que nos permitirá diseñar una interfaz gráfica para nuestra aplicación de manera sencilla e intuitiva, y además, generará los archivos necesarios para incluirlo en nuestro código fuente.

8.1 Requisitos previos

Los requisitos previos necesarios para poder crear un proyecto en GUIX Studio son los siguientes:

- Tener creado un proyecto en e2Studio conforme a los puntos 5.17 y 5.18.
- Tener instalado GUIX Studio.
- Tener descargados y copiados los archivos necesarios, descritos en el punto 5.18

8.2 Creación de un proyecto

Una vez tengamos nuestro proyecto creado en e2 Studio, y hayamos añadido y configurado los módulos necesarios, crearemos una carpeta llamada "gui" dentro de la carpeta "src" de nuestro proyecto.

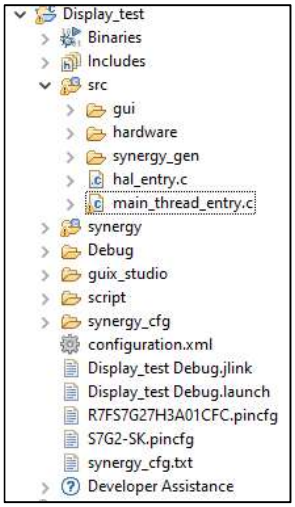


Figura 8-1 Creación de la carpeta “gui”.

De igual manera, en la carpeta raíz, crearemos una carpeta llamada “guix_studio”.

A partir de aquí, ejecutaremos GUIX Studio.

Nos iremos a *Project>New Project* para crear un nuevo proyecto.

Lo primero que nos pedirá el asistente es el nombre del proyecto y su directorio.

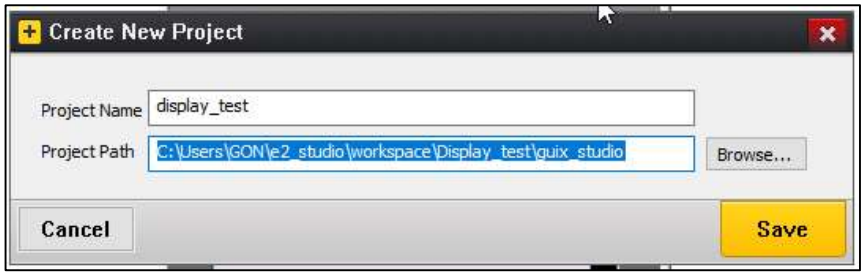


Figura 8-2 Asistente de creación del proyecto.

El directorio debe ser la carpeta “guix_studio” creada anteriormente.

Tras rellenar los dos campos, pulsaremos *Save*.

Nos aparecerá a continuación otra ventana, donde debemos seleccionar las opciones que aparecen en la Figura 8-3

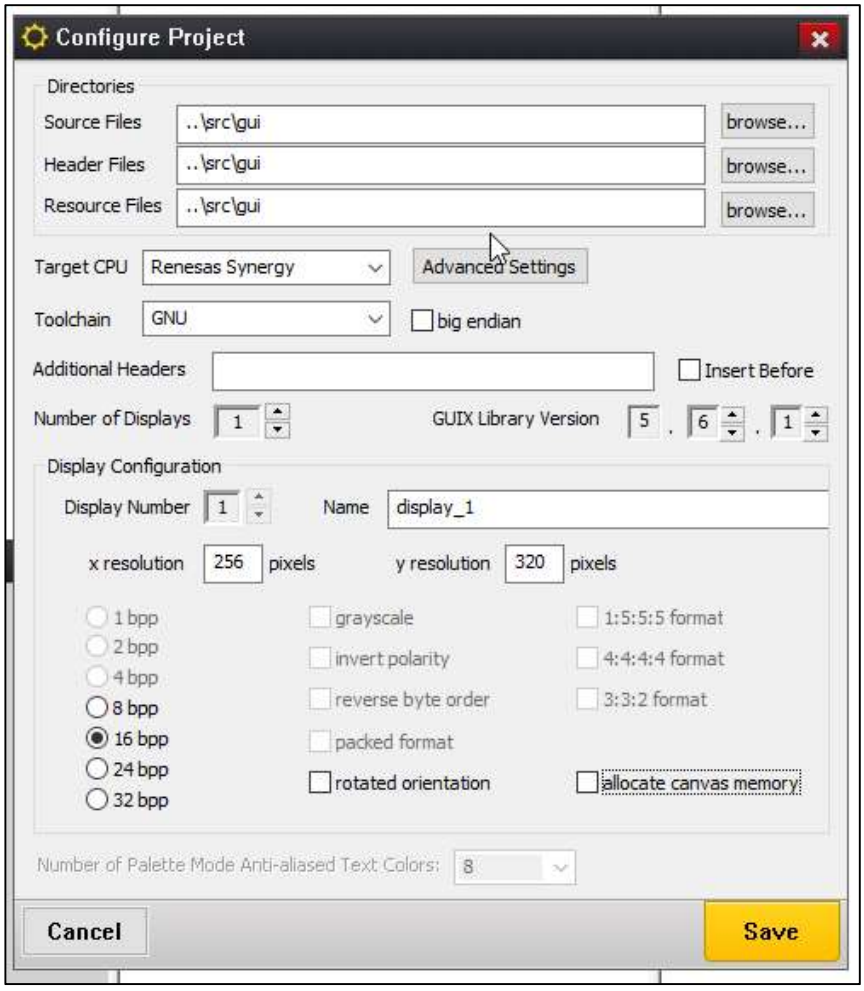


Figura 8-3 Asistente de creación del proyecto.

Es importante que en los directorios de los ficheros fuente, cabeceras y recursos pongamos "..\src\gui".

Pulsaremos *Save* de nuevo y nos creará el proyecto.

Nos abrirá automáticamente el proyecto y veremos que nos ha creado la primera ventana.

Podremos añadir elementos al proyecto pulsando el botón derecho sobre el árbol que encontramos en la zona izquierda de la pantalla.

Podemos incluir ventanas nuevas, botones, cuadros de textos e indicadores, entre otros.

Debajo del árbol, encontramos las propiedades del elemento seleccionado, donde configuraremos el mismo.

En medio de la pantalla, encontraremos una representación del display, donde visualmente veremos donde incluimos cada elemento y como se verá en nuestra aplicación.

Por último, en la zona derecha de la pantalla, encontraremos los recursos disponibles en nuestro proyecto. Desde las fuentes incluidas hasta las imágenes incluidas en nuestro proyecto.

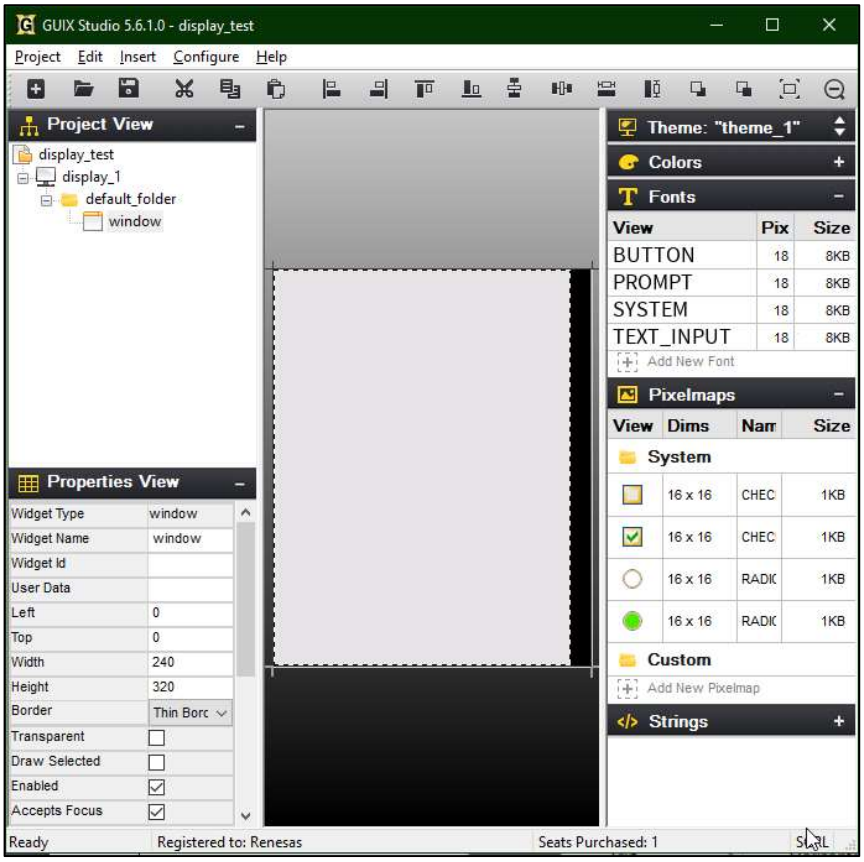


Figura 8-4 Ventana principal de GUIX Studio

A pesar de que, al generar nuestro proyecto, hemos seleccionado una pantalla de 256 px de anchura, la anchura de la pantalla incluida en la placa es de 240px, por lo que cambiaremos la anchura de la ventana a 240px.

En los ajustes del proyecto **siempre** mantendremos los 256px, al igual que en la configuración del proyecto en e2 Studio.

8.3 Elementos principales

En GUIX Studio, existe una gran variedad de elementos o *widgets* que podemos incluir en nuestro proyecto.

Cada uno de ellos contiene una serie de propiedades, entre las cuales encontramos el tamaño y posición del elemento entre otros.

Las propiedades más importantes en cualquier elemento de GUIX son las siguientes:

- *Widget Name*: Es el nombre con el que se identifica a cada elemento y se diferencia de otros. No permite tener el mismo nombre para dos elementos distintos.
- *Widget ID*: Es el identificador del tipo de elemento. Se usa desde e2 Studio para generar los eventos que queremos dentro de cada ventana. Se pueden repetir, pero no se recomienda hacerlo dentro de la misma ventana.
- *Left/Top/Width/Height*: Determinan la posición y tamaño del elemento en la pantalla.
- *Event Function*: La función que ejecutará el microprocesador cuando se genere un evento relacionado con el elemento, como, por ejemplo, una pulsación sobre el mismo. Se recomienda usar este campo solo dentro de cada ventana, pero no para el resto de elementos. La función recibirá el ID del elemento

pulsado dentro de la ventana y actuará en consecuencia, como puede ser realizar una acción tras pulsar un botón en la pantalla.

Los principales elementos que vamos a desarrollar a continuación son los siguientes:

- *Window*
- *Button / Text Button / Pixelmap Button*
- *Prompt / Numeric Pixelmap Prompt / Numeric Pixelmap Prompt*
- *Progress Bar / Radial Progress Bar*
- *Slider / Radial Slider*
- *Circular Gauge*

Para incluir un nuevo elemento, debemos hacer clic derecho sobre el árbol del proyecto. Aparecerá entonces un menú, donde podremos seleccionar el elemento que deseemos.

A continuación, vamos a mostrar los distintos elementos y qué funciones debemos usar para modificarlos en tiempo de ejecución. Tras cada función ejecutada, **siempre** ejecutaremos la siguiente función:

- gx_system_canvas_refresh(): Refrescará la pantalla con los cambios realizados previamente.

8.3.1 Elemento Window

Este es el elemento más básico, en el que incluiremos el resto de elementos (botones, cuadros de texto, imágenes, sliders, indicadores, barras de progreso, etc.).

Adicionalmente, será responsable del procesado de los eventos.

Las propiedades de las ventanas se muestran en la Figura 8-5.

Dentro de la ventana Propiedades, encontramos distintos campos para ajustar el tamaño y posición de la ventana, cambiar el color de la misma, incluir un marco, ponerle un fondo a partir de los *Pixelmaps* incluidos en el proyecto, etc.

La propiedad más importante es, como hemos comentado anteriormente, la *Event Function*. Escribiremos en dicho campo la función que actuará sobre los eventos de la pantalla, mientras nos encontremos dentro de esta ventana. Dicha función debe estar definida en nuestro proyecto en e2Studio.

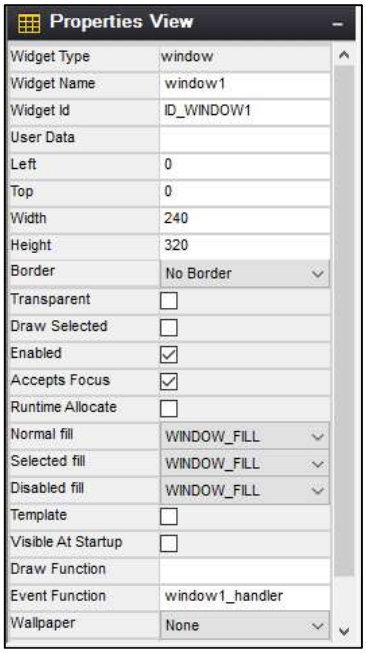


Figura 8-5 Propiedades de una ventana

8.3.2 Elementos Button

El nombre de estos elementos es bastante descriptivo, se trata de un botón en la pantalla.

Las propiedades de las ventanas se muestran en la Figura 8-6.

Dentro de la ventana Propiedades, encontramos distintos campos para ajustar el tamaño y posición del botón, cambiar el color del mismo, incluir un marco, cambiarle el texto (si es un *Text Button*) o la imagen a partir de los *Pixelmaps* incluidos en el proyecto (si es un *Pixelmap Button*), etc.

La propiedad más importante es el *Widget ID*. Este elemento debe ser distinto al *Widget ID* del resto de los elementos incluidos en la misma ventana. Así, desde la función que actúa sobre los eventos de la ventana en la que se encuentra el botón, podemos identificar el botón pulsado, y actuar en consecuencia.

Los elementos *Text Button* y *Pixelmap Button* difieren del elemento básico en la posibilidad de incluir texto dentro del botón, en el primer caso, y la posibilidad de mostrar el botón como una imagen.

En el elemento *Text Button*, encontraremos los campos *String ID* y *Text*, donde podremos seleccionar el texto deseado a incluir.

En el elemento *Pixelmap Button*, encontraremos los campos *Normal Pixelmap*, *Selected Pixelmap* y *Disabled Pixelmap*, donde podremos seleccionar la imagen que mostrará el botón según el estado en el que se encuentre (normal, pulsado o deshabilitado).

En tiempo de ejecución, podemos usar las funciones gx_button_select y gx_button_deselect para cambiar el estado del botón a pulsado/no pulsado. Se podrían usar por ejemplo para asociar un botón físico y un botón en la pantalla.

Los prototipos de las funciones son las siguientes:

- UINT gx_button_select (GX_BUTTON *button): donde *button* es el nombre que le hemos dado en GUIX Studio. Genera automáticamente el evento.
- gx_button_deselect(GX_BUTTON *button, GX_BOOL gen_event); donde *button* es el nombre que le hemos dado en GUIX Studio, y *gen_event* indica que se debe generar un evento si su valor es

GX_TRUE. De lo contrario, usaremos GX_FALSE.

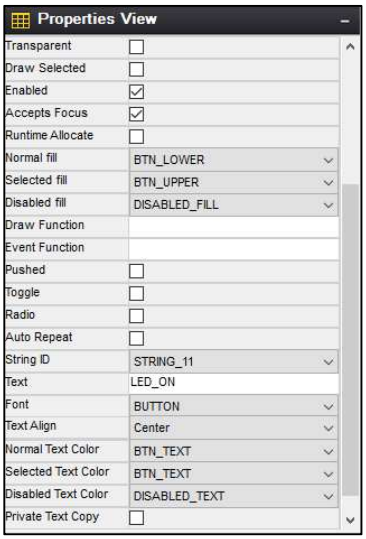


Figura 8-6 Propiedades de un botón.

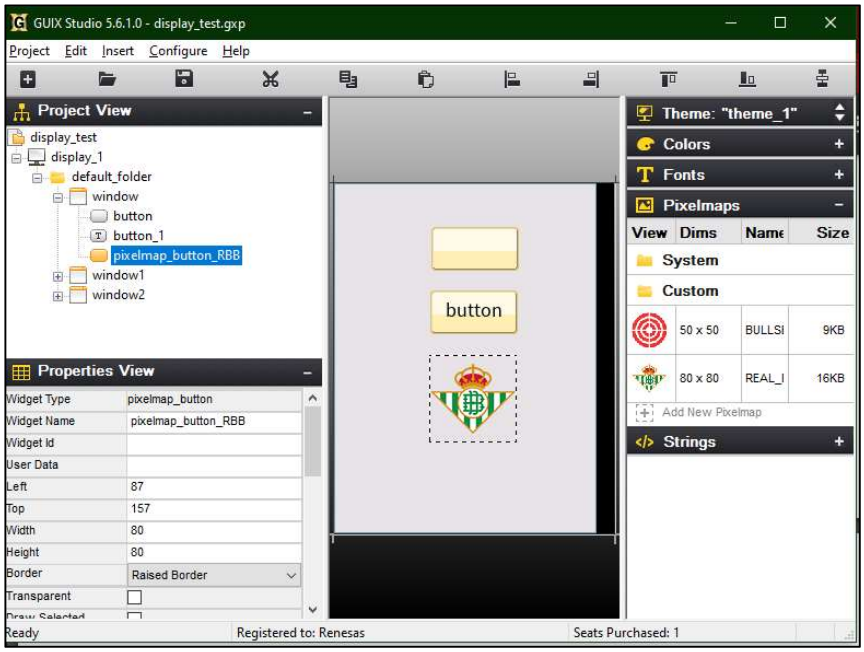


Figura 8-7 Ejemplos de botones

8.3.3 Elementos Prompt

Estos elementos son cuadros de texto.

