UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN

"ENTORNO DE VERIFICACIÓN Y TEST PARA OPTIMIZAR LA LÍNEA DE PRODUCCIÓN DEL EQUIPO VERONTE CEX"

TRABAJO FIN DE MÁSTER

Junio - 2021

AUTOR: Mariela Antillano Fernández
DIRECTOR/ES: David Marroquí Sempere
David Benavente Sánchez

UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE
ESCUELA POLITÉCNICA SUPERIOR DE ELCHE
MÁSTER EN INGENIERÍA DE TELECOMUNICACIÓN

"ENTORNO DE VERIFICACIÓN Y TEST PARA OPTIMIZAR LA LÍNEA DE PRODUCCIÓN DEL EQUIPO VERONTE CEX"

TRABAJO FIN DE MÁSTER

Junio - 2021

AUTOR: Mariela Antillano
DIRECTOR/ES: David Marroquí Sempere
David Benavente Sánchez

Firmado por ANTILLANO FERNANDEZ MARIELA - 51775414E el día 30/06/2021

Mariela Antillano Fernández

Firmado digitalmente por DAVID|MARROQUI|SEMPERE
Fecha: 2021.07.01 12:30:01 +02'00'

David Marroquí Sempere

Resumen

En este proyecto se desarrolló un entorno para optimizar en tiempo, robustez y fiabilidad del proceso de verificación en la línea de producción de los equipos Veronte CAN Expander (CEX) y Veronte CEX MC de la empresa Embention Sistemas Inteligentes. Para ello, se diseñó una Printed Circuit Board (PCB) de verificación para llevar a cabo el proceso de comprobación de las interfaces y periféricos de los productos, así como, el conjunto de herramientas software necesarias para completar el proceso de verificación. El entorno desarrollado ha sido validado e implementado en la línea de producción de los equipos, así como, para la reparación de los mismos.

Palabras clave

Veronte CEX, CEX MC, verificación, línea de producción, Embention, Inter-Integrated Circuit Module (I2C), Controller Area Network (CAN), Universal Asynchronous Receiver-Transmitter (UART), Analog-to-Digital Converter (ADC), Pulse Width Modulator (PWM), Enhanced Capture (eCAP), TMS320F28335.

Abstract

In this project, an environment to optimize time, robustness and reliability of the verification process in the production line of Veronte CEX and Veronte CEX MC equipment from the Embention company was developed. In order to do that, a verification PCB was designed to carry out the process of checking the interfaces and peripherals of the products, as well as the set of software tools necessary to complete the verification process. The developed environment has been validated and implemented in the production line, as well as for their repair.

Keywords

Veronte CEX, CEX MC, verification, production line, Embention, I2C, CAN, UART, ADC, PWM, eCAP, TMS320F28335.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

CAPÍTULO 1:

INTRODUCCIÓN

1.1. Contexto

Los Unmanned Aerial Vehicle (UAV), o también conocidos como drones, son aeronaves robóticas que vuelan sin tripulación, capaces de mantener un nivel de vuelo controlado y sostenido, de manera autónoma y con el objetivo de cumplir una misión. Los drones se han implementado ampliamente en varios aspectos de la vida de las personas, tanto en dominios civiles como militares, con tareas como la vigilancia, la fotografía aérea, el diagnóstico industrial, inspecciones o la cartografía.

Embention es una empresa dedicada al desarrollo de componentes y sistemas críticos para drones. Dentro de los valores de la empresa se pueden mencionar:

- Cumplir procesos de validación y verificación para asegurar la calidad de los productos.
- Políticas continuas de innovación e I+D.
- Cumplimiento de normas aeronáuticas como DO160 y DO254.

Entre los principales productos de la empresa se encuentran el 1xVeronte [1] y 4xVeronte [2], los cuales son autopilotos para drones para el control avanzado de sistemas no tripulados. Incorporan un conjunto de sensores y procesadores junto con un radio enlace de datos. El 1xVeronte es un autopiloto simple, mientras el 4xVeronte es un autopiloto redundante con tres autopilotos simples, para garantizar más seguridad durante las misiones.

Otro producto importante desarrollado en Embention es el Veronte CEX que es

un periférico diseñado para reducir la cantidad de cableado en vehículos autónomos, trabajando como un expansor de puertos y señales. Permite instalar sensores, actuadores, payloads y controladores de motor tan alejados del autopiloto como se requiera, sin riesgo de interferencias. Además mejora la conectividad Entrada/Salida (E/S) al poder conectar varios CEXs al autopiloto Veronte. Su integración es muy simple, por lo que Veronte CEX se convierte en una rápida solución para para incrementar la conectividad, permitiendo una optimización del cableado especialmente en sistemas de gran tamaño. Su funcionamiento básicamente consiste en comunicarse con un autopiloto 1xVeronte o 4xVeronte y que éste cuente ahora con las interfaces del CEX para trabajar. Por ejemplo, si se comunica el CEX por el bus CAN con un 1xVeronte, permitiría que éste tenga además de sus periféricos, 8 PWMs y 4 eCAPs adicionales, éstos de 5V de amplitud, además de un bus I2C, dos interfaces UART, una alimentación de 3.3V y una de 5V, entre otras interfaces. En la Figura 1.1 se puede observar un Veronte CEX, consta de dos conectores donde se puedan conectar los periféricos a utilizar. El datasheet del CEX y CEX MC se puede encontrar en el Anexo A.

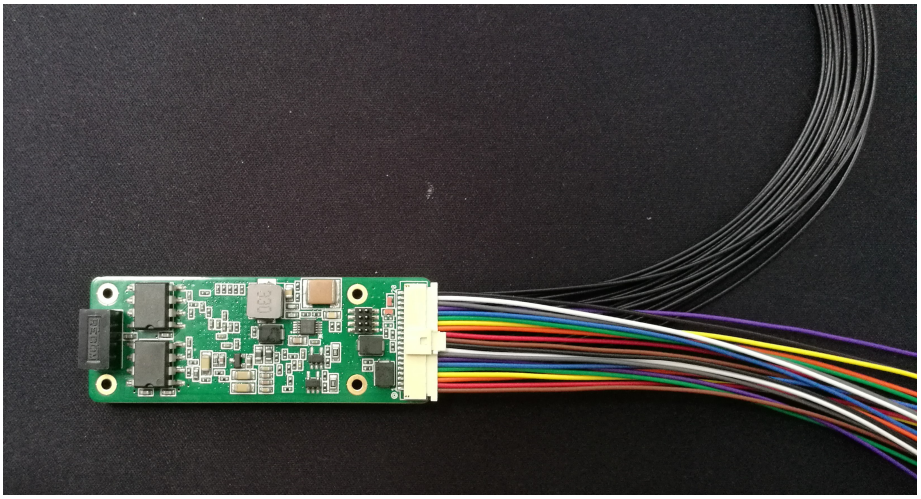


Figura 1.1: Veronte CEX.

El CEX MC (también llamado Veronte CEM) es otro producto de Embention, que consiste en un CEX dentro de una caja metálica para protegerlo de interferencias electromagnéticas. Es una versión más robusta del CEX que además incluye convertidores de señal que pueden adaptar las señales UART en diferentes configuraciones. En la Figura 1.2 se puede observar el CEX MC.

Figura 1.2: CEX MC.

Un proyecto en el que el Veronte CEX es fundamental es un sistema para la realización de vuelos tripulados, que consiste en un drone con dieciocho motores eléctricos independientes que controlan las hélices; cada motor es controlado por un CEX y ellos a su vez por un 4xVeronte.

Mi experiencia en Embention se remonta a 2019, cuando empecé haciendo prácticas en la empresa, formando parte del departamento de verificación, desarrollando tests de diseño de Hardware (HW) comprobando el 1xVeronte y 4xVeronte siguiendo las especificaciones de la DO254. Los tests consistían en validar y verificar las interfaces, periféricos y todo el HW necesario de los productos para cumplir sus funciones. Una vez terminadas las prácticas, cuando me contrataron, me asignaron como proyecto para mi Trabajo de Fin de Máster mejorar la línea de producción del Veronte CEX.

1.2. Motivación

Para garantizar el correcto funcionamiento de los productos por parte de los clientes, en la línea de producción de Embention se realizan distintas pruebas y verificaciones para detectar los fallos en la cadena de montaje, ya sea por parte del fabricante, del montador, al momento de ensamblado o fallo de los componentes en sí.

Esta verificación consiste en probar el HW de cada unidad que conforma cada producto que suministra Embention, con tests básicos para garantizar que todos los componentes están correctamente ubicados, alineados, bien soldados y funcionando correctamente. Es una verificación de HW, más no de diseño y no se comprueba el Software (SW) que va a utilizar el producto al momento de llegar al cliente. Estas dos últimas verificaciones se realizan dentro del departamento de verificación, para cada versión de producto (tanto de HW como de SW) pero no para cada unidad fabricada.

Una correcta verificación de los productos en la línea de producción garantiza su correcto funcionamiento, por lo que hace que la empresa gane clientes satisfechos, con productos robustos, útiles y seguros, y así, que se ahorre dinero evitando devoluciones para reparaciones de fallos que se podrían haber detectado durante el proceso de producción. Además, debido a la naturaleza de los productos de Embention, sus fallos acarrean consecuencias con costos elevados. Concretamente, el fallo de un Veronte CEX puede conducir al fallo de un autopiloto 1xVeronte, de un 4xVeronte, de un drone o incluso de una aeronave más compleja y costosa.

Debido a la importancia del proceso de verificación en la línea de producción, mejorar la cadena de montaje del Veronte CEX era crítico. Antes del desarrollo del presente trabajo, la verificación del CEX consistía en un setup, con loopbacks y conexiones que no estaban bien definidas. Además, no existía documentación en la empresa que explicara las conexiones del setup, el proceso de montaje, ni documentación donde se almacenaran los resultados de los tests ejecutados. Tampoco existía constancia del SW utilizado.

En el anterior entorno de verificación del CEX y CEX MC el sistema no estaba implementado en una PCB por lo que para cada test era necesario desconectar y conectar cables de forma manual. Debido a esto, los errores de carácter humano eran probables y el proceso más lento. Además, no había constancia de los tests ejecutados, ni organización ninguna. Por otro lado, no se comprobaban todas las interfaces y periféricos de los productos y las pruebas no eran tan exhaustivas como es necesario en un proceso de verificación en la línea de producción.

Es importante llevar un control de los productos verificados que se envían a los clientes para tener información de qué tests se han ejecutado, qué interfaces suelen fallar y qué componentes son más problemáticos para mejorar el diseño del producto, si fuera necesario, y mejorar el proceso de fabricación para detectar estos defectos antes de que los productos estén en manos de los clientes. También, un documento con esta información permite tener un registro de la vida del producto, si ha tenido alguna reparación e incluso, si algo ha fallado en algún momento. Por otro lado, es importante que exista un documento explicando cada paso de la verificación para que el proceso pueda ser ejecutado por cualquier persona, sin conocer con totalidad el funcionamiento de cada parte del setup.

1.3. Objetivos

Los objetivos generales y específicos a alcanzar en el desarrollo de este proyecto se enumeran en esta sección.

1.3.1. Objetivos generales

- Mejorar la verificación en la línea de producción de Veronte CEX y CEX MC.
- Desarrollar una PCB de verificación, así como el SW necesario, para la verificación de todas las interfaces y periféricos tanto del Veronte CEX como del CEX MC. Explicados en el Capítulo 3 y Capítulo 4.
- Realizar un manual de usuario para la correcta verificación de los productos (CEX y CEX MC). Explicado en el Capítulo 2.3.1.

1.3.2. Objetivos específicos

- Para reducir costes, la PCB de verificación debe ser de dos capas y se debe utilizar el Microcontrolador (uC) del CEX.
- La PCB de verificación debe adaptarse tanto al CEX como al CEX MC y ser única para todo el proceso de verificación.
- La PCB de verificación debe poder alimentarse a 12 V y 36 V, siendo esta última la alimentación necesaria durante el Burn In Test, que es una verificación que consiste en comprobar el funcionamiento de los productos en temperaturas elevadas para detectar fallos, será explicado más adelante (2.3.9).
- El proceso de verificación tiene que ser simple y rápido.
- La comunicación con el ordenador debe ser a través de una conexión serie.
- El HW desarrollado para la verificación debe soportar temperaturas entre -40ºC y 85ºC para realizar el Burn In test, así como pruebas de temperatura en la cámara climática.
- Todos los procesos tienen que adaptarse a las herramientas disponibles en la empresa.
- Debe existir documentación explicativa de los procesos de verificación para que sean fácilmente reproducibles por cualquier persona, así como los criterios de aceptación de cada test. Dicho documento es el Acceptance Test Procedure (ATP) (2.3.1).
- Debe existir un documento para registrar los resultados obtenidos en la verificación de cada producto, el Acceptance Test Report (ATR) (2.3.1).
- Los fallos en la PCB del CEX deben detectarse lo más pronto posible durante el proceso de producción.

CAPÍTULO 2:

MARCO TEÓRICO

En este capítulo se desarrollan distintos conceptos e ideas necesarios para facilitar la compresión del resto del proyecto. Además, se trata de una recopilación de la información requerida para desarrollar el entono de verificación del Veronte CAN Expander (CEX).

2.1. Veronte CEX

Veronte CEX es un dispositivo diseñado para reducir la cantidad de cableado en vehículos autónomos e incrementar el número de periféricos en el sistema, ya que permite instalar sensores, actuadores, payloads y controladores de motor al Veronte.

La optimización del cableado que permite el CEX es esencial especialmente en los vehículos de gran tamaño, no solamente en términos de reducción de peso o de simplicidad del esquema de cableado, sino también por la robustez y estabilidad en la transmisión de datos que provee, siendo éste resistente a las interferencias electromagnéticas, permitiendo la instalación de cables de longitud sin pérdidas de señal. Especificaciones técnicas más relevantes:

- Entrada de alimentación redundante: el Veronte CEX tiene dos entradas de alimentación independientes entre 6 V y 60 V (V1 y V2) y protección contra polaridad inversa.
- Salidas de voltaje: una de 3.3 V y otra de 5 V (máximo 100mA cada una).
- CAN Bus: dos interfaces Controller Area Network (CAN). Explicado en 2.2.6.
- UART: dos interfaces Universal Asynchronous Receiver-Transmitter (UART)

y una tercera No Populated (NOPOP). Explicado en 2.2.4.

- I2C: una interfaz Inter-Integrated Circuit Module (I2C). Explicado en 2.2.5.
- Salidas digitales: ocho señales Pulse Width Modulator (PWM) de 5 V de amplitud. Explicado en 2.2.1.
- Entradas Digitales: cuatro Enhanced Capture (eCAP) de 5 V. Explicado en 2.2.2.
- Entradas analógicas: dos entre 0 V y 36 V, dos entre 0 V y 12 V, dos entre 0 V y 5 V y dos entre 0 V y 3.3 V. Explicado en 2.2.3.
- Sensor de temperatura analógico [3].

Las señales de entrada y salida se reparten en dos conectores que tiene la Printed Circuit Board (PCB) del Veronte CEX.

El CEX MC cuenta con las mismas especificaciones técnicas, sin embargo tiene solo un conector y en lugar de las dos interfaces UART hay dos opciones de montaje:

- Interfaz RS232 e interfaz RS485.
- Interfaz UART e interfaz RS485.

Para generar todas las señales antes mencionadas y su funcionamiento en general, el Veronte CEX utiliza un Microcontrolador (uC) de Texas Instruments, que es un controlador de señales digitales que cuenta, entre otras, con las siguientes características:

- 6 canales PWM: Enhanced Pulse Width Modulator (ePWM).
- 6 entradas de captura de 32 bits o salidas PWM auxiliares: eCAP.
- Analog-to-Digital Converter (ADC) de 12 bits: 16 canales, 12.5 Megasamples per second (MSPS), tiempo de conversión 80 ns.
- 3 Serial Communications Interface (SCI): A/B/C.
- 2 Enhanced Controller Area Network (eCAN): A/B.
- 1 I2C.
- 88 General-purpose I/O pin (GPIO) (compartidos).
- Salida/entradas 3.3 V.

En este capítulo se van a explicar las distintas interfaces del uC que utiliza Veronte CEX durante su funcionamiento, por su importancia, el objetivo de este proyecto es poder verificarlas todas durante el proceso de montaje en la línea de producción.

2.2. Periféricos del Veronte CEX

En este apartado se detallan los distintos periféricos que proporciona el uC a Veronte CEX. Sin embargo, las especificaciones son particulares del uC que tiene este producto en concreto, el TMS320F28335 de Texas Instruments [4].

2.2.1. Enhanced Pulse Width Modulator (ePWM)

Este periférico admite la generación de PWM de forma independiente y complementaria. Es clave para controlar distintos sistemas electrónicos, tanto comerciales como industriales. Permite el control de motores digitales, la comunicación y la conversión de señales digitales a analógicas. En el caso del CEX, se utiliza, por ejemplo, para controlar los motores del Unmanned Aerial Vehicle (UAV).

El módulo ePWM de este uC representa un canal PWM completo compuesto por dos salidas PWM y a su vez, se crean instancias de ePWM idénticas, de forma que están: EPWMxA y EPWMxB, donde x representa una instancia de ePWM genérica. Incluye también distintos submódulos que se configuran por Software (SW) para que el PWM tenga el comportamiento esperado; entre ellos se encuentran:

- Time-base (TB): escala el reloj del módulo de ePWM para sincronizarlo con el reloj del sistema, configura el Time-Base Counter Register (TBCTR) del PWM en frecuencia o periodo y establece su modo de funcionamiento. El TB determina el tiempo de todos los eventos del ePWM.
- Counter-Compare (CC): especifica el duty cycle de las salidas, así como el tiempo en el que ocurren los eventos de conmutación en ellas.
- Action-qualifier (AQ): especifica el tipo de acción a llevar a cabo cuando un evento de TB o CC ocurre, así como, no tomar ninguna acción, cambiar el

estado de la salida a estado alto o estado bajo, forzar el estado de la salida por control de SW o configurar la banda muerta del PWM por SW.

- Dead-band (DB): es un delay que se implementa para evitar un cortocircuito en los transistores que generan los PWM. Este submódulo controla esa relación permitiendo especificar el retardo en el flanco ascendente o descendente de salida o incluso permite omitir el DB.
- Trip-Zone (TZ): las señales trip-zone alertan de un fallo externo al módulo de ePWM. Este submódulo configura el módulo ePWM para que reaccione a los pines de la zona de disparo y especifica la acción de disparo a llevarse a cabo cuando un fallo ocurre; además establece qué tan seguido va a reaccionar el ePWM para cada pin de TZ.
- Event-trigger (ET): habilita los eventos ePWM que activarán una interrupción o el inicio de conversión del ADC. Además, especifica la velocidad a la que los eventos causan desencadenantes.

La frecuencia del PWM es controlada por el Time-Base Period Register (TBPRD) y el TBCTR. A su vez, el CC toma como entrada el TBCTR y lo compara constantemente con Counter-Compare A Register (CMPA) y Counter-Compare B Register (CMPB), estos eventos se introducen en el AQ donde son calificados por la dirección del contador y se convierten en acciones si éstas están habilitadas. Además, el CC configura el duty cycle del PWM, de forma que cuando, por ejemplo, CMPA es igual a 0, la señal PWM está en estado bajo durante todo el período de tiempo, teniendo así un duty cycle de 0 %; mientras que cuando CMPA es igual a TBPRD el duty es del 100 %.

El AQ tiene el papel más importante en la generación de las ondas PWM debido a que decide qué eventos se convierten en acciones, produciendo así las formas de onda conmutadas requeridas en las salidas EPWMxA y EPWMxB.

TBPRD y CC tienen asociados registros shadow para sincronizarse con el Hardware (HW).

Las señales PWM salen del uC con una amplitud máxima de 3.3 V. Estas señales son transformadas por HW a 5 V utilizando un level shifter en la PCB del CEX.

2.2.2. Enhanced Capture (eCAP)

Es un canal de captura completo que se puede instanciar varias veces según el dispositivo de destino, utiliza una base de tiempo de 32 bits y registra hasta cuatro eventos programables. Una de sus aplicaciones, y para lo que se utiliza en este proyecto, es para medidas de período y duty cycle de señales de tren de pulsos, como PWM. Cuando no se utiliza en el modo de captura, el módulo eCAP se puede configurar como una salida PWM de un solo canal.

El módulo eCAP incluye:

- Multiplexores de selección de polaridad de flanco (ascendente o descendente) por cada evento de captura.
- Cada flanco (hasta 4) está calificado como evento por el secuenciador Mod4.
- El evento de borde se activa en su respectivo registro CAPx (hay un registro CAP por cada CAP del uC, x = 1 a 4) por el contador Mod4. El registros CAPx se carga en el flanco de bajada.

El modo de captura del eCAP puede ser continuo o de una sola captura. En ambos casos, el módulo eCAP controla las funciones de inicio, parada y reinicio del contador Mod4, a través de una acción de disparo único que se puede activar utilizando el comparador de stop-value y después se re-arma por SW.

Una vez armado, el eCAP espera entre 1 a 4 (definido por el stop-value) eventos de captura antes de congelar el contador Mod4 y el contenido de los registros CAP1-4 (time stamps).

Re-armar el módulo eCAP lo prepara para otra secuencia de captura. Además, reinicia a cero el contador Mod4 y permite cargar nuevamente los registros CAP1-4.

En el modo continuo, el contador Mod4 continúa funcionando y se ignora el paso de re-armado y continúa escribiendo valores de captura de forma cíclica.

De forma más simple, el contador Mod4 cuenta cuándo se activa el evento de captura. En modo de continuo, el modo de conteo Mod4 es: 0-1-2-3-0, y el valor

capturado se guarda continuamente en los registros CAP1 a CAP4. En el modo simple, el modo de conteo Mod4 se detiene por defecto de 0 a 3. Es decir, se producen un total de 4 capturas, que se almacenan en los registros CAP1 a CAP4.

El uC acepta tensiones de hasta 3.3 V y los pines de eCAP del CEX hasta 5 V, por lo que estas señales son reducidas por HW en la PCB del CEX antes de entrar al uC.

2.2.3. Analog-to-Digital Converter (ADC)

Es un convertidor analógico digital de 12 bits, es decir que tiene $2^{12} = 4096$ pasos. Está formado por 16 canales, configurables como dos módulos independientes de 8 canales. Sin embargo, aunque hay múltiples canales de entrada y dos secuenciadores, este ADC solo tiene un conversor.

El ADC permite un muestreo simultáneo o secuencial, con una velocidad de 12.5 MSPS.

El secuenciador se puede operar como dos secuenciadores independientes de 8 estados (la palabra "estados" representa el número de conversiones automáticas que se pueden realizar con el secuenciador), donde cada módulo tiene la opción de elegir cualquiera de sus ocho canales utilizando un multiplexor analógico y, tras la conversión, el resultado es almacenado en su respectivo registro ADCRESULT. Los dos módulos pueden secuenciar automáticamente una serie de conversiones, es decir, que cada vez que el ADC reciba una solicitud de inicio de conversión, puede realizar múltiples conversiones automáticamente, lo que permite que el sistema convierta el mismo canal varias veces. De esta forma, la capacidad de secuenciación automática proporciona hasta 16 "conversiones automáticas" en una sola sesión y cada conversión se puede programar para seleccionar uno de los 16 canales de entrada. Por otro lado, con la conexión en cascada de los módulos, puede operarse como un único secuenciador de 16 estados. El valor digital de la entrada analógica es derivado por:

- 0, cuando input ≤ 0 V.

- $$4096 \times \frac{InputAnalogVoltage - ADCLO}{3} \tag{2.1}$$

  cuando 0 V < input < 3 V. Donde ADCLO es voltaje contra el cual se convierten los voltajes del canal de entrada del ADC, en este caso, 0 V.
- 4095, cuando input = 3 V.

El circuito interno del ADC del uC está formado por un circuito Sample and Hold (S+H), para tomar las muestras. En él, los pines de entrada ADCIN pueden ser modelados como un circuito RC en el que, para un correcto funcionamiento, el condensador de muestreo (Ch) del circuito tiene un valor que permite que su tiempo de carga para que la señal de entrada ADC se adquiera (sample) y mantenga el condensador cargado (hold) mientras se procesa. Normalmente, la duración de S+H se elige de modo que Ch se cargue dentro de $\frac{1}{2}$ Less Significant Bit (LSB) o $\frac{1}{4}$ LSB del valor final, dependiendo del error tolerable.

Una vez adquirida la muestra con el circuito S+H, es procesada. Para ello, el ADC tiene un circuito que convierte la señal analógica muestreada por el circuito S+H en una muestra digital discreta.

La salida del ADC del circuito conversor es directamente proporcional a la entrada analógica de voltaje e inversamente proporcional al voltaje de referencia, como demuestra la Ecuación 2.2, la cual es una expresión general de la Ecuación 2.1, y describe la función de transferencia típica de un ADC ideal, donde Code es salida del ADC en forma decimal, V_{IN} es la entrada analógica del ADC (en Voltios), n es el número de bits del ADC y V_{REF} es el voltaje de referencia del ADC. También se observa que la salida Code depende del número de bits, es decir, de la resolución del convertidor [5].

$$Code = (V_{IN}) \times \frac{2^n}{V_{REF}} \tag{2.2}$$

Aunque el uC tiene un V_{REF} interno, que se selecciona por defecto, también pueden utilizarse otros voltajes de referencia de valores 1.024 V, 1.5 V o 2.048 V, conectándolos al pin ADCREFIN del uC.

Debido a la importancia de V_{REF}, en la PCB del CEX se utiliza una referencia de voltaje externa de 2.048 V, que permite mantener un voltaje de salida constante, incluso con variaciones de temperatura o de su alimentación. Para ello, la PCB del CEX utiliza el REF3020 [6], que es un voltaje de referencia con una precisión de 0.2 % y una regulación lineal típica de 110 μV/V; su estabilidad ante variaciones de temperatura, con un valor típico de 30 ppm/ºC entre –40 ºC $\leq$ TA $\leq$ +85ºC, lo hace ideal para esta aplicación; además, es el recomendado por el uC del CEX (TMS320F28335 [4]).

Sin embargo, aunque se use una referencia externa, el rango máximo del ADC es 3.0 V, por lo que las señales de entrada de ADC a los pines del CEX son reducidas por HW utilizando divisores de tensión en la PCB del CEX. Además de eso, las señales pasan por un filtro paso bajo y por un circuito seguidor de tensión antes de entrar al uC, para estabilizar así la señal y garantizar, de esta forma, que no entre más tensión que la soportada por el uC.

2.2.4. Serial Communications Interface (SCI)

La interfaz de comunicaciones en serie es un puerto en serie asíncrono de 2 señales, comúnmente conocido como UART. El SCI contiene un First In First Out (FIFO) de recepción y de transmisión de 16 niveles. Esta interfaz permite comunicaciones entre la Central Processing Unit (CPU) y otro periférico asíncrono que utilice el estándar Non Return to Zero (NRZ), puede funcionar para comunicaciones half dúplex o full dúplex y, para garantizar la integridad de los datos, el SCI verifica los datos recibidos para detectar fallos, paridad y desbordamiento.

En este uC cada interfaz SCI está formada por dos pines, SCITXD (SCI transmit-output pin) y SCIRXD (SCI receive-input pin), con baud rate programable; ambos pines pueden utilizarse como GPIO si no se configuran como SCI. Los datos están formados por un bit de inicio, la longitud de la palabra en bits, un bit de paridad opcional, uno o dos bits de stops y un extra bit para distinguir la dirección de la data, en el modo Address.

En una comunicación full duplex el SCI utiliza:

- Un transmisor (TX) y sus registros principales: Transmit data buffer (SCITXBUF), registro de buffer de datos del transmisor, los cuales son cargados por la CPU para ser transmitidos; y Transmitter shift register (TXSHF), registro de desplazamiento del transmisor, que acepta los datos del registro SCITXBUF y los transfiere al SCI transmit-output pin (SCITXD).
- Un receptor (RX) y sus registros principales: Receive data buffer (SCIRXBUF), registro de buffer de datos del receptor para ser leídos por la CPU; y Receiver shift register (RXSHF), registro de desplazamiento del receptor, que actualiza el SCI receive-input pin (SCIRXD).
- Un generador de baud rate programable.
- Registros de control y estado.

El receptor y el transmisor SCI pueden funcionar de forma independiente o simultánea.

2.2.5. Inter-Integrated Circuit Module (I2C)

I2C proporciona una interfaz entre el uC y otros dispositivos que cumplen con la especificación del bus Inter-IC (bus I2C) de Philips Semiconductors versión 2.1 y se conectan mediante un bus I2C.

Cada dispositivo conectado al bus I2C se reconoce por un address único y trabaja como transmisor o receptor, según su función en el dispositivo: maestro o esclavo. El maestro se encarga de iniciar la transferencia de datos en el bus y de generar la señal de reloj para permitir dicha transferencia. La comunicación empieza con el maestro siendo un maestro-transmisor y normalmente transmite en una dirección para un esclavo en particular. Para recibir datos del esclavo se convierte en un maestro-receptor.

Durante la transferencia, cualquier dispositivo direccionado por este maestro se considera esclavo. Si el módulo I2C es un esclavo, comienza como un esclavo-receptor y normalmente envía un acuse de recibo cuando reconoce su dirección esclava de un maestro. Si el maestro le solicita datos, el módulo debe cambiarse a modo esclavo-

transmisor.

El módulo I2C admite el modo multimaestro, de forma que uno o más dispositivos capaces de controlar un bus I2C pueden conectarse al mismo bus I2C. Está formado por una interfaz serie con un pin de datos en serie System Data I2C (SDA) y un pin de reloj System Clock I2C (SCL); cada uno de ellos debe estar conectado a un voltaje de alimentación positivo mediante una resistencia de pull-up, de forma que cuando el bus está libre, ambos pines están en estado alto; también son bidireccionales. Hay dos técnicas de transferencia de datos de I2C:

- Modo estándar: envía únicamente la cantidad de datos programados.
- Modo repetitivo: se envía constantemente información hasta que el SW recibe una condición de STOP o una de START.

El módulo I2C está formado por:

- Un pin de datos SDA y un pin de reloj SCL
- Registro de datos y FIFO para retener temporalmente los datos recibidos y transmitir los que viajan entre la CPU y el pin SDA.
- Una interfaz de periféricos para permitir que la CPU acceda al registro FIFO y del módulo I2C.
- Un sincronizador para regularizar el reloj del dispositivo y el del pin SCL, así como para sincronizar transferencias de datos con maestros de diferentes velocidades de reloj.
- Un preescalador (I2C Prescaler Register (I2CPSC)) para dividir el reloj de entrada que se dirige al módulo I2C.
- Un filtro de ruido para los pines SDA y SCL.
- Un árbitro para manejar la relación entre el módulo I2C cuando es maestro y otro maestro en el sistema.
- Lógica de interrupciones en el I2C y en el FIFO.

El reloj del módulo I2C determina su frecuencia de funcionamiento. El campo I2CPSC dentro del registro de escalado se inicializa con el valor por el que se quiere

dividir el reloj del sistema System Clock uC (SYSCLK) para obtener el valor del reloj I2C, de forma que:

$$I2CModuleClock(Fmod) = \frac{SYSCLK}{I2CPSC + 1} \tag{2.3}$$

Un segundo divisor afecta a la señal I2C Module Clock para obtener la señal de reloj del I2C en modo maestro que llega al pin SCL.

Para la transmisión de datos, se genera un pulso en el dispositivo maestro para cada bit de datos a ser transmitido. Los valores de los niveles de estado alto y estado bajo no son fijos en el protocolo I2C y dependen del nivel de alimentación asociado. Cuando empieza la transferencia de datos, el maestro manda una condición de START, que es un cambio de estado de nivel alto a bajo mientras el SCL está en estado alto. Después de esta condición y mientras no llegue la señal de STOP, el bus se considera ocupado. La condición STOP se define como una transición de nivel bajo a alto en la línea SDA mientras SCL está en estado alto e indica el final de una transferencia de datos.

La señal de datos tiene un tamaño de 1 a 8 bits y cada bit va con un pulso de la señal SCL. Los datos siempre son enviados con el Most Significant Bit (MSB) primero. Cuando el módulo I2C es receptor, ya sea maestro o esclavo, puede reconocer o ignorar los datos enviados por el transmisor. En el segundo caso, envía una señal de Negative Acknowledgement (NACK).

Bajo condiciones normales de funcionamiento, solo un dispositivo maestro genera la señal SCL. Sin embargo, durante el período de sincronización, si hay dos o más maestros, los relojes deben sincronizarse para que la señal de datos sea procesada correctamente.

En el caso en el que dos o más maestros empiezan la transmisión en el bus aproximadamente a la vez, se inicia un proceso de arbitraje.

2.2.6. Enhanced Controller Area Network (eCAN)

Es un controlador CAN que es compatible con el estándar CAN 2.0B. Se utiliza para comunicarse por serie en entornos eléctricamente ruidosos y otros campos donde se requieren comunicaciones fiables. Admite data rate hasta 1 Megabits por segundo (Mbps) y 32 mailboxes [7].

El CAN es un bus diferencial y la transmisión de mensajes se lleva a cabo a través de dos cables trenzados: CAN high y CAN low. En el estado recesivo, los dos cables están al mismo nivel de tensión, conocido como "common-mode voltage", entre -2 V y 7 V. Necesita una impedancia controlada de 120 Ω entre sus líneas para permitir la transferencia de datos y evitar reflexiones.

El CAN es un protocolo de comunicación de múltiples maestros en serie que admite el control distribuido en tiempo real. Los mensajes están formados hasta por 8 Bytes de datos y son priorizados utilizando un protocolo de arbitraje. También cuentan con un mecanismo de detección de errores, para lograr la integridad de los datos.

Admite cuatro tipos de tramas diferentes para la comunicación:

- Tramas de datos que transportan la información de un nodo transmisor a los nodos receptores.
- Tramas remotas que son transmitidas por un nodo para solicitar la transmisión de una trama de datos con el mismo identificador.
- Tramas de error que son transmitidas por cualquier nodo al detectar una condición de error.
- Tramas de sobrecarga que proporcionan un retraso adicional entre las tramas de datos o tramas remotas anteriores y posteriores.

Además, la versión 2.0B del CAN admite dos formatos diferentes: tramas estándar y tramas extendidas.

Los campos de bits que conforman las tramas están formados por:

- Inicio de trama.
- Campo de arbitraje, contiene el identificador y tipo de mensaje que se envía.
- Campo de control que indica el número de Bytes que se transmiten.
- Hasta 8 Bytes de datos.
- Comprobación de redundancia cíclica (Cyclic Redundancy Check (CRC)).
- Mensaje de reconocimiento (Acknowledgement (ACK)).
- Bits de fin de trama.

El controlador de mensajes del módulo eCAN está formado por una Memory Management Unit (MMU), incluida la interfaz de la CPU, la unidad de control de recepción (filtrado de aceptación) y la unidad de gestión del temporizador; por un registro de control y estado; y por una memoria Random Access Memory (RAM) para almacenamiento de los 32 mensajes del mailbox. Además, el módulo eCAN tiene un CAN protocol kernel (CPK).