Las propiedades de las ventanas se muestran en la Figura 8-8.

Dentro de la ventana Propiedades, encontramos distintos campos para ajustar el tamaño y posición del cuadro de texto, cambiar el color del mismo, incluir un marco, cambiarle el texto o el valor numérico (si es un *Numeric Prompt)* o la imagen a partir de los *Pixelmaps* incluidos en el proyecto (si es un *Pixelmap Numeric Prompt)*, etc.

La propiedad más importante es el *Widget Name*. Sabiendo el nombre del objeto, podremos cambiar el texto o valor en tiempo de ejecución.

Los elementos *Numeric Prompt* y *Pixelmap Numeric Prompt* difieren del elemento básico en la posibilidad de incluir un valor numérico (entero), en el primer caso, y la posibilidad de mostrar una imagen como fondo del cuadro de texto.

En el elemento *Prompt*, encontraremos los campos *String ID* y *Text*, donde podremos seleccionar el texto deseado a incluir.

En el elemento *Numeric Prompt*, encontraremos el campo *Numeric Value*, donde introduciremos el valor numérico inicial.

En el elemento *Pixelmap Numeric Prompt*, encontraremos los campos *Fill Pixmap* y *Sel Pixmap*, donde podremos seleccionar la imagen que mostrará el botón según el estado en el que se encuentre (normal o seleccionado).

En tiempo de ejecución, podemos usar las funciones gx_button_select y gx_button_deselect para cambiar el estado del botón a pulsado/no pulsado. Se podrían usar por ejemplo para asociar un botón físico y un botón en la pantalla.

Los prototipos de las funciones son las siguientes:

- gx_prompt_text_set_ext (GX_PROMPT *prompt, GX_STRING *string): donde *prompt* es el nombre del cuadro de texto y *string* es una estructura del tipo GX_STRING que contiene el texto y su longitud. Sus miembros son los siguientes:
    - GX_CONST GX_CHAR *gx_string_ptr: Puntero a la cadena de caracteres. Podemos definirla directamente sin tener que apuntar a otra variable. Por ejemplo; LED_str.gx_string_ptr = "LED apagado";
    - UINT gx_string_length: Es la longitud de la cadena anterior, sin contar el carácter nulo.
- gx_numeric_prompt_value_set (GX_NUMERIC_PROMPT *prompt, INT value): donde *prompt* es el nombre del cuadro de texto y *value* es el nuevo valor que queremos mostrar.
- gx_numeric_pixelmap_prompt_value_set(GX_NUMERIC_PIXELMAP_PROMPT *prompt, INT value): donde *prompt* es el nombre del cuadro de texto y *value* es el nuevo valor que queremos mostrar.

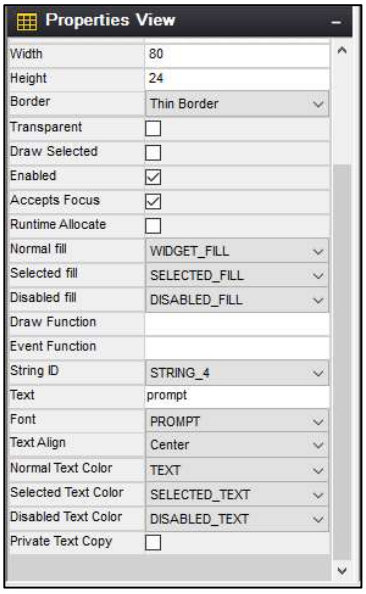


Figura 8-8 Propiedades de un cuadro de texto

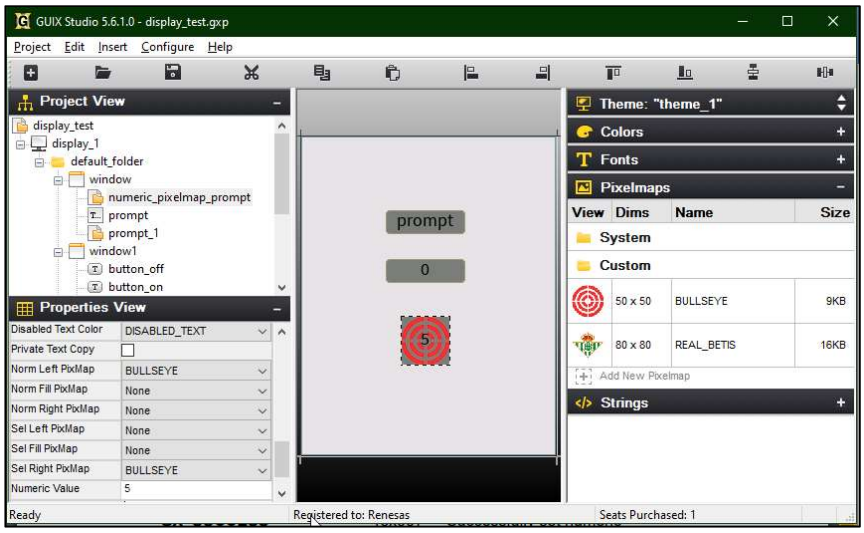


Figura 8-9 Ejemplos de cuadros de texto

8.3.4 Elementos Progress Bar

Estos elementos son barras de progreso.

Las propiedades de las ventanas se muestran en la Figura 8-10.

Dentro de la ventana Propiedades, encontramos distintos campos para ajustar el tamaño y posición de la barra de progreso, cambiar los colores de la misma, o del fondo, elegir mostrar el valor o no, cambiar el valor numérico de inicio, etc.

La propiedad más importante es el *Widget Name*. Sabiendo el nombre del objeto, podremos cambiar su valor en tiempo de ejecución.

El elemento *Radial Progress Bar* difiere del elemento básico en la forma en la forma de la barra. En este caso,

según incrementa el valor, la barra describe una circunferencia, mientras que el elemento básico es una barra de progreso recta

En el elemento *Radial Progress Bar*, encontraremos los campos *Anchor Value* y *Brush Width* donde seleccionaremos el punto de inicio de la barra de progreso radial, y el ancho de la misma.

En tiempo de ejecución, podemos usar las gx_progress_bar_value_set y gx_radial_progress_bar_value_set para cambiar el valor de las mismas.

Los prototipos de la función son las siguientes:

- gx_progress_bar_value_set (GX_PROGRESS_BAR *progress_bar, INT value): donde *progress_bar* es el nombre de la barra de progreso y *value* es el nuevo valor que queremos mostrar.
- gx_radial_progress_bar_value_set(GX_RADIAL_PROGRESS_BAR *progress_bar, GX_VALUE value): donde *progress_bar* es el nombre de la barra de progreso y *value* es el nuevo valor que queremos mostrar.

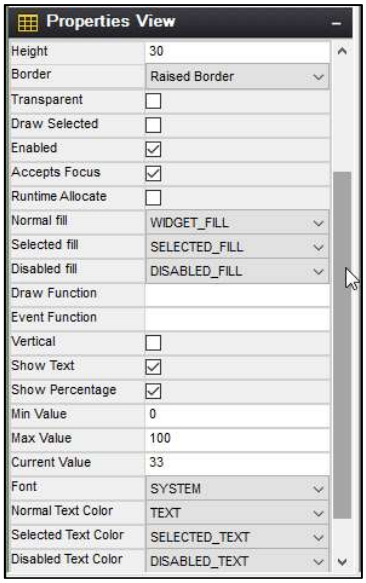


Figura 8-10 Propiedades de una barra de progreso

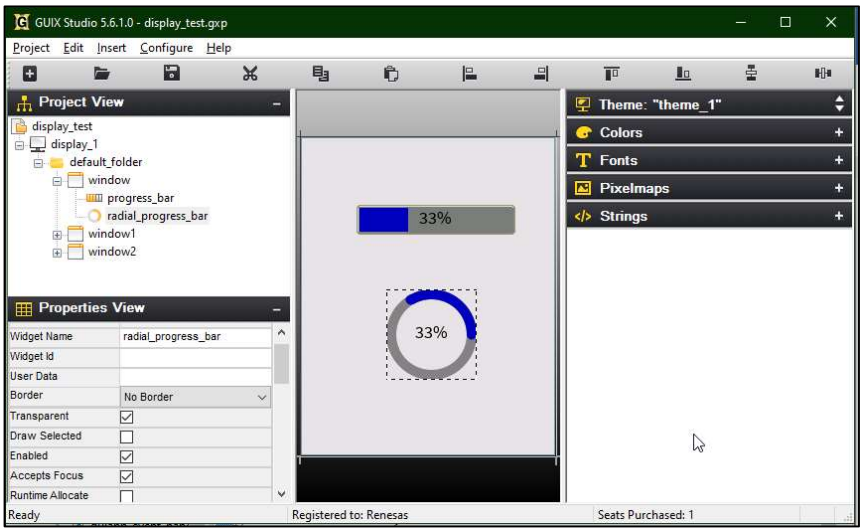


Figura 8-11 Ejemplos de barra de progreso

8.3.5 Elementos Slider

Estos elementos son deslizadores. Deslizaremos el dedo sobre el elemento para fijar un valor, y usar este para realizar algo (por ejemplo, cambiar el duty cycle de un PWM).

Las propiedades de las ventanas se muestran en la Figura 8-12.

Dentro de la ventana Propiedades, encontramos distintos campos para ajustar el tamaño y posición del deslizador, cambiar los colores del mismo, ajustar las marcas, ajustar los valores máximo y mínimo, el incremento mínimo, el tamaño de la aguja, etc.

La propiedad más importante es el *Widget Name*. Sabiendo el nombre del objeto, podremos obtener su valor en tiempo de ejecución.

En tiempo de ejecución, podemos usar la función gx_slider_needle_position_get para obtener el valor del deslizador.

Sin embargo, en esta ocasión, es más sencillo acceder directamente a la estructura donde se guardan los valores relacionados con el deslizador, (valor máximo, mínimo, actual, incremento máximo, etc.).

De la siguiente manera, accederemos a dicho valor:

<ventana >.<ventana>_<nombre del deslizador>.gx_slider_info.gx_slider_info_current_val

Así, podemos, por ejemplo, escribir la siguiente sentencia:

g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl,
window2.window2_slider.gx_slider_info.gx_slider_info_current_val, TIMER_PWM_UNIT_PERCENT, 1);

en la que cambiaremos directamente el duty cycle de nuestro PWM según el valor del deslizador. En este caso, el valor del deslizador debe oscilar entre 0 y 100.

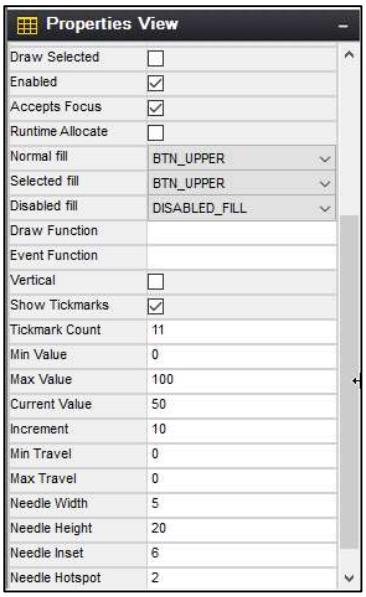


Figura 8-12 Propiedades de un deslizador

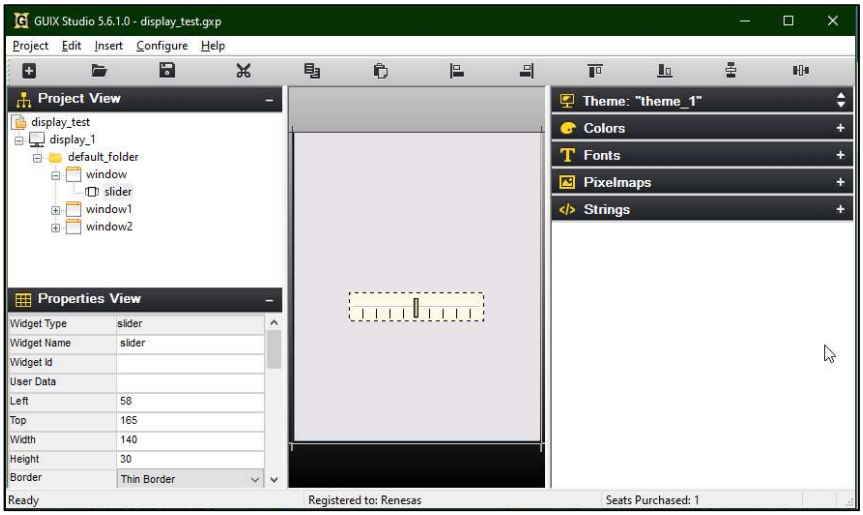


Figura 8-13 Ejemplo de un deslizador

8.3.6 Elemento Circular Gauge

Estos elementos son medidores.

Las propiedades de las ventanas se muestran en la Figura 8-14.

Dentro de la ventana Propiedades, encontramos distintos campos para ajustar el tamaño y posición del medidor, tamaño y posición de la aguja, cambiar el *sprite* del medidor o la aguja, seleccionar el ángulo de inicio, etc.

La propiedad más importante es el *Widget Name*. Sabiendo el nombre del objeto, podremos cambiar su valor en tiempo de ejecución.

También son muy importantes los *Pixelmap*, tanto del fondo como de la aguja, así como las *Needle Xpos, Ypos, Xcor, Ycor*, que indican donde se encuentra la aguja respecto al elemento principal, y donde se encuentra el centro de la aguja.

En tiempo de ejecución, podemos usar función gx_circular_gauge_angle_set para poder cambiar el ángulo de la aguja.

El prototipo de la función es la siguiente:

- gx_progress_bar_value_set (GX_CIRCULAR_GAUGE *gauge, INT angle); donde *gauge* es el nombre del medidor circular y *angle* es el nuevo ángulo que le vamos a dar a la aguja.

Adicionalmente, es muy probable que tengamos que usar las funciones tx_byte_allocate y tx_byte_release, que son las equivalentes a las funciones malloc y free, de la biblioteca estándar de C. Es muy posible que cuando intentemos cambiar el ángulo de la aguja del medidor, esta desaparezca, o se comporte de forma errática. Esto se debe a que la memoria asignada es menor a la necesaria. Para solucionarlo, incluiremos estas funciones en nuestro código.

A continuación, mostramos como implementarlas:

- gx_user_heap.c:

```
#include "gx_user_heap.h"

static VOID * gx_user_malloc(ULONG size);
static VOID   gx_user_free(VOID * p_mem);
static TX_BYTE_POOL _gx_user_heap_pool;
```

```
static UCHAR _gx_user_heap_area[GX_USER_HEAP_SIZE] \
BSP_ALIGN_VARIABLE(8) \
BSP_PLACE_IN_SECTION(GX_USER_HEAP_SECTION);

UINT gx_user_heap_setup(VOID)
{
    UINT status = tx_byte_pool_create(&_gx_user_heap_pool, "GUIX User
Heap",
_gx_user_heap_area, GX_USER_HEAP_SIZE);
    if (TX_SUCCESS == status)
    {
        status = gx_system_memory_allocator_set(gx_user_malloc,
gx_user_free);;
    }
    return status;
}


VOID * gx_user_malloc(ULONG size)
{
    VOID * p_mem;
    if (tx_byte_allocate(&_gx_user_heap_pool, &p_mem, size,
TX_NO_WAIT))
    {
        p_mem = NULL;
    }
    return p_mem;
}

VOID gx_user_free(VOID * p_mem)
{
    tx_byte_release(p_mem);
}
```

- gx_user_heap.h:

```
#ifndef GX_USER_HEAP_H_
#define GX_USER_HEAP_H_

#include "hal_data.h"
#include "tx_api.h"
#include "gx_api.h"

#define GX_USER_HEAP_SECTION        (".bss")
#define GX_USER_HEAP_SIZE           (256 * 320)

UINT gx_user_heap_setup(VOID);

#endif /* GX_USER_HEAP_H_ */
```

Podemos encontrar ambos archivos en el siguiente enlace https://renesasrulz.com/synergy/f/synergy---forum/9486/i-want-to-control-circular_gauge-in-guix/31749#31749.

Para que GUIX haga uso de ellas, debemos llamar a la función gx_user_heap_setup en nuestro hilo principal, tras ejecutar gx_system_initialize.