Después de que el CPK recibe un mensaje válido, la unidad de control de recepción del controlador de mensajes determina si el mensaje debe almacenarse en uno de los 32 mensajes del mailbox de la RAM. La unidad de control de recepción comprueba el estado, el identificador y la máscara de todos los mensajes recibidos para determinar la ubicación apropiada del mailbox. El mensaje recibido se almacena en el mailbox correspondiente después de pasar el filtro de aceptación. Si la unidad de control de recepción no puede encontrar ningún mailbox para almacenar el mensaje recibido, éste se descarta.

Cuando se desea transmitir un mensaje, el controlador de mensajes transfiere el mensaje al búfer de transmisión del CPK para iniciar la transmisión del mensaje. Cuando se desea transmitir más de un mensaje, el controlador de mensajes transfiere al CPK el mensaje disponible para ser transmitido con la prioridad más alta; en el caso de que dos mailboxes tengan la misma prioridad, primero se transmite el que tenga el número más alto.

La unidad de gestión del temporizador comprende un contador de time-stamp y aplica uno a cada mensaje recibido o transmitido. Genera una interrupción cuando hay un time-out. Esta característica es única del módulo eCAN.

El CAN tiene un Standard CAN Controller Mode (SCC) que es una funcionalidad reducida del eCAN. Solo cuenta con 16 mailboxes, el time stamping no está disponible y el número de máscaras de aceptación es reducido. Este modo es el seleccionado por defecto.

2.2.7. General-Purpose Input/Output (GPIO)

Es un pin genérico que puede configurarse tanto como entrada, como salida, de nivel alto o bajo.

Casi todos los pines del uC de CEX pueden configurarse como GPIO. Los registros de multiplexación GPIO se utilizan para seleccionar el funcionamiento de los pines compartidos. El uC cuenta con 88 pines GPIO y cada pin configurado de esta forma se puede seleccionar individualmente para funcionar como Entrada/Salida (E/S) digital, o conectarse a una de hasta tres señales de E/S periféricas, a través de sus registros.

2.2.8. RS232

RS232 es un estándar para comunicaciones serie. Trabaja con tensiones de $\pm$ 15 V y sus valores de tensión se invierten con respecto a los valores lógicos, de forma que el valor lógico positivo se consigue con una tensión negativa.

Aunque el uC de CEX no tiene señales RS232, una de sus señales UART es convertida a este tipo de bus de comunicaciones utilizando un driver RS232 para el modo de montaje de CEX MC.

2.2.9. RS485

RS485 es un estándar de comunicaciones en bus de la capa física del Modelo OSI. Es ideal para transmitir a altas velocidades por largas distancias debido a que está definido como un bus diferencial, por un cable se transporta la señal original y por

el otro la inversa. Trabaja con tensiones entre -7 V y 12 V.

Al igual que 2.2.8, el uC no cuenta con señales RS485, sino que una de las señales UART se transforma a este protocolo utilizando un driver.

2.3. Proceso de Verificación

El Proceso de verificación del Veronte CEX consiste en verificar el funcionamiento del HW de su PCB. Para ello se realizan distintos pasos de verificación en la línea de producción y así comprobar el funcionamiento de todas sus interfaces y periféricos. La idea es detectar los fallos lo antes posible dentro de la cadena de montaje para ahorrar tiempo y recursos.

En este proyecto se desarrolla un entorno para la verificación del CEX y CEX MC, los distintos pasos del proceso serán explicados en este apartado.

2.3.1. Aceptance Test Procedure (ATP) y Acceptance Test Result (ATR)

Todos los pasos de verificación a seguir en la línea de producción, así como el criterio de aceptación de los tests, están explicados en el Acceptance Test Procedure (ATP) del CEX. Por otro lado, todos los resultados son registrados en un Acceptance Test Report (ATR), el cual es un documento único por Serial Number (S/N) de cada CEX a verificar.

Si se detecta algún fallo en la verificación, se registra en el ATR correspondiente al S/N, se etiqueta como CEX defectuoso y será revisado más adelante en el departamento de Producción.

En el Anexo G se encuentran los ATPs del CEX y del CEX MC. En dichos documentos se referencia a un documento más general que explica los procesos de verificación, éste no es anexado por temas de confidencialidad.

El Anexo H corresponde a la plantilla del ATR. Al empezarse la verificación de

un CEX o CEX MC el operario empieza a completar la plantilla. Como el ATR es común para los dos productos, al iniciarse la verificación, se debe seleccionar en el documento el producto a verificar, ya que hay pasos de la verificación únicos para el acCEX MC.

Los procesos de verificación detallados en el ATP son:

1. Verificación Visual pre-Vibrado.
2. Shaking Test.
3. Verificación Visual post-Vibrado.
4. Verificación de Cortocircuitos.
5. Verificación de Corrientes y Voltajes.
6. Verificación de Programado.
7. Verificación Funcional pre-Burn In Test.
8. Burn In Test.
9. Verificación de Tropicalizado, es decir, del recubrimiento protector.
10. Verificación Funcional post-Tropicalizado.
11. Verificación de Montaje en Caja. (CEX MC)
12. Verificación Funcional post-Montaje en Caja. (CEX MC)
13. Verificación Final.

Los pasos 11 y 12 son exclusivos del CEX MC, debido a que es el mismo producto que el CEX pero con interfaces RS232 y RS485, según el tipo de montaje, y una caja metálica protectora. Debido a esto, su proceso de producción es el mismo que el Veronte CEX hasta el tropicalizado.

De forma general, la verificación se puede dividir en tres etapas:

1. Verificación Inicial: formada por las inspecciones visuales, shaking test, verificación eléctrica (cortocircuitos, corrientes y voltajes) y verificación de programado.
2. Burn in Test
3. Verificación de Ensamblado: formada por la verificación de tropicalizado, funcional, de montaje en caja (si aplica) y verificación final.

Para las etapas 1 y 3 se necesita una fuente de alimentación de 12, 24 o 36 V, mientras que para la etapa 2 se necesita una fuente de 36 V. Para las verificaciones eléctricas es necesario un multímetro. Para programar el producto se necesita un Joint Test Action Group (JTAG) y un SW para el programado: "Flashing Tool", desarrollada en Embention, o "Uniflash" de Texas Instruments. Para el Burn In Test se necesita una cámara climática a una temperatura de 65 ºC y para el Shaking Test una mesa de vibración. Para los test funcionales se necesita una terminal en el ordenador, como "Putty" o "Kitty" y un convertidor RS232 - Universal Serial Port (USB).

2.3.2. Verificación Visual pre-Vibrado.

El objetivo de este test es comprobar que los componentes hayan llegado correctamente soldados del fabricante.

Consiste en verificar visualmente la orientación, el montaje y las soldaduras de determinados componentes críticos de la PCB del CEX, los cuales pueden generar problemas en caso de estar defectuosos o mal montados. En la Figura 2.1 se pueden observar los componentes y orientaciones a verificar de la capa Top Bottom del CEX.

2.3.3. Shaking Test

El objetivo de este test es comprobar la correcta soldadura de los componentes a la PCB.

Consiste en someter a una curva de vibración predeterminada al CEX, de forma que si hay componentes mal soldados, con soldadura en frío, sueltos o defectuosos, se caigan o fallen en este punto de la verificación. La curva de vibración que se aplica es siempre la misma para que sea un proceso estandarizado; para ello se utiliza una mesa de vibración. En el Anexo G (ATP) se encuentra la curva de vibración que se aplica al CEX en este punto de la verificación.

La curva de vibración se aplica mientras el CEX permanece panelizado, por lo

que no puede ser agresiva.

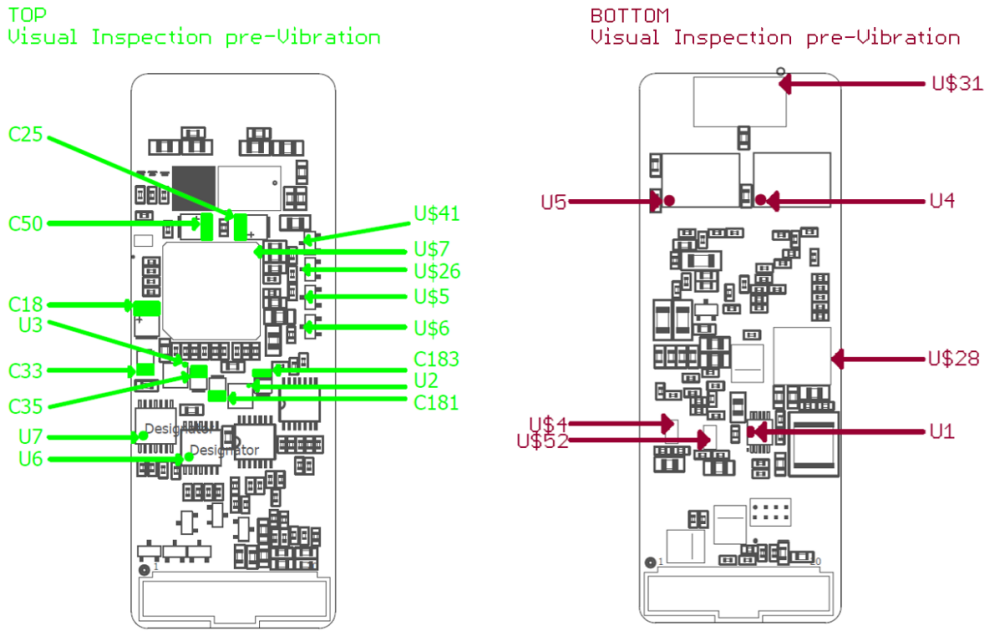


Figura 2.1: Inspección Visual pre-Vibrado, izquierda Top, derecha Bottom.

2.3.4. Verificación Visual post-Vibrado.

El objetivo de este test es comprobar que los componentes siguen soldados correctamente a la PCB del CEX después del Shaking Test.

La verificación consiste en inspeccionar visualmente los mismos componentes que en la verificación visual anterior (2.3.2) y comprobar que siguen soldados correctamente a la PCB.

2.3.5. Verificación de Cortocircuitos

El objetivo de este test es comprobar que no hay conexión entre masa (GND) y las señales críticas. De esta forma se comprueba que las alimentaciones no están cortocircuitadas con GND antes de conectar la PCB a la alimentación y así evitar perjudicar de forma más agresiva el CEX a verificar.

Los puntos a verificar se observan en la Figura 2.2. Para verificar la señal aislada (5V_ISO) se comprueba que no está conectada con la GND aislada, es decir, con

(ISO_GND).

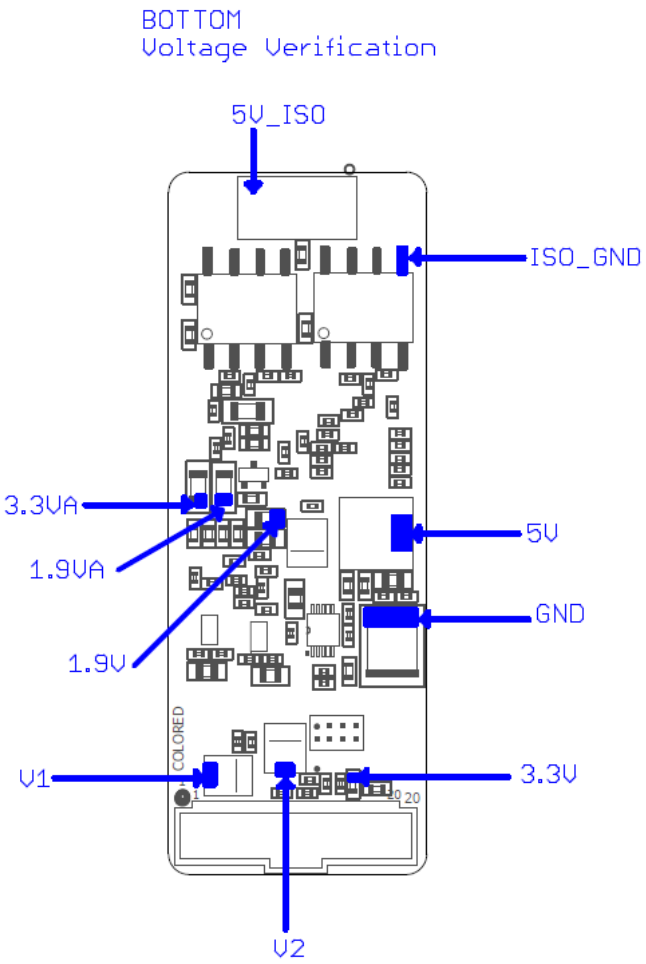


Figura 2.2: Verificación de Cortocircuitos bottom: alimentaciones y GND.

Además, en esta verificación se comprueba que los fusibles y el aislamiento funcionan correctamente. Para ello, se comprueba que hay continuidad entre los pines de los fusibles de la Figura 2.3 (derecha) y que no hay continuidad entre las señales aisladas de la Figura 2.3 (izquierda).

2.3.6. Verificación de Corrientes y Voltajes

El objetivo de este test es comprobar las alimentaciones del CEX así como su consumo de corriente para detectar fallos electrónicos.

En caso de que algún resultado de voltaje se aleje del valor esperado puede significar que hay algún regulador de la PCB que no esté funcionando correctamente

o que hay alguna pista fallando. Por otro lado, un consumo excesivo de corriente puede ser señal de cortocircuito en algún lugar de la PCB; mientras que un consumo por debajo de lo esperado de que hay algún componente de la PCB que no está funcionando correctamente.

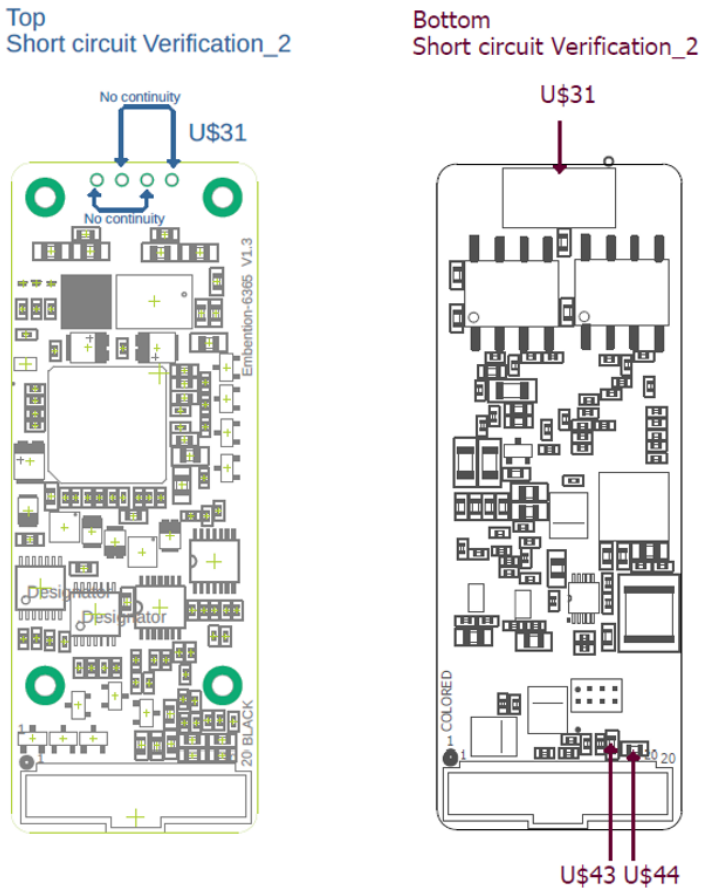


Figura 2.3: Verificación de Cortocircuitos: continuidad entre pines. Izquierda top, derecha bottom.

Los puntos a verificar son los mismos que en la Figura 2.2 y los valores esperados son los siguientes:

- V1 = 7V ± 15 %, debido a las pérdidas en la PCB de verificación.
- V2 = 7V ± 15 %
- 5V = 5V ± 15 %
- 5V_ISO = 5V ± 15 %, se comprueba con ISO_GND.
- 3.3V = 3.3V ± 5 %
- 3.3VA = 3.3V ± 5 %
- 1.9V = 1.9V ± 5 %

- 1.9VA = 1.9V ± 5 %

Por otro lado, el valor esperado del consumo de corriente CEX sin programar, es entre 90 mA y 145 mA.

2.3.7. Verificación de Programado

El objetivo de este test es comprobar que el uC del CEX puede ser programado correctamente. Para ello, utilizando una herramienta de programado, se le carga el SW de verificación: Production Embention Testing Tool (PETT). Después, se comprueba con la terminal que PETT funciona y que la comunicación con el ordenador es posible.

2.3.8. Verificación Funcional pre-Burn In Test

El objetivo de este test es comprobar el funcionamiento de todas las señales, interfaces y periféricos del CEX así como la mayor cantidad de componentes posibles.

Es la primera verificación funcional que se realiza durante el proceso de producción. Para ello se utiliza PETT. En este punto de la producción, también con PETT se le asigna un uAddress al CEX, que es un identificador que se le graba en la memoria flash y se relaciona con el S/N.

2.3.9. Burn In Test

El objetivo de este test es detectar posibles fallos que puedan producirse en una fase temprana del ciclo de vida del CEX (también denominados “fallos de mortalidad infantil”) [8]. Para ello, se somete la PCB del CEX a un estrés eléctrico a temperaturas elevadas.

Consiste en aplicarle al CEX ciclos de encendido y apagado, alternando las alimentaciones entre 7 V y 36 V, entre V1 y V2, a una temperatura de 65 ºC (dentro

de la cámara climática). La idea es repetir de forma iterativa la siguiente secuencia, hasta que se ejecuta, en total, 30 veces PETT:

1. Se enciende CEX a 7 V en V1 y 0 V en V2.
2. Se pasan todos los tests de PETT.
3. Se apaga el CEX.
4. Se enciende CEX a 7 V en V2 y 0 V en V1.
5. Se pasan todos los tests de PETT.
6. Se apaga el CEX.
7. Se enciende CEX a 36 V en V1 y 0 V en V2.
8. Se pasan todos los tests de PETT.
9. Se apaga el CEX.
10. Se enciende CEX a 36 V en V2 y 0 V en V1.
11. Se pasan todos los tests de PETT.
12. Se apaga el CEX.

2.3.10. Verificación de Tropicalizado

El tropicalizado es un proceso que consiste en la aplicación de una capa de barniz cubriendo la PCB con el propósito de protegerla, principalmente contra la humedad.

El objetivo de este test es comprobar que el CEX se ha tropicalizado correctamente. Para ello, visualmente se comprueba que toda la superficie de la PCB está tropicalizada y que los conectores no han sido dañados.

2.3.11. Verificación Funcional post-Tropicalizado

El objetivo de este test, al igual que 2.3.8 es comprobar el funcionamiento de todas las interfaces y periféricos del CEX para verificar que no han sido afectados por el proceso de tropicalizado.

2.3.12. Verificación de Montaje en Caja

Este test solo se realiza para los CEX MC. El objetivo de este test es comprobar que el CEX ha sido correctamente ubicado en su caja y que todas las conexiones se han realizado correctamente.

2.3.13. Verificación Funcional post-Montaje en Caja

Al igual que en las verificaciones funcionales anteriores (2.3.8 y 2.3.11), el objetivo de este test es comprobar el funcionamiento del CEX y verificar que todas las interfaces y periféricos siguen funcionando correctamente después del proceso de montaje en caja (2.3.12). Esta verificación también se realiza únicamente a los CEX MC.

2.3.14. Verificación Final

Este test es el último paso de la verificación antes de que el CEX sea entregado a los clientes. El objetivo es verificar que todas los pasos anteriores se han realizado correctamente, que visualmente todo está correcto, limpio y que no hay elementos sueltos. En este paso también se programa el CEX con el SW correspondiente para su funcionamiento fuera de la empresa.

2.4. Requerimientos

Para realizar los procesos de 2.3 es necesario el SW de PETT para los tests funcionales y de un HW para comprobar las interfaces y periféricos del CEX; ambos se van a desarrollar en este proyecto.

2.4.1. Production Embention Testing Tool (PETT)

Es una herramienta de verificación que se utiliza para comprobar el correcto funcionamiento de todos los periféricos del CEX y de tantos componentes como sea posible. Consiste en un conjunto de tests funcionales que se apoyan en el HW disponible, ya sea el del propio CEX o la PCB de verificación. También tiene operaciones para acciones específicas, como cambiar el uAddress.

Tiene al menos tres modos de funcionamiento:

- Modo Fast.
- Modo Slow.
- Modo Burn In.

En el Capítulo 4 se explicarán las distintas partes de PETT del CEX, así como sus clases, modo de funcionamiento y complicaciones en el proceso de desarrollo.

Modo Fast

Consiste pasar todos los tests justo cuando el CEX se enciende, sin necesidad de una interacción humana. Es el primer test funcional al que se somete el producto, en 2.3.8. En este modo se excluyen los test que necesitan una interacción del operario, como el Test de uAddres.

Si todos los tests se ejecutan correctamente, comprobándose el funcionamiento general del CEX, se enciende en la PCB el Light-Emitting Diode (LED) de "PASS". Por otro lado, si algún test falla, se enciende el LED de "FAIL".

Modo Slow

Está formado por todos los tests funcionales que están desarrollados para el CEX. Se puede ejecutar un test, todos o los que seleccione el operario. Se utiliza en 2.3.11 y 2.3.13.

Al alimentar el CEX y abrir la terminal, se selecciona el test en concreto a ejecutar o la opción de ejecutar todos los tests. Se puede repetir cuantas veces sea necesario y, si el test pasa, comprobando así el funcionamiento de la interfaz a verificar, se imprime en pantalla "PASS", si no, se imprime "FAIL".

Modo Burn In

El modo Burn In de PETT es el utilizado en 2.3.9. Consiste en ejecutar los tests funcionales en ciclos de encendido y apagado.

Para que no haga falta la interacción humana durante el Burn In Test (2.3.9), éste se apoya de Valkiria, que es una herramienta de Embention para automatizar procesos. Debido a esto, el modo Burn In necesita imprimir mensajes en pantalla que Valkiria utiliza para saber cuándo ha empezado y terminado un ciclo de PETT.

Por otro lado, en el Burn In de CEX se excluyen los tests de las alimentaciones, debido a que éstas se alternan durante la prueba.

Cuando se termina la ejecución de PETT se reinicia la PCB para lograr los ciclos de encendido y apagado, para que no haga falta la interacción de una persona para esta acción, se utiliza la PCB de verificación.

2.4.2. PCB de Verificación del CEX

Para comprobar el funcionamiento de las interfaces y periféricos del CEX, además de PETT, es necesario el desarrollo de una PCB de verificación con las siguientes características:

- Tiene un sistema para el reinicio del CEX durante el Burn In Test (2.3.9).
- El funcionamiento de los periféricos SCI y CAN se hace mediante loopbacks y se comprueba que se recibe la misma trama que se envía, por lo que en la PCB de verificación se hacen estas conexiones.
- Para comprobar la señal I2C se establece comunicación con un dispositivo externo, que se conecta en la PCB de verificación.

- Se conectan los PWM con los eCAP y así se comprueban ambos periféricos.
- Se conectan los ADC con las salidas de voltaje del CEX: 5 v y 3.3 V, para comprobar así ambas interfaces.
- Se establece la comunicación del CEX con el ordenador a través de una conexión RS232, a la que se conecta un cable RS232-USB.

Además, la PCB de verificación se utiliza para alimentar la PCB del CEX, para medir su consumo de corriente. Asimismo, permite visualmente saber si pasan los tests y comprobar que los pines de GND del conector del CEX están conectados a GND.

En el Capítulo 3 se detallará el desarrollo de la PCB de verificación del CEX, así como sus interfaces, modo de funcionamiento y complicaciones en el proceso de diseño.

CAPÍTULO 3:

DESARROLLO DE HARDWARE

Para poder verificar los distintos periféricos e interfaces del CAN Expander (CEX) es necesario el desarrollo de un Hardware (HW), como se explica en 2.3.9.

La Printed Circuit Board (PCB) de verificación se diseña con el Software (SW) Eagle y las simulaciones se realizan con el SW LTspice de Analog Devices o TINA-TI de Texas Instruments.

De forma general, la PCB está formada por:

1. Un regulador para reducir la tensión de entrada de la PCB de verificación a los valores necesarios para el funcionamiento del resto de componentes de la PCB.
2. Un conector para el CEX y otro para el CEX MC.
3. Una serie de Light-Emitting Diodes (LEDs) para que con inspección visual a la PCB de verificación se pueda saber si:
   - Las señales de 5V y 3.3V funcionan correctamente.
   - Las conexiones a masa (GND) de los pines de masa de los conectores son correctas.
   - Un test pasa o falla.
4. Un interruptor para seleccionar el modo de funcionamiento: Fast, Slow, Burn In.
5. Loopbacks para comprobar las señales.
6. Un conector para medir consumo de corriente.

En este capítulo se explican las distintas partes de la PCB de verificación desa-

rrollada para el proceso de verificación del CEX en la línea de producción.

3.1. Conectores

La PCB de verificación necesita poder adaptarse tanto al Veronte CEX como al CEX MC, por lo que necesita tener tres conectores, dos correspondientes al CEX y el tercero al CEX MC. La verificación de los distintos pines que conforman los conectores, así como que éstos están bien soldados, se explicará en los siguientes apartados.

3.2. Circuito de alimentación

La PCB de verificación se puede alimentar con fuentes de alimentación entre 12 V y 36 V. La señal de alimentación de entrada (Vin) se regula a 7 V con un circuito reductor utilizando un convertidor buck DC/DC. Durante el Burn In Test se alterna la alimentación del CEX entre la señal Vin y la señal de 7 V.

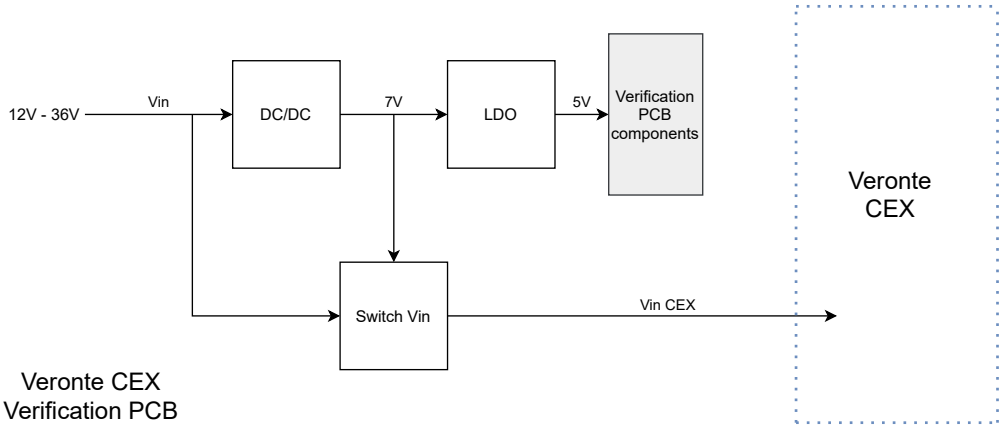


Figura 3.1: Circuito de alimentación.

A partir de la señal de 7 V, utilizando un convertidor LDO, sale una alimentación de 5V, que se utiliza para alimentar los distintos componentes de la PCB.

En la Figura 3.1 se representa el circuito de alimentación de la PCB de verificación.

Para medir el consumo de corriente, se le agrega a la PCB de verificación un conector, de forma que cuando se quiera medir la corriente, se cierra el circuito con el multímetro y cuando no, basta con agregar un jumper para cerrarlo. El CEX, en standby, conectado a la PCB de verificación, es decir, con los LEDs correspondientes encendidos, los loopbacks y las conexiones, consume aproximadamente 340 mA.

Circuito DC-DC buck

El circuito buck para convertir la señal de entrada en 7 V está formado por el LM5164 de Texas Instrumets [9]. Aunque todos los pasos del proceso de verificación están pensados para un voltaje máximo de 36 V, debido a que es la fuente de alimentación de mayor tensión que se utiliza normalmente en cualquier departamento en la empresa, el buck de la PCB de verificación puede reducir a 7 V el voltaje máximo de funcionamiento del CEX, que es 60 V.

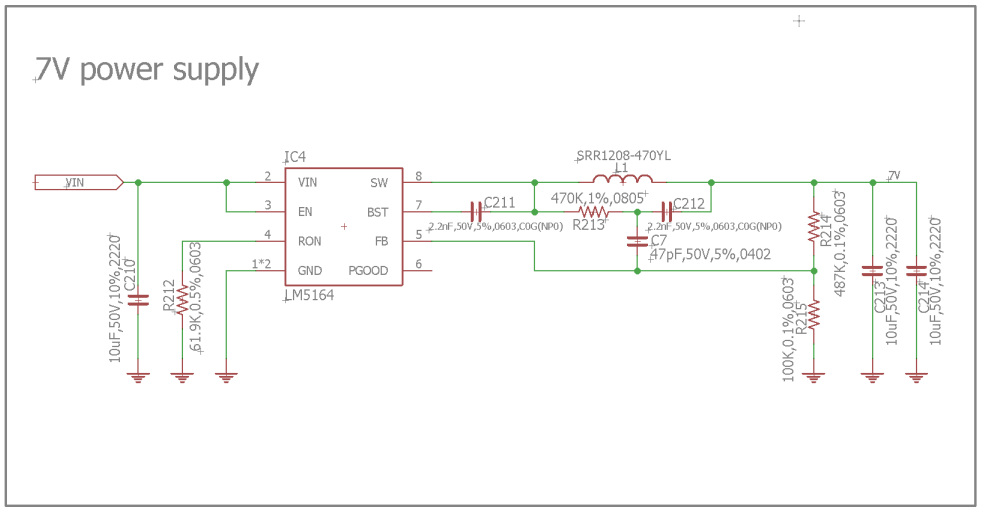


Figura 3.2: Diseño esquemático ddel regulador buck LM5164: 60 V a 7 V.

Sin embargo, este componente por sí mismo no es suficiente para lograr la reducción de voltaje deseada. Requiere de algunos componentes que definen el funcionamiento para la aplicación específica en que se va a usar. Así, en el datasheet [9] se detalla la aplicación típica que recomienda el fabricante del convertidor buck, así como, el circuito que necesita para trabajar óptimamente y el valor de los componentes necesarios. Estas recomendaciones del fabricante se han utilizado como guía para el diseño del circuito reductor buck, así como, las guías de diseño del layout.

El circuito del convertidor buck se puede ver en la Figura 3.2.

El diseño se lleva a cabo partiendo de la tensión de salida deseada, en este caso 7 V. Así, siguiendo las indicaciones del fabricante se establecen, por ejemplo, los valores del condensador de salida, C_{212} y R_{213} y C_7 para reducir el rizado de la tensión de salida. Por otro lado, R_{214} y R_{215} determinan el voltaje del pin feedback del componente; mediante estas dos resistencias se establece la relación entre el voltaje de salida y la referencia interna del componente, de 1.2 V, por lo que se determina de esta manera la tensión de salida. Otro valor importante que determina este circuito es la frecuencia de switch mediante la resistencia R_{212}, ya que afecta al rizado en la salida.

En la Figura 3.3 se observa el resultado de la simulación del circuito buck del LM5164, utilizando el SW TINA-TI de Texas Instruments: en azul se representa la señal de entrada, *VIN*, de 60V, en la que se observa una transición de 0 V a 60 V al inicio de la simulación; en morado se observa la señal *Vout* que a los 3 ms alcanza su voltaje estable, 7.17 V; en verde se puede apreciar *ILOAD*, que es la corriente que pasa por la carga que tiene el circuito de simulación, 20 Ω, que es 357.81 mA. La carga elegida para la simulación es aproximada, porque se utiliza únicamente para estudiar, de forma general, el comportamiento del circuito reductor.

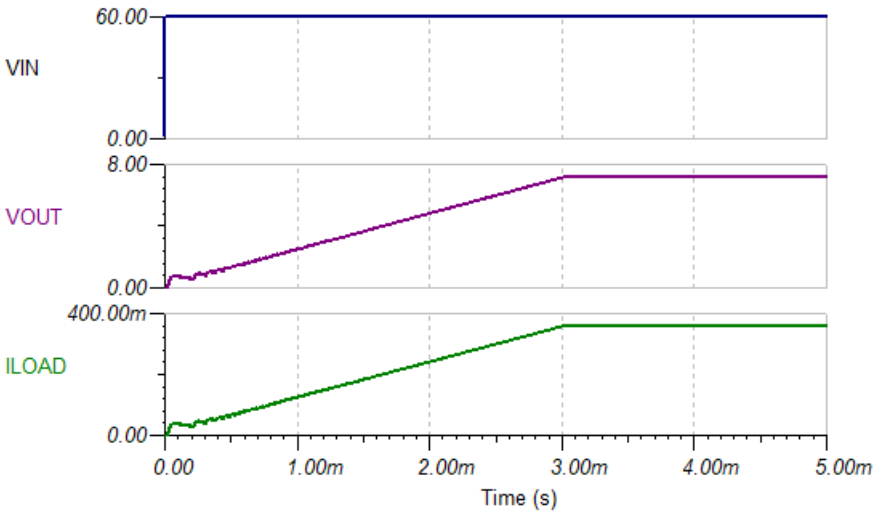


Figura 3.3: Arranque y regulación del regulador buck LM5164: 60 V a 7V.

El convertidor LDO, por su parte, es el encargado de reducir el voltaje de salida del circuito buck a 5 V para alimentar los componentes de la PCB de verificación

así como las puertas lógicas, los relés o el timer. Es el NCV1117ST50T3G de ON Semiconductor [10].

3.3. Comunicación con el ordenador

La comunicación de la PCB de verificación con el ordenador debe ser a través de una comunicación serie, sin embargo, mientras el CEX tiene dos señales Universal Asynchronous Receiver-Transmitter (UART), el CEX MC convierte dichas señales en RS232 y RS485.

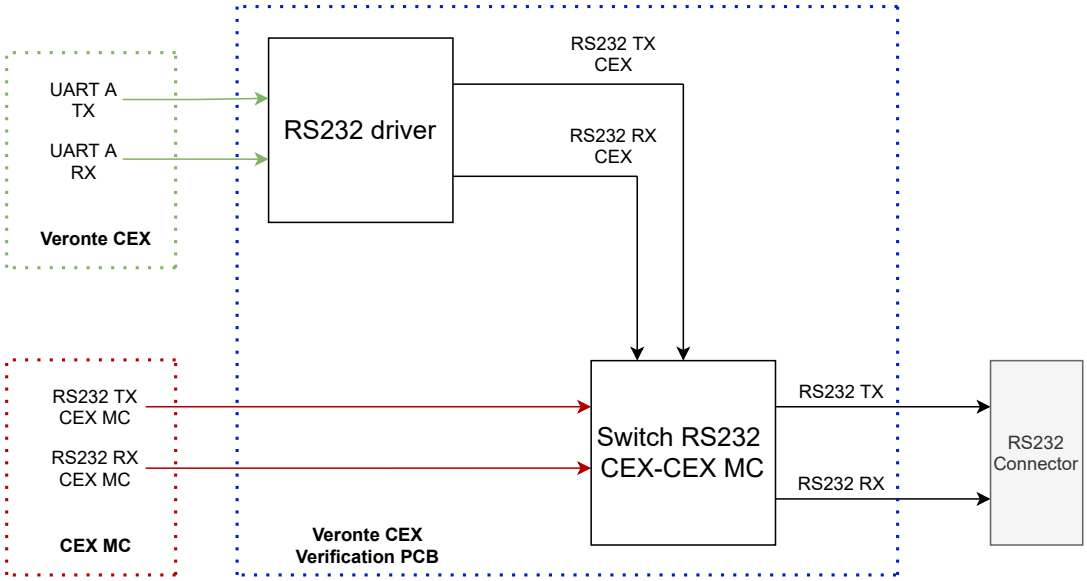


Figura 3.4: Comunicación con el ordenador.