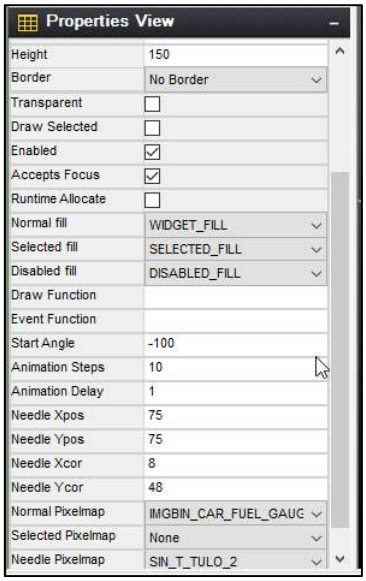


Figura 8-14 Propiedades de un *circular gauge*

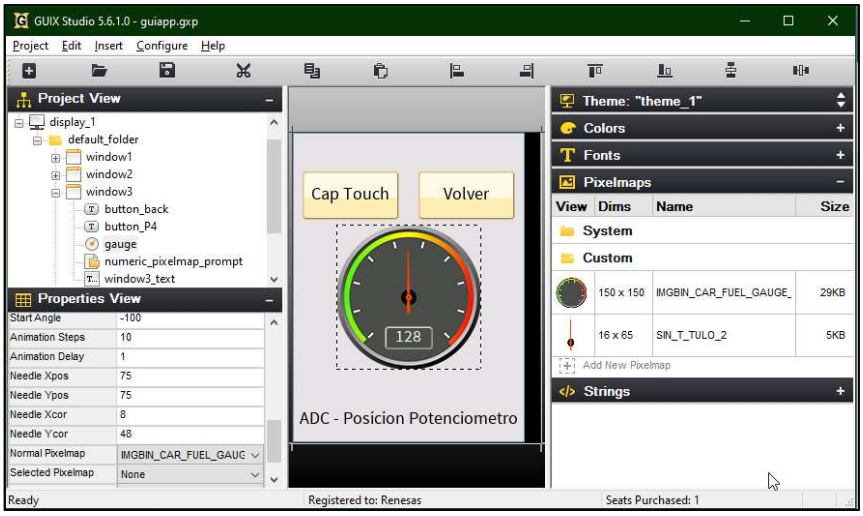


Figura 8-15 Ejemplo de *circular gauge*

8.4 Generación de especificaciones y recursos de GUIX

Una vez tengamos nuestro proyecto terminado en GUIX Studio, y queramos generar los archivos necesarios para usarlos en nuestro proyecto de e2 Studio, guardaremos el proyecto e iremos a *Project > Generate All Output Files*. Pulsaremos el botón *Generate* y ya tendremos nuestros archivos generados en la carpeta *gui* que hemos creado al principio.

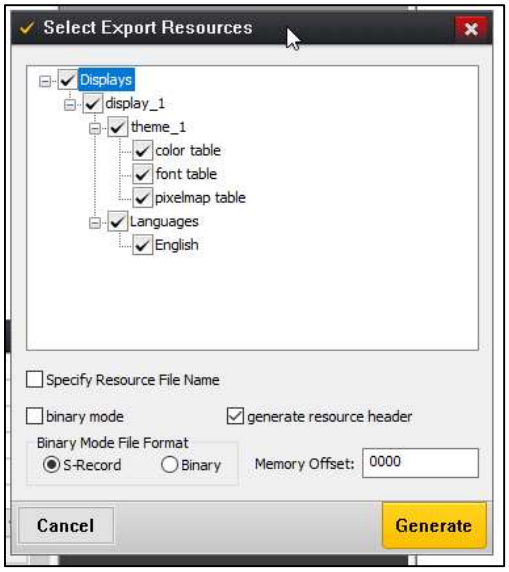


Figura 8-16 Ventana de generación de recursos y especificaciones.

8.5 Ejemplo de programación

Realice un proyecto en e2Studio y GUIX Studio, en el que:

- En una ventana se maneje el estado de un LED mediante botones en la pantalla y muestre su estado actual en un cuadro de texto.
- En una segunda ventana se muestre las coordenadas X e Y de la última pulsación en la pantalla.

8.5.1 GUIX Studio

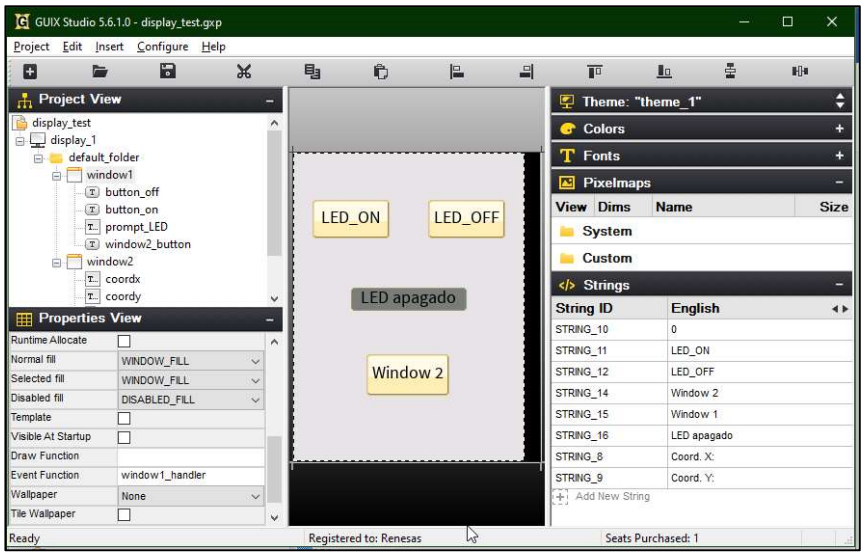


Figura 8-17 Ventana 1

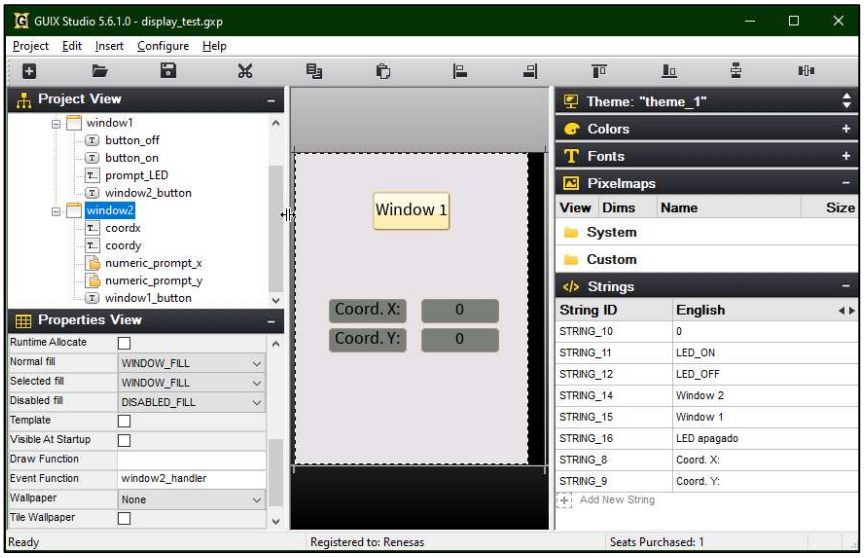


Figura 8-18 Ventana 2

8.5.2 HAL_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
void hal_entry(void){
    /* TODO: add your own code here */
}
```

8.5.3 main_thread_entry.c

```
/* Main Thread entry function */
#include "main_thread.h"
#include "bsp_api.h"
#include "gx_api.h"
#include "gui/display_test_specifications.h"
#include "gui/display_test_resources.h"
#include "hardware/lcd.h"

#define RED_LED_PIN IOPORT_PORT_06_PIN_01
#define GREEN_LED_PIN IOPORT_PORT_06_PIN_00
#define ORANGE_LED_PIN IOPORT_PORT_06_PIN_02
#define ON IOPORT_LEVEL_LOW
#define OFF IOPORT_LEVEL_HIGH

static bool ssp_touch_to_guix(sf_touch_panel_payload_t * p_touch_payload,
GX_EVENT * g_gx_event);
void main_thread_entry(void);

void circulo(GX_WINDOW *widget, int x, int y);

static GX_EVENT g_gx_event;

GX_WINDOW_ROOT * p_window_root;
extern GX_CONST GX_STUDIO_WIDGET *display_test_widget_table[];


void main_thread_entry(void) {
```

```
      ssp_err_t        err;
      sf_message_header_t * p_message = NULL;
      UINT status = TX_SUCCESS;

      g_ioport.p_api->pinWrite(RED_LED_PIN, OFF); //APAGAMOS el LED rojo
      g_ioport.p_api->pinWrite(GREEN_LED_PIN, OFF); //APAGAMOS el LED verde
      g_ioport.p_api->pinWrite(ORANGE_LED_PIN, OFF); //APAGAMOS el LED naranja

    /* Initializes GUIX. */
    status = gx_system_initialize();

    /* Initializes GUIX drivers. */
    err = g_sf_el_gx.p_api->open (g_sf_el_gx.p_ctrl, g_sf_el_gx.p_cfg);

    gx_studio_display_configure ( DISPLAY_1, g_sf_el_gx.p_api->setup,
LANGUAGE_ENGLISH, DISPLAY_1_THEME_1, &p_window_root );

    err = g_sf_el_gx.p_api->canvasInit(g_sf_el_gx.p_ctrl, p_window_root);

    // Create the widgets we have defined with the GUIX data structures and
resources.
    GX_CONST GX_STUDIO_WIDGET ** pp_studio_widget =
&display_test_widget_table[0];
    GX_WIDGET * p_first_screen = NULL;

    while (GX_NULL != *pp_studio_widget) {
        // We must first create the widgets according the data generated in
GUIX Studio.

        // Once we are working on the widget we want to see first, save the
pointer for later.
        if (0 == strcmp("window1", (char*)(*pp_studio_widget)->widget_name))
        {
            gx_studio_named_widget_create((*pp_studio_widget)->widget_name,
(GX_WIDGET *)p_window_root, GX_NULL);
        } else {
            gx_studio_named_widget_create((*pp_studio_widget)->widget_name,
GX_NULL, GX_NULL);
        }
        // Move to next top-level widget
        pp_studio_widget++;
    }
    // Attach the first screen to the root so we can see it when the root is
shown
   gx_widget_attach(p_window_root, p_first_screen);

    /* Shows the root window to make it and patients screen visible. */
    status = gx_widget_show(p_window_root);

    /* Lets GUIX run. */
    status = gx_system_start();

    /** Open the SPI driver to initialize the LCD (SK-S7G2) **/
    err = g_spi_lcdc.p_api->open(g_spi_lcdc.p_ctrl, (spi_cfg_t
*)g_spi_lcdc.p_cfg);

    /** Setup the ILI9341V (SK-S7G2) **/
    ILI9341V_Init();

    /* Controls the GPIO pin for LCD ON (DK-S7G2, PE-HMI1) */
    err = g_ioport.p_api->pinWrite(IOPORT_PORT_10_PIN_03, IOPORT_LEVEL_HIGH);
```

```
    /* Opens PWM driver and controls the TFT panel back light (DK-S7G2, PE-
HMI1) */
      while(1){
            bool new_gui_event = false;

            err = g_sf_message0.p_api->pend(g_sf_message0.p_ctrl,
&main_thread_message_queue, (sf_message_header_t **) &p_message,
TX_WAIT_FOREVER);
            if (err)
            {
                  /** TODO: Handle error. */
            }

            switch (p_message->event_b.class_code)
            {
            case SF_MESSAGE_EVENT_CLASS_TOUCH:
            {
                  switch (p_message->event_b.code)
                  {
                  case SF_MESSAGE_EVENT_NEW_DATA:
                  {
                        /** Translate an SSP touch event into a GUIX event */
                        new_gui_event =
ssp_touch_to_guix((sf_touch_panel_payload_t*)p_message, &g_gx_event);
                  }
                  default:
                        break;
                  }
                  break;
            }
            default:
                  break;
            }
            /** Message is processed, so release buffer. */
            err = g_sf_message0.p_api->bufferRelease(g_sf_message0.p_ctrl,
(sf_message_header_t *) p_message, SF_MESSAGE_RELEASE_OPTION_FORCED_RELEASE);

            /** Post message. */
            if (new_gui_event) {
                  gx_system_event_send(&g_gx_event);
            }
      }
}

static bool ssp_touch_to_guix(sf_touch_panel_payload_t * p_touch_payload,
GX_EVENT * gx_event){
      bool send_event = true;

      switch (p_touch_payload->event_type)
      {
      case SF_TOUCH_PANEL_EVENT_DOWN:
            gx_event->gx_event_type = GX_EVENT_PEN_DOWN;
            break;
      case SF_TOUCH_PANEL_EVENT_UP:
            gx_event->gx_event_type = GX_EVENT_PEN_UP;
            break;
      case SF_TOUCH_PANEL_EVENT_HOLD:
      case SF_TOUCH_PANEL_EVENT_MOVE:
            gx_event->gx_event_type = GX_EVENT_PEN_DRAG;
            break;
      case SF_TOUCH_PANEL_EVENT_INVALID:
            send_event = false;
```

```
            break;
        default:
            break;
        }
        if (send_event){
            /** Send event to GUI */
            gx_event->gx_event_sender = GX_ID_NONE;
            gx_event->gx_event_target = 0;
            gx_event->gx_event_display_handle = 0;

            gx_event->gx_event_payload.gx_event_pointdata.gx_point_x =
p_touch_payload->x;
            gx_event->gx_event_payload.gx_event_pointdata.gx_point_y =
(GX_VALUE)(320 - p_touch_payload->y);  // SK-S7G2

            gx_numeric_prompt_value_set(&window2.window2_numeric_prompt_x,
p_touch_payload->x);
            gx_numeric_prompt_value_set(&window2.window2_numeric_prompt_y,
(320 - p_touch_payload->y));

            gx_system_canvas_refresh();
            if (gx_event->gx_event_type != GX_EVENT_PEN_DRAG )
circulo(&window2, p_touch_payload->x, (320 - p_touch_payload->y));
        }
        return send_event;
}

void g_lcd_spi_callback(spi_callback_args_t * p_args){
    (void)p_args;
    tx_semaphore_ceiling_put(&g_main_semaphore_lcdc, 1);
}

void circulo(GX_WINDOW *widget, int x, int y){
    GX_RECTANGLE drawto;
    GX_CANVAS *my_canvas;

    /* The default window drawing callback function is the default interface
effect drawing */
    gx_window_draw(widget);

    /* Define a rectangle, and subsequent 2D drawing functions are drawn
within the rectangle */
    gx_utility_rectangle_define(&drawto, 0, 0, 240, 320);

    /* Return to the canvas corresponding to the window */
    gx_widget_canvas_get(widget, &my_canvas);

    /* Starts drawing on the specified canvas.*/
    gx_canvas_drawing_initiate(my_canvas, widget, &drawto);

    gx_context_raw_line_color_set(0x00000000);
    gx_context_raw_fill_color_set(0x00000000);
    gx_context_brush_style_set(GX_BRUSH_OUTLINE);
    gx_context_brush_width_set(4);
    gx_canvas_circle_draw(x, y, 6);

    /* After drawing, it is used to force immediate drawing. Note that it
must be consistent with gx_canvas_drawing_initiate calls in pairs */
    gx_canvas_drawing_complete(my_canvas, GX_TRUE);
}
```

8.5.4 display_test_event_handlers.c

```
#include "gui/display_test_resources.h"
#include "gui/display_test_specifications.h"
#include "main_thread.h"

#define RED_LED_PIN IOPORT_PORT_06_PIN_01
#define GREEN_LED_PIN IOPORT_PORT_06_PIN_00
#define ORANGE_LED_PIN IOPORT_PORT_06_PIN_02
#define ON IOPORT_LEVEL_LOW
#define OFF IOPORT_LEVEL_HIGH

extern GX_WINDOW_ROOT * p_window_root;

volatile GX_STRING LED_str;

static UINT show_window(GX_WINDOW * p_new, GX_WIDGET * p_widget, bool
detach_old);

UINT window1_handler(GX_WINDOW *widget, GX_EVENT *event_ptr){
    UINT result = gx_window_event_process(widget, event_ptr);
    switch (event_ptr->gx_event_type){
        case GX_SIGNAL(ID_BUTTON_ON, GX_EVENT_CLICKED): //si se pulsa el
boton LED ON
            g_ioport.p_api->pinWrite(RED_LED_PIN, ON); //encendemos el LED
rojo
            LED_str.gx_string_ptr = "LED encendido";
            LED_str.gx_string_length = 13; //actualizamos el cuadro de texto
            gx_prompt_text_set_ext(&window1.window1_prompt_LED, &LED_str);
            gx_system_canvas_refresh();
            break;
        case GX_SIGNAL(ID_BUTTON_OFF, GX_EVENT_CLICKED): //si pulsan el boton
LED OFF
            g_ioport.p_api->pinWrite(RED_LED_PIN, OFF); //apagamos el LED
rojo
            LED_str.gx_string_ptr = "LED apagado";
            LED_str.gx_string_length = 11; //actualizamos el cuadro de texto
            gx_prompt_text_set_ext(&window1.window1_prompt_LED, &LED_str);
            gx_system_canvas_refresh();
            break;
        case GX_SIGNAL(ID_WINDOW2_BUTTON, GX_EVENT_CLICKED): //si pulsan el
boton "Window 2", nos vamos a la ventana 2
            show_window((GX_WINDOW*)&window2, (GX_WIDGET*)widget, true);
            break;
        default:
            gx_window_event_process(widget, event_ptr);
            break;
    }
    return result;
}

UINT window2_handler(GX_WINDOW *widget, GX_EVENT *event_ptr){
    UINT result = gx_window_event_process(widget, event_ptr);
    switch (event_ptr->gx_event_type){
        case GX_SIGNAL(ID_WINDOW1_BUTTON, GX_EVENT_CLICKED): //si pulsan el
boton "Window 1", nos vamos a la ventana 1
            show_window((GX_WINDOW*)&window1, (GX_WIDGET*)widget, true);
            break;
        default:
            result = gx_window_event_process(widget, event_ptr);
            break;
    }
```

```
    return result;
}

static UINT show_window(GX_WINDOW * p_new, GX_WIDGET * p_widget, bool
detach_old){
    UINT err = GX_SUCCESS;
    if (!p_new->gx_widget_parent)
    {
        err = gx_widget_attach(p_window_root, p_new);
    }
    else
    {
        err = gx_widget_show(p_new);
    }
    gx_system_focus_claim(p_new);