Debido a esto, para que la PCB de verificación sea compatible para los dos productos, se decide que la comunicación con el ordenador sea utilizando el protocolo RS232. Por ello, es necesario un RS232 driver (MAX3232-EP [11]) para transformar la señal UART A del Veronte CEX a RS232.

Ya que solo se quiere transformar la señal del CEX, es necesario un sistema para que se seleccione si se quiere leer del CEX o del CEX MC, como se observa en la Figura 3.4.

El switch RS2323 CEX - CEX MC de la Figura 3.4, cambia entre las señales UART transmisor (TX) y UART receptor (RX) convertidas en señales RS232 del

Veronte CEX y las señales RS232 TX y RS232 RX del Veronte CEX MC. Para ello se utiliza un interruptor para seleccionar el modo de funcionamiento, que activa o desactiva un relé que cambia entre las señales del Veronte CEX y del CEX MC, como se observa en la Figura 3.4. Este sistema de selección, junto a los conectores específicos para cada producto, son los parámetros que hacen que la PCB de verificación sea válida para verificar tanto al CEX como al CEX MC.

3.4. Verificación de salidas de potencia: 3.3V y 5V

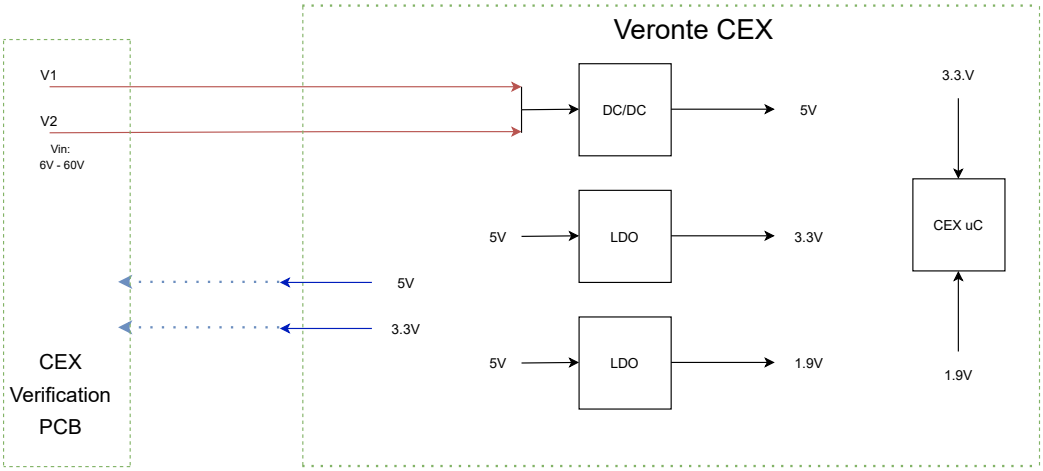


Figura 3.5: Alimentaciones 3.3V y 5V

El correcto funcionamiento de las alimentaciones de 3.3V y 5V que están conectadas a la salida del Veronte CEX garantiza que sus reguladores de tensión están trabajando de manera óptima y asegura que por lo menos una parte del Microcontrolador (uC) del CEX está bien alimentada. Para probarlo, se conectan las alimentaciones de 3.3V y 5V a LEDs conectados a GND, como se observa en la Figura 3.5, de forma que se pueda verificar con una inspección visual rápida que las alimentaciones funcionan correctamente. Para comprobar los niveles de tensión de las salidas son de 3.3V y 5V, respectivamente, se realiza la verificación de Analog-to-Digital Converters (ADCs), que se explicará más adelante (3.6).

3.5. Verificación de GNDs

Para verificar la alimentación de 3.3V a la salida del CEX y que todos los pines de masa del conector del CEX estén conectados a GND, se conecta dicha señal de tensión a cada pin de masa del conector a través de un LED, como se observa en la Figura 3.6. Si un LED no se enciende quiere decir que la conexión a masa de dicho pin no es correcta, o que el LED se ha dañado. La conexión se repite con 7 LEDs y sus correspondientes conexiones a GND, 5 de ellos compartidos para el CEX y el CEX MC y los dos restantes únicamente para el CEX MC.

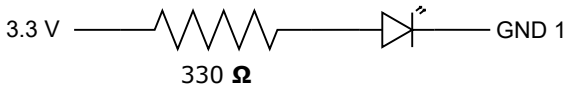


Figura 3.6: Verificación de GNDs.

Comprobar las conexiones de los pines GND es importante porque estos pines pueden utilizarse como referencia a masa de diferentes periféricos conectados al CEX. Si alguna de ellas no estuviera correctamente conectada a GND podría traer problemas en el dispositivo a utilizar, por tener una referencia errónea. Además, los conectores del CEX son bastante delicados, por lo que comprobar cada uno de sus pines es crucial para el correcto funcionamiento del producto.

3.6. Verificación de ADCs

La verificación de ADCs consiste en comprobar que el voltaje de entrada a los pines de ADCs es el mismo que el valor que se lee en el uC. Para ello, se les conectan las salidas de potencia del CEX de 3.3V y 5V y después, por SW se comprueba que el valor recibido es el esperado.

Como se explicó en el apartado 2.2.3, los pines del uC aceptan un voltaje de hasta 3 V, por lo que en la PCB del CEX, para leer los valores deseados sin dañar ningún componente (3.3 V, 5 V, 12 V o 36 V, según el pin del ADC que se utiliza), la señal ADC que entra a la PCB es reducida antes de ser leída por el uC. Las señales a verificar son 8 ADCs:

- Dos de 0V a 3.3 V.
- Dos de 0V a 5 V.
- Dos de 0V a 12 V.
- Dos de 0V a 36 V.

3.6.1. ADC 3.3V

Para comprobar las señales ADCs de 3.3V se les conecta la señal de 3.3 V del CEX a través de un divisor de tensión. De esta forma, a los pines ADCs de 3.3V del CEX les entra una señal de 3.3 V dividida entre dos por el divisor de tensión, es decir, les entran 1.65 V.

El divisor de tensión no es realmente necesario, debido a que las señales analógicas de 3.3V de Veronte CEX están preparadas para recibir tensiones de esa magnitud. Sin embargo, se decide agregarlo para proteger el circuito, evitando que le entren valores de tensiones superiores a lo que puede soportar. Esto se hace porque el objetivo del test es probar las conexiones de HW de la PCB del CEX, más no forzar el amplificador operacional que tiene la placa, debido a que en la PCB del CEX hay un circuito seguidor de tensión por el que pasa la señal antes de entrar al pin del uC del ADC. Las resistencias que se utilizan en el divisor de tensión son de valores altos, 30 kΩ, para limitar la corriente que le entra al operacional.

3.6.2. ADC 5V

Como se observa en la Figura 3.7, y de la misma forma que se explicó en el apartado anterior (3.6.1), para verificar el correcto funcionamiento de las señales ADCs de 5V, se les conecta la salida de potencia de 5 V del CEX.

A una de las señales analógicas de 5V, la señal Analógica 4, se le agrega otro divisor de voltaje en paralelo y se utiliza para seleccionar el modo de funcionamiento de la PCB de verificación, entre Slow, Fast o Burn In. Para ello se utiliza un interruptor, como en la Figura 3.7, que, según su estado, selecciona uno u otro divisor de tensión, de forma que en el pin del uC correspondiente a dicho ADC de la PCB del

CEX, se lee un voltaje u otro, y así, en Production Embention Testing Tool (PETT) se lee el modo seleccionado.

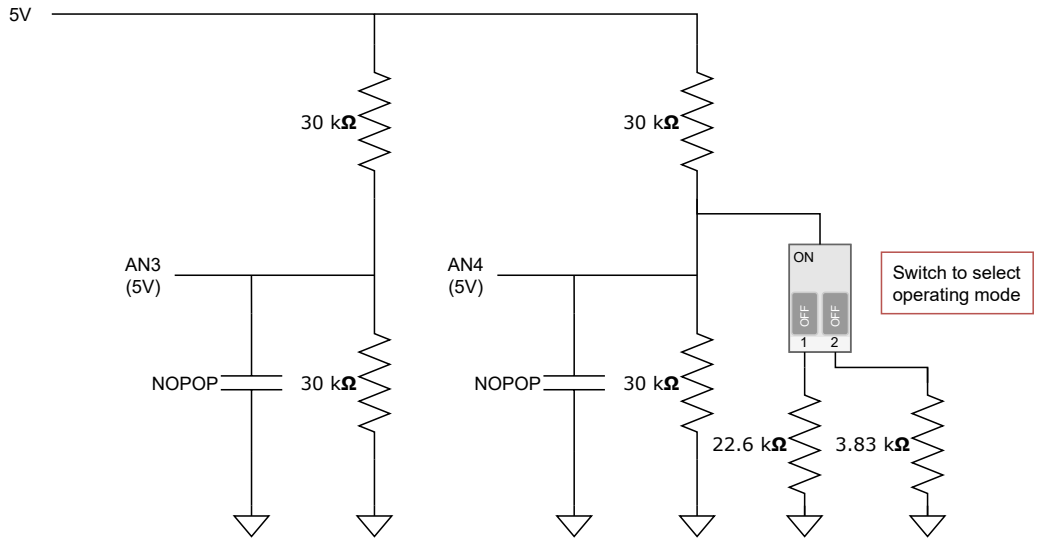


Figura 3.7: Verificación ADCs con señales 3.3V y 5V.

Para la señal Analógica 3, el divisor de tensión está formado por resistencias de 30 kΩ, de forma que los 5 V se dividen entre dos y entran 2.5 V al CEX. Para la señal Analógica 4 los divisores de tensión están formados por las resistencias que se observan en la Figura 3.7, 30 kΩ, 22.6 kΩ y 3.83 kΩ, de forma que:

1. Si el interruptor está apagado (1 en OFF y 2 en OFF), entran 2.5 V al CEX.
2. Si el switch 1 del interruptor está encendido y el 2 apagado, entran 1.5 V a la PCB del CEX.
3. Si el switch 1 del interruptor está apagado y el 2 encendido, entran 508 mV a la PCB del CEX.
4. Si el interruptor está encendido (1 en ON y 2 en ON), entran 448 mV al CEX.

3.6.3. ADC 12V y 36V

Para verificar las señales analógicas de 12V y 36V no hace falta agregar el divisor de tensión, porque las señales están preparadas para más voltaje. Se les conecta la señal de 5 V directamente a los pines de las señales analógicas a estudiar y por SW se comprueba que se leen 5 V. Se agrega una resistencia en serie de 0 Ω en caso de

que sea necesario agregar una resistencia de otro valor para limitar la corriente que entra al pin del ADC.

En todos los casos se deja un condensador como No Populated (NOPOP) por si fuera necesario soldarlo para filtrar señales.

3.7. Verificación de señales PWMs y eCAPs

Para la verificación de los Pulse Width Modulators (PWMs) y de los Enhanced Captures (eCAPs) se hace un loopback como se observa en la Figura 3.8. Se le añade una resistencia de 0 Ω para poder adaptar el circuito y conectarlo de otra forma si fuera necesario. Como hay 8 PWMs a verificar y solo 4 eCAPs, se conectan de dos en dos.

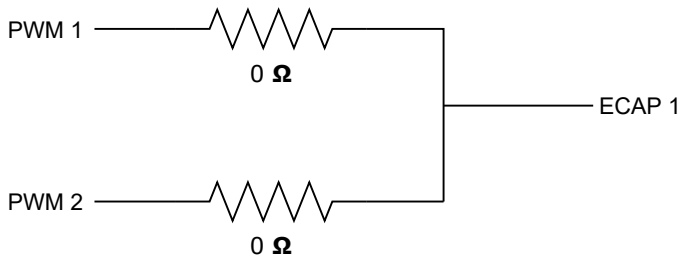


Figura 3.8: Test PWM - eCAP

La idea de la conexión en loopback de esa forma es comprobar que la señal generada por los pines de los PWMs sea la misma que la leída por el eCAPs correspondiente. Si no se lee en el pin del uC la señal que se envía, quiere decir que en la PCB del CEX hay un fallo el circuito que conforma la señal PWM o el que conforma la señal de eCAP.

Debido a que los pines PWMs están conectados de dos en dos con los pines de los eCAPs, como se observa en la Figura 3.8, y que en la PCB del CEX hay un level shifter para cambiar el nivel lógico de los PWMs; mientras se está enviando una señal de PWM, la otra hay que ponerla como salida a nivel bajo, de forma que se garantice un 0 después del level shifter en el pin que no se quiere evaluar en ese momento. Si se configurara como input, habría un estado de alta impedancia y no

estable a la salida del level shiter, dificultando la lectura del PWM que se quiere probar; si la configuración fuera como PWM, la señal que se quiere probar (el otro PWM) podría cambiar por completo y su lectura sería errónea.

3.8. Verificación de señales CAN

Para la verificación de las señales CAN A y CAN B, se hace un loopback entre ellas conectando las partes positivas y negativas entre sí, como se observa en la Figura 3.9.

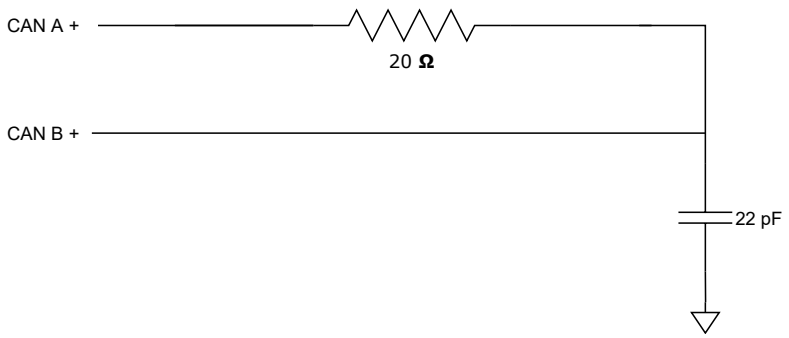


Figura 3.9: Verificación señales CAN.

Al igual que en los otros loopbacks, la idea es comprobar que la señal enviada por el uC es la misma que la que recibe. De forma que se envía una trama por el Controller Area Network (CAN) A y se recibe por el CAN B y viceversa. Si las tramas recibidas no son las mismas que las enviadas, quiere decir que alguna parte del HW del CAN de la PCB del CEX está fallando.

En este loopback, al igual que en el resto de ellos utilizados para comprobar señales, se agregan a la conexión resistencias y condensadores para simular un cable [12]. Esto se hace para lograr una aproximación sencilla a un cable, sin inductancia, porque lo que se quiere comprobar es el HW de la PCB del CEX, más no la efectividad de su diseño.

3.9. Verificación de señales UART

Para la verificación de UART, al igual que en los loopbacks mencionados anteriormente, la idea es comprobar que la señal que se envía desde el uC es la misma que la que se recibe. Para ello se realizan conexiones como en la Figura 3.10, tanto para comprobar las interfaces UART B, como las de UART C que, aunque dicha señal está NOPOP en la PCB del CEX, se agrega a la PCB de verificación el looback por si se quiere probar su funcionamiento.

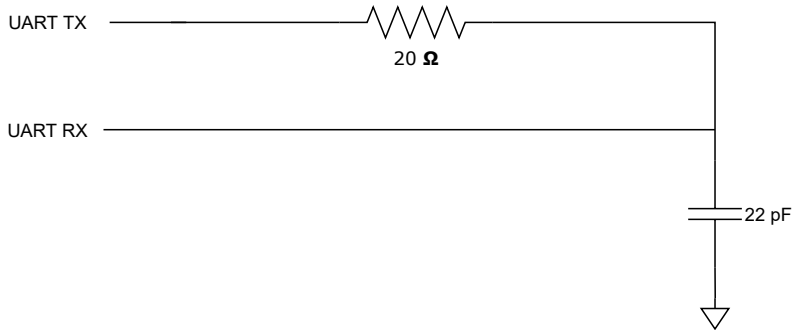


Figura 3.10: Verificación de señales UART.

La señal UART A se utiliza para la comunicación en serie con el ordenador, por lo que enviar y recibir mensajes con la terminal garantiza su funcionamiento.

3.10. Verificación de I2C

Para comprobar el correcto funcionamiento de la interfaz Inter-Integrated Circuit Module (I2C), se conecta la señal de datos, System Data I2C (SDA), y de reloj, System Clock I2C (SCL), a un magnetómetro externo, como se observa en la Figura 3.11. El dispositivo se alimenta a 5V, por lo que para ellos se utiliza la señal de 5V del CEX y, de esta forma, se comprueba también que dicha señal es correcta. Para comprobar la interfaz I2C se hace funcionar el magnetómetro externo.

El magnetómetro externo es otro producto desarrollado en Embention que se comunica por I2C. Como lo que se quiere probar es el funcionamiento del bus y no del dispositivo, la prueba consiste en establecer una comunicación con él.

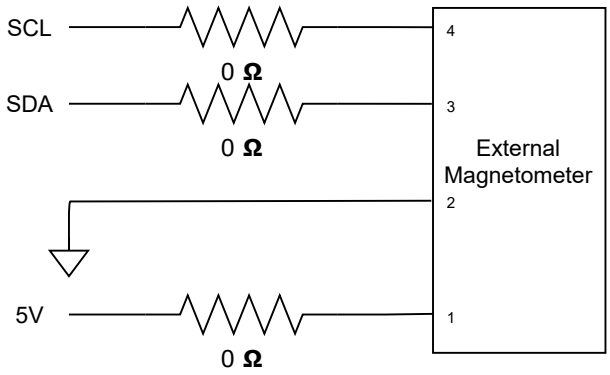


Figura 3.11: Verificación de I2C con magnetómetro externo.

3.11. Burn In Test

Durante el Burn In Test (2.3.9), el CEX es reiniciado después de cada ciclo de PETT en Modo Burn In (2.4.1). Para ello es necesario un HW específico que, cuando PETT termine de pasar los tests, reinicie el CEX.

La PCB de verificación se encarga de reinicio del CEX por medio de un circuito de relés, MOSFETs y puertas lógicas. En cada ciclo de apagado del CEX se cambia la alimentación (7 V o 36 V) y cada dos ciclos, la señal a alimentar (V1 o V2).

El relé que selecciona entre V1 y V2 tiene una resistencia de 0 Ω conectada entre sus salidas para los modos de funcionamiento en los que no se quiera alternar entre las alimentaciones (los modos diferentes al Burn In). Los relés de ON/OFF y de 7V/36V tienen una resistencia NOPOP uniendo sus salidas, por si en algún momento se quieren fijar a un estado.

Para hacer cambiar los relés de estado se utilizan MOSFETs tipo N [13] , que tienen conectados a su surtidor la señal GND y a su drenador el relé correspondiente, de forma que si cambia de estado la señal conectada a su puerta, el voltaje en el drenador cambia de estado.

Cuando termina una iteración de PETT, éste pone en estado alto las señales PWM3 y PWM6 del CEX. Ambas señales están conectadas a la entrada de una puerta AND que, cuando detecta que están las dos en estado alto, saca una señal en estado alto que está conectada al pin TRIG del timer TLC555 a través de un

MOSFET para invertir la señal. Es necesaria la puerta lógica AND para activar el TLC555 porque de otra forma, durante alguno de los test de los PWM, éste se activaría. La salida del TLC555 está conectada a un flip-flop para alternar los voltajes de las alimentaciones (7 V o 36 V), que a su vez está conectado a otro flip-flop para alternar las fuentes de alimentación (V1 o V2).

El TLC555 [14], por su parte, es un circuito temporizador cuya función es producir pulsos de temporización. Cuando la señal de trigger (TRIG) está en estado alto, su salida se mantiene en estado bajo, que es el estado de reposo. Cuando el pin TRIG del TLC555 se pone a un nivel inferior del trigger (aproximadamente 1/3 de su voltaje de alimentación), su salida se pone en estado alto, hasta transcurrido el tiempo determinado por la ecuación 3.1, donde RA es la resistencia entre el pin DISCH y RESET, en este caso 1 MΩ, y C el condensador conectado entre THRES y GND, en este caso de 10 uF.

$$T = 1{,}1 * RA * C \tag{3.1}$$

CEX ON/OFF

La salida del TLC555, llamada 555_OUT en el diseño esquemático, está conectada a un relé que activa o desactiva la conexión del CEX con la alimentación. De esta forma, lo apaga cuando las señales PWM3 y PWM6 están en estado alto y lo enciende un ciclo del TLC555 después.

Supply switching

La señal 555_OUT también está conectada a la señal de reloj de un flip-flop tipo D (SN74LVC1G80 [15]) con trigger en el flanco de subida, para el circuito "Supply switching", encargado de alternar las alimentaciones entre 7 V y 36 V.

En el SN74LVC1G80 la entrada de datos (D) se transfiere a su salida ($\bar{Q}$) cuando la señal de reloj (CLK) tiene un flanco de subida, es decir, cuando el TLC555 cambia a estado alto, lo que ocurre cuando se activan PWM3 y PWM6.

El pin D está conectado al pin $\bar{Q}$ por medio de una resistencia. PWR_SWITCH, es la señal de salida y a su vez, está conectada a GND por medio de una resistencia, como se observa en Figura 3.11.

De esta forma, la señal PWR_SWITCH cambia de estado con cada flanco de subida de la señal 555_OUT.

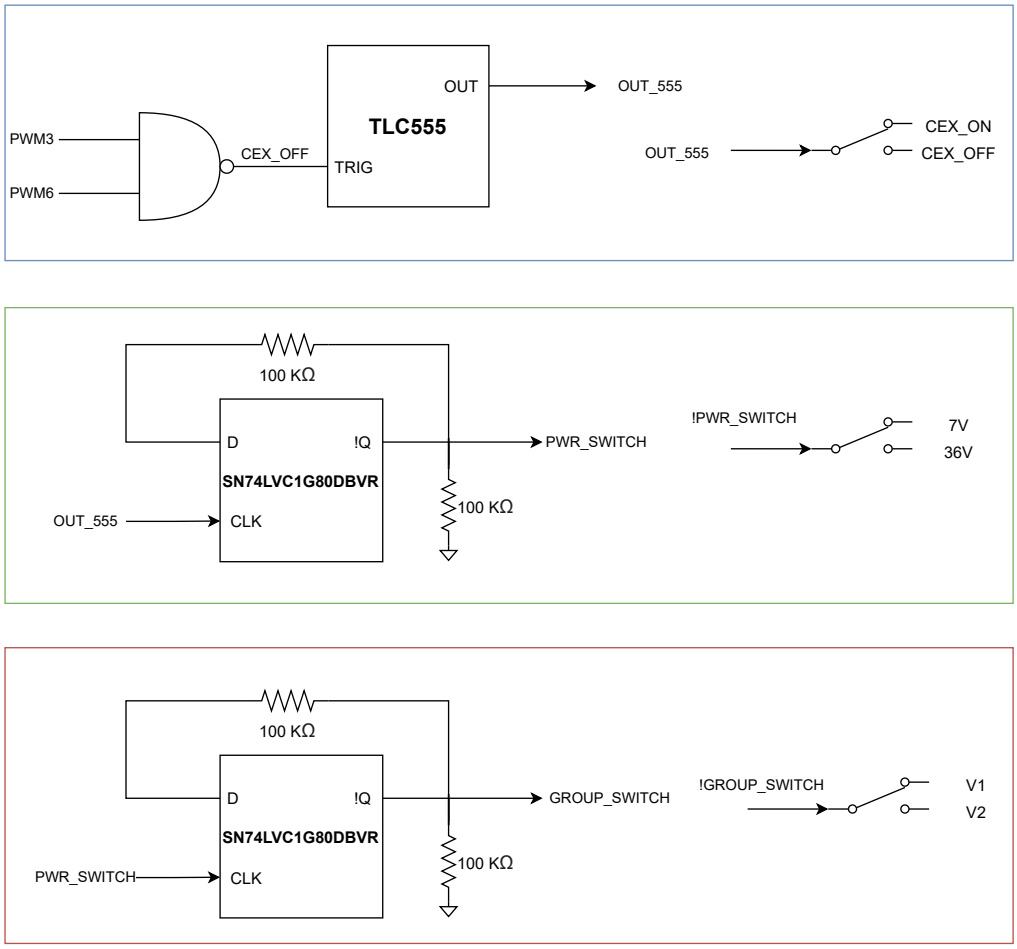


Figura 3.12: Circuito Burn In Test.

Supply group select

La señal, PWR_SWITCH es salida del circuito "Supply switching" y está conectada al pin CLK del flip-flop tipo D (SN74LVC1G80) del circuito "Supply group select". El circuito es básicamente igual que el anterior (3.11) y su salida, llamada GROUP_SWITCH, activa el relé que cambia entre V1 y V2.

De esta forma, la señal GROUP_SWITCH cambia de estado con cada flanco

de subida de la señal PWR_SWITCH y cada dos flancos de subida de la señal 555_OUT.

En la Figura 3.12 se representa un esquema de las conexiones realizadas. Aunque según el diagrama las señales PWM3 y PWM6 están conectadas a una puerta NAND, realmente la conexión es a una AND y un MOSFET que la niega. En la Figura 3.13 se observa el resultado de la simulación del circuito del Burn In Test.

3.12. LEDs de PASS y FAIL

Además de los LEDs de las alimentaciones de 3.3V, de 5V y los de GND, a la PCB de verificación se le agregan dos LEDs más, uno rojo para señalar si un test ha fallado y uno verde para señalar si ha pasado. El LED de *PASS* está conectado a la señal PWM8 y, por el SW PETT se enciende cuando corresponde; mientras que, por su parte, el LED de *FAIL* está conectado a la señal PWM1 y, de la misma manera, por SW, se enciende cuando corresponde.

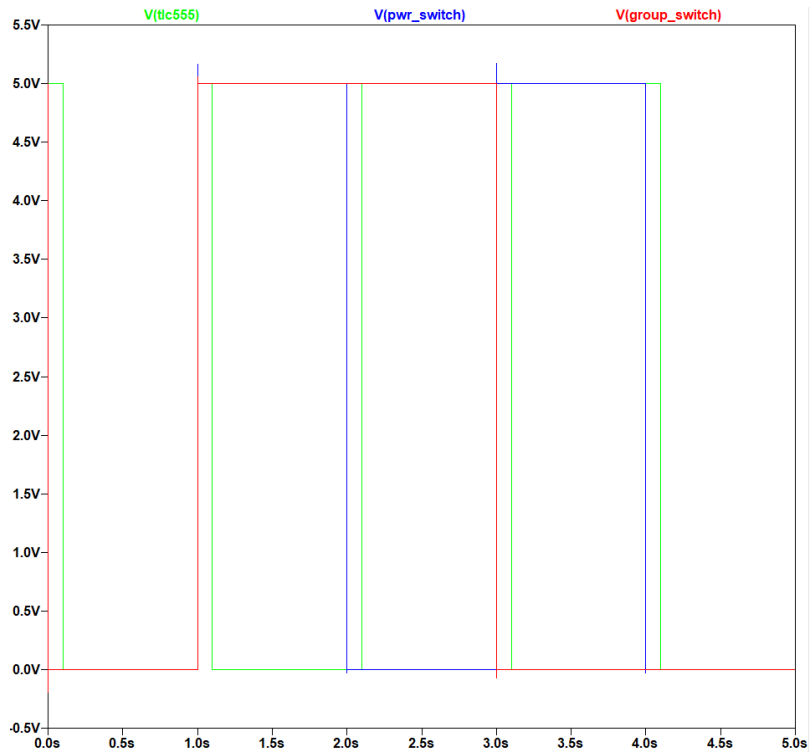


Figura 3.13: Simulación Circuito Burn In Test.

3.13. Señales RS232 y RS4885

La señal RS232 del CEX MC, como se explicó en el apartado 3.3, se comprueba estableciendo una comunicación con el ordenador. Para ello, el interruptor de selección del producto se pone en el estado "CEX MC" y al conector RS232 se le conecta un cable RS232-USB.

La señal RS485 del CEX MC, por su parte, al igual que el resto de señales de comunicación, se comprueba a través de un loopback, verificando que la señal enviada es la misma que la recibida. A nivel de PETT, el funcionamiento es el mismo que el Test de UART, porque la señal del uC sale como una señal Serial Communications Interface (SCI), sin embargo, a nivel de HW, el loopback se realiza uniendo las señales TX+ con RX+ y TX- con RX-, debido a que la señal RS485 es diferencial.

3.14. Desarrollo del Prototipo Funcional

La PCB de verificación del CEX fue desarrollada siguiendo los requisitos planteados como objetivos en este trabajo:

- Se mejora la línea de producción al desarrollar una PCB de verificación simple, versátil y fácil de utilizar. Además, se agregan los distintos pasos al proceso de verificación:
    - Inspección Visual
    - Shaking Test
    - Verificación de Cortocircuitos
    - Verificación de Corrientes y Voltajes
    - Verificación Funcional, verificando todas sus interfaces
    - Burn In Test
    - Verificación de Tropicalizado

- El diseño permite la verificación de las interfaces del CEX y CEX MC:
  - 2 entradas de alimentación: se comprueba su funcionamiento en la alimentación del CEX y la redundancia de la alimentación durante el Burn In Test.
  - 2 salidas de potencia (3.3V y 5V): se comprueban con el test de las GNDs, de los ADCs. Además, con una inspección visual a la la PCB de verificación, se puede saber si están conectadas las señales, porque hacen que se enciendan los correspondientes LEDs.
  - 8 PWMs y 4 eCAPs: se verifican haciendo loopbacks entre ellos de forma que se conectan dos PWMs a un eCAP.
  - 2 CAN: se verifican utilizando loopbacks entre CAN A y CAN B, se comprueba que se recibe la misma señal que se envía.
  - 3 UART (1 de ellas NOPOP): UART A se verifica con la comunicación con el ordenador. UART B, por su parte, utilizando loopbacks entre la señal TX y RX, se comprueba que se recibe la misma señal que se envía. UART C de la misma forma que UART B, en el caso de que esté soldada.
  - 1 I2C: se verifica utilizando un magnetómetro externo y comprobando que es posible la comunicación con el dispositivo.
  - 8 ADCs: se comprueban leyendo en los pines correspondientes del uC los valores de las salidas de potencia.
  - RS232 del CEX MC: se comprueba utilizándolo para la comunicación con el ordenador.
  - RS485 del CEX MC: se verifica, al igual que las señales UART, haciendo un loopback entre su señal TX y RX y comprobando que la señal que se envía es la misma que la que se recibe.
- Debido a que el diseño de la PCB de verificación debía ser en dos capas para ahorrar costes, por comodidad, tiene componentes en solo una de ellas.
- Para permitir la compatibilidad con el CEX y CEX MC, consta de, por un lado, los conectores del CEX, y por otro, el del CEX MC. Además, tiene un interruptor para seleccionar si se quiere la comunicación utilizando una señal UART, que es el caso del CEX, o una señal RS232, para el CEX MC.

- Permite la alimentación de 12 V a 36 V, teniendo un convertidor DC/DC que puede trabajar con ese rango de voltajes de entrada.
- El HW soporta temperaturas hasta 85 ºC, debido a que los componentes han sido elegidos para trabajar en esa temperatura.
- Para detectar los fallos lo antes posible en la línea de producción, primero se revisan visualmente los componentes, a continuación se comprueba su correcta soldadura, se buscan cortocircuitos, se revisan voltajes y después de un test funcional, se realiza el Burn In Test, con el objetivo de detectar posibles fallos futuros.

En el Anexo B se encuentra el diseño esquemático de la PCB de verificación, mientras que en el Anexo C la ubicación de componentes en el diseño board, así como las pistas de la cara *top* y *bottom* de la PCB.

En la Figura 3.14 se puede observar la PCB de verificación ya fabricada.

Fueron fabricadas 5 PCBs de verificación, cuyo Bill of Materials (BoM) se encuentra en el Anexo D. En el BoM está añadido el rango de temperatura de operación de cada componente de la PCB, todos se encuentran dentro del rango deseado, es decir, entre -40 ºC y 85 ºC, menos el interruptor para seleccionar el modo de funcionamiento y el interruptor para seleccionar entre CEX y CEX MC; sin embargo, como los interruptores, estando en otros productos de la empresa, se han sometido a las temperaturas máximas deseadas en este proyecto, se utilizan de todas formas en la PCB de verificación.

El presupuesto y la orden de producción se encuentran en el Anexo E.

3.14.1. Revisión del Prototipo Funcional

Una vez fabricadas las PCBs, al querer comprobar su funcionamiento se detectaron los siguientes problemas o detalles que no se tomaron en cuenta. Después de resolverlos, se hicieron las modificaciones necesarias en las PCB de verificación. Sin embargo, no se fabricaron más porque ya trabajaban correctamente.

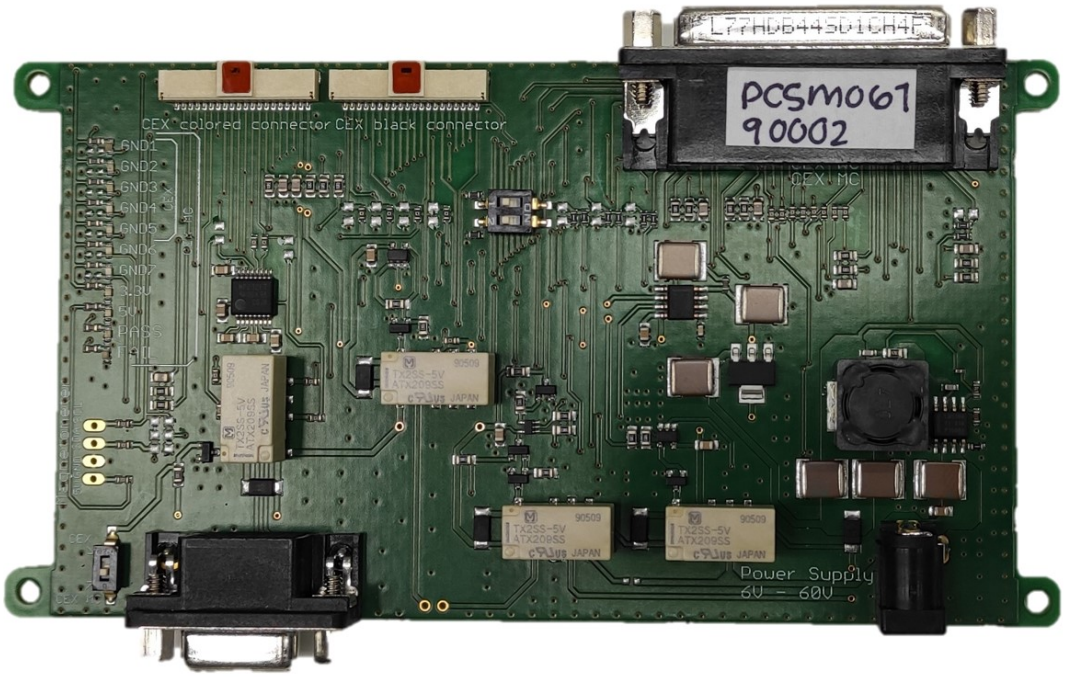


Figura 3.14: PCB de verificación del CEX.

En el Acceptance Test Procedure (ATP) quedan registrados todos los re-trabajos que hay que hacer en las PCBs, en caso de que quieran replicarse.

En los siguientes apartados se encuentran las modificaciones que se realizaron a la PCB de verificación para que funcionara correctamente.

Relé ON/OFF

Diseñando la PCB de verificación hubo un error en el estado inicial de un relé, por lo que el CEX no se encendía. Se solución cambiando la conexión del relé.

Señales CAN

Para que el bus CAN trabaje correctamente necesita una resistencia de terminación de 120 Ω. Sin embargo, al momento de diseñar la PCB de verificación no se tomó en cuenta que el CEX, en la última release de su diseño, no las tiene soldadas, sino que están NOPOP.

Debido a esto, tuvo que soldarse una resistencia de terminación a la PCB de

verificación entre la señal CAN_A_P y CAN_A_N, es decir, entre las señales diferenciales del bus CAN A, tanto del CEX como del CEX MC. Solo es necesario soldarlas a un bus y no también al CAN B porque al hacer un loopback entre las señales y que la distancia entre ellas es corta, la misma resistencia sirve para los dos.

Alimentación driver RS232

Para alimentar el driver RS232 se utilizaba la alimentación de potencia del CEX de 3.3V. Sin embargo, para evitar problemas de tener el CEX conectado al driver de RS232 sin estar éste alimentado, es decir, una conexión Universal Serial Port (USB) - UART directa, sin conversor, se cambió la alimentación del driver de RS232 por los 5V que tiene la PCB de verificación.

Resistencias de PWMs

Durante el diseño se agregaron a los PWMs resistencias de 0 Ω por si hacía falta sustituirlas por unas de otro valor.

Haciendo pruebas con la PCB de verificación, surgió la necesidad de configurar los PWMs que no se están probando como General-purpose I/O pin (GPIO), output, nivel bajo, por lo explicado en 3.7. Debido a esta configuración, para evitar que entre corriente a los pines, específicamente al pin del uC del PWM que está conectado al PWM que se quiere probar (debido a que están conectados dos PWMs a un eCAP), se tuvieron que cambiar las resistencias de los PWMs por unas de 10 kΩ.

Resistencias de LEDs

Hay dos LEDs que se encienden al final de cada test, uno verde de *PASS* y uno rojo de *FAIL*. Para activar los LEDs, por SW se envía una señal utilizando las señales PWM1 y PWM8 y las resistencias de los LEDs eran de 330 Ω. Sin embargo, como se añaden las resistencias a los PWMs (3.14.1), hay que cambiar el valor de las resistencias de los LEDs a 910 Ω, porque se forma un divisor de tensión entre las señales PWMs y eCAP y otro en las señales PWM1 y PWM8 con los LEDs,

ocasionado que se redujera el nivel de tensión de los PWMs y que el eCAP no fuera capaz de ver el cambio de nivel de la señal.

Resistencias de ADCs

Al momento de diseñar el circuito de verificación de los ADCs (3.6) no se tomó en cuenta que dentro de la PCB del CEX también hay divisores de tensión para reducir el nivel lógico de las señales analógicas que entran y poderlas leer en el uC. Debido a esto, conectando la PCB de verificación, quedan dos divisores de tensión unidos, como se observa en la Figura 3.15, y el valor de tensión recibido en el pin del ADC es menor que el esperado al momento de diseñar la PCB.

Para las señales analógicas de 3.3V, que son divididas entre 2 por un divisor de tensión en la PCB de verificación, en vez de llegarle 1.65 V al uC, como se esperaba, llegan 670 mV, debido al divisor de tensión en la PCB del CEX.

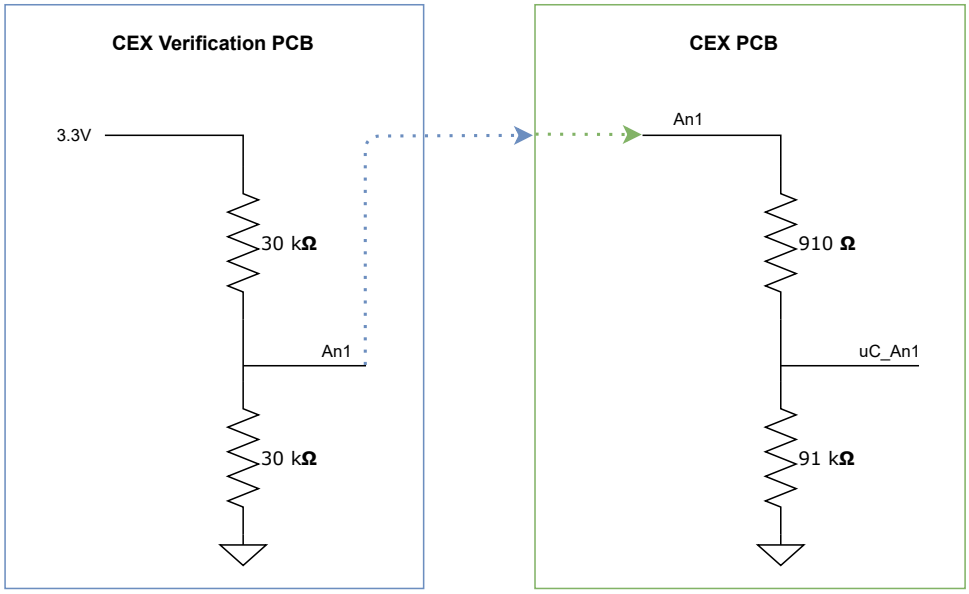


Figura 3.15: Problema ADCs.

Para las señales de 3.3V en concreto, no es problemático, porque se resuelve cambiando el parámetro de multiplicación en PETT, sin embargo, en el caso del interruptor que selecciona el modo de funcionamiento, en lugar de tener 4 estados disponibles, como se explicó en 3.6.2, hay 3, porque en lugar de recibirse 2.5 V, 1.5 V, 508 mV y 448 mV en el CEX, se reciben 1 V, 802 mV, 390 mV y 353 mV,

siendo los últimos valores muy parecidos. Esto puede causar que se lean resultados incorrectos al seleccionar el modo de funcionamiento debido a la precisión del uC.

Burn In Test

Al probar el Burn In Test se observó que uno de los flip-flop no trabajaba correctamente, por lo que las alimentaciones no se alternaban de la forma esperaba. Haciendo pruebas se observó que se debía a que el flip-flop que seleccionaba entre V1 y V2, en algunos casos, de forma no cíclica ni reproducible, cambiaba de estado cuando se encendía la PCB del CEX, cuando el comportamiento esperado era que lo hiciera solo al apagarla. La secuencia era:

1. Se enciende el CEX con V1 a 7 V y V2 a 0 V.
2. Se apaga, cambian los flip-flop para conectar V1 a 36 V y V2 a 0 V, se enciende.
3. Se apaga, cambian los flip-flop para conectar V1 a 0 V y V2 a 7 V, se enciende.
4. Se apaga, cambian los flip-flop para conectar V1 a 0 V y V2 a 36 V, se enciende, cambian los flip-flop para conectar V1 a 7 V y V2 0 V.

Debido a esto, se saltaba un estado en la conmutación, haciendo que una alimentación no se midiera. Aunque era un error que no se repetía de forma continua, impedía comprobar de la forma deseada las alimentaciones y la redundancia de ellas en la PCB del CEX.

En la Figura 3.16 se observa una captura de la pantalla de la medida del osciloscopio en el momento en el que la señal GROUP_SW cambia de estado cuando se enciende la PCB del CEX. Se observa también que, además del cambio de la señal GROUP_SW, hay una oscilación de casi 1 V en la señal de alimentación del flip-flop (señal 5V_SERVICE, CH4 en 3.16).

Para intentar solucionar el problema se probó:

1. Agregar a la resistencia de la salida de PWR_SW un condensador de 0.1 uF en paralelo, con el objetivo de desacoplar el ruido que se genera en el cambio de estado del flip-flop.

2. Agregar un condensador en paralelo al condensador de la alimentación del flip-flop de la señal GROUP_SW de 22 uF, con el objetivo de estabilizar la alimentación y que el flip-flop no cambie de estado.
3. Agregar un condensador de 68 uF en la entrada de 36V (Vin) de la PCB de verificación, para minimizar la oscilación en el arranque del CEX.
4. Como los pasos anteriores no funcionaron, se desoldaron los condensadores, porque se habían hecho muchos cambios y no se estaba consiguiendo resolver el problema.
5. Agregar una resistencia a GND a la entrada de la señal CLK del flip-flop (señal PWR_SW) de 4.7 kΩ (pull down) y la oscilación no hiciera cambiar al flip-flop de estado.
6. Agregar un condensador de tantalio de 4.7 uF en paralelo al condensador de la alimentación del flip-flop de la señal GROUP_SW, para reducir la oscilación de la alimentación. Con este condensador si se consiguió el objetivo.
7. Agregar un condensador de 330 nF en paralelo a la resistencia que se agregó en el paso 5, para reducir la oscilación.

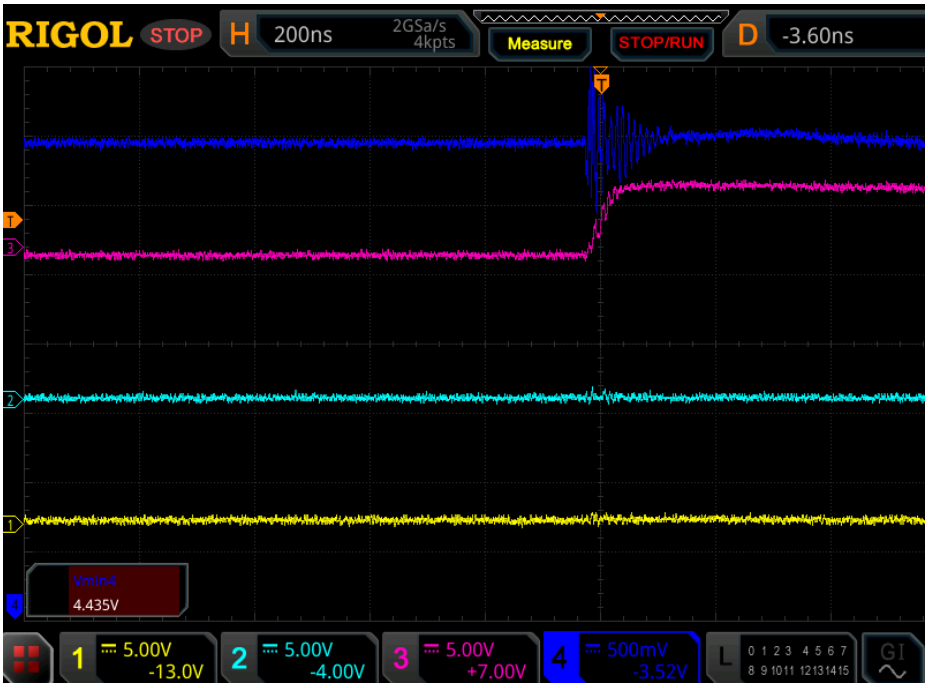


Figura 3.16: Problema Burn In. CH1 signal 555OUT, CH2 signal PWR_SW, CH3 signal GROUP_SW, CH4 signal 5V_SERVICE (alimentación del flip flop)

Después de todas las pruebas y con todas esas modificaciones, se consiguió que el flip-flop siguiera el comportamiento esperado y que cambiara de estado de forma

correcta durante todo el Burn In Test en la cámara climática. De esta forma se cumplió el objetivo que durante el Burn In test se siguiera la siguiente secuencia:

1. Se enciende el CEX con V1 a 7 V y V2 a 0 V.
2. Se apaga, cambian los flip-flop para conectar V1 a 36 V y V2 a 0 V, se enciende.
3. Se apaga, cambian los flip-flop para conectar V1 a 0 V y V2 a 7 V, se enciende.
4. Se apaga, cambian los flip-flop para conectar V1 a 0 V y V2 a 36 V, se enciende.
5. Se repite hasta alcanzar las 30 iteraciones de PETT.

El correcto funcionamiento de los flip-flops durante el Burn In Test permite verificar las dos alimentaciones en dos niveles de voltaje durante dicha prueba de temperatura. Al ser una prueba de estrés, cuando el Burn In Test termina de manera exitosa se asegura que los componentes están correctamente soldados y operativos.

CAPÍTULO 4:

DESARROLLO DE SOFTWARE

Para la verificación del CAN Expander (CEX) y del CEX MC en la línea de producción, además de la Printed Circuit Board (PCB) de verificación, hace falta un Software (SW) para la ejecución de los tests.

Los tests de verificación se programan en el Microcontrolador (uC) del Veronte CEX. Para ello, se utiliza el Integrated Development Environment (IDE) Code Composer Studio (CCS), de Texas Instruments. Para desarrollar el código se utilizan distintas librerías comunes de Embention programadas en C++, por lo que el código de verificación, conocido como Production Embention Testing Tool (PETT), también está desarrollado en ese lenguaje. Dichas librerías comunes sirven de base para el código de PETT, se utilizan para llenar los registros, configurar las interfaces, programar las interrupciones y todo el funcionamiento a más bajo nivel.

PETT está conformado por distintos archivos header (.h) con las instancias de las clases y métodos utilizados; y archivos de implementación (.cpp), donde se implementan esos métodos y clases.

De esta forma, el código se organiza, a grandes rasgos, en un archivo principal (main.cpp), un archivo de control (SerialTestCtrl) y archivos para los distintos tests para verificar.

Además, el código de PETT se apoya en templates para pasar parámetros entre clases. Los templates en C++ son funciones especiales que pueden operar con tipos genéricos. Esto permite crear un template de una función que puede ser adaptado a más de una clase sin repetir todo el código para cada una de ellas. Los parámetros de template permiten pasar también tipos a una función, de la misma forma que los

parámetros de función normales se pueden usar para pasar valores a una función. Estos templates de funciones pueden utilizar los parámetros que reciban como si fueran cualquier otro tipo normal.

Por otro lado, para programar el uC se necesita un archivo .cmd [16] con las especificaciones de la gestión de la memoria flash del uC, también común con el del SW del CEX pero adaptado a las necesidades de PETT.

Para utilizar algunas variables definidas en otras librerías, como Array, utilizada para trabajar con vectores de datos, se necesita reservar memoria. Para ello se utiliza la clase Memmgr (Memory Manager) y se selecciona si se reserva memoria externa o interna. Aunque el CEX no tiene memoria externa, en el .cmd se reserva una parte de la memoria flash como "externa" para poder utilizar dicha clase.

Para escribir por pantalla se utilizan las distintas funciones de Logger.h, explicado más adelante (4.12).

A continuación, en este apartado se van a explicar las distintas clases desarrolladas para el correcto funcionamiento de PETT del CEX.

4.1. main.cpp

Es la función principal. En ella se declaran las interrupciones, se reserva la memoria necesaria para el funcionamiento de PETT, se declaran las instancias necesarias para el desarrollo de la clase y se enumeran los tests a pasar. Además, se desarrolla la lógica del Burn In Test.

La dirección de la flash donde se almacena el address se pasa como parámetro al Test Address, la Aplicación Write Address y al SerialTestCtrl.

Además de los tests para los que se necesita la PCB de verificación, explicados en el Capítulo 2, en PETT se comprueban las señales de alimentación V1 y V2 y el sensor de temperatura de la PCB del CEX.

Adicionalmente, el *main* tiene un bucle infinito que llama a la función *step* de la

clase SerialTestCtrl, que será explicada más adelante (4.2).

Configuración

A continuación, en el main se configuran todos los pines de los Pulse Width Modulators (PWMs) en el uC como señales General-purpose I/O pin (GPIO), salidas, a nivel bajo. Esto se debe a que los PWMs salen de la PCB del CEX a través de un level shifter que cambia su nivel lógico de 3.3V a 5V, como se observa en la Figura 4.1. Debido a esto y a que, como se explicó anteriormente (3.11), los PWMs en la PCB de verificación se utilizan, además de para el test PWM-Enhanced Capture (eCAP) (3.7), para reiniciar la PCB en el Burn In Test, se necesita garantizar un 0 a la salida del level shifter cuando no se desea el reinicio del CEX. Con la configuración de los pines de los PWMs como salida a nivel bajo se garantiza que de la PCB del CEX sale un 0 después del level shifter y no un estado incierto, como se obtendría configurando los pines como inputs.

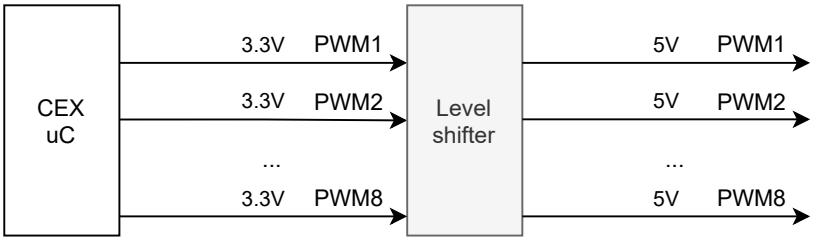


Figura 4.1: PWM level shifter.

Para configurar los GPIOs a nivel bajo se llama a la función *clear* de la clase GPIO del código del CEX, que se encarga de escribir en el registro GPxCLEAR (x = A, B, C) correspondiente.

Modo de funcionamiento

A continuación, en el main se lee el modo de funcionamiento seleccionado en la PCB de verificación en el pin del Analog-to-Digital Converter (ADC) 4 y se ejecutan los tests según dicha selección, como se observa en la Figura 4.2. Para ello se llama a la función *get_ execution_ mode* de la clase SerialTestCtrl, que será explicada más adelante (4.2).

Si el modo fuera Fast Test, los tests empiezan a ejecutarse justo cuando se enciende el producto a verificar. Sin embargo, en este caso se excluye el test del uAddress, debido a que este modo se selecciona cuando el producto acaba de ser programado, después de la verificación de voltajes y corrientes, con el objetivo de descartar cualquier fallo antes de hacer todo el proceso de verificación y producción y, en ese momento, no se ha establecido el uAddress del producto a verificar.

Si el modo seleccionado fuera Burn In Test, se excluyen los tests de alimentación de V1 y V2, porque durante el Burn In se alterna entre las alimentaciones y siempre hay una de ellas conectada a masa; esto es importante porque durante dicha prueba, al momento de fallar un test se detienen las ejecuciones. Debido a esto, para verificar las alimentaciones, en este caso, se hace con una función especial, *test_power_an*.

El modo Slow puede ejecutar los tests que se quieran. En este caso no hay ninguna exclusión y se pueden repetir los tests cuantas veces sea necesario.

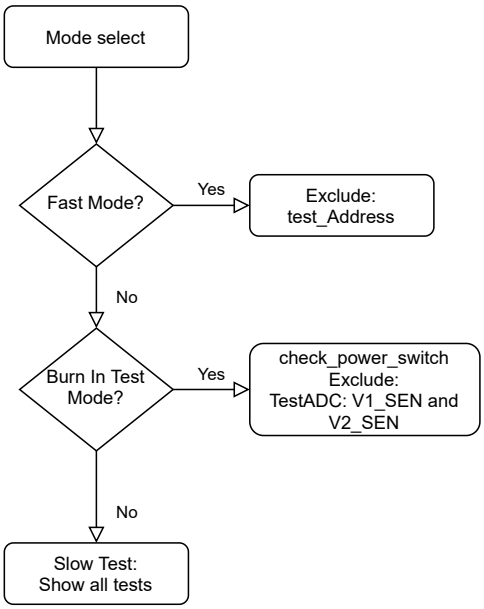


Figura 4.2: Mode select.

A continuación, se enumeran los tests que conforman PETT del CEX:

1. Test Address
2. Test ADC: V1 Sen
3. Test ADC: V2 Sen
4. Test ADC: AN1(3.3V) and 3.3V Regulator (needs 3.3V to AN1(3.3V))

5. Test ADC: AN2(3.3V) and 3.3V Regulator (needs 3.3V to AN2(3.3V))
6. Test ADC: AN3(5V) and 5V Regulator (needs 5V to AN3(5V))
7. Test ADC: AN4(5V) and 5V Regulator (needs 0.7V|2.15V to AN4(5V))
8. Test ADC: AN5(12V) and 5V Regulator (needs 5V to AN5(12V))
9. Test ADC: AN6(12V) and 5V Regulator (needs 5V to AN6(12V))
10. Test ADC: AN7(36V) and 5V Regulator (needs 5V to AN7(36V))
11. Test ADC: AN8(36V) and 5V Regulator (needs 5V to AN8(36V))
12. Test Serial Communications Interface (SCI), UART B: UART B loopback
13. Test Controller Area Network (CAN): CAN A-B loopback
14. Test Inter-Integrated Circuit Module (I2C): MMC5883MA External
15. Test PWM: PWM and PWM to eCAP1
16. Test PWM: PWM and PWM to eCAP2
17. Test PWM: PWM and PWM to eCAP3
18. Test PWM: PWM and PWM to eCAP4
19. Test Temperature: Temperature sensor test

En PETT, además de tests, hay operaciones. Éstas son opciones disponibles para necesidades que no siempre están presentes. En el CEX las operaciones son:

- App change uAddress: para cambiar el address del CEX o CEX MC.
- LTTest: para realizar un test de una duración determinada que consiste en iteraciones continuas de PETT.
- Test SCI, Universal Asynchronous Receiver-Transmitter (UART) C: en el caso en que la UART C esté soldada (por defecto está No Populated (NOPOP)).

Burn In Test

A continuación, en el main se desarrolla el Burn In Test, como se representa en la Figura 4.3.

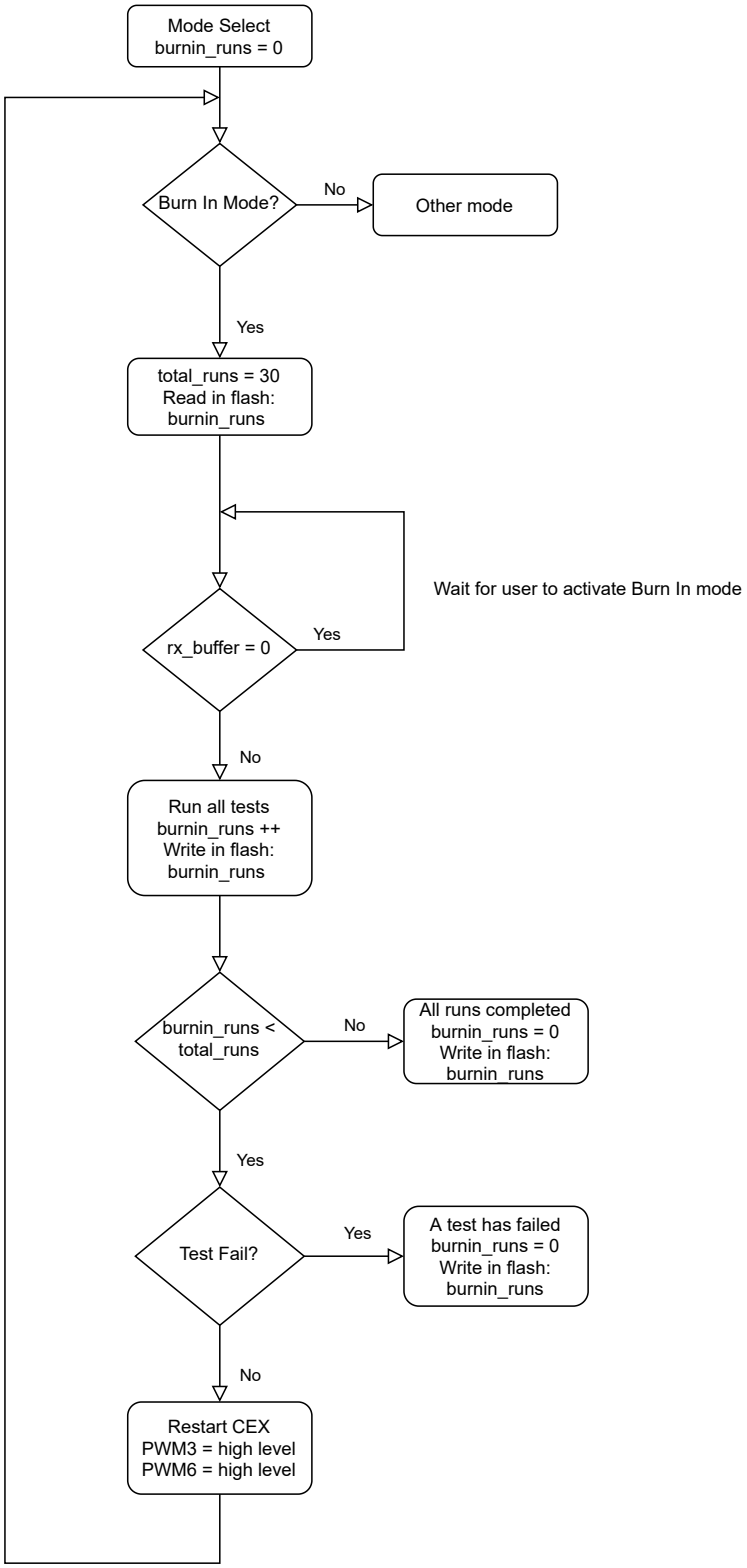


Figura 4.3: Burn In Test.

Como se explicó anteriormente (2.3.9), este test consiste en ciclos de encendido y apagado del CEX alternando las alimentaciones. Durante cada ciclo de encendido

se realizan todos los tests para comprobar el correcto funcionamiento del producto. Al entrar en el Burn In Test Mode se imprime en pantalla el uAddress, se lee en la flash la cantidad de iteraciones que lleva el test, se muestra un mensaje en pantalla de "Press any key to start Burn In" y se espera recibir algo en el buffer (rx_buffer) para iniciar el test. A continuación se suma 1 a la variable que cuenta las iteraciones del burn in (burnin_runs) y se ejecutan todos los tests. Si algún test falla, no se reinicia el CEX, se pone burnin_runs = 0 y se imprime un mensaje de error en la pantalla; en caso contrario, se escribe en la flash las iteraciones que lleva el test (burnin_runs), se reinicia el CEX configurando los pines de los PWM3 y PWM6 a nivel alto y se vuelve a repetir el proceso hasta llegar al total de iteraciones, que es 30.

Para evitar la necesidad de la interacción humana durante el Burn In Test, este se desarrolla pensando en utilizar Valkiria, una herramienta de Embention que se encarga de automatizar el proceso. Para adaptar el test a esta herramienta, se utiliza la frase "Press any key to start Burn In:", al momento de iniciar el test, para que automáticamente empiece el proceso, la frase "A test has failed! " cuando un test falla y "All runs completed! " cuando las iteraciones terminan; sin embargo, debido a la rapidez del uC del CEX y que Valkiria está diseñada pensando en el producto principal de la empresa, Veronte, es necesario repetir este último mensaje porque en algunos casos no le da tiempo a la herramienta de detectarlo y avisar al operario que han terminado las iteraciones.

En la Figura 4.4 se observa el contador que se agrega para esta tarea.

En algunos casos, debido al divisor de tensión que se forma por el loopback PWMs-eCAP, activar las señales PWM3 y PWM6 no basta para reiniciar el CEX, por lo que si después de un tiempo no se ha reiniciado, se encienden las señales PWM4 y PWM5 también (los PWMs también conectados al loopback) para forzar el reinicio, como se observa en la Figura 4.4.

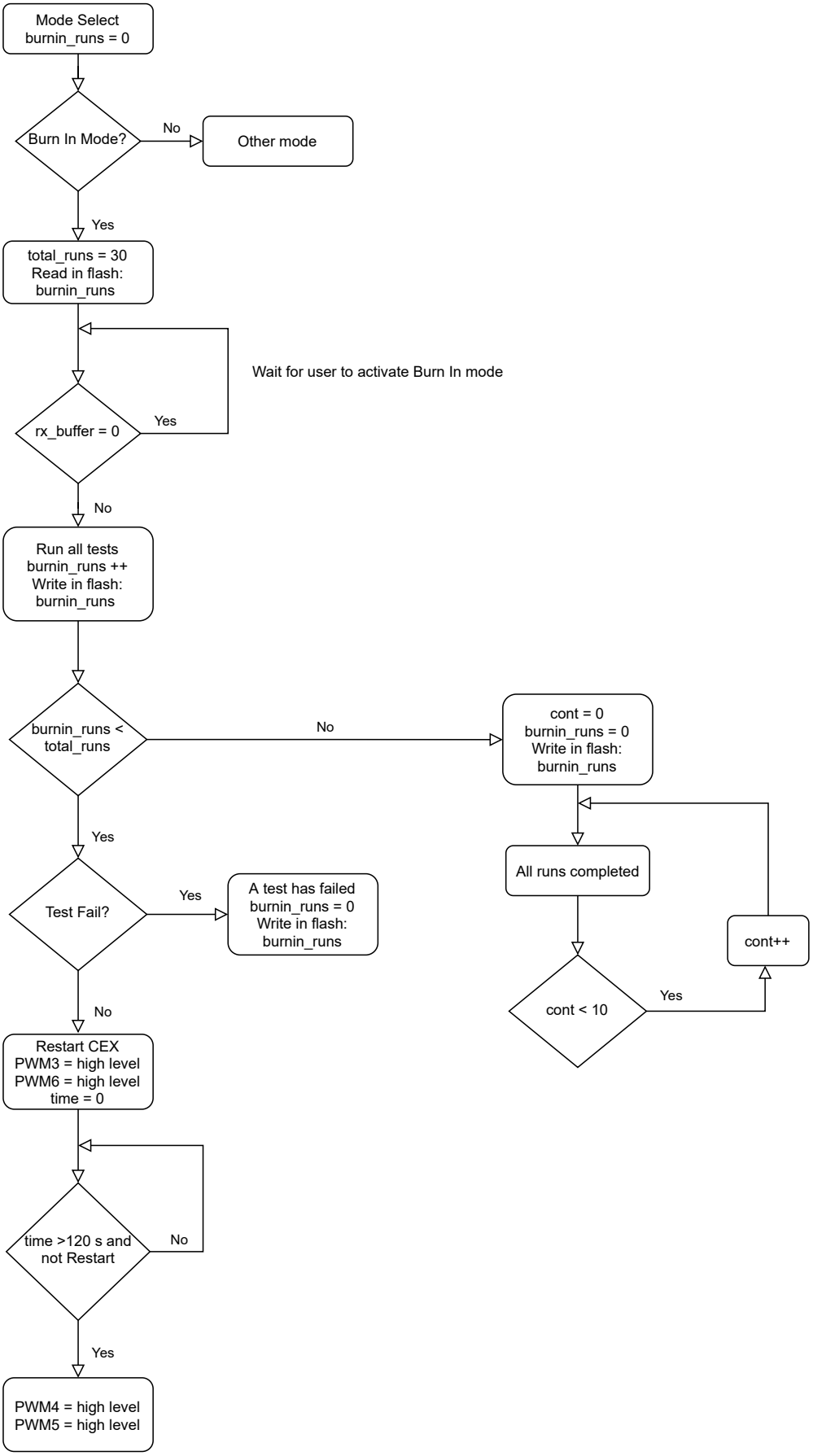


Figura 4.4: Burn In Test.

4.2. SerialTestCtrl

Después del main (4.1), SerialTestCtrl es la clase más importante del código. En ella:

- Se configura el modo de funcionamiento de la PCB de verificación: Fast, Slow o Burn In.
- Se establece la comunicación con el ordenador:
  - Se establece que la comunicación es por UART A.
  - Se escribe en pantalla (una terminal): se imprimen el main (los tests), el modo de funcionamiento seleccionado y los resultados.
  - Se lee la opción seleccionada por el usuario.
- Se diferencian los tests Slow, Fast y las operaciones.
- Se registra si un test tuvo como resultado "Pass" o "Fail" y se enciende el Light-Emitting Diode (LED) correspondiente.
- Se ejecutan los tests según la opción seleccionada por el usuario.

Además, al inicio de la clase se definen dos variables que se utilizan para fijar la versión de PETT.

El constructor de la clase inicializa las variables a utilizar:

- sci: variable para establecer la comunicación por SCI. Se inicializa pasándole como parámetro el puerto SCI-A, ya que es por el que se quiere establecer la comunicación.
- rx_buffer: variable de tipo array, se inicializa con un tamaño de 300 y se almacena en la memoria externa. Se utiliza para almacenar lo leído por serie.
- state: hace referencia a una enumeración que tiene elementos de los estados posibles: desconectado, conectado, espera, datos disponibles, procesando y test automático finalizado.
- tests_slow: variable de tipo array, se inicializa con un tamaño de 50, una constante definida como máximo número de test, y se almacena en la memoria externa. Se utiliza para registrar los tests del modo slow.

- tests_fast: al igual que *tests_ slow*, pero para registrar los tests fast.
- tests_ok: variable de tipo booleano. Se utiliza para registrar si un test ha pasado correctamente o no.
- ops: variable de tipo array, se inicializa con un tamaño de 16, una constante definida como máximo número de operaciones, y se almacena en la memoria externa. Se utiliza para registrar las operaciones de PETT.
- cr: variable de tipo Chrono. Se utiliza para registrar tiempo. Se pasa como parámetro un booleano en true para empezar el registro del tiempo.
- ran: variable de tipo entero para registrar cuántos tests han sido ejecutados.
- failed: variable de tipo entero para registrar cuántos tests han fallado.

Para llevar a cabo dichas tareas, la clase *SerialTestCtrl* está formada por distintas funciones que serán explicadas en esta sección.