    GX_WIDGET * p_old = p_widget;
    if (p_old && detach_old)
    {
        if (p_old != (GX_WIDGET*)p_new)
        {
            gx_widget_detach(p_old);
    }
    return err;
}
```

9 APLICACIÓN DE DEMOSTRACIÓN

Descripción de una aplicación de demostración que incluye los periféricos anteriormente descritos

Con todos los elementos descritos anteriormente, hemos desarrollado una aplicación de demostración, en el que hemos incluido casi todos los elementos que hemos descrito en esta guía de programación.

La aplicación consiste en una interfaz gráfica a través de la cual el usuario interactúa con los distintos periféricos, dependiendo de la ventana de la interfaz gráfica en la que se encuentre.

Las ventanas que conforman la interfaz son las siguientes:

- Menú principal: Está formada por una serie de botones que nos llevará a las otras pantallas, en las cuales se interactúa con los periféricos.

Figura 9-1 Pantalla 1

- Ventana Timer/PWM: En esta pantalla encontramos un deslizador, que nos permitirá modificar el duty cycle de un PWM al que tenemos conectado un LED. Un cuadro de texto mostrará el valor (porcentual) del duty cycle. Encontramos también dos botones que nos permitirán ir a la siguiente pantalla, o volver al menú principal.

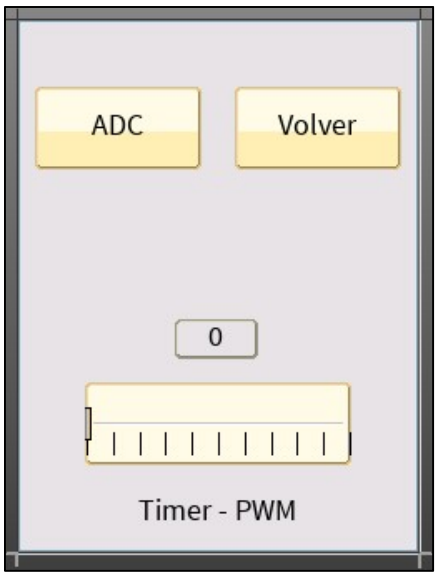


Figura 9-2 Pantalla 2

- Ventana ADC: En esta pantalla encontramos un medidor, el cual nos mostrará la posición de un potenciómetro conectado a uno de los pines asociados a un canal del conversor analógico-digital. Según movamos el potenciómetro, también lo hará el medidor. Dentro del mismo se encuentra también un cuadro de texto que muestra el valor convertido (con una resolución de 8 bits), en decimal. Encontramos también dos botones que nos permitirán ir a la siguiente pantalla, o volver al menú principal.

Figura 9-3 Pantalla 3

- Ventana Sensores Capacitivos: En esta pantalla encontramos dos botones asociados a los dos botones capacitivos, que se muestran pulsados cuando su análogo capacitivo lo esté. También encontramos una barra de progreso, que nos indicará la posición en la que se encuentra el deslizador capacitivo capacitivo, con un valor de entre 0 y 100. Encontramos también dos botones que nos permitirán ir a la siguiente pantalla, o volver al menú principal.

Figura 9-4 Pantalla 4

- Ventana Pantalla Táctil: En esta pantalla mostramos las coordenadas de las pulsaciones en la pantalla táctil, a través de sus correspondientes cuadros de texto. Encontramos también dos botones que nos permitirán ir a la siguiente pantalla, o volver al menú principal.

Figura 9-5 Pantalla 5

- Ventana DAC: En esta pantalla encontramos dos botones y una barra de progreso. Los botones incrementan o decrementan el valor de salida del conversor digital-analógico en incrementos del 10% del valor de tensión de Vcc (3.3V). Una barra de progreso mostrará, junto con un cuadro de texto superpuesto el valor de tensión aplicado al pin correspondiente. Encontramos también dos botones que nos permitirán ir a la siguiente pantalla, o volver al menú principal.

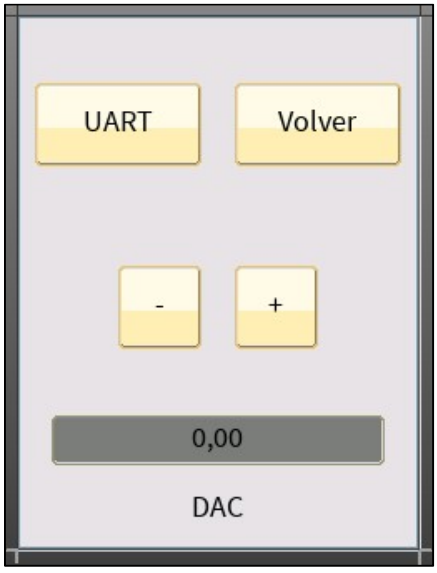


Figura 9-6 Pantalla 6

- Ventana UART: En esta pantalla encontramos el valor leído por UART y la respuesta programada, según la cadena de caracteres leída. Si la cadena leída corresponde a uno de los comandos programados, actuarán en consecuencia. La acción de estos comandos consiste en encender o apagar los LEDs integrados en la placa. Encontramos también un botón que nos permitirá volver al menú principal.

Figura 9-7 Pantalla 7

Para realizar esta aplicación, hemos dividido el proyecto en distintos hilos de programación, uno por periférico.

La excepción es el hilo que gobierna la pantalla y su sensor táctil resistivo, que conforman el hilo principal.

Cada hilo queda a la espera (bloqueante) de un flag configurado previamente usando la ventana de configuración del proyecto. Cada vez que se cambie de ventana, se cambian los estados de los flags asociados al hilo que bloquearemos y al que permitiremos ejecutar su código. De esta manera, hemos conseguido sincronizar la ejecución de cada hilo.

El único hilo que no se bloqueará nunca será el principal, ya que siempre lo necesitaremos para responder a los eventos generados por GUIX.

9.1 Funcionamiento de la aplicación

9.1.1 Hilo principal (main_thread_entry)

Este hilo es el más importante. En este hilo se inicializa los drivers de la pantalla, se crea el hilo GUIX y se controlan los eventos táctiles. El hilo GUIX es una caja negra y se crea al llamar a las funciones *gx_system_initialize* y *gx_system_start* durante la inicialización de los drivers. Durante la inicialización, se llama a las función *gx_user_heap_setup*, que se encarga de asignar la memoria necesaria al *widget* que lo necesite. En nuestro caso, lo usamos para asignar la memoria necesaria al *widget Circular Gauge*. Sin añadir esta función, la aguja "desaparece" al moverla de su posición inicial.

Tras la inicialización, entramos en un bucle while, donde se va a leer la cola de mensajes por la cual se reciben los eventos de la pantalla táctil. Tras una espera bloqueante, se reanuda la ejecución del hilo al generarse un evento táctil. Este es leído y se llama a una función que traduce a un tipo de evento que el hilo GUIX pueda reconocer y actuar en consecuencia. Junto con el tipo de evento, van algunos argumentos, como las coordenadas del evento. Adicionalmente, se ha incluido en esta función la lectura de un flag (ventana 5, pantalla táctil), para poder actualizar los cuadros de texto incluidos en dicha ventana, con los valores de las coordenadas de las pulsaciones.

Por último, se envía el evento al hilo GUIX.

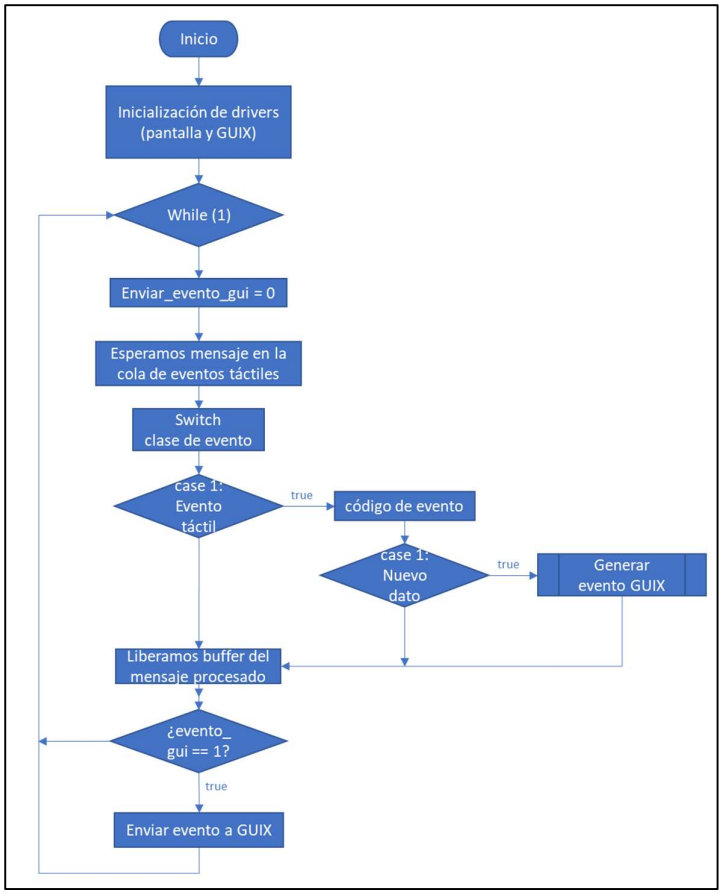


Figura 9-8 Hilo principal

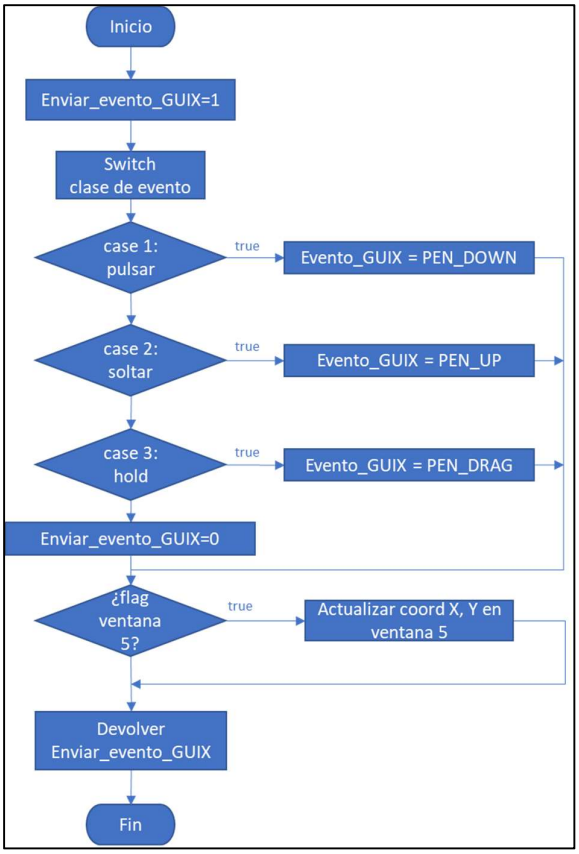


Figura 9-9 Función Genera Eventos GUIX

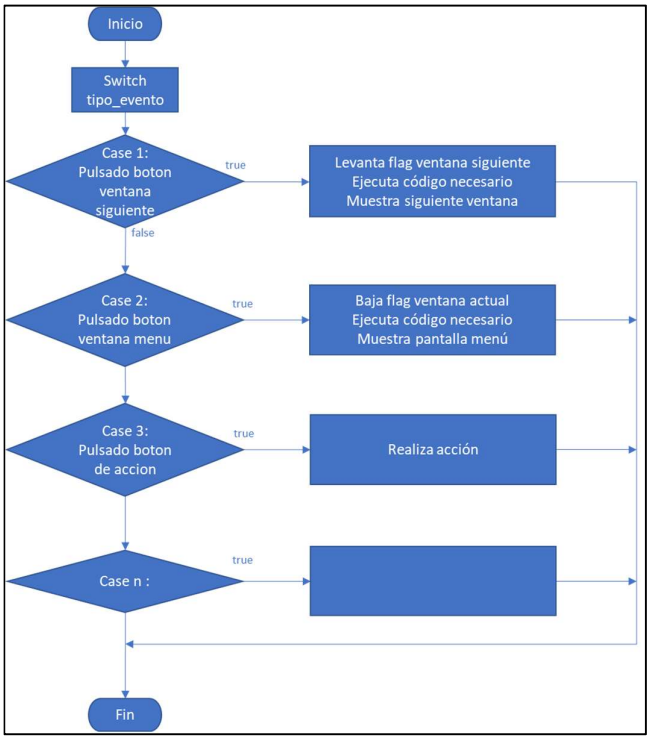


Figura 9-10 Manipulador de eventos (genérico)

El hilo GUIX, hace uso de unas funciones para saber como actuar según se pulsen los *widgets* de la pantalla.

En la Figura 9-10, incluimos un diagrama de flujo de una función genérica para manipular los eventos GUIX. Estas funciones comparten estructura y se compone principalmente de una sentencia *switch*-case, en la que según el tipo de evento, se actua de una manera u otra. Por ejemplo, al detectar una pulsación en cada uno de los botones que encontramos en la ventana menú principal, se activa el flag correspondiente y se ejecuta la función que cambia la ventana mostrada por la pantalla. Si es necesario, se ejecuta alguna función *enable* u *open* para desbloquear el uso del periférico en cuestión.

9.1.2 Hilo PWM

Este hilo se encarga de inicializar, configurar y actuar sobre un pin con salida PWM. Tras la inicialización y configuración del timer correspondiente, se entra en un bucle while, donde se va a esperar la activación del flag de la ventana 2. Una vez se activa el flag, se lee otro flag (sin espera bloqueante) que indica una actualización del valor del slider presente en la ventana 2, activado por la función *window2_handler*. Si se detecta la activación del flag, se actualiza el *duty cycle* del PWM y el valor porcentual del mismo, mostrado en el cuadro de texto presente en esta ventana. Finalmente, se desactiva el flag.

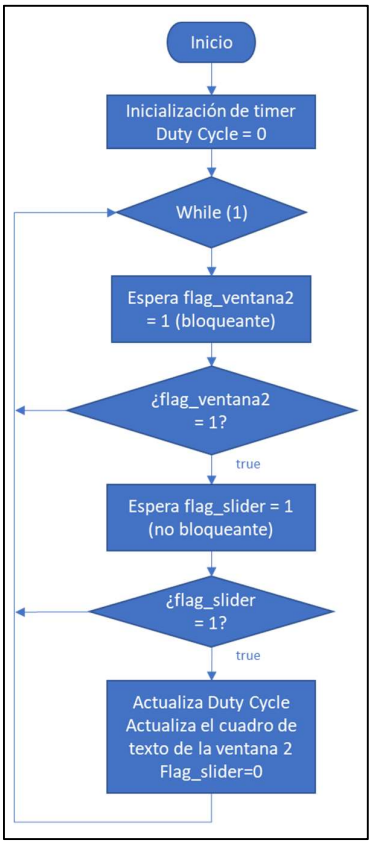


Figura 9-11 Hilo PWM

9.1.3 Hilo ADC

Este hilo se encarga de inicializar, configurar y leer el ADC. Tras la inicialización y configuración del conversor, se entra en un bucle while, donde se va a esperar la activación del flag de la ventana 3. Una vez se activa el flag, se lee el valor convertido, y se actualizan los *widgets* que se muestran en la ventana 3 (cuadro de texto con el valor convertido y posición de la aguja del *gauge*).

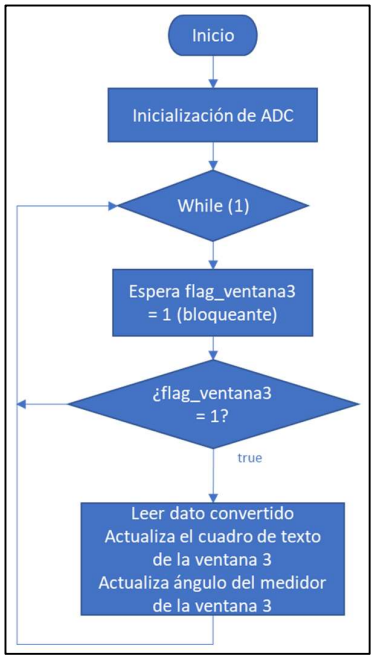


Figura 9-12 Hilo ADC

9.1.4 Hilo Touch

Este hilo se encarga de inicializar, configurar y leer los sensores capacitivos. Tras la inicialización y configuración de los sensores, se entra en un bucle while, donde se va a esperar la activación del flag de la ventana 4. Una vez se activa el flag, se leen 3 variables booleanas, que indican si se han pulsado los botones 1 y 2, y si el slider se ha pulsado. No es necesario usar *event flags* al ser variables que solo se modifican dentro del hilo. Tras la lectura de las variables, se actualiza el estado de los botones virtuales y de la barra de progreso que representa la posición del slider.

Los eventos generados por los sensores capacitivos generan una interrupción, que llama a dos funciones *callback* (una para los botones, otra para el slider). Estas funciones reciben los eventos y cambian las variables de estado booleanas a *true* o *false* según la información transmitida con el evento; comprueban el *id* del sensor que ha generado el evento así como el tipo de evento (pulsado, levantado, pulsación larga, inválido, etc.) y actúan en consecuencia. Adicionalmente, la función *callback* del *slider* guarda la posición de la pulsación en una variable, que es leída en la función principal del hilo y que esta pueda actualizar la barra de progreso.

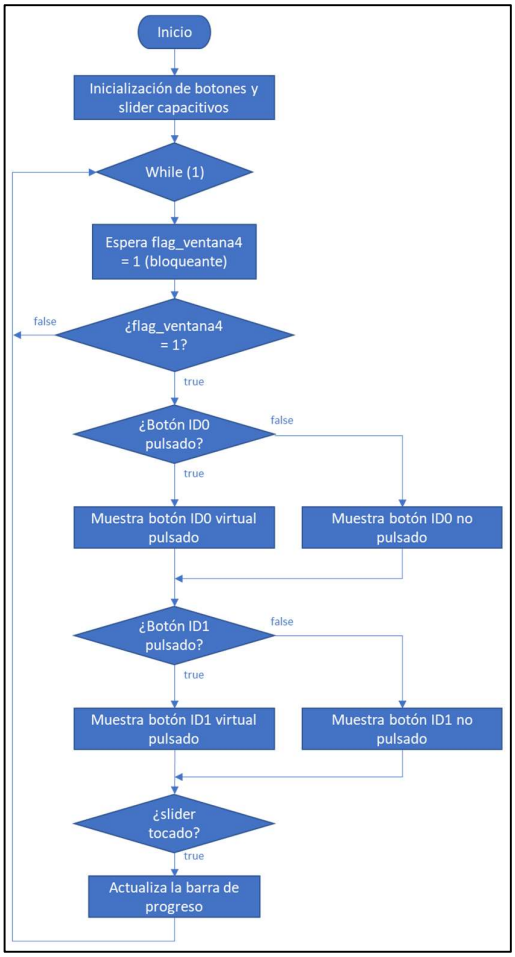


Figura 9-13 Hilo Touch

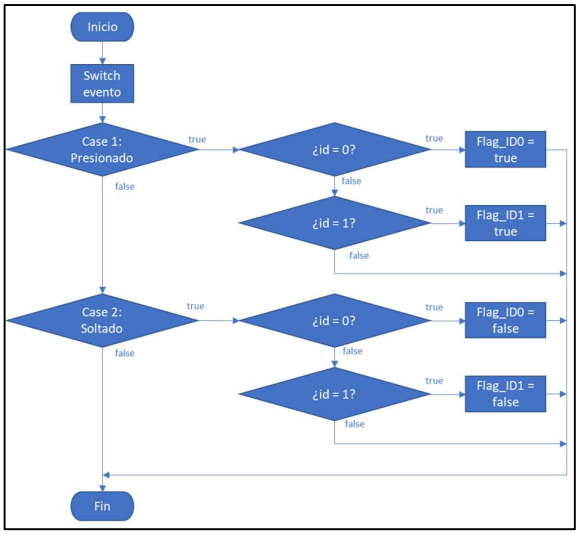


Figura 9-14 Callback botones capacitivos