En el SerialTestCtrl.h, antes del inicio de la clase, se definen dos tipos, *Test* y *Operation*, que son booleanos, pero se definen de esta forma para clarificar el código y que su organización sea más simple.

set_addr

Se utiliza para recibir el puntero a la dirección del address desde el main y poderlo utilizar en la clase SerialTestCtrl.

get_instance

Al igual que en otras clases, la función get_instance se utiliza para crear una instancia de la clase, de forma que se le permita a otras clases acceder a sus datos y funciones. Cuando se crea la instancia se cambia el tamaño del buffer de las variables tests_slow, tests_fast, ops y rx_buffer, utilizando una función de la clase Array, para que al momento de crear los objetos de la clase SerialTestCtrl estén vacíos; se actualiza la lectura de tiempo de la variable *cr*; y se llama a la función register_test, pasando como parámetros la función run_tests y la cadena de texto "Run all tests", para registrar todos los tests deseados de PETT.

read, write, wr_available

Cada una de ellas recibe como parámetro rx_buffer, inicializado en el constructor, y devuelve un booleano en true si la función terminó correctamente. Se inicializa el pin de SCI que se quiere utilizar, es decir, el correspondiente a UART A y se realiza la acción de la función.

read, a diferencia de write y wr_available lo que recibe como parámetro es la dirección de memoria de rx_buffer, y se encarga de llamar a una función de una clase del CEX que lee los datos contenidos en ella. *write* escribe los datos almacenados en el buffer, llamando a una función de escritura, mientras que *wr_available* consulta si se puede escribir en el pin SCI porque el buffer está disponible, utilizando una función encargada de ello.

run_tests

Esta función llama a la función run y es llamada por la función register_test.

get_execution_mode

Como su nombre lo indica, se encarga de establecer el modo de ejecución de PETT entre modo Slow, modo Fast y modo Burn In. Para ello, lo leído en el pin ADC del uC correspondiente al pin de selección de modo de funcionamiento, es analizado.

Al inicio de la función se calcula el factor del divisor de tensión de la señal ADC de la PCB del CEX y el de la PCB de verificación como en la Ecuación 4.1.

$$factor = 2 \times \frac{10{,}1}{6{,}2} \tag{4.1}$$

A continuación, se crean las instancias necesarias, se declaran las variables y se configura el pin ADC a utilizar, AN4, se establece que utilice la referencia externa de 2.048 V. Luego, se lee el valor del ADC.

Para leer el ADC, durante medio segundo se leen las muestras del ADC en voltios, con la función *get_sample_volts* de la clase ADC del CEX, y se van sumando para luego hacer la media. Si no hay muestras leídas, la variable mtot_conv se iguala a cero; en caso contrario, se hace la media de las muestras leídas y ese valor se multiplica por el *factor* y se almacena en mtot_conv.

A continuación, en función de lo leído, se elige un modo de funcionamiento u otro:

- Si mtot_conv = 2,15 ± 25 % modo Fast.
- Si mtot_conv = 1,60 ± 25 % modo Burn In.
- En los otros casos, modo Slow.

Se deja un margen del 25 % por lo explicado en 3.14.1 y que el segundo divisor de tensión que se forma en la PCB de verificación no se toma de cuenta en el cálculo del *factor*, en la Ecuación 4.1.

El modo elegido se imprime en pantalla por facilidad del usuario.

Por otro lado, si mtot_conv es igual a 0 quiere decir que la señal ADC no está funcionando correctamente. Sin embargo, en este caso, se elegiría el modo slow y se ejecutarían los tests. En caso de que el usuario no detecte el error al leer en pantalla el 0 en en valor medido no sería un inconveniente, porque se detectaría el error en los Tests de ADCs.

register_test

Se utiliza para registrar los tests. Es llamada desde el main por cada uno de los test, recibiendo como parámetro el test a pasar y la cadena de texto que aparecerá en pantalla con el nombre del test. Cada uno de ellos se va guardando en el array tests_fast, con su descripción y con estado "unexecuted". En caso que se registren más tests que el número máximo de tests establecido (50), salta un excepción y se detiene la ejecución.

register_op

Su funcionamiento es similar al de la función register_test, pero en este caso para registrar las operaciones. Recibe como parámetro la operación a realizar, la letra con la que se llama la operación y la descripción de la misma. A continuación, se comprueba si se puede añadir la operación al array de operaciones y en caso de que no, salta una excepción y se detiene la ejecución.

to_lower

Recibe como parámetro un carácter y devuelve el mismo en letra minúscula. Es llamada por la función to_lower y se utiliza para poder leer de la terminal las letras utilizadas en las operaciones, sin importar que estén en mayúscula.

op_ind

Comprueba que la letra tecleada por el usuario pertenece a una operación, ya sea escrita en mayúscula o en minúscula (comprobándola con op_ind), en caso de que sí, regresa el número de operación a la que le corresponde dicha letra y, en caso contrario, regresa un -1, señalando que hubo un error.

Esta función es llamada por la función *step*, recibe como parámetro la letra tecleada por el usuario y, según lo que devuelva, se ejecuta o no una operación.

print_ops

Como se explicó anteriormente, las operaciones se diferencian de los tests en que son opcionales.

La función *print_ops* imprime en pantalla las operaciones de PETT. Para ello, con un bucle for desde 0 hasta el tamaño del array operaciones, la función recorre las operaciones registradas y va imprimiendo la letra con la que se llaman y su descripción.

print_kstring

Se utiliza para imprimir en pantalla (terminal). Es llamada por todos los "log" que aparecen en el código de PETT a través del Logger.h.

print_kstring comprueba si la escritura está disponible, en caso de que no, hace un "poll". Si la escritura está disponible, se llama a la función *write* (4.2) pasando como parámetro cada carácter que conforma la cadena que se quiere imprimir en pantalla.

El "poll" es una función dentro de la estructura Halsuite del CEX. De forma general, para las señales SCI, se encarga de:

- Si hay datos en el dispositivo SCI de transmisión, se escriben en la First In First Out (FIFO) de transmisión.
- Si hay datos en el dispositivo SCI de recepción, se escriben en la FIFO de recepción.
- Si está activado el flag de error de recepción y a su vez está activado el flag de detección de rotura o de error de trama, se reinicia el SCI y la FIFO de recepción.

print_main

Esta función escribe en pantalla los tests de PETT y es llamada por la función *step*.

Primero, comprueba el address y, si está dentro del rango deseado (entre 40000 y 49999), se imprime en pantalla, si no lo está, se imprime "No uaddress". A continuación, se imprime la versión de PETT, definida al principio de la clase SerialTestCtrl. Después, con un for desde 0 hasta el máximo de tests registrados, se imprimen en pantalla los tests y su estado (pass, faill o sin ejecutar).

print_results

Es llamada por la función *run* y *step*. Recibe como parámetro el identificador del test y su estado. Luego, se comprueba su estado y si el test ha pasado se imprime en pantalla un "PASS", si el test ha fallado, se imprime un "FAIL".

print_results_total

Es llamada por la función *run_all* y *step*. Recibe como parámetro cuántos tests han pasado correctamente y cuántos han fallado y los imprime en pantalla.

get_state

Es llamada por la función *process*. Devuelve un booleano en false si algún test de los ya ejecutados ha fallado y un booleano en true si todos los tests han pasado. Para ello, con un bucle for que recorre desde 0 hasta el tamaño del array de tests registrados (tests_slow + tests_fast), siempre y cuando una variable *ret*, la cual registra el estado del test analizado, sea verdadera, se comprueba si los tests ejecutados tienen un estado fail; si alguno de los tests falla (estado fail), la variable *ret* se pone en falso, se sale del for y se devuelve *ret*.

get_size

Devuelve el tamaño del array de tests registrados, es decir, tests_fast.size + tests_slow.size.

get_test

Recibe el índice de un test y devuelve el array tests_slow en la posición del índice, siempre y cuando el índice sea menor que el tamaño del array.

run

Es llamada por las funciones *run_tests* y *run_all*. Se encarga de ejecutar los tests.

Al inicio de la función, se enciende el LED verde y se apaga el rojo de la PCB del CEX. Luego, con un operador ternario se obtiene el número de tests a ejecutar con la función *get_size*, de forma que si el modo seleccionado es Slow, se ejecutan todos los tets, pero si es Fast, se excluyen los tests más lentos. A continuación, con un bucle for desde 0 hasta el tamaño máximo de los tests registrados, se recorren los tests. Luego, se llama a la función *process* por cada uno de ellos y se almacena la respuesta en una variable de tipo booleano que se pasa como parámetro, junto al identificador del test, cuando se llama a la función print_results. Por último, cambia de estado el LED verde y el rojo de la PCB del CEX.

La función devuelve true si no falla ningún test y false si falla al menos uno.

run_all

Es llamada cuando se quieren ejecutar todos los tests: en el Burn In Test y en el Long-Term Test (LTT). Al inicio de la función las variables *ran* y *failed* se ponen a 0 y se llama la función *run* pasándolas como parámetros. Después, se llama a la función *print_results_total* pasando dichas variables como parámetros para imprimir los resultados.

process

Se encarga de procesar los tests para evaluar si han fallado o pasado. Recibe como parámetro el índice del test a ejecutar y devuelve un booleano.

Debido a que dependiendo del resultado de los tests, se encienden los LEDs de la PCB de verificación de *Pass* y *Fail* y que, como se explicó en el capítulo anterior (3.12), esto hace a través de las señales PWM1 y PWM8, al inicio de la función se configuran los pines de dichas señales como GPIOs y salidas.

A continuación, se ejecuta el test y se guarda su resultado en una variable de tipo booleano que devolverá como parámetro, llamada res. Si *res* es verdadero, se actualiza el estado del test a "pass", si no, se actualiza a "fail".

run_op

Se utiliza para ejecutar las operaciones. Es de tipo booleano y recibe como parámetro el identificador de la operación (un entero).

Si el identificador es mayor que el tamaño del array de operaciones (ops) salta una excepción y se termina la ejecución de la función. En caso contrario, se ejecuta la operación y se guarda el resultado en una variable de tipo booleano, res, que devuelve al final de la función.

is_num

Recibe como parámetro un carácter y se encarga de comprobar si es número entre 0 y 9. Es llamada por las funciones *atoi* y *step*.

atoi

Se utiliza para que convertir una cadena de texto formada por números en un entero con el valor de la cadena.

La función *atoi* recibe como parámetro una cadena de texto y, si su primer elemento no es un número, comprobándolo con *is_num*, devuelve un entero de valor *0xFFFF*; en caso contrario, la convierte en un entero.

Para ello, con un bucle que recorre la cadena de texto y, si el carácter es un número, se calcula su posición en el número entero de la siguiente manera: ret *= 10 y luego, ret += str[i] - '0'.

La función devuelve la variable *ret* y es llamada por la función *step*.

step

Se utiliza para evaluar los distintos estados de ejecución de PETT, es llamada por el main de forma recursiva cada vez que termina de ejecutarse. Al inicio de la función, se hace un *poll* de los periféricos: I2C, SCI A, SCI B, SCI C, CAN A, CAN B, eCAPx (x = 1 a 4).

Como se explicó en la Sección 4.2, la función *poll* se encuentra dentro de la Hal-suite del CEX. Para el caso de I2C, se encarga de, si se activa el flag de interrupción de arbitraje por pérdidas, regsitrar el error y reiniciar el módulo. En el caso de las señales CAN, se encarga de manejar sus mensajes de recepción. Por otro lado, para las señales eCAP, si hubiera datos de captura, los pone en la cola FIFO del eCAP.

A continuación, la función *step* evalúa la variable *state*, con un switch-case, de la siguiente forma:

Case:

- desconectado: sale de la función.
- conectado: detiene la variable cr, contadora de tiempo. Llama a la función *print_main* (4.2) y actualiza state a *espera*.
- espera: si hay datos en el buffer por ser leídos, se actualiza el estado a *datos disponibles* y aumenta el tamaño del buffer de lectura una unidad; en caso contrario se sale de la función.
- datos disponibles: se utiliza para leer los datos del buffer de recepción. Este caso, por su complejidad e importancia, será explicado más adelante.
- procesando: si la escritura está disponible, se pasa al estado de espera y se ignoran los datos pendientes.
- test automático finalizado: como su nombre lo dice, es el caso en el que ha finalizado el test automático. Si la variable tests_ok, la cual es un booleano que devuelve true si los tests pasaron correctamente, es verdadera, se enciende el LED verde y parpadea el rojo de la PCB del CEX, en caso contrario, se enciende el rojo y parpadea el verde, señalando que ha fallado un test.

El caso *datos disponibles*, al principio de su ejecución, comprueba la variable cr

y, en caso de que esté contando tiempo, se actualiza su valor. A continuación, se espera un poco (300 ms) para el caso en que el usuario escriba un número de más de un dígito.

Después, se lee el buffer de recepción y, si hay más datos que el tamaño del buffer máximo, salta una excepción, sino, se va actualizando el tamaño del buffer mientras se lee.

Luego, se comprueba el tamaño del buffer y, si mide más que cero, quiere decir que se han leído datos.

Ahora, se procesan los datos. Primero, se actualiza el estado a *procesando*y se declara una variable de tipo entero, idx, con valor $0xFFFF$. Si el tamaño del buffer es 1 y la función *op_ind* (4.2) devuelve un valor diferente a -1, indicando que el usuario ha seleccionado una operación, se ejecuta la operación seleccionada con la función *run_op* (4.2) y se imprime en pantalla *App ended*. Por otro lado, si lo que hay en el buffer es un número, comprobado con la función *is_num* (4.2), se almacena en la variable *idx* el entero escrito por el usuario, valor devuelto por *atoi* (4.2); si el valor de *idx* está dentro del tamaño del buffer de tests, comprobado con la función *get_size* (4.2), se imprime en pantalla el test a ejecutar, se actualizan las variables *ran*, *failed* y *tests_ok* a cero, se ejecuta el test seleccionado, con la función process (4.2), y se guarda su resultado (pass o fail) en la variable *tests_ok*; si *idx* es cero, quiere decir que el usuario seleccionó la opción "Run all tests", por lo que se llama a la función *print_results_total* (4.2), pasando como parámetros las variables *ran* y *failed*; si *idx* es diferente de cero, se llama *print_results* (4.2), pasándo como parámetros *idx* y *tests_ok*. Si lo leído en el buffer de recepción no es un número dentro de la lista de tests, ni una operación, se imprime en pantalla el menú con la función *print_main* (4.2).

4.3. Test Address

El Test del Address consiste en comprobar que el CEX tiene un uaddress escrito en la memoria flash y que se encuentra entre el rango definido para el producto:

entre 40000 y 49999. Este test es el primero en ejecutarse debido a que Valkiria en el Burn In Test lo necesita para identificar el CEX a verificar.

Para comprobar el uAddress se sigue el diagrama de la Figura 4.5. Se lee en la flash lo almacenado en la celda del address, se comprueba si está dentro del rango y se imprime en pantalla; en caso de que no lo esté, se le asigna el valor de addr_uav_unknown (999).

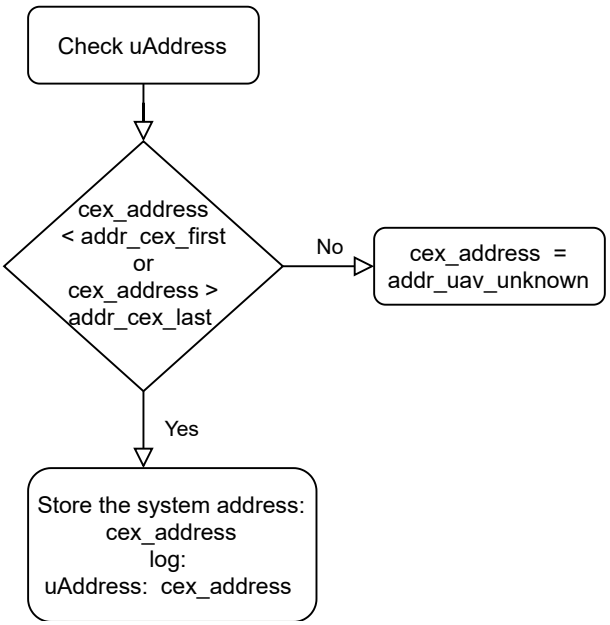


Figura 4.5: Test Address.

Si durante la ejecución el test falla puede significar que no se ha escrito el uaddress, por lo que se utiliza la operación "Write Address" y se escribe el uaddress correspondiente, que debe coincidir con el número de serie del producto.

4.4. Test ADC

La clase TestADC se utiliza para comprobar los valores leído en los pines ADC del uC del CEX. De forma general, está formada por dos tipos de test: test de lectura de voltaje y test de temperatura.

El test de lectura de voltaje lee el valor en voltios conectado al pin ADC. Para su funcionamiento, en el main se utiliza el siguiente template, en el que ADC es el canal de ADC que se quiere leer, IN es el valor esperado y FACTOR es el valor del divisor

de tensión que tiene la PCB del CEX: template<Dsp28335_ent::ADC::ADCchannel ADC, Uint16 IN, Uint16 FACTOR>.

El test de temperatura consiste en leer el valor de temperatura medido por el sensor ubicado en la PCB del CEX y conectado a un pin ADC del uC. En este caso, el template solo recibe como parámetro el pin del ADC que se quiere leer: template<Dsp28335_ent::ADC::ADCchannel ADC>.

Para ejecutar los tests, la clase *TestADC* está formada básicamente por las funciones: test_power_an, test_temp, check_power_switch, get_error, run y runtemp, que serán explicadas en esta sección.

test_power_an

Está definida en el archivo .h de la clase. Se encarga de ejecutar la función run, a partir de la instancia de la clase. Recibe desde el main los parámetros del template: canal ADC, valor esperado (llamado IN) y factor del divisor de tensión (llamado FACTOR) y los pasa a *run* como parámetros, la cual devuelve un booleano que representa si el test ha pasado o fallado. Los parámetros se pasan a *run* convertidos en floats y representan el valor real del voltaje esperado y factor de conversión. Esto se hace porque IN y FACTOR se pasan en el template como enteros de 16 bits y luego son convertidos en floats. Los parámetros no se pasan en el template como floats porque los parámetros del template floating-point no son estándar.

Para convertir *IN* y *FACTOR* en floats, se multiplican los valores recibidos por una fracción proporcional al valor que se desea y, de esta forma, se obtiene el valor esperado y el factor del divisor de tensión.

test_temp

Al igual que la función *test_power_an*, esta función está desarrollada en el archivo .h de la clase. A través de un template recibe como parámetro el pin del uC del ADC que tiene conectado el sensor de temperatura en la PCB del CEXs. *test_temp* llama a la función runtemp, pasando como parámetro el valor que recibe

del template y espera un booleano que representa si el test ha pasado o fallado.

check_power_switch

Debido a que en el Burn In Test se alternan las alimentaciones entre V1 y V2, por SW en PETT se necesita comprobar que al menos una de las dos variables tiene un valor correcto, para lo que se utiliza la función check_power_switch.

A través de dos operadores ternarios, se comprueba si en los pines ADCs de V1 o V2 se leen 7 V, 12 V o 36 V, las alimentaciones normalmente utilizadas en la empresa. Para ello, se llama a la función test_power_an.

get_error

La función get_error recibe como parámetros dos floats, value y mtot_conv, y devuelve otro float, percent_error. Es llamada por la función *run* y se encarga, de forma general, de obtener el error de una medida cuando su valor es variable, es decir, cuando se trata de uno de los pines ADC encargado de leer las alimentaciones (V1 y V2) o el modo de funcionamiento.

Cuando un pin es de valor variable se diferencia pasando como parámetro en la variable *IN* del template un valor específico en la variable *value*: un 0, cuando se trata del pin que lee el modo de funcionamiento, y un 1 cuando se trata de V1 o V2.

Por otro lado, el parámetro mtot_conv contiene la información leída en el pin ADC del uC, es decir, la lectura analógica.

Debido a esto, lo primero que hace la función get_error es diferenciar entre los valores de value. De forma que si es 0, se calcula el error considerando si el modo de funcionamiento es Fast, Slow, o Burn In y si es 1, se calcula considerando si la alimentación es 7 V, 12 V o 36 V.

Si value es igual a 0, se espera que el pin ADC reciba 2.15 V para el caso de Fast mode, 0.70 V para el caso de Burn In mode y 1.60 V para el caso de Slow mode. A partir de esos valores esperados, se calcula el error relativo y se devuelve como

parámetro.

Para el caso en que value es igual a 1, se calcula el error relativo cuando las alimentaciones son 7 V, 12 V o 36 V. En caso de que la alimentación leída no sea ninguna de ellas, se imprime en pantalla que el valor leído es incorrecto y que el error relativo es del 100 %. Para saber si la alimentación es 7 V, 12 V o 36 V, se compara el valor leído en el ADC con cada uno de esos valores y, si la diferencia entre ellos no es más de 1.5, se entiende que es la alimentación correcta.

En la Figura 4.6 se representa la función get_error.

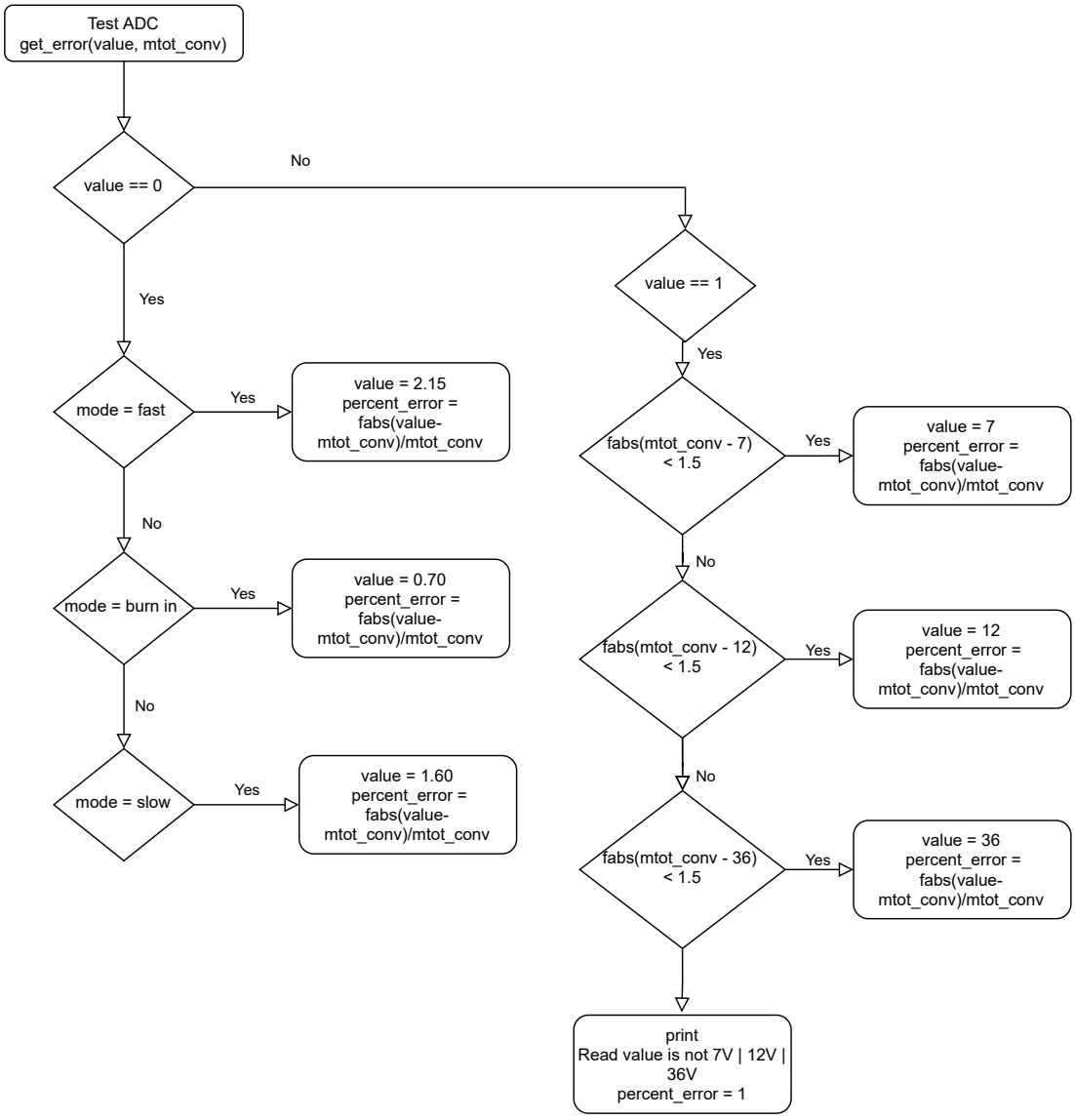


Figura 4.6: Test ADC get error.

run

Es la función principal del TestADC. Recibe como parámetros los valores del template: canal del ADC, value (IN en el template) y factor.

run es llamada por la función test_power_an y devuelve un booleano en true si el valor que se lee es correcto y pasa el test, y fail si el test falla debido a que el valor leído está fuera del rango esperado. Un fail puede significar un fallo de la señal ADC en cualquier punto, ya sean los integrados, las pistas, o incluso, la señal de entrada (señal de potencia 3.3V o 5V).

La función *run* se encarga de leer el valor del pin ADC del uC del CEX.

Al inicio de la función se inicializa la clase ADC, pasando como parámetro la referencia que se va a utilizar, en este caso la externa de 2.048 V, por lo explicado en 2.2.3. Se espera 500 ms para que se estabilice la lectura del ADC y se inicializan variables temporales. Mientras el tiempo sea menor que 0.5 s, cada 5 ms se leen las muestras.

Para leer el valor recibido en el pin del uC, se utiliza la función get_sample_volts de la clase ADC del CEX, que devuelve el valor en voltios leído por el pin del ADC pasado como parámetro. Los valores leídos se van sumando y se van contando las muestras leídas, de forma que después se hace una media. La media de valores leídos se multiplica por el factor, para convertirla en el voltaje real a la entrada del divisor de tensión de la PCB del CEX.

Para los casos que value es 0 o 1, se llama la función get_error (4.4), para comprobar si el valor leído se encuentra dentro de los rangos esperados, para los casos en que se trata de un pin ADC de alimentación o de selección de modo de funcionamiento.

A continuación, se calcula el error relativo de la medida y, si no es mayor que el 15 %, se considera que el test pasó correctamente. Después, se imprime en pantalla el valor leído y el error de la medida y la función devuelve un true si el test ha finalizado correctamente y un false, en caso contrario.

En la Figura 4.7 se observa un diagrama del funcionamiento de la función run.

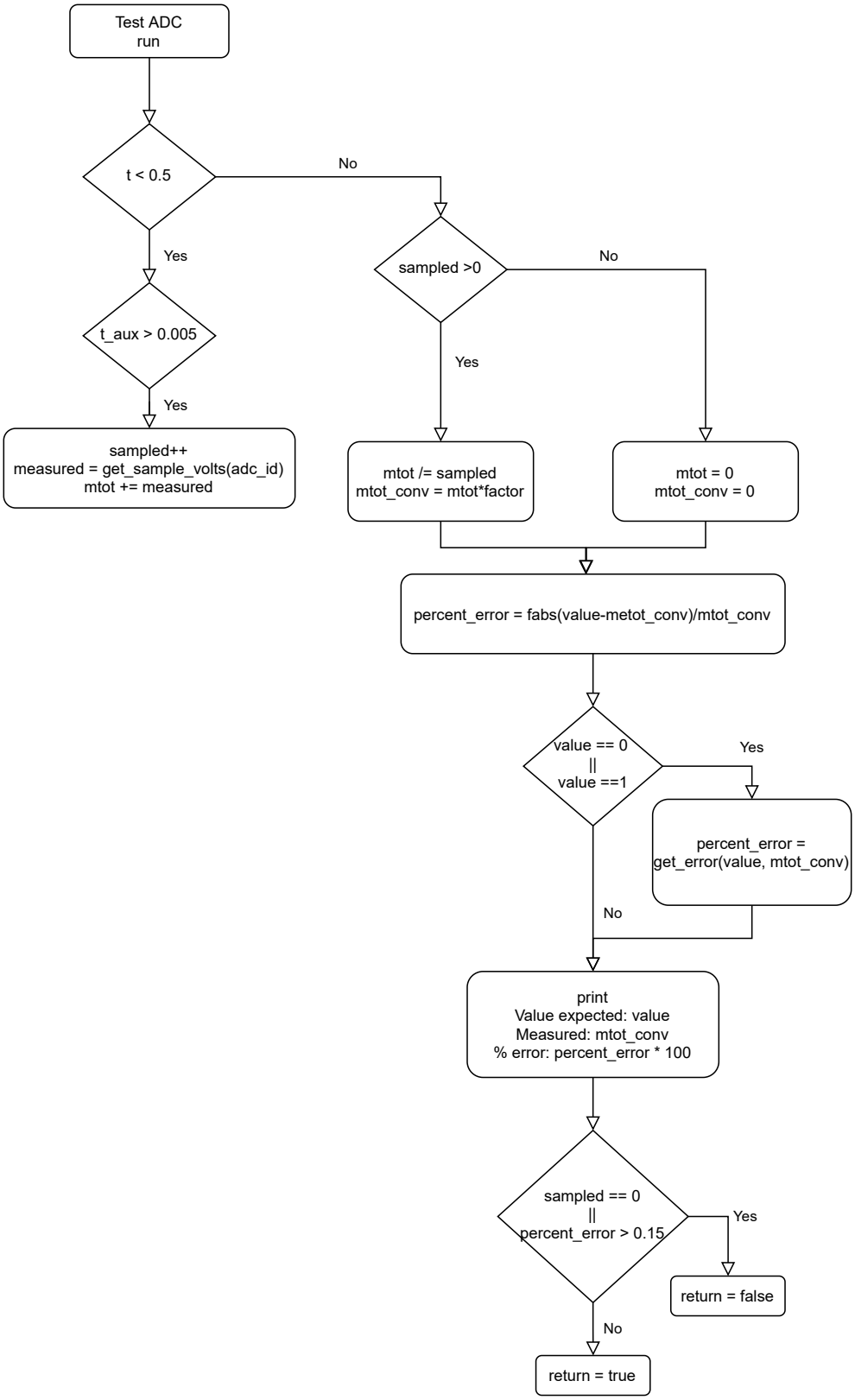


Figura 4.7: Test ADC.

runtemp

En un pin ADC hay conectado un sensor de temperatura ubicado en la PCB del CEX, encargado de medir la temperatura de la placa.

Para comprobar el funcionamiento del sensor, así como que se lee correctamente en el uC, se utiliza la función *runtemp*. Ésta es llamada por test_temp, recibe como parámetro el pin del ADC a leer y devuelve un booleano en true si la temperatura leída se encuentra dentro del rango esperado, y false, en caso contrario.

Para hacer este test, se utiliza la función get_value de la clase ADCsuite del CEX que, al recibir el canal donde está conectado el sensor de temperatura, devuelve el valor de la temperatura en grados Kelvin. Se comprueba en PETT que el valor de temperatura esté entre 370 ^{o}K y 290^{o}K, debido a que en el Burn In Test y otras pruebas de temperatura, la PCB del CEX se calienta y puede alcanzar ese valor, mientras que el límite inferior se debe a la temperatura mínima que se ha medido con el CEX durante las pruebas de PETT realizadas durante su validación.

4.5. Test SCI

El Test SCI se utiliza para comprobar el funcionamiento de las interfaces SCI del CEX: UART B y UART C (UART A se utiliza para comunicarse con el ordenador, por lo que una comunicación exitosa garantiza que la interfaz SCI A funciona correctamente.)

El test está conformado por distintas funciones, así como *run*, encargada de ejecutar el test, y testUARTB y TtestUARTC, que se utilizan para crear una instancia del test, configurar los pines correspondientes como SCI y llamar a la función run.

Para la configuración SCI en las funciones testUARTB y TtestUARTC, primero, se crea una instancia del pin a utilizar, ya sea, SCI B o SCI C, según corresponda. Después, se configura el puerto SCI: se establece un baudrate de 115200, la longitud de trama de 8 bits, un bit de stop, se deshabilita la paridad y se establece que no es modo Address.

Debido a que le señal UART C por defecto está NOPOP y en su lugar está soldada una señal I2C, la función testUARTC, además, de lo explicado anteriormente, configura los pines de la señal I2C como GPIO de entrada, para evitar daños en el uC en caso de que no se hayan desoldado las resistencias que hacen que el bus sea I2C y no UART.

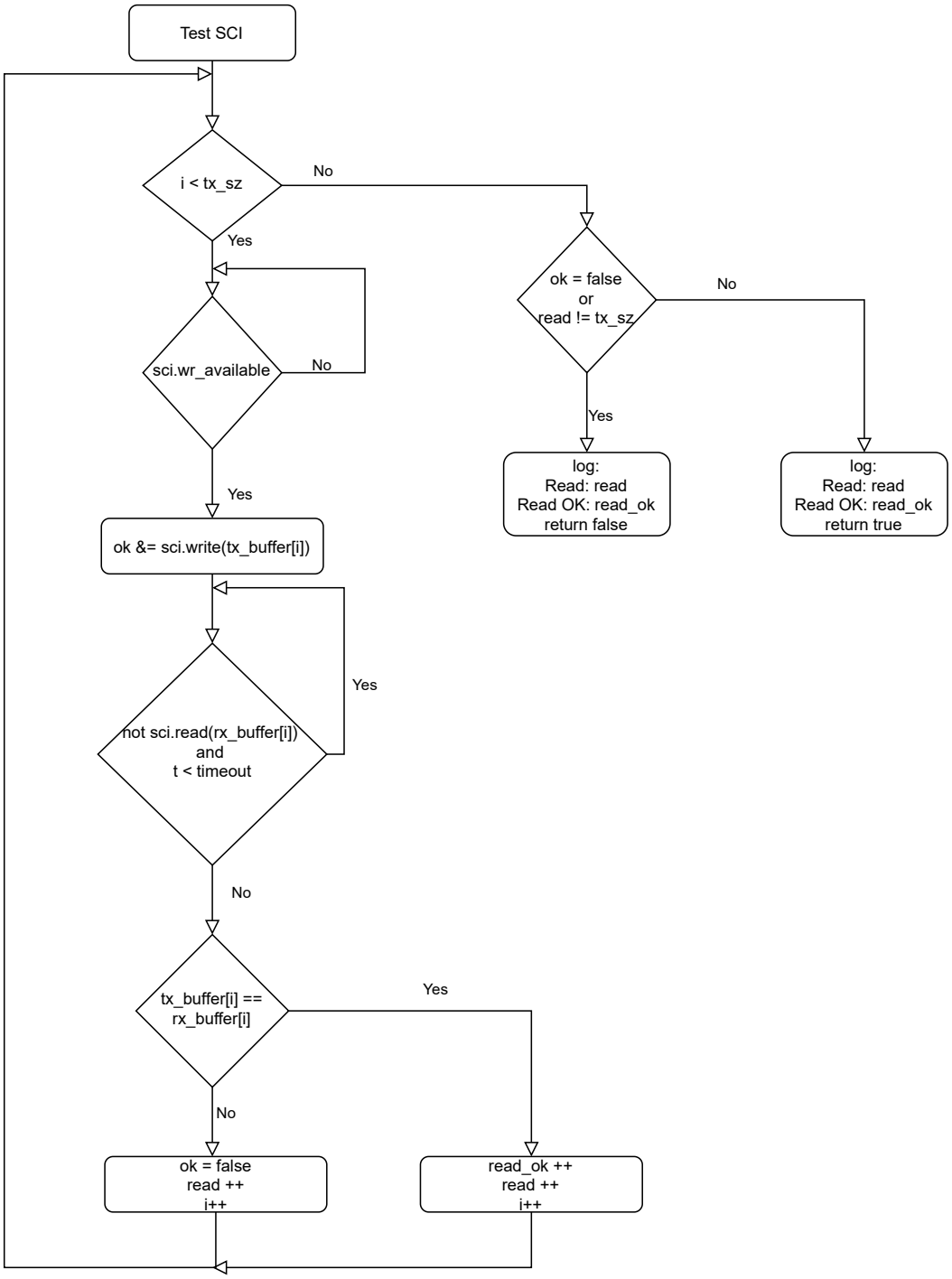


Figura 4.8: Test SCI.

En el constructor de la clase se inicializan las variables, como un objeto de tipo SCI y los arrays de los buffers de envío y recepción, pasando como parámetro el tamaño y el lugar donde se almacenan. Además, se rellena el array de transmisión, tx_buffer.

La función run es la encargada de ejecutar el test. Tras instanciar la clase CEX necesaria para acceder a todos los periféricos e interfaces del producto, se instancia la clase chrono, para medir tiempo. A continuación, se vacía la información que puede haber en el buffer de recepción, se establece un timeout de 0.5 segundos y se declaran las variables a utilizar. Luego, byte a byte se envía lo que hay en el buffer de transmisión, siempre que esté disponible, y se espera a recibir en el buffer de recepción lo enviado en un tiempo inferior al timeout. Si el byte transmitido coincide con el recibido, se aumenta el contador de leídas correctas, en caso contrario, no se aumenta el contador y se pone una variable de control a false, como se observa en la Figura 4.8. En ambos casos, se aumenta un contador de bytes enviados. Si los bytes recibidos correctamente coinciden con el tamaño del buffer de transmisión, se considera que el test ha pasado correctamente. Por otro lado, si ha habido algún error tanto en la transmisión como en la recepción, el resultado del test es un false.

4.6. Test CAN

El Test CAN consiste en comprobar que la señal que se envía es la misma que se recibe, debido a que en la PCB de verificación hay un loopback.

Al inicio de la función que ejecuta el test, se configuran los pines de CAN A y CAN B. Para ello, se establece un baudrate de 1 Megabits por segundo (Mbps), ya que es el máximo soportado por Hardware (HW). Luego, se configura cada bus de receptor (RX): se le establece un identificador de recepción, un número de mailboxes y un tamaño de buffer, además, debido a que el módulo Enhanced Controller Area Network (eCAN) del uC permite una configuración para tramas extendidas, se configuran las tramas del test como normales; también se les asigna una máscara, $0xFFFFFFFF$, la misma para ambos buses. Para ello se utilizan distintas funciones de la estructura CANcfg del código del CEX, de forma que, siendo cfg_a una

variable de tipo CANcfg:

- cfg_a.br se iguala a 1000000 para establecer el baudrate.
- cfg_a.rx.resize(1) se utiliza para cambiar el tamaño del buffer de RX a 1.
- cfg_a.rx[0].sz se iguala a 16 para establecer el número de mailboxes del elemento 0 del array de recepción, que es el único que se utiliza.
- cfg_a.rx[0].flt.id.extended se pone a false para configurar un formato de trama estándar.
- cfg_a.rx[0].flt.id.id se igual al ID del que se espera recibir una trama.
- cfg_a.rx[0].flt.msk para establecer la máscara deseada, $0xFFFFFFFF$.

La máscara de aceptación del bus CAN se utiliza para indicar cuáles de los bits de identificación de la trama recibidos serán utilizados por el filtro de CAN. El filtro de CAN, por su parte, compara los bits de identificación de los mensajes CAN recibidos con los bits del registro del filtro, si se observa una coincidencia entre el filtro y la identificación del CAN, se acepta la trama y se transfiere al buffer de CAN de recepción; si no hay coincidencia, se descarta. La máscara $0xFFFFFFFF$ quiere decir que solo deja pasar las tramas con el ID de RX configurado.

A continuación, se asigna una trama para que envíe cada bus, formada por el identificador del CAN de transmisión, un booleano para señalar que no es CAN extendido y los datos. De acuerdo a esta configuración y a la máscara establecida, el bus A, solo recibirá tramas del B y viceversa.

Luego, se configuran cuatro tramas de recepción y se igualan a cero. Se vacían los buffers de recepción, se inicializa una variable temporal y se escribe en cada bus CAN la trama definida.

A continuación, se espera un período de 0.25 s para que se envíen las tramas y se comprueba lo recibido. Una representación del test se puede observar en la Figura 4.9.

El test CAN pasa si:

1. CAN A lee correctamente una trama y la almacena en frame_arx1.

2. CAN B lee correctamente una trama y la almacena en frame_brx1.
3. CAN A y CAN B intentan leer otra trama, pero no pueden, debido a que ya se han leído las tramas disponibles.
4. La trama transmitida por CAN A es la misma recibida por CAN B y viceversa.

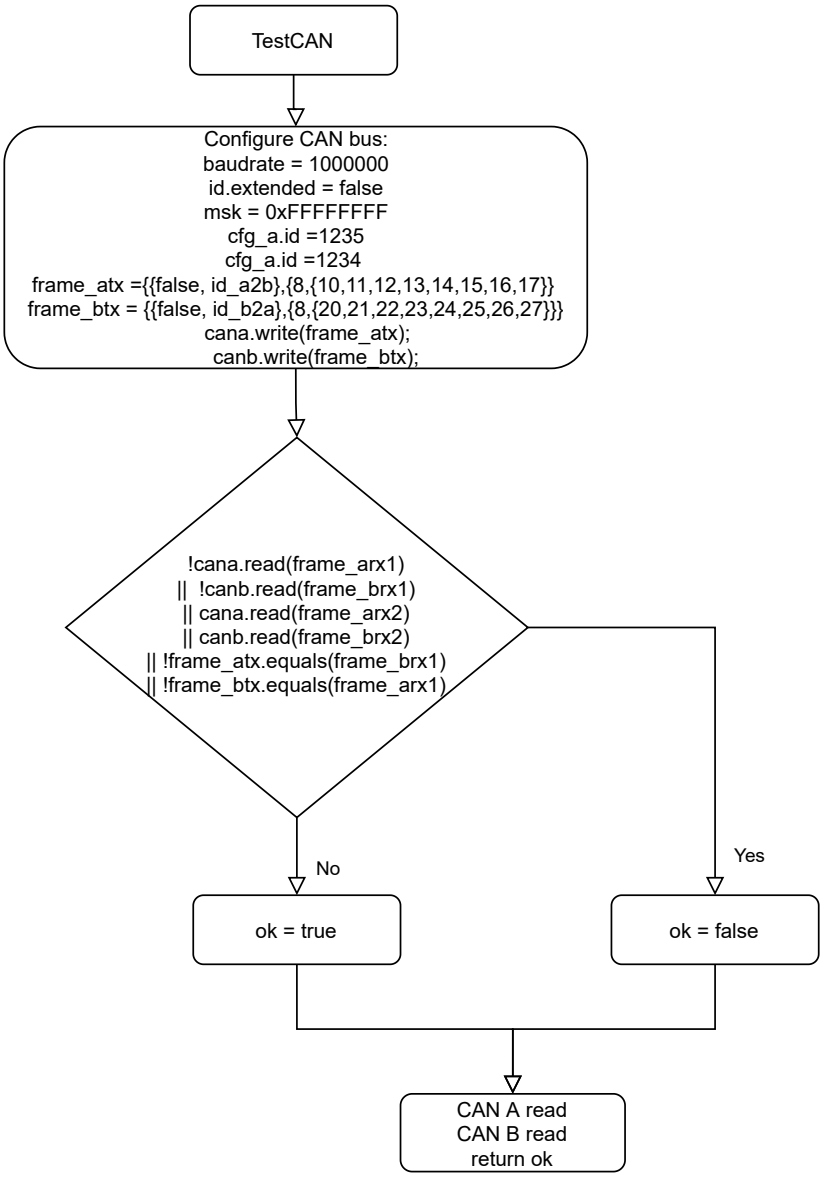


Figura 4.9: Test CAN.

La comprobación 3 de la lista anterior se realiza porque cuando CAN envía una trama y no recibe Acknowledgement (ACK) la continúa enviando hasta que le llega un mensaje de confirmación, es decir, hasta que un dispositivo recibe la trama, o hasta que pasa un timeout establecido por el bus. Por lo que dicha comprobación confirma que en efecto el dispositivo destino ha recibido la trama, enviado un ACK

y el bus de transmisión ha dejado de enviar tramas.

Para imprimir las tramas recibidas se llama a la función print_canframe de la clase TestCAN, que recibe como parámetro la trama a imprimir y la recorre mientras la imprime con la función *log* de Logger.h (4.12).

4.7. Test I2C

Así como se explicó en 3.10, el test del I2C consiste en establecer comunicación con un magnetómetro externo. Debido a que se quiere probar la interfaz I2C del CEX más no que el magnetómetro funciona correctamente, con lograr una comunicación exitosa con el dispositivo, se considera que el test ha pasado correctamente.

El magnetómetro a utilizar es el MMC5883MA de MEMSIC. Es un sensor magnético de 3 ejes que puede conectarse directamente al uC sin necesidad de conversores Analógicos-Digitales. Permite medir campos magnéticos en un rango de ± 8 Gauss (G), con resolución de 0,25 mG por Less Significant Bit (LSB) en modo de operación de 16 bits y nivel de ruido RMS total de 0,4 mG, sin embargo, el dispositivo no puede leer temperatura y campo magnético a la vez. Para configurarlo, se utiliza su address, la cual es 0110000, es decir 0x30 en hexadecimal [17]. Aunque no puede conectarse a 5 V, está conectado a una PCB que tiene un regulador de tensión, para poder alimentarlo según sus especificaciones y se alimenta con la salida de potencia del CEX de 5 V. Su rango de operación es entre -40 ºC y 85 ºC, lo que permite utilizarlo en la cámara climática durante el Burn In Test.

Como el diseño de la PCB del CEX permite que los pines I2C del conector del CEX sean en su lugar UART C, que por defecto está NOPOP, al inicio del test los pines UART C del uC se configuran como entradas GPIO, de forma que estén protegidos en el caso en que no se hayan desoldado las resistencias que conectan las señales con los pines del conector del CEX.

Después de crear las instancias y declarar las variables necesarias, inicia el test. El constructor del dispositivo necesita como parámetros la interfaz I2C a la que está conectado, un booleano de referencia, una variable de tipo array para almacenar los

datos leídos, otra para guardar los datos en crudo de las medidas y otra para la temperatura leída.

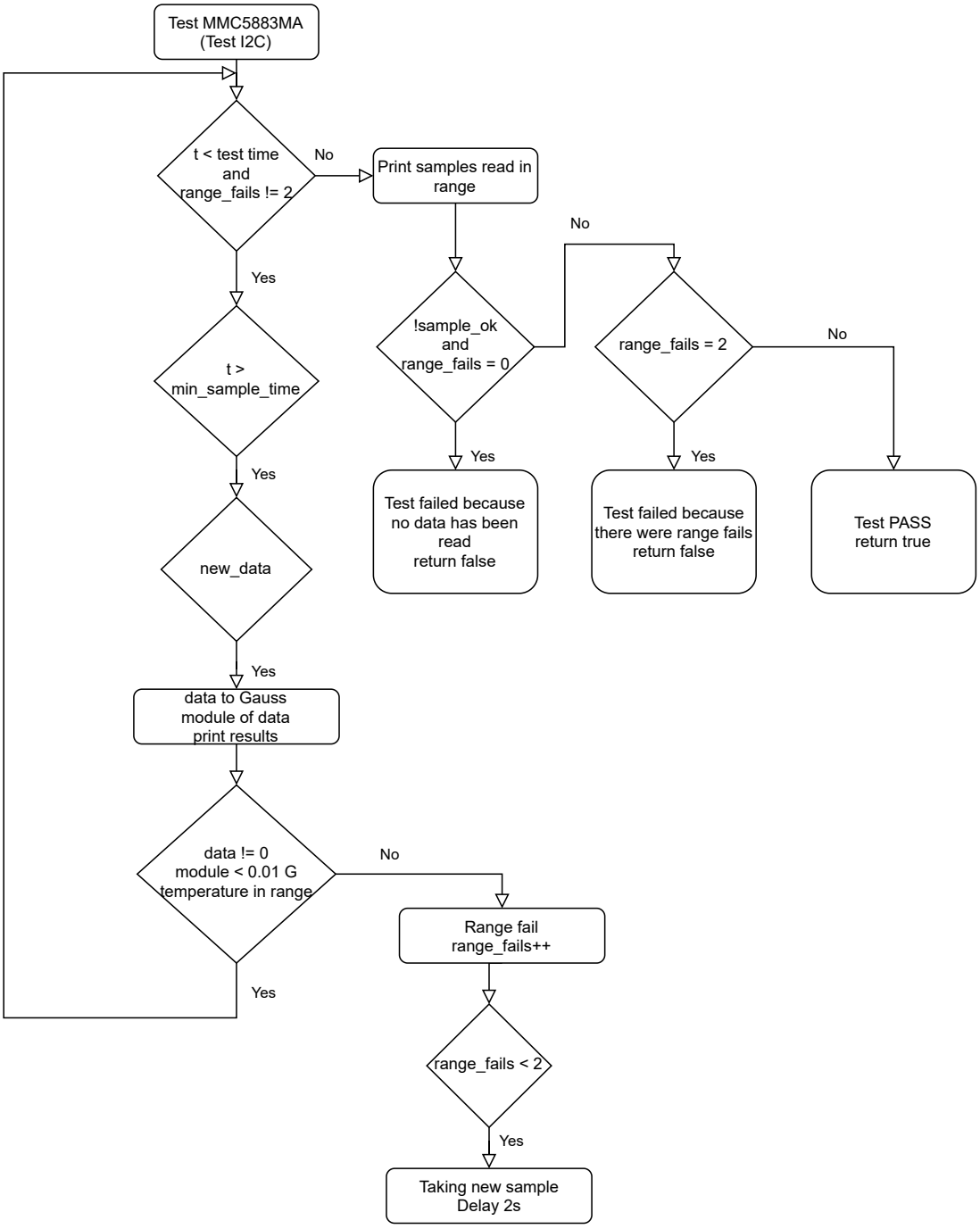


Figura 4.10: Test MMC5883MA (Test I2C).

Para el magnetómetro se establece una tasa de muestreado deseada que, debido a que se necesitan 10 ms para muestrear dos ciclos, se asume de 80 Hz. Esto se debe a que en un primer ciclo se dice de qué registro se quiere leer la información y en el segundo se lee dicha información. Debido a que no se necesita que la lectura sea

precisa, no se calibra el magnetómetro.

Al iniciar el test, se establece un mínimo de tiempo de muestreo igual a la inversa de la tasa de muestreo deseada. Además, se fija un tiempo máximo del test y una condición de que si hay dos fallos también finalice. Luego, se recoge cada muestra hasta llegar al tiempo de muestreo, se lee el valor, se convierte de Tesla a Gauss, se imprime el módulo del valor leído y se analiza.

Para verificar que la muestra leída es correcta, se comprueba que el dato no es cero y que su módulo es mayor que 0.01 G. El módulo se calcula debido a que es un magnetómetro de 3 ejes. De igual forma, se verifica que cada eje tiene un valor diferente a cero. Debido a que no se quiere comprobar el funcionamiento del sensor, sino comprobar que se puede hablar con él, lo que se comprueba es que los datos son diferentes de cero y que se leen medidas, por esto se espera un módulo mayor que 0.01 G. También, se comprueba que la temperatura leída está entre -40 ºC y 85 ºC. En la Figura 4.10 se observa un diagrama del test I2C.

4.8. Test PWM

Al igual que otros tests, se comprueba mediante loopbacks por HW que la señal que se recibe es la misma que se envía. En este caso, se envía una señal PWM y se lee en un pin de eCAP, sin embargo, como la conexión en la PCB de verificación es que cada pin de eCAP está conectado a dos pines de PWM, en el test se comprueba uno y a continuación, el otro.

La función recibe como parámetros los pines de los PWMs que se quieren verificar y su correspondiente eCAP. Para facilitar la comprensión del test, en este apartado el primer PWM del loopback será el PWM1 y el segundo PWM2.

Después de crear las instancias necesarias para continuar con la ejecución, se configuran los pines: PWM1 como PWM y PWM2 como GPIO output nivel bajo, por lo explicado en 3.7.

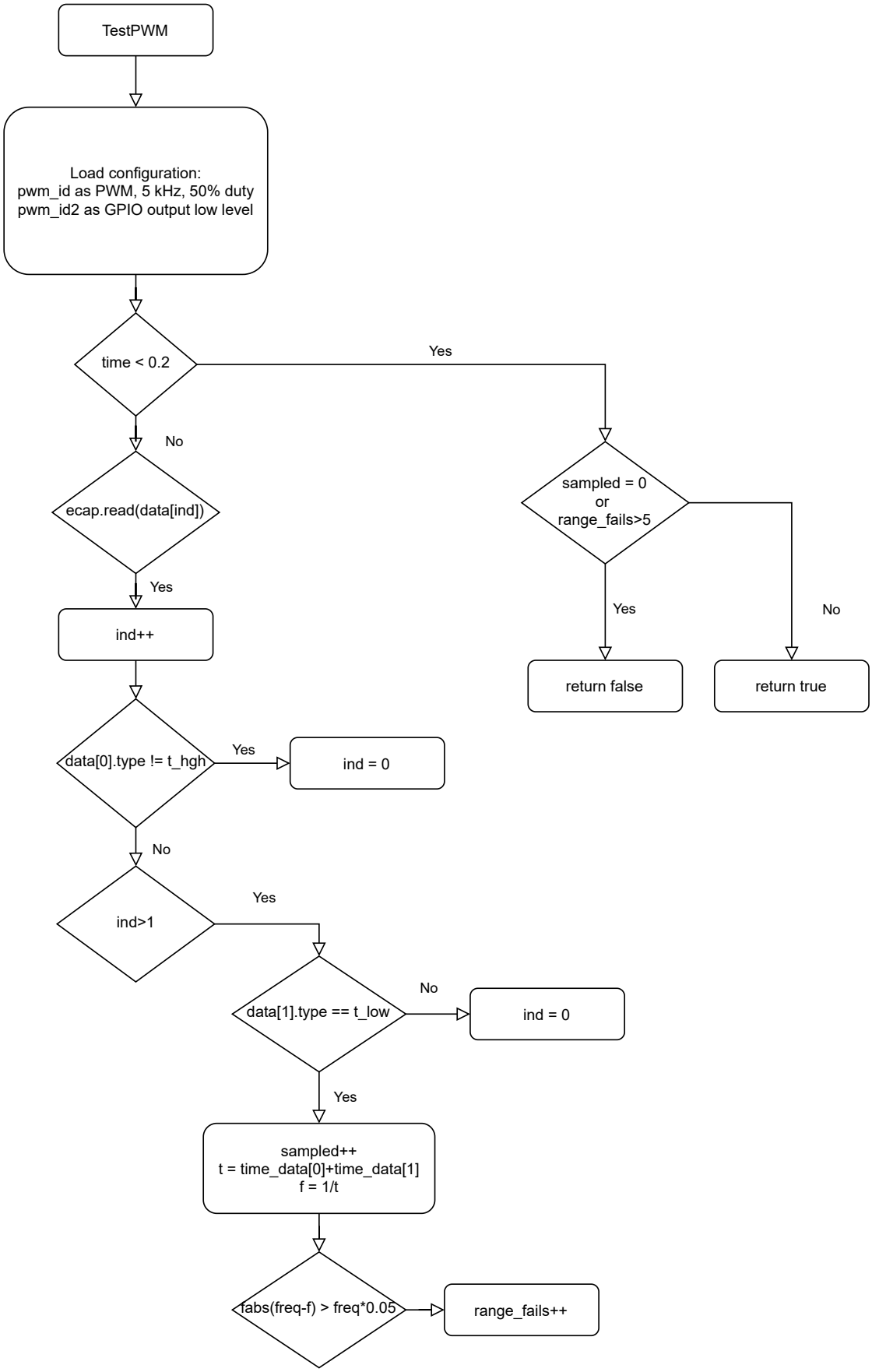


Figura 4.11: Test PWM.

Mientras que el pin eCAP se configura de la siguiente manera:

- Reloj del eCAP activado.
- Se asigna el identificador del pin, que se recibe como parámetro cuando se llama la función desde el main.
- Número de "captured wraps" = número de eventos que se desean capturar, el máximo permitido es 4, sin embargo, se configuran 2 porque son los necesarios.
- Eventos a capturar, en este caso son flancos de subida y de bajada.

Luego, a PWM1 se le configura una frecuencia de 5 kHz (una frecuencia normal dentro del rango de funcionamiento). Se inicializan variables contadoras para controlar los pulsos que se reciben correctamente, se esperan 400 ms para que el PWM se estabilice, se inicia un contador de tiempo y se empieza a leer lo que llega al pin del eCAP. A continuación, durante 200 ms se ejecuta un bucle en el que se guarda el primer flanco de subida que se captura, el flanco de bajada que le sigue y el tiempo entre ambos para obtener la frecuencia. Si la frecuencia calculada es igual a la configurada ($\pm$ 5 %), se considera que el test se ha ejecutado correctamente, en caso contrario, como fallo. Este proceso, para comprobar un PWM, se detalla en la Figura 4.11.

A continuación, se deshabilita PWM1, se configura como GPIO con dirección salida en estado bajo, PWM2 como PWM y se repite el proceso.

Si el rango de fallos es mayor que 5 se considera que hay un error en el PWM o en el eCAP, en caso contrario, el test concluye e imprime por pantalla que todo ha funcionado correctamente.

4.9. Operación Write Address

Es la operación que permite escribir un Address en la flash del CEX. Es llamada AppWruAddress.

El address es importante porque permite identificar el CEX durante el Burn In

Test. Además, cuando los CEXs se entregan a los clientes, deben tener el Address configurado, que coincida con el número de serie del producto.

La clase *AppWruAddress* está formada por las funciones run, set_addr y run_op. Su constructor inicializa una variable de tipo Itport_u8 llamada *sci*, para seleccionar le puerto de la señal UART por la que se recibe la información, UART A; y una variable de tipo Array, *rx_ buffer*, para almacenar lo leído por sci.

La función *set_ addr* se utiliza para recibir desde el main el puntero a la memoria flash donde se almacena el address.

La función *run* crea una instancia del constructor AppWruAddress y llama a la función run_op.

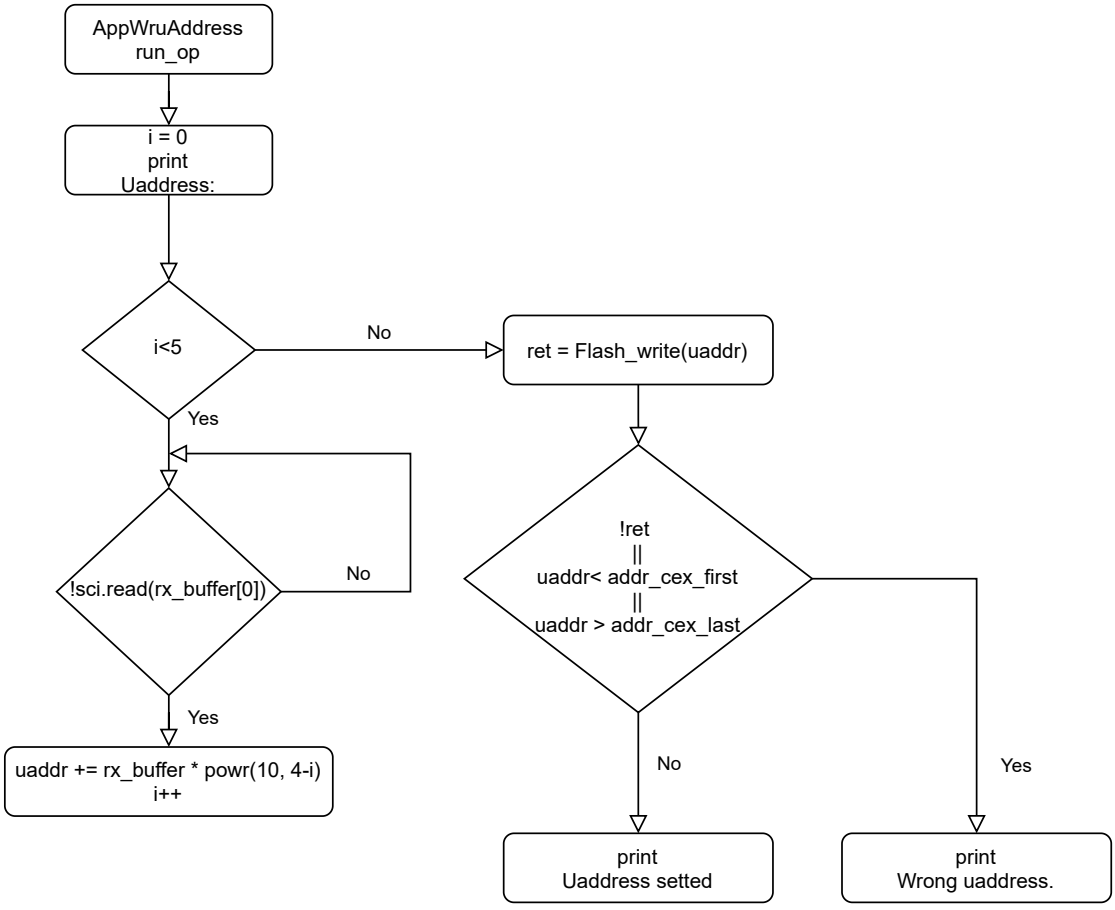


Figura 4.12: Operación Write Address.

La función *run_ op*, a su vez, se encarga de preguntar el uaddress y de almacenarlo en la flash. Cuando se llama la operación AppWruAddress, escribiendo la letra "w" en la terminal, se imprime en pantalla "Uaddress" y se reciben los primeros

5 números que se escriben en la terminal. A continuación, se escribe en la flash el valor recibido. Luego, se comprueba si el valor pudo ser escrito de forma correcta y si se encuentra dentro del rango del uaddress del CEX, es decir, entre 40000 y 49999; en caso que sea correcto se devuelve un true, en caso contrario, se imprime "Wrong uaddress" y se devuelve un false. Después finaliza la operación.

En la Figura 4.12 se observa un diagrama simplificado de la operación AppWruAddress.

Si se escribe un número dentro del rango y aún así el test no finaliza correctamente puede significar que hay un error en la memoria flash, por lo que no se puede escribir en ella. Esto podría ser debido a un fallo en el uC del CEX.

4.10. Operación Long-Term Test

La operación LTT se utiliza para ejecutar un test de larga duración sobre la PCB del CEX. El tiempo de duración del test es de 1 hora, pero puede ser fácilmente modificable cambiando un parámetro del código.

Consiste en ejecutar en bucle todos los tests de PETT para comprobar de forma continua el correcto funcionamiento de todas las interfaces y periféricos del Veronte CEX. De forma simple, es como un Burn In Test pero sin reiniciar la PCB.

Un LTT es útil para probar el funcionamiento del CEX en entornos estresantes, como en pruebas de temperatura o de vibraciones, como las establecidas por la DO-160, que es el estándar que define las condiciones de pruebas ambientales y los procedimientos y criterios de pruebas aplicables para equipos de aviónica.

Para empezar el test, en el menú principal del modo slow de PETT, se selecciona la operación LTT escribiendo una letra "l". El test empieza a ejecutarse y finaliza si transcurre el tiempo deseado o si falla algún test.

La clase LTT está formada por la función run, la cual ejecuta todos los tests de PETT mientras no falla ninguno y el tiempo es menor que 3600 segundos. Después de cada iteración imprime en pantalla el tiempo que lleva el test y las ejecuciones

correctas. La operación finaliza correctamente si, una vez transcurrido el tiempo, todos los tests han terminado exitosamente.

4.11. tiiw.cpp

Hay un bug conocido del compilador de Texas Instruments en el que, si hay un template declarado en un .h de una librería e instanciado en otra, el linker no es capaz de resolver y encontrar dicha instancia, por lo que hay que volverlo a instanciar en el proyecto principal y que de esta forma encuentre la instancia del template y sea capaz de resolver.

El tiiw.cpp cumple la función de almacenar los distintos templates que se necesitan en el proyecto.

4.12. Logger.h

Está formado por distintas funciones *log* y *lognl* y, de forma general, se utiliza por escribir en la terminal a través de la función *print_kstring* (4.2).

log se utiliza para imprimir en pantalla caracteres, cadena de caracteres o enteros, mientras que *lognl*, además, agrega un salto de línea.

Las funciones de *Logger.h* son llamadas por distintas clases de PETT y reciben como parámetros caracteres, cadenas de caracteres, enteros, flotantes o flotantes, según sea el caso, para procesarlos y pasárselos a como parámetros a *print_kstring*.

CAPÍTULO 5:

VALIDACIÓN EXPERIMENTAL

Una vez diseñado el sistema, es importante validarlo y comprobar que funciona para los casos en los que se quiere usar. Para ello, una vez diseñada la Printed Circuit Board (PCB) de verificación y desarrollado el Software (SW) Production Embention Testing Tool (PETT), se hicieron distintas pruebas y análisis para corregir errores y lograr que funcionara correctamente. Dichos errores han sido explicados a lo largo de esta memoria.

El proceso de verificación se considera un elemento *vivo*, ya que, en caso de que sea necesario, podría modificarse para mejorar.

Después de su implementación en la línea de producción, ha servido para detectar fallos en casi todas las interfaces de distintos CAN Expanders (CEXs), así como, para detectar errores y arreglar CEXs de Return Merchandise Authorization (RMA).

El nuevo proceso de verificación se considera una mejora de la línea de Producción porque se prueban de forma más exhaustiva y controlada las distintas interfaces y periféricos del CEX. Además, al estar el proceso explicado en el Acceptance Test Procedure (ATP), se ha convertido un sistema más simple y, al estar los resultados registrados en el Acceptance Test Report (ATR), en más fiable para los clientes.

Se han verificado al menos 100 CEX con el proceso de verificación desarrollado y, para el momento de escritura de esta memoria, se han detectado fallos en al menos el 6 % de ellos.

Los datos estadísticos no son exactos porque una vez implementado el sistema de verificación en la línea de producción, el proceso continúa en el departamento de Producción y no es necesaria la intervención del departamento de Verificación, a

menos que haga falta soporte por algún problema más complejo.

Por otro lado, ninguno de los CEX verificados con este nuevo sistema en Embention antes de salir a los clientes, ha llegado como RMA por fallos de Hardware (HW).

Sobre si el proceso es estadísticamente mejor que el anterior no hay datos, porque no se registraba información más que el número de serie y el cliente que lo recibe.

En la Figura 5.1 se observa un CEX a verificar, conectado a la PCB de verificación antes de empezar una verificación funcional.

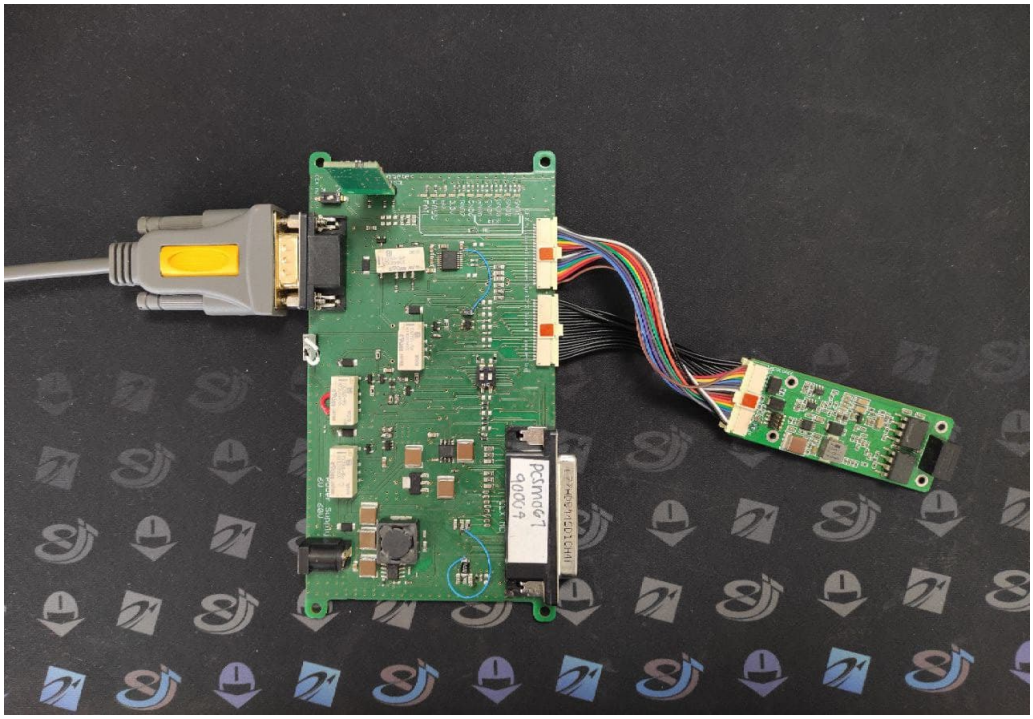


Figura 5.1: PCB de verificación y CEX.

5.1. Funcionamiento

Para verificar un CEX, se realiza el proceso explicado en el Capítulo 2 (2.3). Para las verificaciones funcionales, según el momento de la verificación, se selecciona el modo de funcionamiento Slow o Burn In Test, siendo el Modo Slow necesario para la verificación Funcional pre-Burn In Test (2.3.8), verificación Funcional post-Tropicalizado (2.3.11) y verificación Funcional post-Montaje en Caja (2.3.13); mientras que el Modo Burn In se utiliza para el Burn In Test (2.3.9).

Una vez elegido el modo de funcionamiento a utilizar, el interruptor correspondiente se pone en la posición adecuada y se selecciona con el otro interruptor el producto a verificar, entre CEX y CEX MC. Después, se conecta la PCB de verificación al ordenador, a través del conector RS232-USB, y a la alimentación.

Para la comunicación con el ordenador en Modo Slow es necesario abrir una terminal, seleccionar el COM correspondiente y utilizar un baudrate de 115200. Para el Modo Burn In, por su parte, aunque en la línea de Producción utilizan el SW Valkiria, también puede verse en la terminal.

5.1.1. Modo Slow

Cuando se conecta el CEX al ordenador en modo Slow, PETT imprime todos los tests en pantalla, junto al uaddress del producto, si lo tuviera, y el valor de tensión leído en el pin del Analog-to-Digital Converter (ADC) 4, junto al modo de funcionamiento. En la Figura 5.2 puede verse la lista de tests en el modo slow de PETT sin ejecutar.