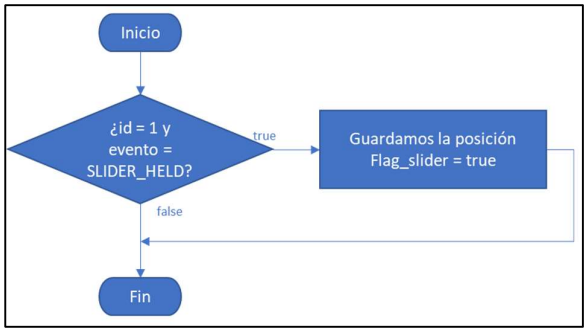


Figura 9-15 Callback slider capacitivo

9.1.5 Hilo DAC

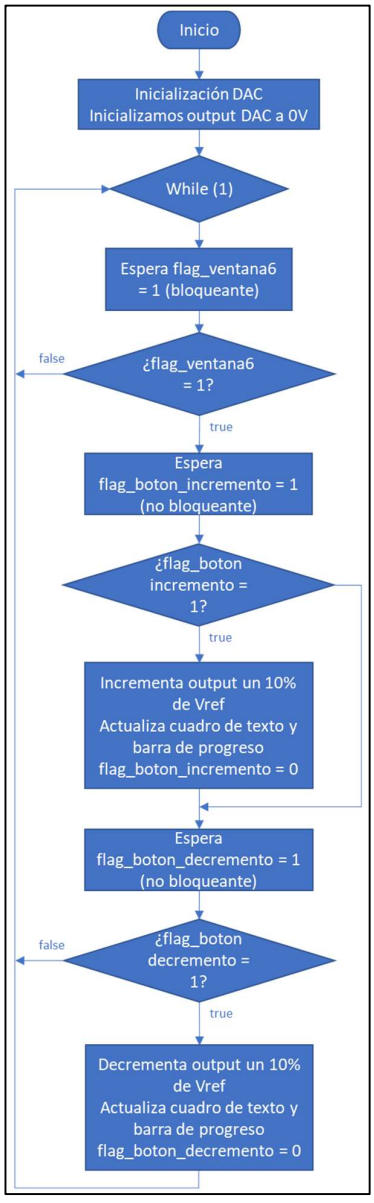


Figura 9-16 Hilo DAC

Este hilo se encarga de inicializar, configurar y leer el DAC. Tras la inicialización y configuración del conversor, se entra en un bucle while, donde se va a esperar la activación del flag de la ventana 6. Una vez se activa el flag, se leen 2 *flags* (mediante una espera no bloqueante), que representan la pulsación de dos botones virtuales. Si se pulsa uno de ellos, se aumenta el valor de tensión de salida un 10% (entre 0V y Vcc). Si se ha pulsado el otro, disminuye el valor de tensión un 10%. Se satura al llegar a Vcc en el primer caso y a 0V en el segundo. En ambos casos, se actualizan el cuadro de texto y barra de progreso y se "bajan" los respectivos *flags*.

9.1.6 Hilo UART

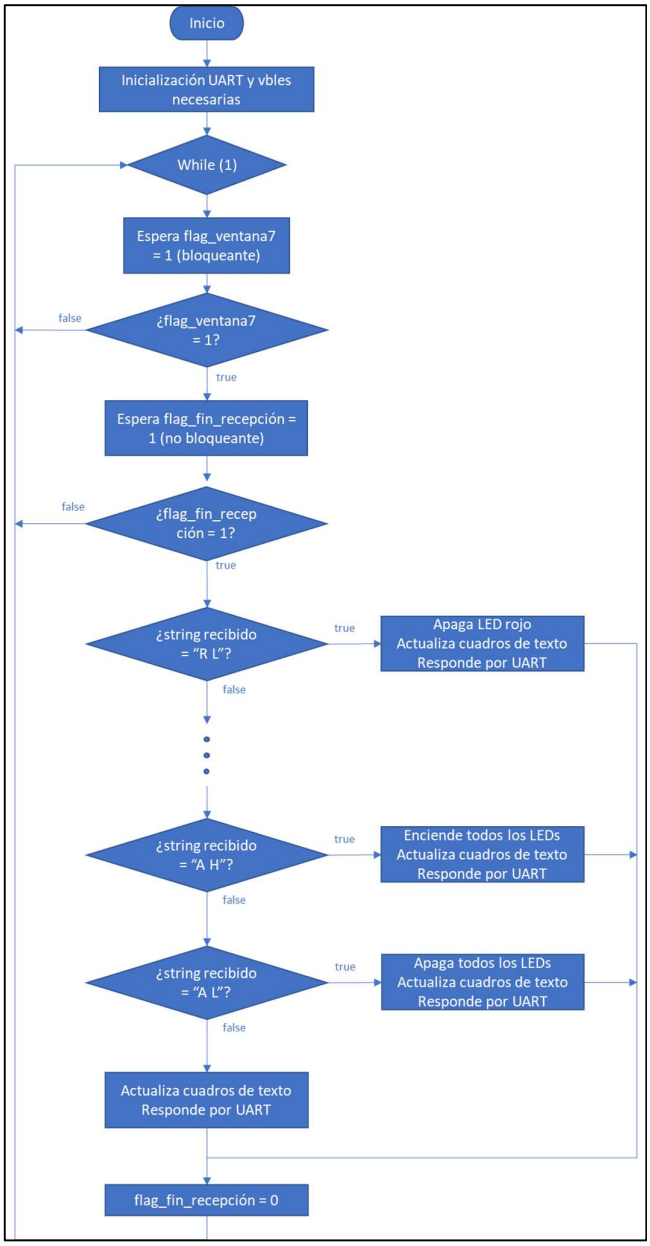


Figura 9-17 Hilo UART

Este hilo se encarga de inicializar y configurar la UART, así como recibir y transmitir datos a través de la misma. Tras la inicialización y configuración de la UART, y de la inicialización de las cadenas de caractéres a transmitir posteriormente, se entra en un bucle while, donde se va a esperar la activación del flag de la ventana 7. Una vez se activa el flag, se leen un *flags* que indica el fin de la recepción de una cadena de caracteres. Si se ha recibido

una cadena de caracteres, entramos en varias sentencias *if* anidadas, en las que se compara la cadena recibida, con unas cadenas definidas previamente. Si coincide con alguna cadena, se enciende/apaga un LED (rojo, verde o naranja), se transmite por UART que se ha encendido/apagado un LED y se actualizan los cuadros de texto de la pantalla, mostrando la cadena recibida y la cadena transmitida en consecuencia. Si no coincide, se transmite una cadena indicando que el comando recibido no es válido, y se actualizan los cuadros de texto de la pantalla consecuentemente. Finalmente, "bajamos" el *flag* de recepción.

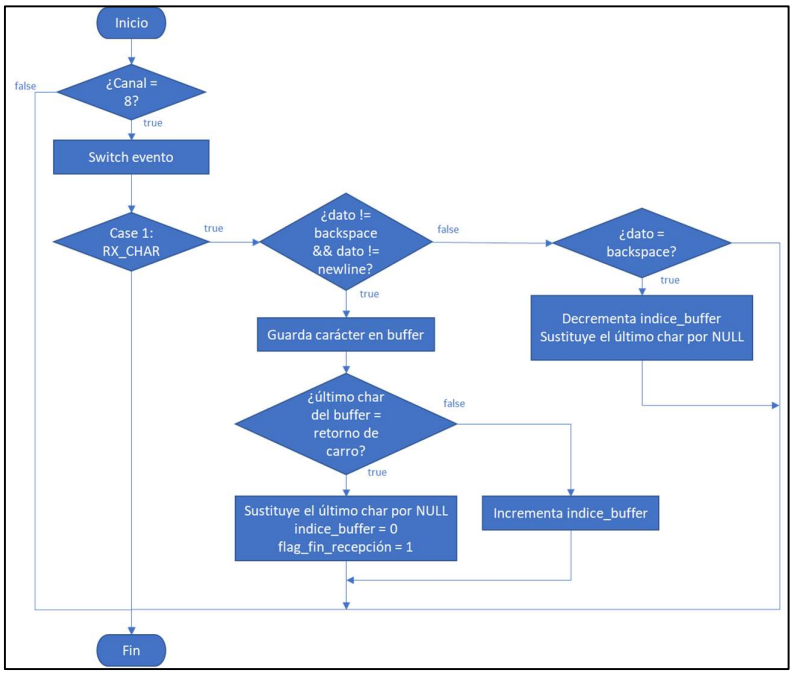


Figura 9-18 Callback UART

La recepción de datos por la UART dispara una interrupción, que llama a una función *callback*. Esta función se encarga de recibir los eventos generados y actuar en consecuencia. En nuestro caso, solo hemos definido la actuación tras la recepción de caracteres.

Al recibir un carácter por el canal 8 (nuestra UART) se decide qué hacer en el caso de recibir el carácter *backspace*. Si se recibe otro carácter, se guarda en el buffer y se incrementa el índice. Si resulta ser un retorno de carro, la función interpreta que la cadena ha terminado. Se resetea entonces el valor del índice y se levanta el flag de recepción. Si se recibe un *backspace*, se decrementaría el índice y se borraría el último carácter recibido previamente.

10 CONCLUSIÓN

Aplicaciones del trabajo

TRas el desarrollo de la guía de programación y comprobar el funcionamiento del kit de programación, se describen las conclusiones en este capítulo, mostrando posibles aplicaciones del trabajo realizado.

10.1 Aplicación en asignaturas de grado

En la Escuela Técnica Superior de Ingeniería de Sevilla, al igual que en otras facultades o escuelas de la Universidad de Sevilla, se imparten diversas asignaturas en las que se hace uso de microcontroladores, ya sea para enseñar a programarlos, como pueden ser las asignaturas de Sistemas Electrónicos, Sistemas Electrónicos para Automatización (al menos en el Grado en Ingeniería Electrónica, Robótica y Mecatrónica) o en las que se hace uso de ellos, como por ejemplo, Proyectos Integrados o Laboratorio de Robótica.

En estas asignaturas se usan generalmente microcontroladores de Texas Instruments, que generalmente se programan de manera básica, a bajo nivel, actuando sobre los propios registros del microcontrolador, que están limitados en recursos y potencia de procesamiento, y que pueden ser beneficiosos para los alumnos para aprender a programar optimizando el uso de los recursos disponibles, pero que acaban limitando las posibilidades de programación a la hora de querer programar una aplicación más exigente a nivel de procesamiento o recursos.

Por otro lado, la familia de microcontroladores Synergy presenta una sintaxis sencilla, de alto nivel y prácticamente independiente del periférico a programar. Adicionalmente la familia Synergy es escalable, contiene distintos microcontroladores con distintos recursos y potencia de procesamiento, pero permitiendo en gran medida reutilizar código escrito para otros microcontroladores de la familia.

De esta manera, se podría usar un rango de microcontroladores que presenta la familia Synergy para impartir las asignaturas mencionadas anteriormente, manteniendo la forma de programarlos y centrándonos en los recursos

disponibles (microcontroladores de menos recursos para aquellas asignaturas que no lo requieran, o que sean introductorias a la programación de microcontroladores, y los de mayores recursos para aquellas asignaturas posteriores).

Adicionalmente, esta familia de microcontroladores hace uso de un Sistema Operativo en Tiempo Real, y actualmente se imparte la asignatura de Informática Industrial en el mismo curso y cuatrimestre que la asignatura de Sistemas Electrónicos para Automatización. Dado que, en la primera asignatura, se enseña lo que es un RTOS, sus elementos (colas de mensajes, semáforos, colas, mutexes), la concurrencia, planificación, etc. y a programar haciendo uso de estos elementos o teniéndolos en cuenta, se puede generar una sinergia entre ambas asignaturas al poder programar aplicaciones sobre un RTOS en un microcontrolador y no únicamente en una máquina virtual con QNX Neutrino.

10.2 Continuación del trabajo

Este trabajo se centra en como programar los principales periféricos que encontramos en el kit (interfaz gráfica, sensores táctiles, comunicación serie, conversores ADC y DAC, etc.). Sin embargo, existen diversos elementos que no se han tratado. Un ejemplo de ello sería el uso de la interfaz Ethernet (con distintos protocolos), del uso de ficheros o de la programación usando memoria flash QSPI. Adicionalmente, existen drivers que ofrecen encriptación, modos de bajo consumo, el uso de la interfaz USB como *Host*, dispositivo o almacenamiento, entre otros.

A partir de trabajo realizado, otro alumno podría seguir investigando el uso y funcionamiento de los periféricos, interfaces y drivers que faltan por incluir en esta guía de programación.

11 BIBLIOGRAFÍA

Documentación y referencias usadas para la realización de este TFG.

1. Renesas Electronics. (2020). Renesas Synergy Software Package (SSP) v1.7.8 User's Manual. 11/06/2020, de Renesas Electronics. Sitio web: https://synergygallery.renesas.com/media/products/1/390/en-US/r11um0140eu0110-synergy-ssp-v178.pdf
2. Renesas Electronics. (2015). Renesas Synergy™ Starter Kit SK-S7G2 User's Manual. Oct. 2015, de Renesas Electronics. Sitio web: https://www.renesas.com/us/en/document/mat/s7g2-starter-kit-sk-s7g2-users-manual
3. Renesas Electronics. (2020). GUIX Studio User Guide. Mayo 2020, de Renesas Electronics. Sitio web: https://synergygallery.renesas.com/media/products/8/391/en-US/r11um0002eu0561-synergy-guix-studio.pdf
4. Oed, R.. (2020). Basics of the Renesas Synergy Platform: Introduction to the Renesas Synergy™ Platform. Abril 2020, de Renesas Electronics. Sitio web: https://www.renesas.com/us/en/document/gde/introduction-renesas-synergy-platform
5. Oed, R.. (2020). Basics of The Renesas Synergy Platform: Details of the Renesas Synergy™ Software. Abril 2020, de Renesas Electronics. Sitio web: https://www.renesas.com/us/en/document/gde/details-renesas-synergy-software
6. Oed, R.. (2020). Basics of the Renesas Synergy Platform: An Introduction to the APIs of the Synergy™ Software Package. Abril 2020, de Renesas Electronics. Sitio web: https://www.renesas.com/us/en/document/gde/introduction-apis-synergy-software
7. Oed, R.. (2020). Basics of the Renesas Synergy Platform: Getting the Renesas Synergy™ Platform Toolchain up and running. Abril 2020, de Renesas Electronics Sitio web: https://www.renesas.com/us/en/document/gde/getting-renesas-synergy-platform-toolchain-and-running
8. Oed, R.. (2020). Basics of the Renesas Synergy Platform: Working with the Development Environments for the Renesas Synergy™ Platform. Abril 2020, de Renesas Electronics. Sitio web: https://www.renesas.com/us/en/document/gde/working-development-environments-renesas-synergy-platform
9. Renesas Electronics. (2017). Transitioning to ThreadX: Event Flags. Abril 2017, de Renesas Electronics. Sitio web: https://renesasrulz.com/synergy/b/weblog/posts/transitioning-to-threadx-event-flags

10. Renesas Electronics. (2017). Transitioning to ThreadX: Message Queues. Abril 2017, de Renesas Electronics. Sitio web: https://renesasrulz.com/synergy/b/weblog/posts/transitioning-to-threadx-message-queues
11. Renesas Electronics. (2017). I want to control circular_gauge in GUIX. Nov. 2017, de Renesas Electronics. Sitio web: https://renesasrulz.com/synergy/f/synergy---forum/9486/i-want-to-control-circular_gauge-in-guix/31749#31749

12 ANEXO CÓDIGOS

Código de la aplicación de demostración.

Este anexo contiene los códigos de la aplicación de demostración descrita en el capítulo 9 de este proyecto.

12.1 hal_entry.c

```
/* HAL-only entry function */
#include "hal_data.h"
void hal_entry(void)
{
    /* TODO: add your own code here */
}
```

12.2 main_thread_entry.c

```
/* Main Thread entry function */
#include "main_thread.h"
#include "bsp_api.h"
#include "gx_api.h"
#include "gui/guiapp_specifications.h"
#include "gui/guiapp_resources.h"
#include "hardware/lcd.h"
#include "gx_user_heap.h"
```

```
//macros para manejar los LEDs fácilmente.
#define RED_LED_PIN IOPORT_PORT_06_PIN_01
#define GREEN_LED_PIN IOPORT_PORT_06_PIN_00
#define ORANGE_LED_PIN IOPORT_PORT_06_PIN_02
#define ON IOPORT_LEVEL_LOW
#define OFF IOPORT_LEVEL_HIGH

static bool ssp_touch_to_guix(sf_touch_panel_payload_t * p_touch_payload,
GX_EVENT * g_gx_event);
void main_thread_entry(void);
void circulo(GX_WINDOW *widget, int x, int y);

static GX_EVENT g_gx_event;

GX_WINDOW_ROOT * p_window_root;
extern GX_CONST GX_STUDIO_WIDGET *guiapp_widget_table[];

static UINT status_flag;    //definimos variable para recoger el estado de la
función que realiza la espera no bloqueante del flag w5
ULONG actual_events_w5;     //definimos la variable donde se va a volcar el
valor del grupo de flags w5. Puede albergar hasta 32 banderas (una por bit,
unsigned long). Por simplicidad usaremos una bandera por grupo.

void main_thread_entry(void) {
      ssp_err_t        err;
      sf_message_header_t * p_message = NULL;
      UINT      status = TX_SUCCESS;

    /* Initializes GUIX. */
    status = gx_system_initialize();

    gx_user_heap_setup();

    /* Initializes GUIX drivers. */
    err = g_sf_el_gx.p_api->open (g_sf_el_gx.p_ctrl, g_sf_el_gx.p_cfg);

    gx_studio_display_configure ( DISPLAY_1, g_sf_el_gx.p_api->setup,
LANGUAGE_ENGLISH, DISPLAY_1_THEME_1, &p_window_root );

    err = g_sf_el_gx.p_api->canvasInit(g_sf_el_gx.p_ctrl, p_window_root);

    // Create the widgets we have defined with the GUIX data structures and
resources.
    GX_CONST GX_STUDIO_WIDGET ** pp_studio_widget = &guiapp_widget_table[0];
    GX_WIDGET * p_first_screen = NULL;

    while (GX_NULL != *pp_studio_widget){
        // We must first create the widgets according the data generated in
GUIX Studio.
        // Once we are working on the widget we want to see first, save the
pointer for later.
        if (0 == strcmp("window1", (char*)(*pp_studio_widget)->widget_name))
        {
            gx_studio_named_widget_create((*pp_studio_widget)->widget_name,
(GX_WIDGET *)p_window_root, GX_NULL);
        } else {
            gx_studio_named_widget_create((*pp_studio_widget)->widget_name,
GX_NULL, GX_NULL);
        }
        // Move to next top-level widget
        pp_studio_widget++;
```

```
    }
    // Attach the first screen to the root so we can see it when the root is
shown
   gx_widget_attach(p_window_root, p_first_screen);

   /* Shows the root window to make it and patients screen visible. */
   status = gx_widget_show(p_window_root);

   /* Lets GUIX run. */
   status = gx_system_start();

   /** Open the SPI driver to initialize the LCD (SK-S7G2) **/
   err = g_spi_lcdc.p_api->open(g_spi_lcdc.p_ctrl, (spi_cfg_t
*)g_spi_lcdc.p_cfg);

   /** Setup the ILI9341V (SK-S7G2) **/
   ILI9341V_Init();

   /* Controls the GPIO pin for LCD ON (DK-S7G2, PE-HMI1) */
   err = g_ioport.p_api->pinWrite(IOPORT_PORT_10_PIN_03, IOPORT_LEVEL_HIGH);
   while(1){
       bool new_gui_event = false; //reseteamos el valor tras el manejo de
cada evento

       err = g_sf_message0.p_api->pend(g_sf_message0.p_ctrl,
&main_thread_message_queue, (sf_message_header_t **) &p_message,
TX_WAIT_FOREVER); //cola de mensajes bloqueante, espera un evento en la
pantalla.
       switch (p_message->event_b.class_code){
           case SF_MESSAGE_EVENT_CLASS_TOUCH:              //si el evento de
es de la clase tactil
               switch (p_message->event_b.code){
                   case SF_MESSAGE_EVENT_NEW_DATA:             //y se ha
recibido un nuevo dato
                       /** Translate an SSP touch event into a GUIX event */
                       new_gui_event =
ssp_touch_to_guix((sf_touch_panel_payload_t*)p_message, &g_gx_event);
//enviamos los datos recibido a la función que generará el evento GUIX en
función del evento en la pantalla.
                       break;
//le pasamos el payload y el puntero a la estructura a rellenar en función
del payload
                   default:
                       break;
               }
               break;
       default:
           break;
       }
            /** Message is processed, so release buffer. */
            err = g_sf_message0.p_api->bufferRelease(g_sf_message0.p_ctrl,
(sf_message_header_t *) p_message, SF_MESSAGE_RELEASE_OPTION_FORCED_RELEASE);
//borramos el buffer por el que hemos leído los datos de la cola.

            /** Post message. */
            if (new_gui_event) gx_system_event_send(&g_gx_event);  //si hay un
nuevo evento guix, lo enviamos
     }
}

static bool ssp_touch_to_guix(sf_touch_panel_payload_t * p_touch_payload,
GX_EVENT * gx_event){
```

```
      bool send_event = true;
      switch (p_touch_payload->event_type){                 //clasificamos el
evento
          case SF_TOUCH_PANEL_EVENT_DOWN:                       //se ha pulsado
la pantalla
              gx_event->gx_event_type = GX_EVENT_PEN_DOWN;
              break;
          case SF_TOUCH_PANEL_EVENT_UP:
              gx_event->gx_event_type = GX_EVENT_PEN_UP;      //se ha dejado
de pulsar
              break;
          case SF_TOUCH_PANEL_EVENT_HOLD:                       //se mantiene la
pulsación
          case SF_TOUCH_PANEL_EVENT_MOVE:                       //se mueve la
pulsación
              gx_event->gx_event_type = GX_EVENT_PEN_DRAG;
              break;
          case SF_TOUCH_PANEL_EVENT_INVALID:                    //es invalido
              send_event = false;                               //no devolvemos
datos
              break;
          default:
              break;
      }

      if (send_event){ /** Send event to GUI */
            gx_event->gx_event_sender = GX_ID_NONE;
            gx_event->gx_event_target = 0;
            gx_event->gx_event_display_handle = 0;
            gx_event->gx_event_payload.gx_event_pointdata.gx_point_x =
p_touch_payload->x;  //guardamos las coordenadas para enviarlas
            gx_event->gx_event_payload.gx_event_pointdata.gx_point_y =
(GX_VALUE)(320 - p_touch_payload->y);

            status_flag = tx_event_flags_get(&w5_flag, (ULONG) 1, TX_OR,
&actual_events_w5, TX_NO_WAIT);        //espera no bloqueante del flag w5, si
estamos en la pantalla 5, actuamos sobre la misma, mostrando por pantalla los
valores de las coordenadas x e y.
            if (actual_events_w5){
                gx_numeric_prompt_value_set(&window5.window5_numeric_
prompt_X, p_touch_payload->x);
                gx_numeric_prompt_value_set(&window5.window5_numeric
_prompt_Y, (320 - p_touch_payload->y));
                gx_system_canvas_refresh();
                if (gx_event->gx_event_type != GX_EVENT_PEN_DRAG )
circulo(&window5, p_touch_payload->x, (320 - p_touch_payload->y));
            }
      }
      return send_event;
}

void g_lcd_spi_callback(spi_callback_args_t * p_args)
{
    (void)p_args;
    tx_semaphore_ceiling_put(&g_main_semaphore_lcdc, 1);
}


void circulo(GX_WINDOW *widget, int x, int y){
    GX_RECTANGLE drawto;
    GX_CANVAS *my_canvas;
```

```
    /* The default window drawing callback function is the default interface
effect drawing */
    gx_window_draw(widget);

    /* Define a rectangle, and subsequent 2D drawing functions are drawn
within the rectangle */
    gx_utility_rectangle_define(&drawto, 0, 0, 240, 320);

    /* Return to the canvas corresponding to the window */
    gx_widget_canvas_get(widget, &my_canvas);

    /* Starts drawing on the specified canvas.*/
    gx_canvas_drawing_initiate(my_canvas, widget, &drawto);

    gx_context_raw_line_color_set(0xffff0000);
    gx_context_raw_fill_color_set(0xffff0000);
    gx_context_brush_style_set(GX_BRUSH_OUTLINE);
    gx_context_brush_width_set(2);
    gx_canvas_circle_draw(x, y, 6);

    /* After drawing, it is used to force immediate drawing. Note that it
must be consistent with gx_canvas_drawing_initiate calls in pairs */
    gx_canvas_drawing_complete(my_canvas, GX_TRUE);
}
```

12.3 guiapp_event_handlers.c

```
#include "uart_thread.h"
#include "adc_thread.h" //para poder usar el flag de estos hilos y
"activarlos"/"desactivarlos" cuando toque.
#include "dac_thread.h"
#include "touch_thread.h"
#include "pwm_thread.h"
#include "tx_api.h"     //header de la api de threadx. permite uso de
aquellos elementos inherentes a un RTOS (mutex, colas, etc.)
#include "gui/guiapp_resources.h"       //headers con los recursos y
especificaciones creados con GUIX Studio
#include "gui/guiapp_specifications.h"
#include "main_thread.h"

#define RED_LED_PIN IOPORT_PORT_06_PIN_01     //defines usados para el uso de
los LEDs de la placa
#define GREEN_LED_PIN IOPORT_PORT_06_PIN_00
#define ORANGE_LED_PIN IOPORT_PORT_06_PIN_02
#define ON IOPORT_LEVEL_LOW
#define OFF IOPORT_LEVEL_HIGH

extern GX_WINDOW_ROOT * p_window_root;

static UINT show_window(GX_WINDOW * p_new, GX_WIDGET * p_widget, bool
detach_old);

UINT window1_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)
{
    UINT result = gx_window_event_process(widget, event_ptr);

    switch (event_ptr->gx_event_type)
    {
        case GX_SIGNAL(ID_BUTTON_PANTALLA_2, GX_EVENT_CLICKED): //vamos a la
pantalla 2
```

```
            tx_event_flags_set(&w2_flag, (ULONG) 1, TX_OR); //activamos el
flag correspondiente
            show_window((GX_WINDOW*)&window2, (GX_WIDGET*)widget, true);
            break;

        case GX_SIGNAL(ID_BUTTON_PANTALLA_3, GX_EVENT_CLICKED): //vamos a la
pantalla 3
            tx_event_flags_set(&w3_flag, (ULONG) 1, TX_OR); //activamos el
flag correspondiente
            show_window((GX_WINDOW*)&window3, (GX_WIDGET*)widget, true);
            break;

        case GX_SIGNAL(ID_BUTTON_PANTALLA_4, GX_EVENT_CLICKED): //vamos a la
pantalla 4
            tx_event_flags_set(&w4_flag, (ULONG) 1, TX_OR); //activamos el
flag correspondiente
            g_sf_touch_button.p_api->enable(g_sf_touch_button.p_ctrl, 0);
//habilitamos los sensores capacitivos
            g_sf_touch_button.p_api->enable(g_sf_touch_button.p_ctrl, 1);
            g_sf_touch_slider.p_api->enable(g_sf_touch_slider.p_ctrl, 1);
            show_window((GX_WINDOW*)&window4, (GX_WIDGET*)widget, true);
            break;

        case GX_SIGNAL(ID_BUTTON_PANTALLA_5, GX_EVENT_CLICKED): //vamos a la
pantalla 5
            tx_event_flags_set(&w5_flag, (ULONG) 1, TX_OR); //activamos el
flag correspondiente
            show_window((GX_WINDOW*)&window5, (GX_WIDGET*)widget, true);
            break;

        case GX_SIGNAL(ID_BUTTON_PANTALLA_6, GX_EVENT_CLICKED): //vamos a la
pantalla 6
            tx_event_flags_set(&w6_flag, (ULONG) 1, TX_OR); //activamos el
flag correspondiente
            g_dac0.p_api->start(g_dac0.p_ctrl); //iniciamos el DAC
            show_window((GX_WINDOW*)&window6, (GX_WIDGET*)widget, true);
            break;

        case GX_SIGNAL(ID_BUTTON_PANTALLA_7, GX_EVENT_CLICKED): //vamos a la
pantalla 7
            g_uart8.p_api->open(g_uart8.p_ctrl, g_uart8.p_cfg); //iniciamos
la UART
            g_uart8.p_api->baudSet(g_uart8.p_ctrl, (uint32_t)9600);
            g_uart8.p_api->write(g_uart8.p_ctrl,(const uint8_t
*)"\033[2J\033[0;32mUART OPEN:\r\n\033[0;35m - Comandos:  +
LED(\033[4;31mR\033[0;31med\033[0;35m, \033[4;32mG\033[0;32mreen\033[0;35m,
\033[4;33mO\033[0;33mrange\033[0;35m, \033[4;39mA\033[0;39mll\033[0;35m)
[espacio] valor (\033[4mH\033[0;35migh, \033[4mL\033[0;35mow)\r\n\033[0;39m",
strlen("\033[2J\033[0;32mUART OPEN:\r\n\033[0;35m - Comandos:  +
LED(\033[4;31mR\033[0;31med\033[0;35m, \033[4;32mG\033[0;32mreen\033[0;35m,
\033[4;33mO\033[0;33mrange\033[0;35m, \033[4;39mA\033[0;39mll\033[0;35m)
[espacio] valor (\033[4mH\033[0;35migh,
\033[4mL\033[0;35mow)\r\n\033[0;39m"));  //contiene códigos de color VT100
            tx_event_flags_set(&w7_flag, (ULONG) 1, TX_OR); //activamos el
flag correspondiente
            show_window((GX_WINDOW*)&window7, (GX_WIDGET*)widget, true);
            break;
    default:
        gx_window_event_process(widget, event_ptr);
        break;
    }
    return result;
```

```
}

UINT window2_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)
{
    UINT result = gx_window_event_process(widget, event_ptr);
    switch (event_ptr->gx_event_type){
        case GX_SIGNAL(ID_BUTTON_PANTALLA_3, GX_EVENT_CLICKED): //vamos a la
pantalla 3
            tx_event_flags_set(&w3_flag, (ULONG) 1, TX_OR); //activamos el
flag correspondiente
            tx_event_flags_set(&w2_flag, (ULONG) 0, TX_AND); //desactivamos
el flag correspondiente
            show_window((GX_WINDOW*)&window3, (GX_WIDGET*)widget, true);
            g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl, 0,
TIMER_PWM_UNIT_PERCENT, 1); //apagamos el LED
            break;

        case GX_SIGNAL(ID_BUTTON_BACK, GX_EVENT_CLICKED): //volvemos a la
pantalla principal
            tx_event_flags_set(&w2_flag, (ULONG) 0, TX_AND); //desactivamos
el flag correspondiente
            show_window((GX_WINDOW*)&window1, (GX_WIDGET*)widget, true);
            g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl, 0,
TIMER_PWM_UNIT_PERCENT, 1); //apagamos el LED
            break;

        case GX_SIGNAL(ID_SLIDER_PANTALLA_3, GX_EVENT_SLIDER_VALUE):
            tx_event_flags_set(&slider_flag, (ULONG) 1, TX_OR); //activamos
el flag al detectar evento en el slider de la pantalla.
            break;

        default:
            result = gx_window_event_process(widget, event_ptr);
            break;
    }
    return result;
}

UINT window3_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)
{
    UINT result = gx_window_event_process(widget, event_ptr);

    switch (event_ptr->gx_event_type){

        case GX_SIGNAL(ID_BUTTON_PANTALLA_4, GX_EVENT_CLICKED): //vamos a la
pantalla 4
            tx_event_flags_set(&w3_flag, (ULONG) 0, TX_AND);    //activamos
el flag correspondiente
            tx_event_flags_set(&w4_flag, (ULONG) 1, TX_OR); //desactivamos el
flag correspondiente
            g_sf_touch_button.p_api->enable(g_sf_touch_button.p_ctrl,
0);//habilitamos los sensores capacitivos
            g_sf_touch_button.p_api->enable(g_sf_touch_button.p_ctrl, 1);
            g_sf_touch_slider.p_api->enable(g_sf_touch_slider.p_ctrl, 1);
            show_window((GX_WINDOW*)&window4, (GX_WIDGET*)widget, true);
            break;

        case GX_SIGNAL(ID_BUTTON_BACK, GX_EVENT_CLICKED):   //volvemos a la
pantalla principal
            tx_event_flags_set(&w3_flag, (ULONG) 0, TX_AND); //desactivamos
el flag correspondiente
            show_window((GX_WINDOW*)&window1, (GX_WIDGET*)widget, true);
```