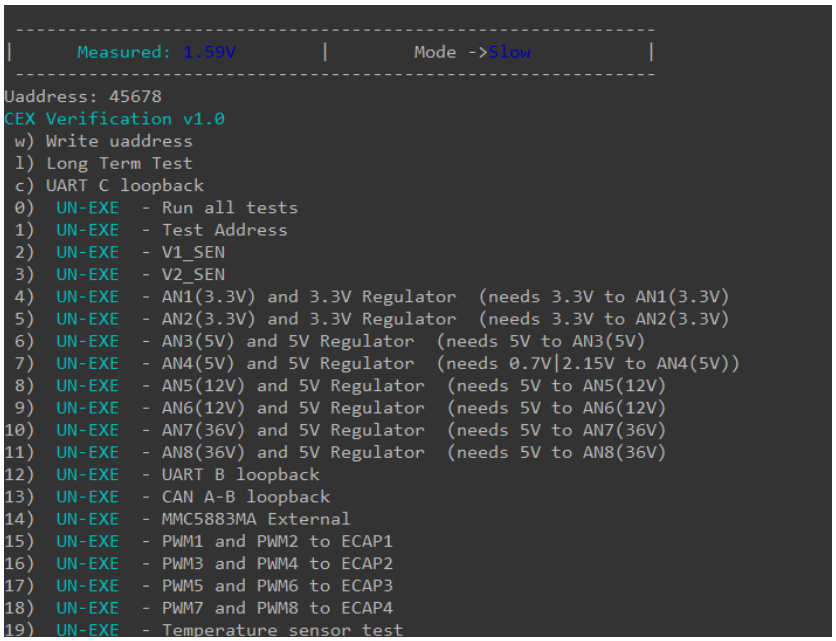


Figura 5.2: Modo slow, tests sin ejecutar.

A continuación, si el usuario escribe un "0" se ejecutan todos los tests y, a medida que se ejecutan, se imprime en pantalla el resultado de cada uno de ellos y

un "PASS" en letras verdes si el resultado del test ha sido favorable, y un "FAIL" en rojo si ha fallado el test. En la Figura 5.3 se observan los tests ejecutados, en ella los tests 3 y 14 han fallado, el test "V2_SEN" porque la PCB de verificación que se utilizaba era la correspondiente al Burn In y en ella no está soldada la resistencia que permite alimentar V1 y V2 a la vez, porque en dicho test se alternan las alimentaciones; mientras que el test "MMC5883MA External" ha fallado porque no estaba conectado el magnetómetro externo.

Por otro lado, si en lugar del 0 se escribe otro número que se encuentra dentro del menú de tests, se ejecuta el test seleccionado. Si se escribe una letra dentro del menú de operaciones, se ejecuta la operación. Sin embargo, si se escribe un carácter que no está dentro de las opciones de PETT, se vuelve a imprimir el menú en la pantalla.

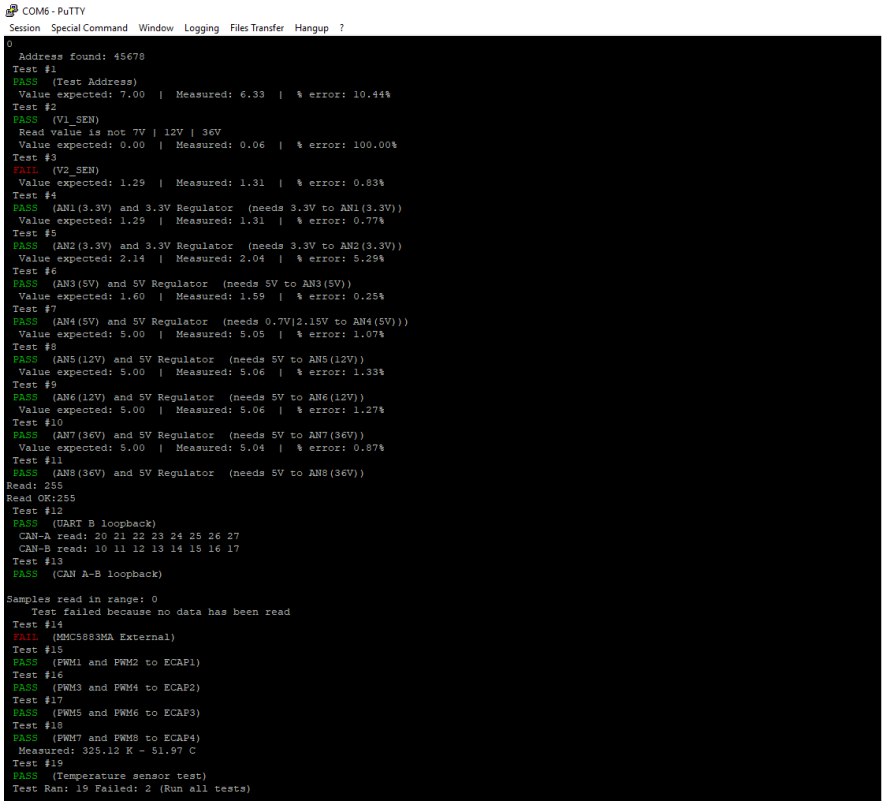


Figura 5.3: Modo slow, tests ejecutados. 2 fallos.

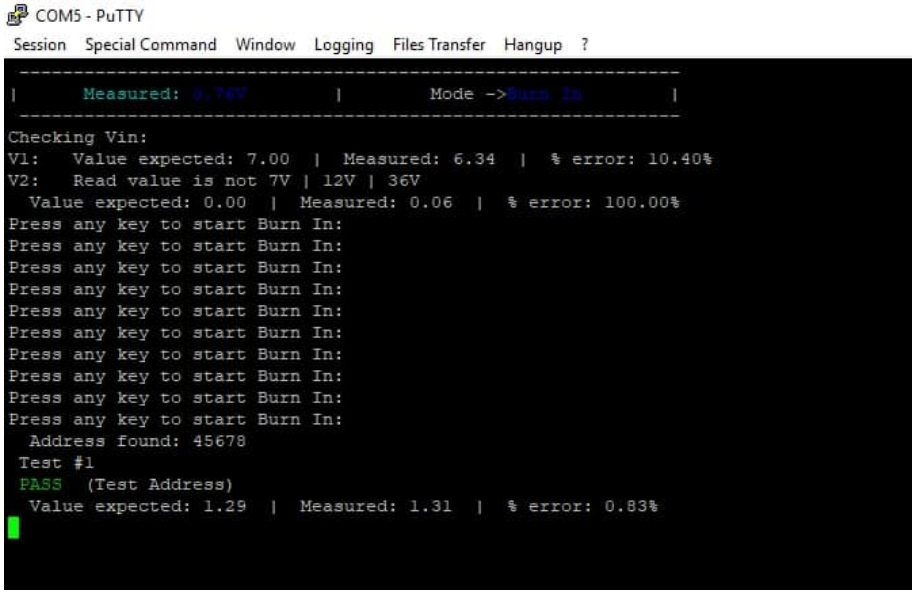


Figura 5.4: Modo Burn In "Press any key to start"

5.2. Modo Burn In

Cuando se selecciona este modo, el CEX imprimen en pantalla "Press any key to start Burn In:", hasta que recibe un carácter, como se observa en la Figura 5.4. Por lo que el usuario, a través de la terminal, o Valkiria, presiona una tecla y empiezan a ejecutarse todos los tests.

Al terminar de ejecutarse se imprime en pantalla el número de tests ejecutados y un mensaje para indicar que se reiniciará el CEX, como se observa en la Figura 5.5. Después, se activan los correspondientes Pulse Width Modulators (PWMs) y se reinicia el CEX.

Si un test falla, al terminar la ejecución de PETT, se imprime en pantalla "A test has failed!" y no se reinicia el CEX, por lo que en Valkiria se ve un cuadro en rojo señalando el fallo y en la terminal se lee lo impreso en pantalla, como se observa en la Figura 5.6.

Como se explicó en el Capítulo 2 (2.4.1) Valkiria es una herramienta desarrollada en Embention que se utiliza para automatizar los tests. Permite que no haga falta una interacción humana para cada ejecución de PETT durante el Burn In Test y que si un test falla o si se acaba el Burn In, sea fácilmente detectable.

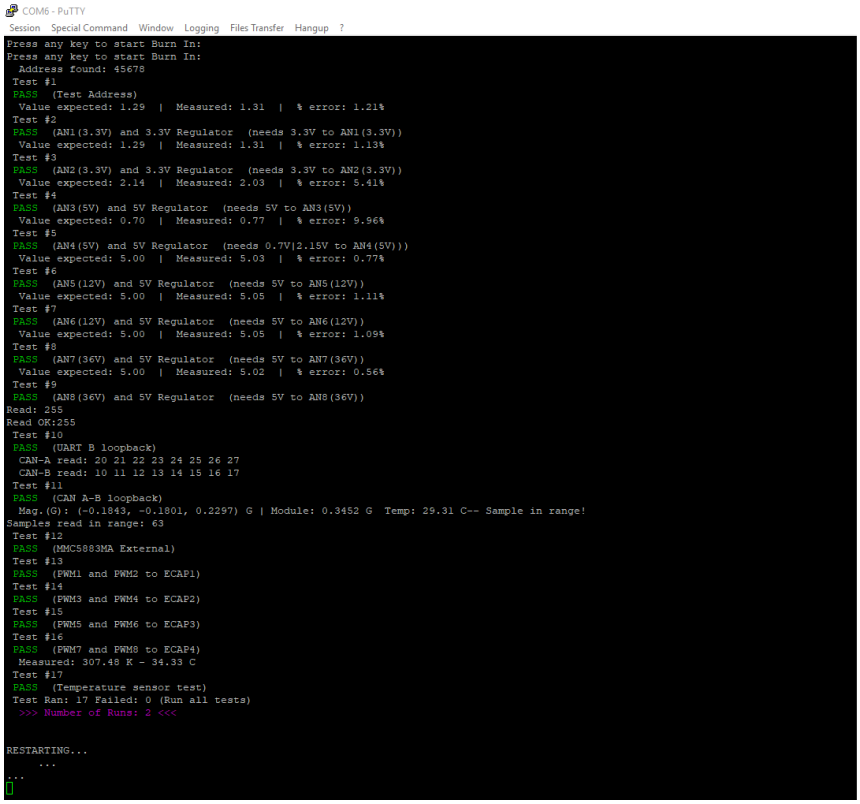


Figura 5.5: Modo Burn In, ejecución de tests.

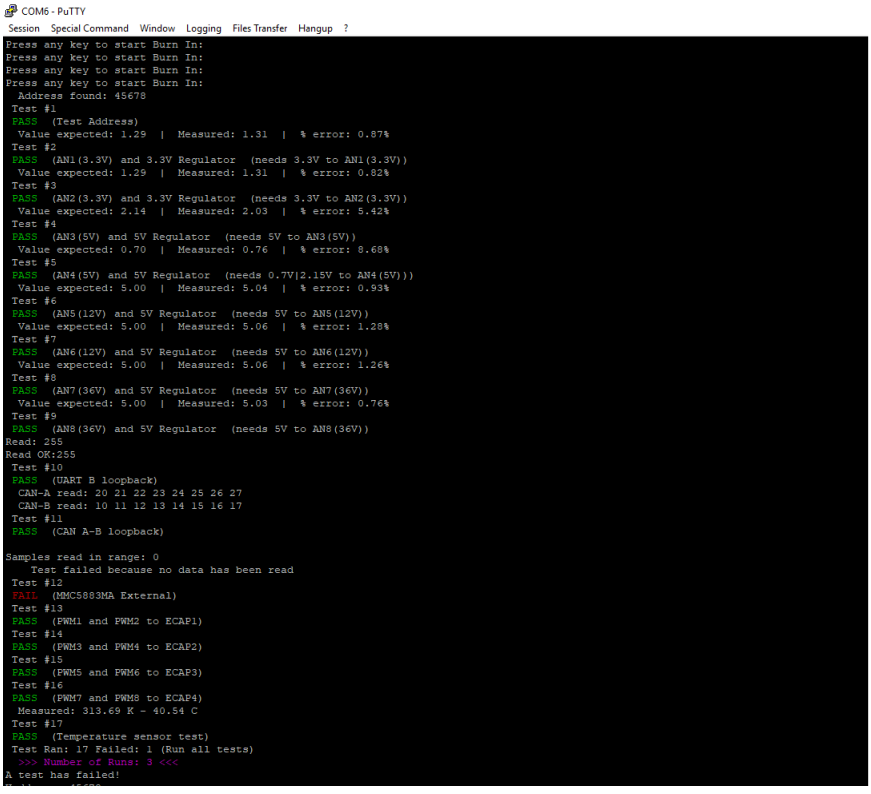


Figura 5.6: Modo Burn In, fallo de tests.

El resultado en Valkiria se observa como en la Figura 5.7, donde un cuadro verde con el address del CEX indica que se han terminado todas las ejecuciones.

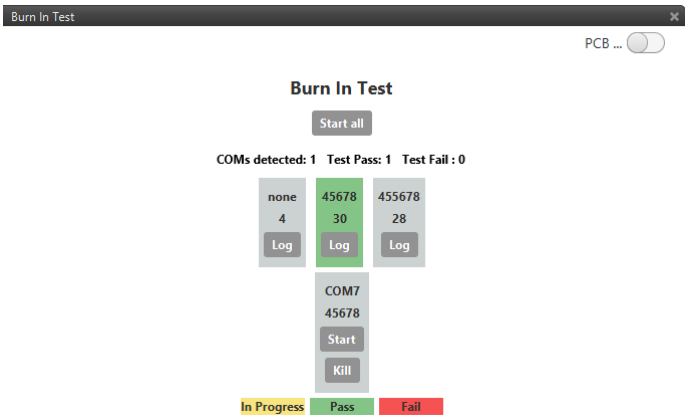


Figura 5.7: Modo Burn In, Valkiria.

En la Figura 5.8 se observa un CEX siendo verificado, conectado a una PCB de Verificación. En la ejecución del test funcional pre-Burn In Test falló un test de PETT por lo que tiene un Light-Emitting Diode (LED) rojo encendido indicando el error.

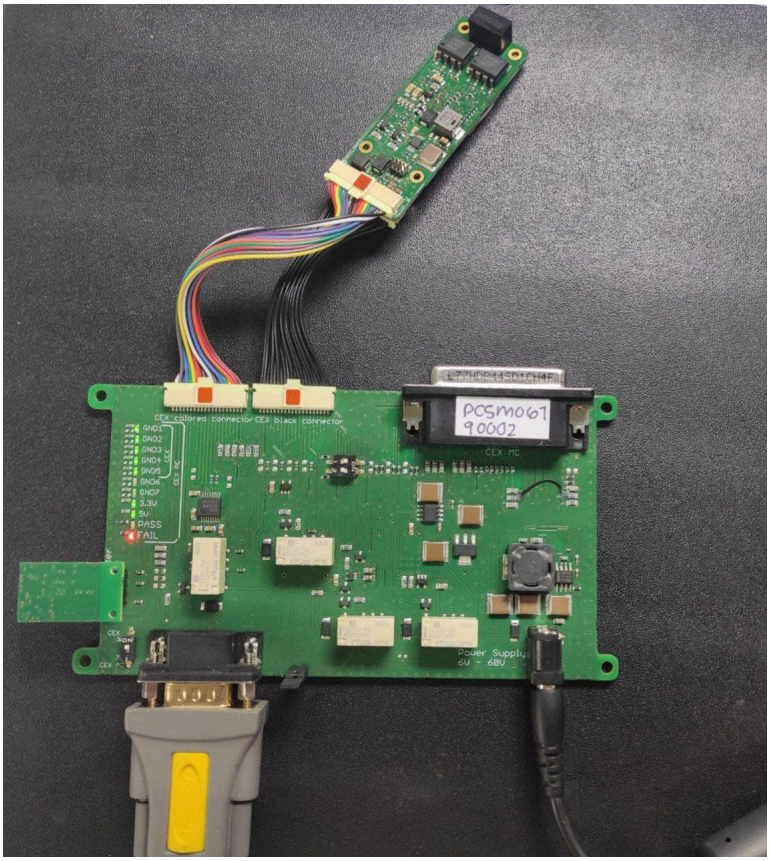


Figura 5.8: CEX y PCB de verificación, resultado fail.

5.3. Evaluación de la PCB de Verificación y PETT

Después de fabricarse las PCBs de verificación y de desarrollarse PETT, llegaron como RMAs CEXs MC con el tipo de montaje Serial Communications Interface (SCI) half duplex y RS485, por lo que tuvo que adaptarse una PCB de verificación y PETT a esa variante del producto.

Como se explicó en el Capítulo 2 (2.1), el CEX MC tiene las siguientes formas de montaje: con interfaz RS232 e interfaz RS485 y con interfaz Universal Asynchronous Receiver-Transmitter (UART) e interfaz RS485. Sin embargo, durante el diseño de la PCB de verificación y de PETT no se tomó en cuenta la tercera forma de montaje de dicho producto: una interfaz UART half duplex y una interfaz RS485.

La interfaz UART half duplex consiste en una interfaz formada por una única línea encargada de transmisión y recepción, de forma que si se están enviando mensajes, no se pueden recibir. Dentro del CEX MC está el HW necesario para convertir la interfaz SCI A del CEX en este tipo de señal. La interfaz half duplex normalmente se utiliza en el CEX MC para conectarse utilizando JETI.

Debido a este tipo de montaje, la comunicación con el ordenador ya no puede ser a través de la señal UART A, por lo que la PCB de verificación y PETT deben adaptarse.

5.3.1. Modificaciones de HW

Debido a que la comunicación del CEX con el ordenador ya no puede ser a través de la señal SCI A, debe ser utilizando la señal SCI B, que para este tipo de montaje es una señal RS485. Por ello, debe desoldarse el loopback de la señal RS485, por lo que se quitan las resistencias y condensadores que lo forman. También, debe eliminarse la conexión de la señal SCI half duplex al conector RS232 para evitar problemas al momento de la comunicación. Luego, debe conectarse un conversor RS485-USB para la comunicación con el ordenador.

5.3.2. Modificaciones de SW

Una vez desarrollado el código de PETT y puesto en marcha en la línea de producción, se observó que el test Fast no era necesario. La idea de llevarlo a cabo era que se pudiera comprobar fácilmente, de forma general, si todas las interfaces del CEX funcionan correctamente. Sin embargo, a la hora de implementarlo, después de la verificación de Programado no se ejecutaba el test Fast si no el Slow, porque no representaba ninguna ventaja. Debido a esto, en el código de PETT se sustituyó el Modo Fast por el Modo Comunicación RS85, para tener una forma de seleccionar en qué momento la comunicación cambiaba por el nuevo modo de montaje, lo que implicó los siguientes cambios:

1. Se eliminó la condición de Fast Mode en el main, lo que quiere decir que el Test del Address se ejecuta en todos los modos de funcionamiento.
2. Se agregó una condición en la que se comprueba el modo de funcionamiento y si es diferente a *rs485_comm* se agrega a la lista de tests a ejecutar el test de UART B, si no, se agrega el test de JETI.
3. La comunicación con el ordenador es utilizando el UART A cuando el modo de funcionamiento es Slow o Burn In, y en caso contrario, utilizando UART B.

Para elegir por cuál interfaz SCI es la comunicación con el ordenador, en las funciones read, write y wr_available (4.2) de la clase SerialTestCtrl, se agrega un condicional que comprueba si el modo es *rs485_comm* y en caso de que lo sea, se establece que la comunicación es por SCI B, si no, por SCI A.

5.3.3. Test JETI

JETI es un protocolo que utilizan distintos equipos que, algunos casos, uno de ellos se conecta al CEX MC. Utiliza una topología punto a punto y físicamente las líneas de transmisión y de recepción están conectadas por una resistencia de unos pocos kΩ [18]. Debido a esto, las señales transmisor (TX) y receptor (RX) en la

PCB del CEX MC están unidas y el Test JETI se debe ajustar a eso.

Ya que el test consiste en una comprobación de HW más no de SW, no se verifica que el protocolo JETI funciona, sino que todos los componentes están soldados y funcionando correctamente.

A diferencia de los otros tests de señales, en este caso no se puede comprobar que se recibe la misma señal que se envía, porque están conectadas las señales de TX y RX. Por lo que el Test JETI consiste en enviar una señal, recibirla en un Arduino, comprobarla, transmitir otra señal y leerla en el CEX. Para el desarrollo del test se tiene en cuenta que las señales de Arduino tienen una amplitud de 5 V, mientras que las del CEX de 3.3 V; sin embargo, con el montaje de HW de JETI, a ese pin del CEX pueden entrar señales de hasta 5 V, por lo que no es un inconveniente la comunicación con Arduino.

Para ello, se agrega a PETT la clase TestJETI, que está formada por las funciones test, time_counter y run.

El constructor de la clase inicializa los pines de transmisión y recepción como General-purpose I/O pin (GPIO) porque la señal se genera con cambios de estados del pin TX de la señal UART A. Esto es debido a que el Arduino a utilizar no tiene conexiones serie disponibles para tratar los datos como UART half-duplex, porque, aunque tiene dos conexiones serie, éstas están unidas y se utilizan en el test para la comunicación del Arduino con el ordenador. Debido a que las señales a enviar son una combinación de estados altos y bajos y a que no se puede coordinar el tiempo de ejecución del Microcontrolador (uC) del CEX y del de Arduino, el test, en rasgos generales, consiste en evaluar la duración de los pulsos y comprobar que es correcta, de forma que se comprueba así que se están enviando y recibiendo los pulsos correctamente.

A continuación se van a explicar las distintas funciones que forman el test.

test

Se utiliza para crear una instancia de la clase. Es llamada en el main por la función *register_test* (4.2).

run

Ejecuta el test, es llamada por *test* y devuelve un booleano en true si pasa el test y en false si falla.

Al inicio de la función se declaran las variables necesarias: t para almacenar el tiempo y res para guardar el resultado del test. A continuación se configuran los pines, tanto de transmisión como de recepción, como entradas. Luego, se configura el pin de transmisión como salida y se realiza la siguiente secuencia:

1. TX a nivel bajo por 1ms.
2. TX a nivel alto por 1 ms.
3. TX a nivel bajo por 1ms.
4. TX a nivel alto por 1 ms.
5. TX a nivel bajo por 1ms.
6. TX a nivel alto por 1 ms.
7. TX a nivel bajo por 1ms.
8. TX a nivel alto por 5 ms.
9. TX a nivel bajo por 5ms.

Después, se configura el pin como entrada, se llama la función *time_counter* y se almacena su resultado en la variable *t*. Si se cumple que $0{,}0045 < t < 0{,}007$, se imprime en pantalla un 1 y la variable *res* cambia a true. Luego, se vuelve a llamar a la función *time_counter* y se hace la misma comparación para comprobar dos veces la lectura del Arduino. El rango de tiempo aceptado que se establece se debe al resultado de la validación del test.

time_counter

Es llamada por la función *run*, recibe como parámetro un booleano cont y devuelve una varibale t de tipo float. La variable *cont* es true si es la segunda vez que entra a la función *time_counter* y la primera vez que lo hizo devolvió un valor de t dentro del rango deseado, lo que quiere decir que la primera lectura de Arduino fue correcta.

La función *time_counter* se encarga de comprobar el tiempo entre los pulsos de Arduino para verificar así que la conexión entre el pin de Arduino y el pin SCI RX A es correcta.

Al inicio del test se inicia una variable de tipo Chrono para contar el tiempo y la variable flotante t.

Si *scia_rx.get()* devuelve true, lo que quiere decir que se lee un estado alto en el pin GPIO, y *cont* también lo es, empieza a contarse el tiempo, se espera a que Arduino envíe todo el pulso en estado alto, luego a que envíe todo el pulso en estado bajo y después que se ponga de nuevo en estado alto, se actualiza la variable t con la diferencia de tiempo entre los pulsos en estado alto y se imprime en pantalla.

Por otro lado, si *cont* no es true, se espera que el primer pulso que envíe Arduino sea en estado bajo, por lo que se comprueba que *!scia_rx.get()* sea true, lo que quiere decir que se lee un estado bajo en el pin GPIO, después se espera a que pase el pulso en estado bajo, empieza el contador de tiempo cuando Arduino envía un estado alto, se espera a que pase ese pulso y el siguiente pulso en estado bajo, cuando se lee un estado alto se actualiza la variable t con la diferencia de tiempo entre los pulsos en estado alto y se imprime en pantalla.

5.3.4. Arduino

Como se explicó anteriormente, no se puede hacer un loopback para comprobar que la señal que se recibe es la misma que la enviada. Debido a esto, se utiliza un Arduino nano que recibe una señal, la procesa y envía otra. En esta sección se

explicarán las distintas funciones que forman el código desarrollado en Arduino para esta tarea.

Para la ejecución del código se utilizan dos variables globales: digital_byte, de tipo byte, y se le asignal el valor PD2, refiriéndose al pin digital 2; y digital_value, de tipo entero. Para la comunicación con el CEX se conecta un cable desde la salida de del CEX MC al pin PD2 de Arduino.

La PCB de verificación con el conector RS485-USB y el Arduino puede observarse en la Figura 5.9

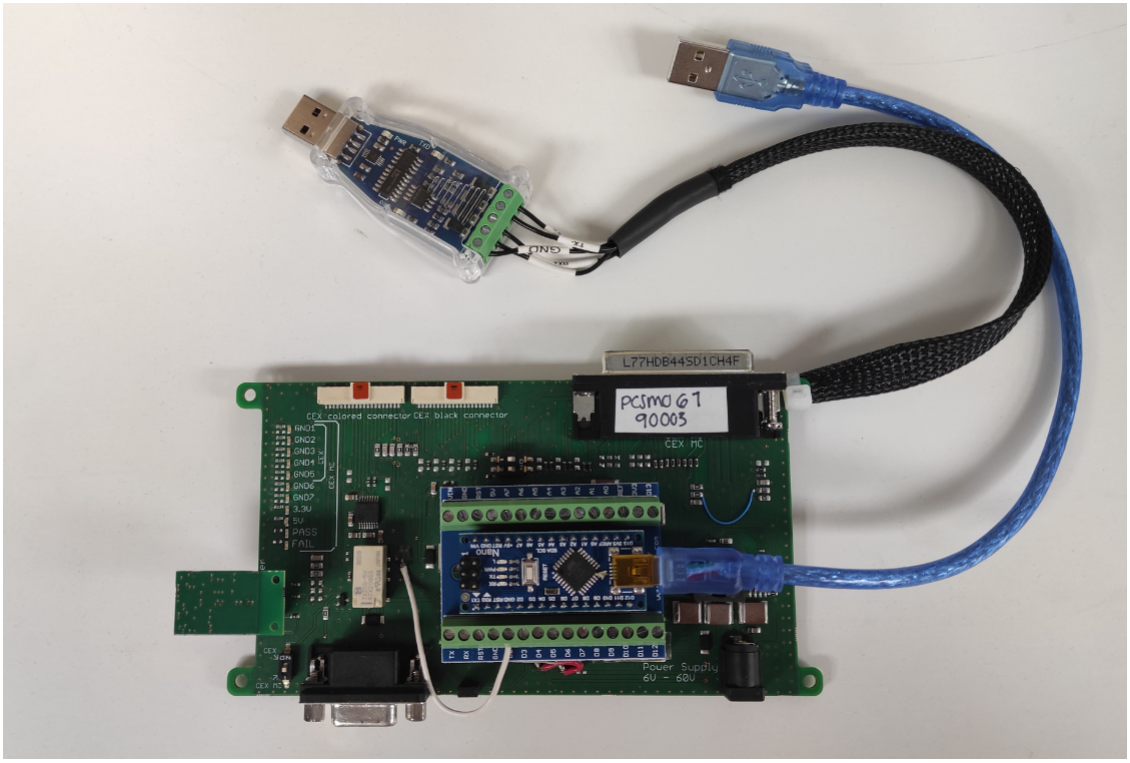


Figura 5.9: PCB de verificación con conexión RS485 y Arduino.

setup

Se inicializa el puerto serie para la comunicación con el ordenador, aunque ésta es opcional, y se hace configura un baudrate de 115200 porque es lo normalmente usado en Embention. A continuación, se configura el pin digital PD2 como entrada y se imprime en pantalla "Start test:". Luego, se configura una interrupción que espera un flanco de bajada en el pin PD2 que, cuando ocurre, llama a la función read_data.

read_data

Se encarga de leer los recibido en el pin PD2. Con la función *pulseIn* se calcula el tiempo de duración de un pulso en estado alto y lo pasa como parámetro a la función *rx_ data*.

rx_data

Se encarga de evaluar la información recibida en *read_ data*. Se evalúa la duración del pulso y, si se cumple que $1110 < signal_rx < 1170$, se comprueba que el pulso se recibió correctamente, por lo que se imprime en pantalla "Signal OK", se desactivan las interrupciones y se llama la función *tx_ data*. En caso que la duración del pulso sea otra se imprime un "Fail" y se desactivan las interrupciones.

El rango de tiempo aceptado fue establecido después de una media de distintas pruebas durante la validación del test.

tx_data

Se encarga de enviar los datos para que los analice PETT. Tras un delay de 1 segundo y configurar el pin PD2 como salida, se envía la siguiente secuencia:

1. PD2 a nivel bajo por 1 s.
2. PD2 a nivel alto por 0.5 s.
3. PD2 a nivel bajo por 0.5 s.
4. PD2 a nivel alto por 0.5 s.
5. PD2 a nivel bajo por 0.5 s.
6. PD2 a nivel alto por 0.5 s.
7. PD2 a nivel bajo por 0.5 s.
8. PD2 como entrada.

Validación Experimental

En la Figura 5.10 se puede observar la medida del osciloscopio de la señal TX conectada al pin del CEX MC. Se observa, al inicio, el tren de pulsos de transmisión delCEX y a continuación el del Arduino, que tiene un estado lógico diferente.

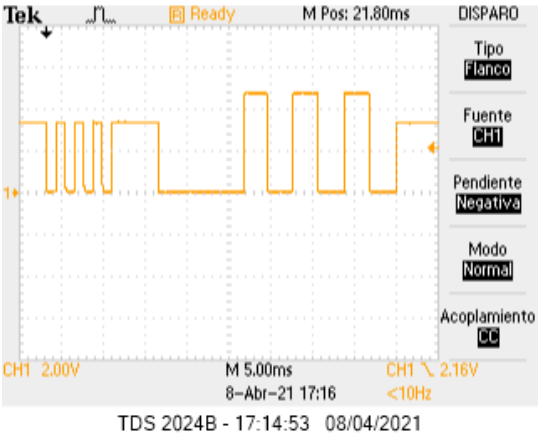


Figura 5.10: Test JETI OK.

Si la conexión no es correcta o el HW del JETI no está bien, el Arduino no responde, como en la Figura 5.11

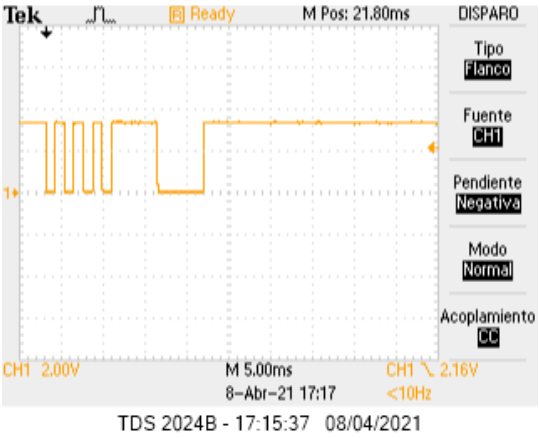


Figura 5.11: Test JETI NOK.

CAPÍTULO 6:

CONCLUSIONES

Por la importancia del CAN Expander (CEX) y CEX MC en los Unmanned Aerial Vehicle (UAV) su correcto funcionamiento es crucial para vuelos seguros y evitar accidentes. En este proyecto se ha mejorado el proceso de verificación del CEX y CEX MC en la línea de producción y se ha creado la documentación asociada al proceso de verificación y resultados, el Acceptance Test Procedure (ATP) y el Acceptance Test Report (ATR).

La Printed Circuit Board (PCB) de verificación, diseñada y fabricada, funciona correctamente. Ha facilitado y agilizado un trabajo tan importante en la empresa como es la verificación de ambos productos, haciendo el proceso de verificación más confiable y práctico. Se ha logrado, también, aumentar la eficiencia de la línea de producción mediante la automatización de los procesos de verificación. Asimismo, se han añadido pasos en la verificación como el Burn In Test y los tests del CEX MC, que hacen el sistema más completo y fiable.

La PCB de verificación ha permitido, también, comprobar el funcionamiento del CEX y CEX MC en otras pruebas de Hardware (HW) en Embention, como tests ambientales para cumplir lo establecido por la DO-160. En la Figura 6.1 se puede observar un CEX MC en una prueba de vibración para la DO-160, siendo comprobado con una PCB de Verificación.

Por otro lado, el desarrollo de Production Embention Testing Tool (PETT) ha permitido que el proceso de verificación sea más robusto y que se comprueben todas las interfaces y periféricos de los productos, en distintas condiciones.

Por todo esto, se puede afirmar que se han cumplido los objetivos propuestos, ya

que se ha mejorado el proceso de verificación en la línea de producción, desarrollando una PCB y un Software (SW) para ello y la documentación asociada al respecto.

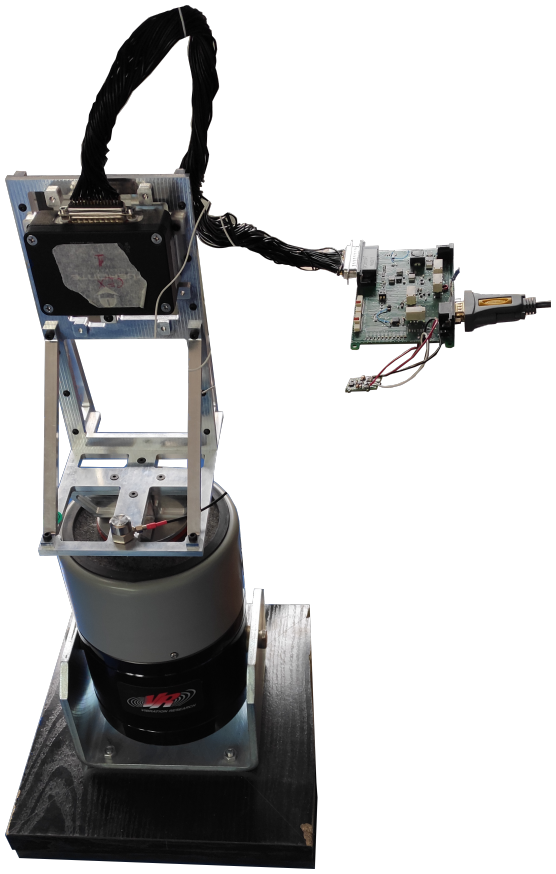


Figura 6.1: CEX en Shaking Test DO-160.

6.1. Líneas Futuras

A pesar del éxito general de este proyecto, quedan aspectos por mejorar. Es por ello por lo que se proponen varias líneas de investigación para continuar con el proyecto desarrollado durante este trabajo.

Las líneas futuras de investigación y desarrollo de la mejora de la verificación de los productos CEX y CEX MC de Embention pueden dividirse en tres partes, que serán explicadas en esta sección: mejora del proceso de verificación, mejora de la PCB de verificación y mejora del SW de verificación.

6.1.1. Mejora del Proceso de Verificación

Aunque el Proceso de verificación en la línea de producción ha mejorado considerablemente, aún hay procesos que optimizar para una verificación más eficaz de cada producto.