```
            break;
        default:
            result = gx_window_event_process(widget, event_ptr);
            break;

    }
    return result;
}

UINT window4_handler(GX_WINDOW *widget, GX_EVENT *event_ptr)
{
    UINT result = gx_window_event_process(widget, event_ptr);

    switch (event_ptr->gx_event_type){
        case GX_SIGNAL(ID_BUTTON_PANTALLA_5, GX_EVENT_CLICKED): //vamos a la
pantalla 5
            tx_event_flags_set(&w4_flag, (ULONG) 0, TX_AND);
            tx_event_flags_set(&w5_flag, (ULONG) 1, TX_OR);
            g_sf_touch_button.p_api->disable(g_sf_touch_button.p_ctrl, 0);
//deshabilitamos sensores capacitivos
            g_sf_touch_button.p_api->disable(g_sf_touch_button.p_ctrl, 1);
            g_sf_touch_slider.p_api->disable(g_sf_touch_button.p_ctrl, 1);

            show_window((GX_WINDOW*)&window5, (GX_WIDGET*)widget, true);
            break;

        case GX_SIGNAL(ID_BUTTON_BACK, GX_EVENT_CLICKED):   //volvemos a la
pantalla principal
            tx_event_flags_set(&w4_flag, (ULONG) 0, TX_AND);
            g_sf_touch_button.p_api->disable(g_sf_touch_button.p_ctrl, 0);
            g_sf_touch_button.p_api->disable(g_sf_touch_button.p_ctrl, 1);
            g_sf_touch_slider.p_api->disable(g_sf_touch_button.p_ctrl, 1);

            show_window((GX_WINDOW*)&window1, (GX_WIDGET*)widget, true);

            break;
        default:
            result = gx_window_event_process(widget, event_ptr);
            break;

    }
    return result;
}
UINT window5_handler(GX_WINDOW *widget, GX_EVENT *event_ptr){
    UINT result = gx_window_event_process(widget, event_ptr);

    switch (event_ptr->gx_event_type){

        case GX_SIGNAL(ID_BUTTON_BACK, GX_EVENT_CLICKED): //volvemos a la
pantalla principal
            tx_event_flags_set(&w5_flag, (ULONG) 0, TX_AND);
            show_window((GX_WINDOW*)&window1, (GX_WIDGET*)widget, true);
            break;

        case GX_SIGNAL(ID_BUTTON_PANTALLA_6, GX_EVENT_CLICKED): //vamos a la
pantalla 6
            tx_event_flags_set(&w6_flag, (ULONG) 1, TX_OR);
            tx_event_flags_set(&w5_flag, (ULONG) 0, TX_AND);
            g_dac0.p_api->start(g_dac0.p_ctrl);
            show_window((GX_WINDOW*)&window6, (GX_WIDGET*)widget, true);
            break;
```

```
        default:
            result = gx_window_event_process(widget, event_ptr);
            break;
        }
    return result;
}

UINT window6_handler(GX_WINDOW *widget, GX_EVENT *event_ptr){
    UINT result = gx_window_event_process(widget, event_ptr);

    switch (event_ptr->gx_event_type){

        case GX_SIGNAL(ID_BUTTON_BACK, GX_EVENT_CLICKED): //volvemos a la
pantalla principal
            tx_event_flags_set(&w6_flag, (ULONG) 0, TX_AND);
            g_dac0.p_api->stop(g_dac0.p_ctrl); //paramos el conversor
            show_window((GX_WINDOW*)&window1, (GX_WIDGET*)widget, true);
            break;
        case GX_SIGNAL(ID_BUTTON_MAS, GX_EVENT_CLICKED): //boton + pulsado
            tx_event_flags_set(&w6_mas_flag, (ULONG) 1, TX_OR);
            break;
        case GX_SIGNAL(ID_BUTTON_MENOS, GX_EVENT_CLICKED): //boton - pulsado
            tx_event_flags_set(&w6_menos_flag, (ULONG) 1, TX_OR);
            break;
        case GX_SIGNAL(ID_BUTTON_PANTALLA_7, GX_EVENT_CLICKED): //vamos a la
pantalla 7
            g_uart8.p_api->open(g_uart8.p_ctrl, g_uart8.p_cfg);
            g_uart8.p_api->baudSet(g_uart8.p_ctrl, (uint32_t)9600);
            g_uart8.p_api->write(g_uart8.p_ctrl,(const uint8_t
*)"\033[2J\033[0;32mUART OPEN:\r\n\033[0;35m - Comandos:  +
LED(\033[4;31mR\033[0;31med\033[0;35m, \033[4;32mG\033[0;32mreen\033[0;35m,
\033[4;33mO\033[0;33mrange\033[0;35m, \033[4;39mA\033[0;39mll\033[0;35m)
[espacio] valor (\033[4mH\033[0;35migh, \033[4mL\033[0;35mow)\r\n\033[0;39m",
strlen("\033[2J\033[0;32mUART OPEN:\r\n\033[0;35m - Comandos:  +
LED(\033[4;31mR\033[0;31med\033[0;35m, \033[4;32mG\033[0;32mreen\033[0;35m,
\033[4;33mO\033[0;33mrange\033[0;35m, \033[4;39mA\033[0;39mll\033[0;35m)
[espacio] valor (\033[4mH\033[0;35migh,
\033[4mL\033[0;35mow)\r\n\033[0;39m"));
            tx_event_flags_set(&w7_flag, (ULONG) 1, TX_OR);
            show_window((GX_WINDOW*)&window7, (GX_WIDGET*)widget, true);
            break;

        default:
            result = gx_window_event_process(widget, event_ptr);
            break;
        }
    return result;
}

UINT window7_handler(GX_WINDOW *widget, GX_EVENT *event_ptr){
    UINT result = gx_window_event_process(widget, event_ptr);

    switch (event_ptr->gx_event_type){

        case GX_SIGNAL(ID_BUTTON_BACK, GX_EVENT_CLICKED): //volvemos a la
pantalla principal
            tx_event_flags_set(&w7_flag, (ULONG) 0, TX_AND);
            show_window((GX_WINDOW*)&window1, (GX_WIDGET*)widget, true);
            g_uart8.p_api->communicationAbort(g_uart8.p_ctrl,
UART_DIR_RX_TX); //abortamos las posibles transmisiones o recepciones
```

```
            g_ioport.p_api->pinWrite(ORANGE_LED_PIN, OFF); //APAGAMOS los
LEDs
            g_ioport.p_api->pinWrite(GREEN_LED_PIN, OFF);
            g_ioport.p_api->pinWrite(RED_LED_PIN, OFF);
            g_uart8.p_api->close(g_uart8.p_ctrl); //cerramos la UART
            break;

        default:
            result = gx_window_event_process(widget, event_ptr);
            break;
        }
    return result;
}

static UINT show_window(GX_WINDOW * p_new, GX_WIDGET * p_widget, bool
detach_old)
{
    UINT err = GX_SUCCESS;

    if (!p_new->gx_widget_parent)
    {
        err = gx_widget_attach(p_window_root, p_new);
    }
    else
    {
        err = gx_widget_show(p_new);
    }

    gx_system_focus_claim(p_new);

    GX_WIDGET * p_old = p_widget;
    if (p_old && detach_old)
    {
        if (p_old != (GX_WIDGET*)p_new)
        {
            gx_widget_detach(p_old);
        }
    }
    return err;
}
```

12.4 gx_user_heap.c

```
/*
 * gx_user_heap.c
 *
 *  Created on: 10 Oct 2017
 *      Author: Renesas Karol
 */

#include "gx_user_heap.h"

static VOID * gx_user_malloc(ULONG size);
static VOID   gx_user_free(VOID * p_mem);

static TX_BYTE_POOL _gx_user_heap_pool;

static UCHAR _gx_user_heap_area[GX_USER_HEAP_SIZE] \
                BSP_ALIGN_VARIABLE(8) \
                BSP_PLACE_IN_SECTION(GX_USER_HEAP_SECTION);
```

```
UINT gx_user_heap_setup(VOID)
{
    UINT status = tx_byte_pool_create(&_gx_user_heap_pool, "GUIX User Heap",
                                      _gx_user_heap_area, GX_USER_HEAP_SIZE);
    if (TX_SUCCESS == status)
    {
        status = gx_system_memory_allocator_set(gx_user_malloc,
gx_user_free);;
    }

    return status;
}


VOID * gx_user_malloc(ULONG size)
{
    VOID * p_mem;

    if (tx_byte_allocate(&_gx_user_heap_pool, &p_mem, size, TX_NO_WAIT))
    {
        p_mem = NULL;
    }

    return p_mem;
}


VOID gx_user_free(VOID * p_mem)
{
    tx_byte_release(p_mem);
}
```

12.5 gx_user_heap.h

```
/*
 * gx_user_heap.h
 *
 *  Created on: 10 Oct 2017
 *      Author: Renesas Karol
 */

#ifndef GX_USER_HEAP_H_
#define GX_USER_HEAP_H_

#include "hal_data.h"
#include "tx_api.h"
#include "gx_api.h"

#define GX_USER_HEAP_SECTION        (".bss")
#define GX_USER_HEAP_SIZE           (256 * 320)

UINT gx_user_heap_setup(VOID);

#endif /* GX_USER_HEAP_H_ */
```

12.6 adc_thread_entry.c

```
/* ADC Thread entry function */
#include "adc_thread.h"
#include "tx_api.h"
#include "main_thread.h"
#include "bsp_api.h"
#include "gx_api.h"
#include "gui/guiapp_specifications.h"
#include "gui/guiapp_resources.h"

static UINT status_flag;
bool flag=false;
ULONG actual_events_w3;     //definimos la variable donde se va a volcar el
valor del grupo de flags w3

void adc_thread_entry(void){
    /* TODO: add your own code here */

    ssp_err_t estado;
    uint16_t dato_convertido;       //guardamos aquí el dato leído por el pin
ADC.

    estado = g_adc0.p_api->open(g_adc0.p_ctrl, g_adc0.p_cfg);
//abrimos ADC0 (8-bit)
    estado = g_adc0.p_api->scanCfg(g_adc0.p_ctrl, g_adc0.p_channel_cfg);
//configuramos el modo de escaneo, establecido previamente en el BSP. Modo
continuo en este caso
    estado = g_adc0.p_api->scanStart(g_adc0.p_ctrl);
//escaneamos (SW trigger)

    while (1){
        actual_events_w3=0;         //reseteamos la variable donde guardamos
el flag
        status_flag = tx_event_flags_get(&w3_flag, (ULONG) 1, TX_OR,
&actual_events_w3, TX_WAIT_FOREVER); //flag activado tras salir de la 2a
pantalla hacia la 3a.
        if(actual_events_w3){   //Puede que otro evento rompa la espera, por
lo que debemos comprobar siempre si el flag se ha activado
            estado = g_adc0.p_api->read(g_adc0.p_ctrl, ADC_REG_CHANNEL_0,
&dato_convertido);        //leemos y volcamos dato a una variable

gx_numeric_pixelmap_prompt_value_set(&window3.window3_numeric_pixelmap_prompt
, (int) dato_convertido); //actualizamos el cuadro de texto con el valor
obtenido.
            gx_circular_gauge_angle_set(&window3.window3_gauge,-
135+(dato_convertido*270/255)); //actualizamos el ángulo de la aguja (el
medidor tiene un arco de 270 deg de recorrido, por lo que pasamos del valor
entre 0 y 255 leído a un valor sobre 270 haciendo una regla de 3. aplicamos
después un offset de 135 grados, donde se encuentra el 0.)
            gx_system_canvas_refresh();
        }
        tx_thread_sleep (10);
    }
}
```

12.7 dac_thread_entry.c

```
#include "dac_thread.h"
```

```c
#include "tx_api.h"
#include "main_thread.h"
#include "bsp_api.h"
#include "gx_api.h"
#include "gui/guiapp_specifications.h"
#include "gui/guiapp_resources.h"
#include "stdio.h"

dac_size_t valor_dac[]={0, 410, 819, 1229, 1638, 2048, 2458, 2867, 3277,
3686, 4095}; //12 bits, incrementos del 10%

float Vref = 3.3;   //valor de referencia del DAC, en este caso, Vcc
float Vo = 0;   //valor teórico de salida, lo usaremos para mostrar en
pantalla

static UINT status_flag;
ssp_err_t status_dac;

int i=0;

ULONG actual_events_w6, actual_events_w6_mas, actual_events_w6_menos;

/* DAC Thread entry function */
void dac_thread_entry(void){
    char salida[5]; //cadena de caracteres usada para mostrar valor Vo en
pantalla
    GX_STRING Vo_str; //estructura admitida por función de GUIX para cambiar
string en pantalla
    Vo_str.gx_string_ptr = "0,00"; //cadena en sí
    Vo_str.gx_string_length = strlen(Vo_str.gx_string_ptr); //longitud cadena

    status_dac=g_dac0.p_api->open(g_dac0.p_ctrl, g_dac0.p_cfg); //abrimos
modulo DAC para pin P014
    status_dac=g_dac0.p_api->write(g_dac0.p_ctrl, valor_dac[i]);
//"escribimos" valor inicial, 0V
    status_dac=g_dac0.p_api->stop(g_dac0.p_ctrl);                //forzamos
que no se active el DAC al principio

    while (1){
        actual_events_w6=0;         //variable donde almacenamos los valores
leídos de las flags. w6 para ejecutar este hilo.
        actual_events_w6_mas=0;     //w6_mas cuando se ha pulsado el boton +
en pantalla. w6_menos cuando se ha pulsado el boton - en pantalla
        actual_events_w6_menos=0;

        status_flag = tx_event_flags_get(&w6_flag, (ULONG) 1, TX_OR,
&actual_events_w6, TX_WAIT_FOREVER); //flag activado tras salir de la 5a
pantalla hacia la 6a.
        if(actual_events_w6){   //Puede que otro evento rompa la espera, por
lo que debemos comprobar siempre si el flag se ha activado
            status_flag = tx_event_flags_get(&w6_mas_flag, (ULONG) 1, TX_OR,
&actual_events_w6_mas, TX_NO_WAIT);    //miramos si ha pulsado el boton +
            if(actual_events_w6_mas){
                i++;
                if (i>10){
                    i=10;
                }
                status_dac=g_dac0.p_api->write(g_dac0.p_ctrl, valor_dac[i]);
//cambiamos valor de salida del DAC
                Vo=Vref*i/10.0f;
//cambiamos valor teórico
```

```
                sprintf(salida, "%1.2f", Vo);
//lo guardamos en una cadena
                Vo_str.gx_string_ptr=salida;
//guardamos la cadena en la estructura necesaria
                Vo_str.gx_string_length = strlen(Vo_str.gx_string_ptr);


gx_prompt_text_set_ext(&window6.window6_window6_text_progress_bar, &Vo_str);
//cambiamos texto en pantalla
                gx_progress_bar_value_set(&window6.window6_progress_bar, i);
//cambiamos valor barra progreso
                gx_system_canvas_refresh();
//actualizamos contenido de la pantalla

                tx_event_flags_set(&w6_menos_flag, (ULONG) 0, TX_AND);
//bajamos el flag tras actualizar los valores
                tx_event_flags_set(&w6_mas_flag, (ULONG) 0, TX_AND);
//bajamos el flag tras actualizar los valores
            }
            status_flag = tx_event_flags_get(&w6_menos_flag, (ULONG) 1,
TX_OR, &actual_events_w6_menos, TX_NO_WAIT);    //miramos si ha pulsado el
boton -
            if(actual_events_w6_menos){
                i--;
                if (i<0){
                    i=0;
                }
                status_dac=g_dac0.p_api->write(g_dac0.p_ctrl, valor_dac[i]);
                Vo=Vref*i/10.0f;
                sprintf(salida, "%1.2f", Vo);
                Vo_str.gx_string_ptr=salida;
                Vo_str.gx_string_length = strlen(Vo_str.gx_string_ptr);


gx_prompt_text_set_ext(&window6.window6_window6_text_progress_bar, &Vo_str);
                gx_progress_bar_value_set(&window6.window6_progress_bar, i);
                gx_system_canvas_refresh();
                tx_event_flags_set(&w6_menos_flag, (ULONG) 0, TX_AND);
                tx_event_flags_set(&w6_mas_flag, (ULONG) 0, TX_AND);
            }
        }
        tx_thread_sleep (10);
    }
}
```

12.8 pwm_thread_entry.c

```
#include "pwm_thread.h"
#include "tx_api.h"
#include "main_thread.h"
#include "bsp_api.h"
#include "gx_api.h"
#include "gui/guiapp_specifications.h"
#include "gui/guiapp_resources.h"

/* PWM Thread entry function */
ULONG actual_events_w2, actual_events_slider;     //definimos la variable
donde se va a volcar el valor del grupo de flags w2
static UINT status_flag;
```

```
uint8_t rx_DC_PWM[2];

void pwm_thread_entry(void){
    g_timer9.p_api->open(g_timer9.p_ctrl, g_timer9.p_cfg);
//inicializamos el timer que usaremos para el PWM.
    g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl, 0, TIMER_PWM_UNIT_PERCENT,
1);    //Forzamos duty cycle nulo

    while (1){
        actual_events_w2=0;         //reseteamos la variable donde guardamos
el flag
        status_flag = tx_event_flags_get(&w2_flag, (ULONG) 1, TX_OR,
&actual_events_w2, TX_WAIT_FOREVER); //flag activado tras salir de la 1a
pantalla hacia la 2a.
        if(actual_events_w2){ //si estamos en la pantalla 2
            status_flag = tx_event_flags_get(&slider_flag, (ULONG) 1, TX_OR,
&actual_events_slider, TX_NO_WAIT);    //miramos si ha cambiado el valor del
slider
            if(actual_events_slider){   //si hay un nuevo valor
                g_timer9.p_api->dutyCycleSet(g_timer9.p_ctrl,
window2.window2_slider.gx_slider_info.gx_slider_info_current_val,
TIMER_PWM_UNIT_PERCENT, 1); //actualizamos el Duty Cycle

gx_numeric_pixelmap_prompt_value_set(&window2.window2_numeric_pixelmap_prompt
, window2.window2_slider.gx_slider_info.gx_slider_info_current_val);
//actualizamos el valor mostrado en pantalla
                gx_system_canvas_refresh(); //redibujamos la pantalla
                tx_event_flags_set(&slider_flag, (ULONG) 0, TX_AND);
//bajamos el flag tras actualizar los valores
            }
        }
        tx_thread_sleep (10);
    }
}
```

12.9 touch_thread_entry.c

```
#include "touch_thread.h"
#include "tx_api.h"

#include "main_thread.h"
#include "bsp_api.h"
#include "gx_api.h"
#include "gui/guiapp_specifications.h"
#include "gui/guiapp_resources.h"

#define RED_LED_PIN IOPORT_PORT_06_PIN_01
#define GREEN_LED_PIN IOPORT_PORT_06_PIN_00
#define ORANGE_LED_PIN IOPORT_PORT_06_PIN_02
#define ON IOPORT_LEVEL_LOW
#define OFF IOPORT_LEVEL_HIGH

bool flag_ID0=0;
bool flag_ID1=0;
bool flag_act_display=0;
bool flag_SLIDER=0;

static uint32_t pos;
static UINT status_flag;
ULONG actual_events_w4=0;
```

```
/* Touch Thread entry function */
void touch_thread_entry(void){
    g_ioport.p_api->pinWrite(RED_LED_PIN, OFF); //APAGAMOS el LED rojo
    g_ioport.p_api->pinWrite(GREEN_LED_PIN, OFF); //APAGAMOS el LED verde
    g_ioport.p_api->pinWrite(ORANGE_LED_PIN, OFF); //APAGAMOS el LED naranja

    g_sf_touch_button.p_api->disable(g_sf_touch_button.p_ctrl, 0);
//deshabilitamos los botones táctiles y el slider, solo se activarán desde el
hilo guiapp_event_handlers cuando pasemos a la pantalla 4
    g_sf_touch_button.p_api->disable(g_sf_touch_button.p_ctrl, 1);
    g_sf_touch_slider.p_api->disable(g_sf_touch_button.p_ctrl, 1);
    while (1){
        status_flag = tx_event_flags_get(&w4_flag, (ULONG) 1, TX_OR,
&actual_events_w4, TX_WAIT_FOREVER); //hilo se queda bloqueado esperando
evento
        if(actual_events_w4){   //Puede que otro evento rompa la espera, por
lo que debemos comprobar siempre si el flag está activo.

            if(flag_ID0) gx_button_select(&window4.window4_button_S1);
//mostramos el boton de pantalla S1 como activo al pulsar el boton tactil ID0
            else gx_button_deselect(&window4.window4_button_S1,
GX_EVENT_TOGGLE_OFF);

            if(flag_ID1) gx_button_select(&window4.window4_button_S3);
            else gx_button_deselect(&window4.window4_button_S3,
GX_EVENT_TOGGLE_OFF);

            if(flag_SLIDER)
gx_progress_bar_value_set(&window4.window4_progress_bar, (pos));  //en una
barra de progreso mostramos la posición del slider
            if(flag_act_display) gx_system_canvas_refresh();
flag_act_display=0;                //actualizamos la pantalla
        }
        tx_thread_sleep (1);
    }
}

void g_button_framework_user_callback(sf_touch_ctsu_button_callback_args_t
*p_args){
    switch (p_args->event){
            case TOUCH_BUTTON_STATE_PRESSED:
                if(p_args->id == 0){
                    g_ioport.p_api->pinWrite(ORANGE_LED_PIN, ON);
                    flag_ID0=1;
                    flag_act_display=1;
                }
                else if (p_args->id == 1){
                    g_ioport.p_api->pinWrite(RED_LED_PIN, ON);
                    flag_ID1=1;
                    flag_act_display=1;
                }

                break;

            case TOUCH_BUTTON_STATE_RELEASED:
                if(p_args->id == 0){
                    g_ioport.p_api->pinWrite(ORANGE_LED_PIN, OFF);
                    flag_ID0=0;
                    flag_act_display=1;
                }
                else if (p_args->id == 1){
```

```
                    g_ioport.p_api->pinWrite(RED_LED_PIN, OFF);
                    flag_ID1=0;
                    flag_act_display=1;
                }

                break;

            default:
                break;
        }
}

void g_slider_framework_user_callback(sf_touch_ctsu_slider_callback_args_t
*p_args){
    if (p_args->id == 1 && (p_args->event ==
SF_TOUCH_CTSU_SLIDER_STATE_HELD)){// || p_args->event ==
SF_TOUCH_CTSU_SLIDER_STATE_RELEASED )){// || p_args->event ==
SF_TOUCH_CTSU_SLIDER_STATE_TOUCHED)){
        pos = 100-p_args->current_position/5; //hemos calibrado a la inversa
los botones capacitivos que componen el slider.
        flag_act_display=1;
        flag_SLIDER=1;
    }
}
```

12.10uart_thread_entry.c

```
#include "uart_thread.h"
#include "string.h"
#include "stdio.h"
#include "tx_api.h"
#include "main_thread.h"
#include "bsp_api.h"
#include "gx_api.h"
#include "gui/guiapp_specifications.h"
#include "gui/guiapp_resources.h"
/* UART Thread entry function */

//defines usados para el uso de los LEDs de la placa
#define RED_LED_PIN IOPORT_PORT_06_PIN_01
#define GREEN_LED_PIN IOPORT_PORT_06_PIN_00
#define ORANGE_LED_PIN IOPORT_PORT_06_PIN_02
#define ON IOPORT_LEVEL_LOW
#define OFF IOPORT_LEVEL_HIGH

//variables donde se guardan los resultados de la lectura de flags
ULONG actual_events_w7, actual_events_w7_uart;
static UINT status_flag;
static uart_event_t uart_event;

//Buffer de datos recibidos por la UART, e índice
char RX_Buffer[1024];
int RX_Buffer_index=0;

void uart_thread_entry(void)
{
    /* TODO: add your own code here */
    volatile GX_STRING input_str; //estructura admitida por función de GUIX
para cambiar string en pantalla
```

```
    volatile GX_STRING output_str; //estructura admitida por función de GUIX
para cambiar string en pantalla

    //cadenas de caracteres usadas para enviar por UART o mostrar por
pantalla
    char display_RedOn_tx[]="LED Rojo encendido";
    char display_RedOff_tx[]="LED Rojo apagado";
    char display_GreenOn_tx[]="LED Verde encendido";
    char display_GreenOff_tx[]="LED Verde apagado";
    char display_OrangeOn_tx[]="LED Naranja encendido";
    char display_OrangeOff_tx[]="LED Naranja apagado";
    char display_AllOn_tx[]="LEDs encendidos";
    char display_AllOff_tx[]="LEDs apagados";
    char display_Invalid_tx[]="Comando invalido";

    int length_display_RedOn_tx = strlen(display_RedOn_tx);
    int length_display_RedOff_tx = strlen(display_RedOff_tx);
    int length_display_GreenOn_tx = strlen(display_GreenOn_tx);
    int length_display_GreenOff_tx = strlen(display_GreenOff_tx);
    int length_display_OrangeOn_tx = strlen(display_OrangeOn_tx);
    int length_display_OrangeOff_tx = strlen(display_OrangeOff_tx);
    int length_display_AllOn_tx = strlen(display_AllOn_tx);
    int length_display_AllOff_tx = strlen(display_AllOff_tx);
    int length_display_Invalid_tx = strlen(display_Invalid_tx);


    char dato_tx[] ="\033[0;32mDato recibido\r\n\033[0;39m";
    char RedOn_tx[]="\033[0;32mLED Rojo encendido\r\n\033[0;39m";
    char RedOff_tx[]="\033[0;32mLED Rojo apagado\r\n\033[0;39m";
    char GreenOn_tx[]="\033[0;32mLED Verde encendido\r\n\033[0;39m";
    char GreenOff_tx[]="\033[0;32mLED Verde apagado\r\n\033[0;39m";
    char OrangeOn_tx[]="\033[0;32mLED Naranja encendido\r\n\033[0;39m";
    char OrangeOff_tx[]="\033[0;32mLED Naranja apagado\r\n\033[0;39m";
    char AllOn_tx[]="\033[0;32mLEDs encendidos\r\n\033[0;39m";
    char AllOff_tx[]="\033[0;32mLEDs apagados\r\n\033[0;39m";
    char Invalid_tx[]="\033[0;32mComando invalido\r\n\033[0;39m";

    int length_dato_tx = strlen(dato_tx);
    int length_RedOn_tx = strlen(RedOn_tx);
    int length_RedOff_tx = strlen(RedOff_tx);
    int length_GreenOn_tx = strlen(GreenOn_tx);
    int length_GreenOff_tx = strlen(GreenOff_tx);
    int length_OrangeOn_tx = strlen(OrangeOn_tx);
    int length_OrangeOff_tx = strlen(OrangeOff_tx);
    int length_AllOn_tx = strlen(AllOn_tx);
    int length_AllOff_tx = strlen(AllOff_tx);
    int length_Invalid_tx = strlen(Invalid_tx);

    while (1)
    {
        //reseteamos vbles
        actual_events_w7=0;
        actual_events_w7_uart=0;

        //esperamos que se active el flag de este hilo
        status_flag = tx_event_flags_get(&w7_flag, (ULONG) 1, TX_OR,
&actual_events_w7, TX_WAIT_FOREVER);
        //si se ha activado, ejecutamos
        if(actual_events_w7){
            //miramos si hemos terminado de recibir algún dato
            status_flag = tx_event_flags_get(&w7_uart_flag, (ULONG) 1, TX_OR,
&actual_events_w7_uart, TX_NO_WAIT);
```

```
            //si hemos terminado de recibir datos
            if(actual_events_w7_uart){
                //actuamos en consecuencia (encender/apagar leds según
comandos) y actualizamos la pantalla
                if (strcmp(RX_Buffer, "R L") == 0) {
                    g_ioport.p_api->pinWrite(RED_LED_PIN, OFF);
                    g_uart8.p_api->write(g_uart8.p_ctrl,(const uint8_t
*)RedOff_tx, length_RedOff_tx);
                    output_str.gx_string_ptr = display_RedOff_tx;
                    output_str.gx_string_length = length_display_RedOff_tx;

gx_prompt_text_set_ext(&window7.window7_window7_uart_tx_text, &output_str );
                }
                else {
                    if (strcmp(RX_Buffer, "G L") == 0) {
                        g_ioport.p_api->pinWrite(GREEN_LED_PIN, OFF);
                        g_uart8.p_api->write(g_uart8.p_ctrl,(const uint8_t
*)GreenOff_tx, length_GreenOff_tx);
                        output_str.gx_string_ptr = display_GreenOff_tx;
                        output_str.gx_string_length =
length_display_GreenOff_tx;

gx_prompt_text_set_ext(&window7.window7_window7_uart_tx_text, &output_str );
                    }
                    else {
                        if (strcmp(RX_Buffer, "O L") == 0) {
                            g_ioport.p_api->pinWrite(ORANGE_LED_PIN, OFF);
                            g_uart8.p_api->write(g_uart8.p_ctrl,(const
uint8_t *)OrangeOff_tx, length_OrangeOff_tx);
                            output_str.gx_string_ptr = display_OrangeOff_tx;
                            output_str.gx_string_length =
length_display_OrangeOff_tx;

gx_prompt_text_set_ext(&window7.window7_window7_uart_tx_text, &output_str );
                        }
                        else {
                            if (strcmp(RX_Buffer, "R H") == 0) {
                                g_ioport.p_api->pinWrite(RED_LED_PIN, ON);
                                g_uart8.p_api->write(g_uart8.p_ctrl,(const
uint8_t *)RedOn_tx, length_RedOn_tx);
                                output_str.gx_string_ptr = display_RedOn_tx;
                                output_str.gx_string_length =
length_display_RedOn_tx;

gx_prompt_text_set_ext(&window7.window7_window7_uart_tx_text, &output_str );
                                }
                            else{
                                if (strcmp(RX_Buffer, "G H") == 0) {
                                    g_ioport.p_api->pinWrite(GREEN_LED_PIN,
ON);
                                    g_uart8.p_api-
>write(g_uart8.p_ctrl,(const uint8_t *)GreenOn_tx, length_GreenOn_tx);
                                    output_str.gx_string_ptr =
display_GreenOn_tx;
                                    output_str.gx_string_length =
length_display_GreenOn_tx;

gx_prompt_text_set_ext(&window7.window7_window7_uart_tx_text, &output_str );
                                    }
                                else {
                                    if (strcmp(RX_Buffer, "O H") == 0) {
```

```
                                        g_ioport.p_api-
>pinWrite(ORANGE_LED_PIN, ON);
                                        g_uart8.p_api-
>write(g_uart8.p_ctrl,(const uint8_t *)OrangeOn_tx, length_OrangeOn_tx);
                                        output_str.gx_string_ptr =
display_OrangeOn_tx;
                                        output_str.gx_string_length =
length_display_OrangeOn_tx;

gx_prompt_text_set_ext(&window7.window7_window7_uart_tx_text, &output_str );
                                    }
                                    else {
                                        if (strcmp(RX_Buffer, "A H") == 0) {
                                            g_ioport.p_api-
>pinWrite(ORANGE_LED_PIN, ON);
                                            g_ioport.p_api-
>pinWrite(GREEN_LED_PIN, ON);
                                            g_ioport.p_api-
>pinWrite(RED_LED_PIN, ON);
                                            g_uart8.p_api-
>write(g_uart8.p_ctrl,(const uint8_t *)AllOn_tx, length_AllOn_tx);
                                            output_str.gx_string_ptr =
display_AllOn_tx;
                                            output_str.gx_string_length =
length_display_AllOn_tx;

gx_prompt_text_set_ext(&window7.window7_window7_uart_tx_text, &output_str );
                                        }
                                        else {
                                            if (strcmp(RX_Buffer, "A L") ==
0) {
                                                g_ioport.p_api-
>pinWrite(ORANGE_LED_PIN, OFF);
                                                g_ioport.p_api-
>pinWrite(GREEN_LED_PIN, OFF);
                                                g_ioport.p_api-
>pinWrite(RED_LED_PIN, OFF);
                                                g_uart8.p_api-
>write(g_uart8.p_ctrl,(const uint8_t *)AllOff_tx, length_AllOff_tx);
                                                output_str.gx_string_ptr =
display_AllOff_tx;
                                                output_str.gx_string_length =
length_display_AllOff_tx;

gx_prompt_text_set_ext(&window7.window7_window7_uart_tx_text, &output_str );
                                            }
                                            else {
                                                g_uart8.p_api-
>write(g_uart8.p_ctrl,(const uint8_t *)Invalid_tx, length_Invalid_tx);
                                                output_str.gx_string_ptr =
display_Invalid_tx;
                                                output_str.gx_string_length =
length_display_Invalid_tx;

gx_prompt_text_set_ext(&window7.window7_window7_uart_tx_text, &output_str );
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
```

```
                }

                input_str.gx_string_ptr = RX_Buffer;
                input_str.gx_string_length = strlen(input_str.gx_string_ptr);
                gx_prompt_text_set_ext(&window7.window7_window7_uart_rx_text,
&input_str );
                gx_system_canvas_refresh();
                //reseteamos el buffer
                strcpy(RX_Buffer, "");
                //"bajamos" el flag
                tx_event_flags_set(&w7_uart_flag, (ULONG) 0, TX_AND);
            }
        }
        tx_thread_sleep (10);
    }
}

void user_uart_callback(uart_callback_args_t *p_args){

    if (p_args->channel == 8){ //si pasa algo en nuestra UART
        switch (p_args->event){
            case UART_EVENT_RX_CHAR: //y recibimos un carácter
                if (p_args->data != 10 && p_args->data != 8) { //si el dato
recibido no es un line-feed o backspace
                    RX_Buffer[RX_Buffer_index]=p_args->data; //lo guardamos
en el buffer

                    if (RX_Buffer[RX_Buffer_index] == 13){ //si el dato es un
retorno de carro
                        RX_Buffer[RX_Buffer_index] = NULL; //cerramos la
cadena
                        RX_Buffer_index=0;                  //reseteamos el
índice
                        tx_event_flags_set(&w7_uart_flag, (ULONG) 1, TX_OR);
//levantamos la bandera
                    }
                    else {
                        RX_Buffer_index++;      //incrementamos el índice
                    }
                }
                else if (p_args->data == 8) {   //si recibimos un backspace
                    RX_Buffer_index--;          //decrementamos el índice
                    if (RX_Buffer_index<0) {    //saturamos
                        RX_Buffer_index=0;
                    }
                    RX_Buffer[RX_Buffer_index] = NULL;  //borramos el
carácter anterior
                }
                break;
            default:
                break;
        }
    }
}
```