Inspección Visual

La Inspección Visual es un paso importante del proceso de verificación porque permite detectar fallos en las pistas, en la polaridad de los componentes o en su ensamblaje. Sin embargo, siempre puede haber un error humano que de resultados falsos positivos. Para evitarlo, puede mejorarse este paso del proceso utilizando el método Automated Optical Inspection (AOI) o Automated X-ray Inspection (AXI).

AOI consiste en utilizar cámaras de 2D o 3D y una fuente de luz para tomar fotografías precisas de la PCB y compararlas con un esquema detallado, para comprobar así el correcto ensamblado y soldadura de los componentes de la PCB. AXI por su parte, tiene la capacidad de inspeccionar elementos ocultos a la vista, porque utiliza rayos X para verificar, por ejemplo, soldaduras debajo de los componentes [19].

In-Circuit Test

Aunque se realiza una verificación de cortocircuitos, antes de alimentar la PCB del CEX y de los tests de PETT, solo se hace de las señales de alimentación principales, más no de todas las señales del producto. Añadir un In-Circuit Test (ICT) permitirá verificar la PCB probando cortocircuitos de todas las señales, circuitos abiertos donde no corresponden, comprobando la resistencia y capacitancia, así como, de forma general, verificar que los componentes están ensamblados y la PCB fabricada correctamente. Un ejemplo de ICT es la *Bed of nails*, que consiste en un dispositivo de prueba electrónica con numerosos pines insertados que están alineados de forma que hagan contacto con los puntos de la PCB que se quieren probar. Sin embargo, dicho sistema de verificación necesita una ensamblaje mecánico para

mantener la PCB en su lugar y, en algunos casos, suele marcar la PCB a verificar en los puntos en los que los pines han hecho contacto; además, se requeriría una bed of nails por cada producto a verificar, por lo que implementarlo en todos los productos de la empresa sería costoso y complicado.

Otra opción para verificar la fabricación y ensamblado de la PCB del CEX es una *Flying Probe*, que consiste en sondas que se mueven de puntos de prueba a otros, según las instrucciones dadas por el programa de SW específico para el producto a verificar. Sus ventajas respecto a la bed of nails son varias: su relación costo-efectividad mejora para lotes pequeños de producción, además, al ser más pequeñas que los pines de la bed, permiten acceder a puntos más específicos; por otro lado, al estar su movimiento controlado por un SW, es más flexible y adaptable. Adicionalmente, la flying probe cuenta con una cámara para verificar la polaridad de los componentes, que es útil también para mejorar la Inspección Visual del proceso de verificación [20].

Verificación de Embarcado

El proceso de verificación en la línea de producción está diseñado para comprobar los productos desde que las PCBs llegan a Embention, hasta que se programan con el SW de embarcado, es decir, con el programa con el que funcionan en manos de los clientes. Sin embargo, como test final de la línea de producción, podría diseñarse un test en el que se comprueben todas las interfaces y periféricos del CEX y CEX MC, durante un tiempo determinado, cuando se programan con su versión de embarcado, para simular así su comportamiento real y verificar que no fallarán cuando lo utilicen los clientes.

6.1.2. Mejora de la PCB de Verificación

Además de modificar la PCB del CEX para que no haga falta hacer los retrabajos que se detallan en el apartado 3.14.1, se proponen las siguientes mejoras para hacer el proceso de verificación más efectivo:

1. La comunicación por RS485 para que sea compatible tanto con el CEX (UART B) como con el CEX MC y sus tres tipos de montaje: RS232 y RS485, UART y RS485, UART half duplex y RS485.
2. El sistema de reinicio puede realizarse con un Microcontrolador (uC), en lugar del circuito de temporizador y puertas lógicas.
3. Estudiar la posibilidad de añadir inductancia en los loopbacks para simular cables o cambiar la forma de simular cables para que sean de mayor longitud.
4. Para mejorar el test del JETI, se propone la realización del test con una emisora u otro dispositivo que utilice el protocolo; o en su defecto, un Arduino con el que se pueda mantener una comunicación serie para que en lugar de pulsos, sea una comunicación half duplex.
5. Mejorar la forma de registrar los ATR para poder generar estadísticas de los resultados.

Además se plantea comprobar un CEX frente a otro, de forma que el uC no se compruebe consigo mismo, sino que la verificación sea comprobando las señales generadas por otro CEX. Esto permite una verificación más robusta y versátil, evitando resultados "falsos positivos".

6.1.3. Mejora del SW de Verificación (PETT)

PETT de CEX ha ido mejorando mientras se validaba junto a la PCB de verificación, pero al ser un SW, es mucho más flexible y adaptable. Como propuesta de mejora se plantea estudiar si es viable desarrollar un sistema para poder programar el CEX sin necesidad del conector Joint Test Action Group (JTAG), de forma que se pueda reducir la interacción humana en el proceso de verificación en la línea de producción.

BIBLIOGRAFÍA

[1] *Veronte Autopilot Datasheet*, Embention, 2019, disponible en https://www.embention.com/wp-content/uploads/2019/02/veronte-autopilot-datasheet.pdf.

[2] *Veronte Autopilot 4x Datasheet*, Embention, 2021, disponible en https://www.embention.com/wp-content/uploads/2021/05/Datasheet-veronte-4x-opt.pdf.

[3] *STLM20 Ultra-low current 2.4 V precision analog temperature sensor*, ST, 2010, disponible en https://www.st.com/content/ccc/resource/technical/document/datasheet/61/83/f6/8b/2f/2a/47/35/CD00119601.pdf/files/CD00119601.pdf/jcr:content/translations/en.CD00119601.pdf.

[4] *TMS320x2833x, TMS320x2823x Technical Reference Manual*, Texas Instruments, 2020, disponible en https://www.ti.com/lit/pdf/sprui07.

[5] "Tips and Tricks for Designing with Voltage References," Texas Instruments, 2021, disponible en https://www.ti.com/lit/eb/slyc147a/slyc147a.pdf. Fecha de consulta: 10/05/2021.

[6] *REF30xx 50-ppm/°C Max, 50-µA, CMOS Voltage Referencein SOT-23-3*, Texas Instruments, 2018, disponible en https://www.ti.com/lit/gpn/ref3020.

[7] "Introduction to the Controller Area Network (CAN)," Texas Instruments, 2016, disponible en https://www.ti.com/lit/an/sloa101b/sloa101b.pdf. Fecha de consulta: 30/11/2020.

[8] "How Burn-In Testing can save you a lot of money – and trouble," Matric Group, 2018, disponible en https://blog.matric.com/burn-in-testing. Fecha de consulta: 07/07/2020.

[9] *LM5164 100-V Input, 1-A synchronous buck DC/DC converter with ultra-low IQ*, Texas Instruments, 2019, rev. A, disponible en https://www.ti.com/lit/gpn/lm5164.

[10] *1.0 A Low-Dropout Positive Fixed and Adjustable Voltage Regulators*, Semiconductor Components Industries, 2021, rev. 30, disponible en https://www.onsemi.com/pdf/datasheet/ncp1117-d.pdf.

[11] *MAX3232 3-V to 5.5-V Multichannel RS-232 line Driver/Receiver with ±15-kV ESD Protection*, Texas Instruments, 2017, rev. N, disponible en https://www.ti.com/lit/gpn/max3232.

[12] "Chapter 4. The Wire," Berkeley University of California, 1999, disponible en http://bwrcs.eecs.berkeley.edu/Classes/icdesign/ee141_f01/Notes/chapter4.pdf. Fecha de consulta: 10/07/2020.

[13] *IRLML6344TRPbF*, International Rectifier, 2014, disponible en http://www.farnell.com/datasheets/1911844.pdf.

[14] *TLC555-Q1 LinCMOS TIMER*, Texas Instruments, 2015, rev. B, disponible en https://www.ti.com/lit/gpn/tlc555-q1.

[15] *SN74LVC1G80 Single Positive-Edge-Triggered D-Type Flip-Flop*, Texas Instruments, 2016, rev. S, disponible en https://www.ti.com/lit/gpn/sn74lvc1g80.

[16] "TI Linker Command File Primer," Texas Instruments, 2021, disponible en http://software-dl.ti.com/ccs/esd/documents/sdto_cgt_Linker-Command-File-Primer.html. Fecha de consulta: 03/11/2020.

[17] *MMC5883MA ±8Gauss, High Performance, Low Cost 3-axis Magnetic Sensor*, MEMSIC, 2017, disponible en https://www.mouser.es/datasheet/2/821/mcic_s_a0004454845_1-2295297.pdf.

[18] *JETI Telemetry communication protocol*, JETI Model, 2015.

[19] "Auto Inspection makes your electronic contract manufacturer productive," Matric Group, 2018, disponible en https://blog.matric.com/

automated-inspection-electronic-contract-manufacturers. Fecha de consulta: 30/06/2020.

[20] "How Flying Probe Testing Works for PCB Assembly," Sierra Circuits, 2020, disponible en https://www.protoexpress.com/blog/how-flying-probe-testing-works-for-pcb-assembly/. Fecha de consulta: 20/05/2021.

ACRÓNIMOS

ACK	Acknowledgement
ADC	Analog-to-Digital Converter
AOI	Automated Optical Inspection
AQ	Action-qualifier
ATP	Acceptance Test Procedure
ATR	Acceptance Test Report
AXI	Automated X-ray Inspection
BoM	Bill of Materials
CAN	Controller Area Network
CC	Counter-Compare
CCS	Code Composer Studio
CEX	CAN Expander
CMPA	Counter-Compare A Register
CMPB	Counter-Compare B Register
CPK	CAN protocol kernel
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DB	Dead-band
E/S	Entrada/Salida
eCAN	Enhanced Controller Area Network
eCAP	Enhanced Capture
ePWM	Enhanced Pulse Width Modulator
ET	Event-trigger

FIFO	First In First Out
G	Gauss
GND	masa
GPIO	General-purpose I/O pin
HW	Hardware
I2C	Inter-Integrated Circuit Module
I2CPSC	I2C Prescaler Register
ICT	In-Circuit Test
IDE	Integrated Development Environment
JTAG	Joint Test Action Group
LED	Light-Emitting Diode
LSB	Less Significant Bit
LTT	Long-Term Test
Mbps	Megabits por segundo
MMU	Memory Management Unit
MSB	Most Significant Bit
MSPS	Megasamples per second
NACK	Negative Acknowledgement
NOPOP	No Populated
NRZ	Non Return to Zero
PCB	Printed Circuit Board
PETT	Production Embention Testing Tool
PWM	Pulse Width Modulator
RAM	Random Access Memory
RMA	Return Merchandise Authorization
RX	receptor
RXSHF	Receiver shift register

S/N	Serial Number
S+H	Sample and Hold
SCC	Standard CAN Controller Mode
SCI	Serial Communications Interface
SCIRXBUF	Receive data buffer
SCIRXD	SCI receive-input pin
SCITXBUF	Transmit data buffer
SCITXD	SCI transmit-output pin
SCL	System Clock I2C
SDA	System Data I2C
SW	Software
SYSCLK	System Clock uC
TB	Time-base
TBCTR	Time-Base Counter Register
TBPRD	Time-Base Period Register
TX	transmisor
TXSHF	Transmitter shift register
TZ	Trip-Zone
UART	Universal Asynchronous Receiver-Transmitter
UAV	Unmanned Aerial Vehicle
uC	Microcontrolador
USB	Universal Serial Port

ANEXOS

ANEXO A:

DATASHEET CEX Y CEX MC

EMBENTION

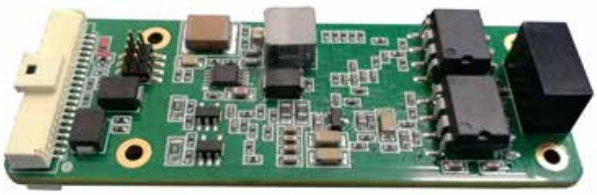
VERONTE CAN EXPANDER

VERONTE CAN EXPANDER

GENERAL **DESCRIPTION**

The CAN Expander greatly increases the input and output port capacity in Veronte Autopilot, by multiplexing several analogue and digital I/O ports and buses into the redundant CAN bus.

It can also optimise wiring by concentrating multiple sensors at a remote section of the vehicle and permits to take advantage of the robustness of CAN bus communications for long distances.

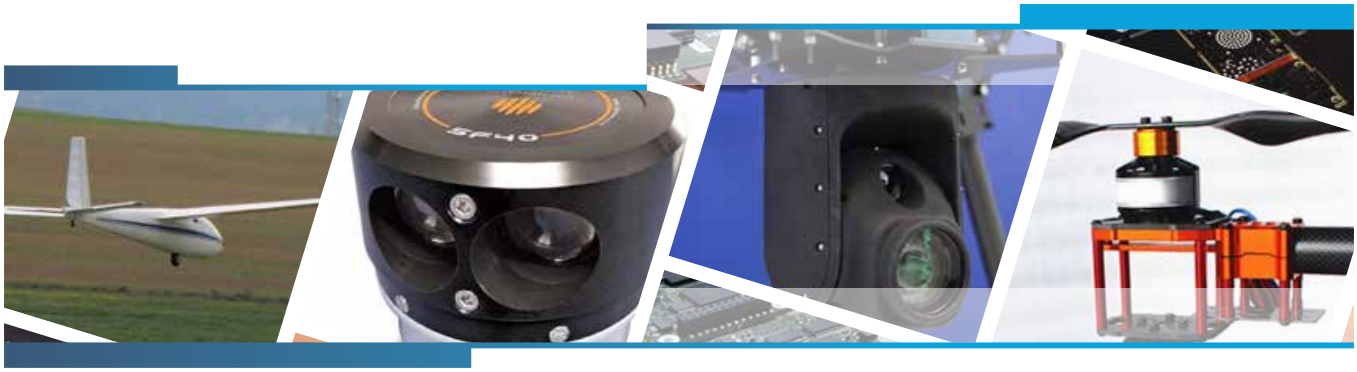


APPLICATIONS

- Payload Control
- Wiring Optimisation
- Long Distance Wiring
- Sensor Feedback
- Complex Telemetry
- Large Vehicles

MAIN **FEATURES**

- Redundant CAN Bus
- CAN Isolation
- Low Consumption
- Multiple Interfaces
- Easy Integration
- Compact Design



embention.com

EMBENTION

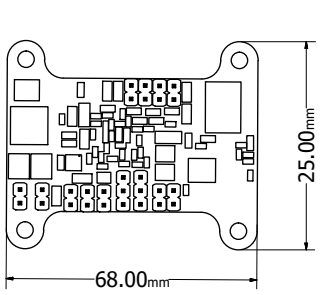
VERONTE CAN EXPANDER

SPECIFICATIONS & HIGHLIGHTS

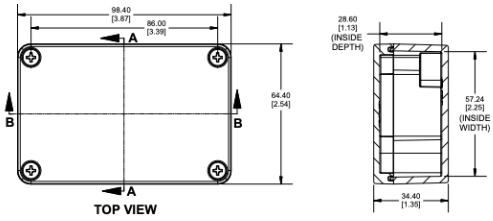
Features	Description
Redundant Power Input	2x (6 – 60VDC) - Independent supplies - Reverse Polarity Protection
Power Outputs	1x 3.3VDC (100 mA) - 1x 5VDC (100mA)
CAN Bus	2x CAN bus (2 or 3 wires)
UART	2x TTL2C (RS232 + RS485 for Veronte CAN Expander MC)
I2C	1
Digital outputs (PWM)	8x 5VDC
Digital inputs (ECAP)	4x 5VDC
Analogue inputs	8 - 2x 0-36VDC, 2x 0-12V, 2x 0-5V, 2x 0-3.3V
Safety	Temperature sensor, isolated redundant CAN bus
Dimensions	68 x 25 mm | 10g
Light Installation	2 wire CAN bus installation
Robust Design	High-performance components and strict design standards
Redundancy	CAN bus redundancy management
Versatility	Expansion ports, long wiring, sensor condenser, failsafe configurations...

DIMENSIONAL DRAWING

VEPE006 – CAN Expander (without enclosure)



VEPE005 – CAN Expander MC (with enclosure)

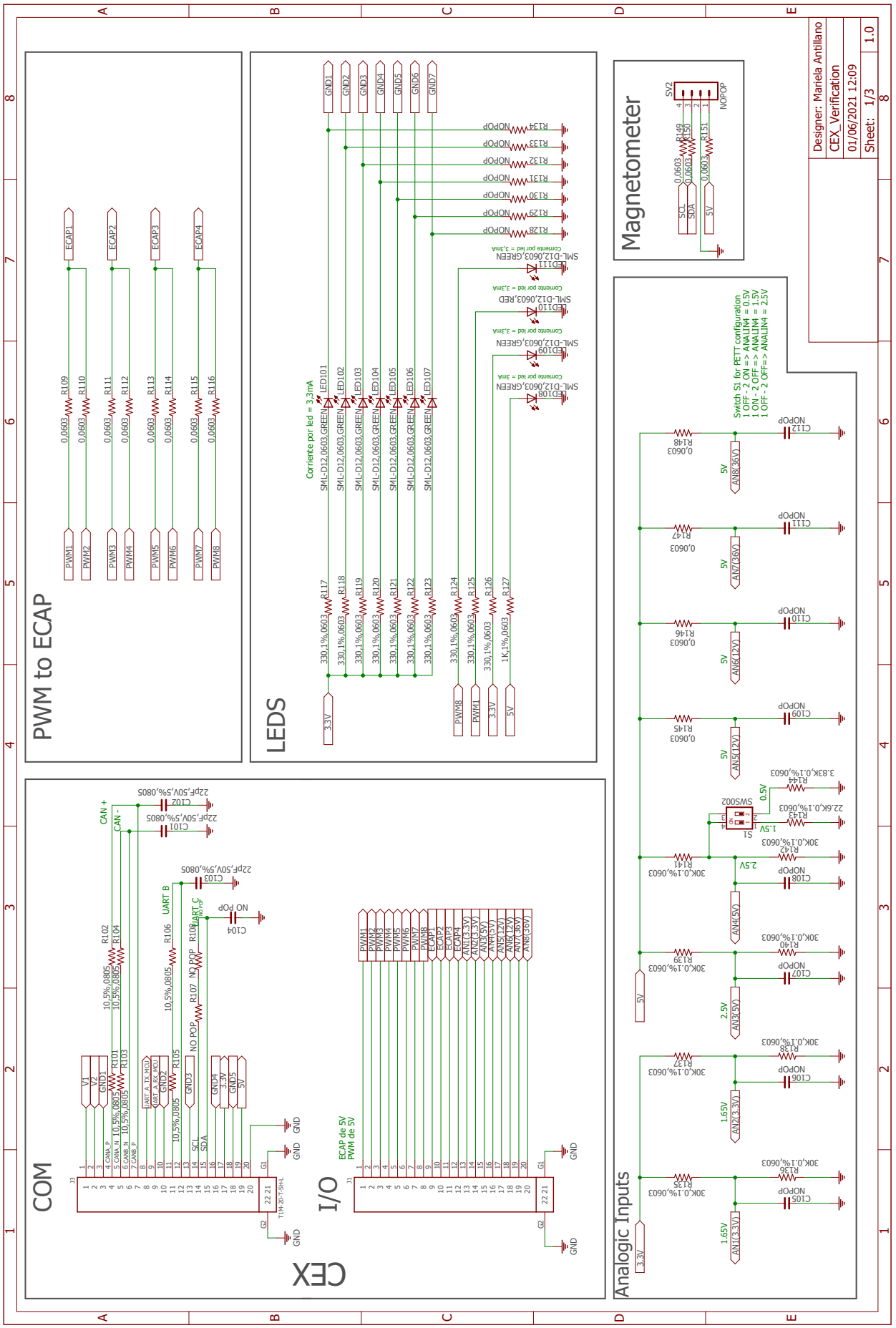


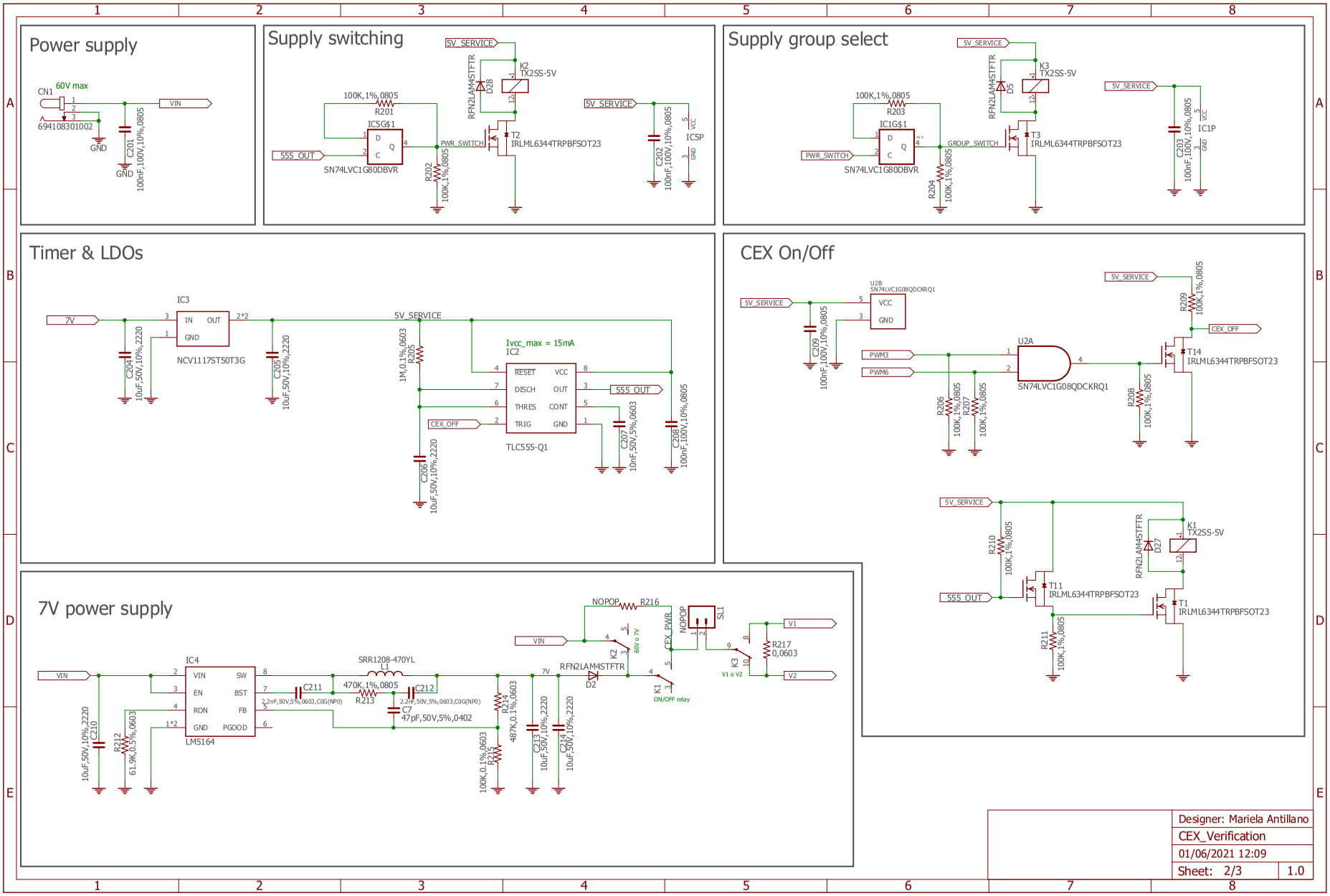
embention.com
sales@embention.com
+34 965 115 421

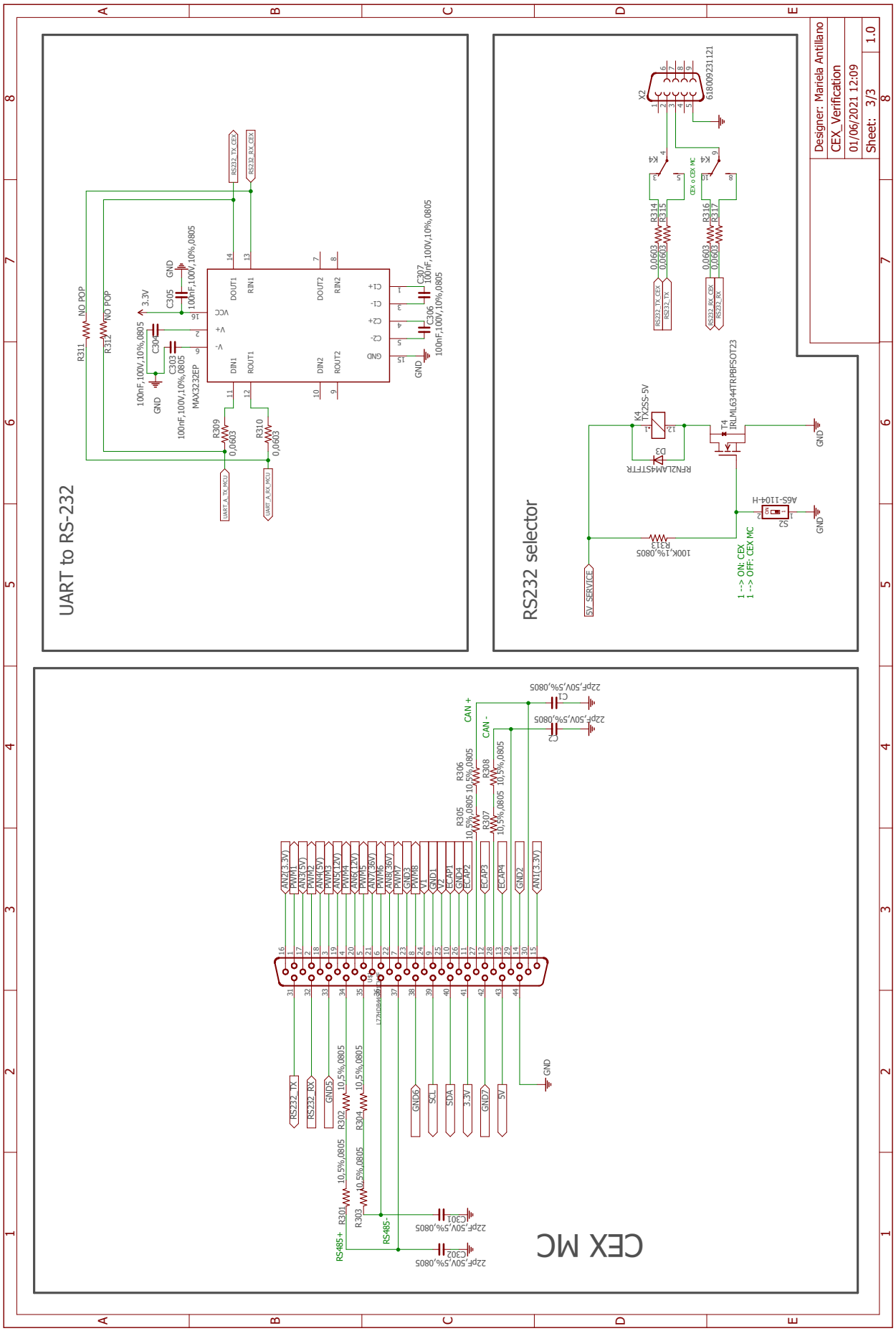
ANEXO B:

DISEÑO ESQUEMÁTICO DE LA PCB DE VERIFICACIÓN

El diseño de HW de la PCB de Verificación se puede encontrar también en el siguiente enlace: *CEX Verification PCB*


PWM to ECAP
LEDS
Magnetometer
Analogic Inputs
COM
I/O
CEX
Designer: Mariela Antillano
CEX_Verification
01/06/2021 12:09
Sheet: 1/3
1.0

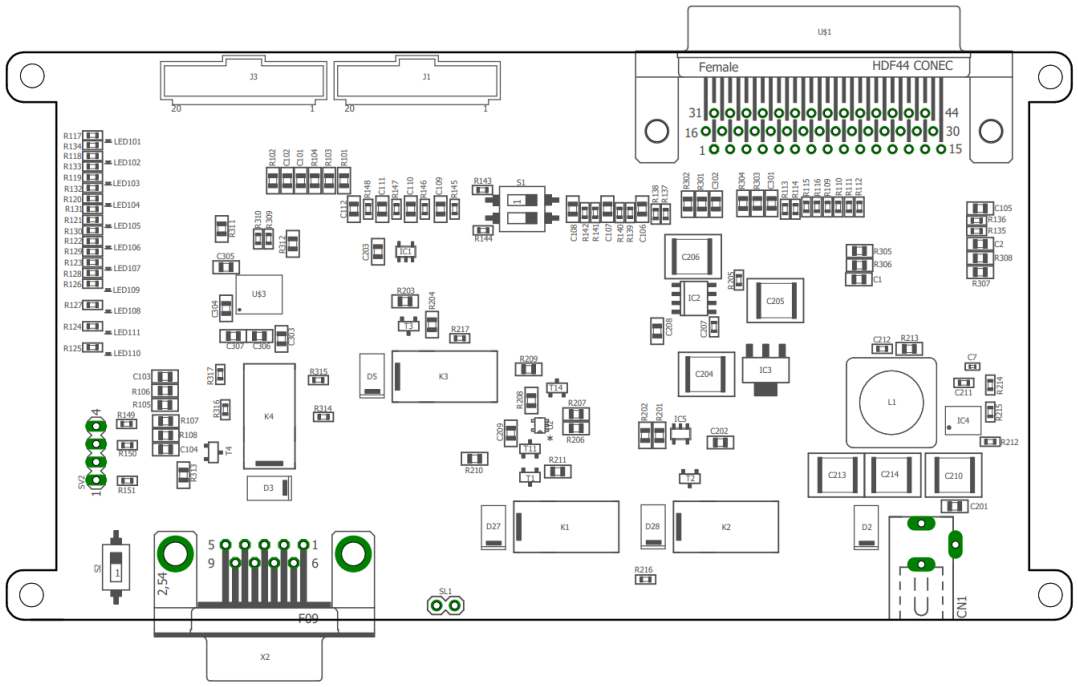

Power supply
CN1
60V max
694108301002
VIN
GND
C201
100nF,100V,10%,0805
Supply switching
5V_SERVICE
K2
TX2SS-5V
RFN2LAM4STFTR
D28
100K,1%,0805
R201
IC5G$1
555_OUT
PWR_SWITCH
SN74LVC1G80DBVR
T2
IRLML6344TRPBFSOT23
R202
100K,1%,0805
C202
100nF,100V,10%,0805
IC5P
Supply group select
K3
TX2SS-5V
D5
R203
GROUP_SWITCH
T3
R204
IC1P
C203
Timer & LDOs
IC3
7V
IN
OUT
GND
NCV1117ST50T3G
C204
10uF,50V,10%,2220
C205
2*2
5V_SERVICE
Ivcc_max = 15mA
IC2
RESET
VCC
DISCH
THRES
CONT
TRIG
TLC555-Q1
R205
1M,0.1%,0603
C206
C207
10nF,50V,5%,0603
C208
100nF,100V,10%,0805
CEX On/Off
U2B
SN74LVC1G08QDCKRQ1
C209
100nF,100V,10%,0805
U2A
PWM3
PWM6
R206
100K,1%,0805
R207
R208
R209
CEX_OFF
T14
T11
T1
R210
R211
7V power supply
IC4
VIN
EN
RON
SW
BST
FB
PGOOD
LM5164
SRR1208-470YL
L1
C211
2.2nF,50V,5%,0603,C0G(NP0)
470K,1%,0805
R213
C212
C7
47pF,50V,5%,0402
RFN2LAM4STFTR
D2
NOPOP
R216
CEX_PWR
SL1
K1
ON/OFF relay
K3
V1 o V2
R217
0,0603
V1
V2
C210
R212
61.9K,0.5%,0603
R214
487K,0.1%,0603
R215
100K,0.1%,0603
C213
C214
Designer: Mariela Antillano
CEX_Verification
01/06/2021 12:09
Sheet: 2/3
1.0


CEX MC
UART to RS-232
RS232 selector
Designer: Mariela Antillano
CEX_Verification
01/06/2021 12:09
Sheet: 3/3
1.0
MAX3232EP
TX2SS-5V
IRLML6344TRPBFSOT23
A6S-1104-H
618009231121
1 --> ON: CEX
1 --> OFF: CEX MC
5V_SERVICE
CAN +
CAN -
RS232_TX
RS232_RX
RS485+
RS485-
SCL
SDA
3.3V
5V
GND

ANEXO C:

DISEÑO BOARD DE LA PCB DE VERIFICACIÓN

C.1. Ubicación de Componentes



C.2. Cara Top

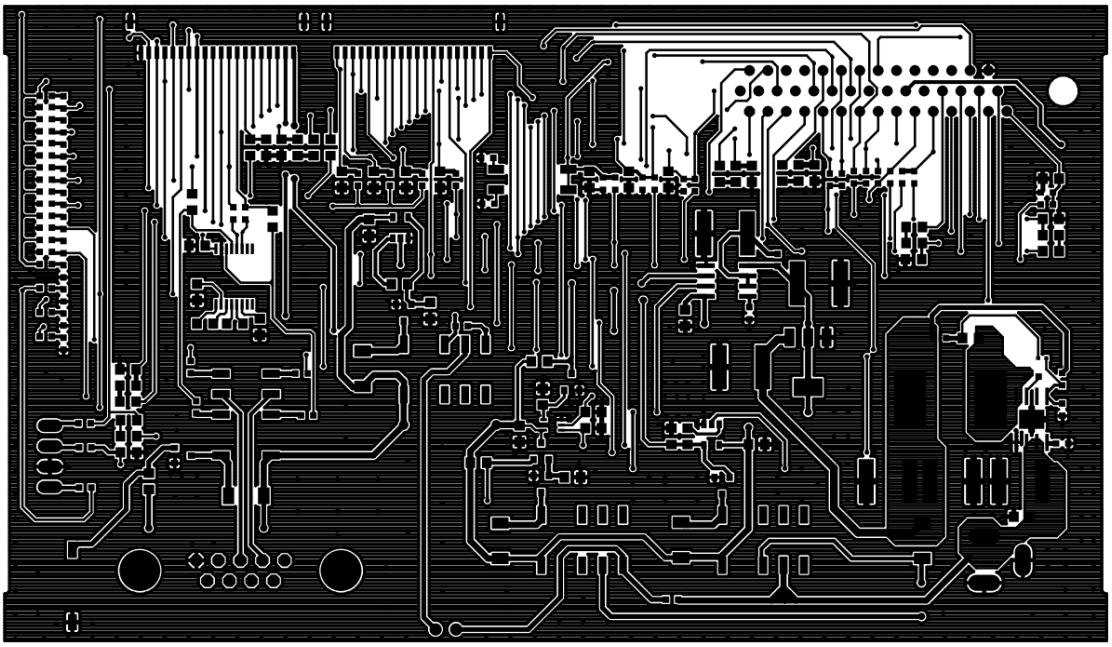


Figura C.1: Copper, component side (cmp)

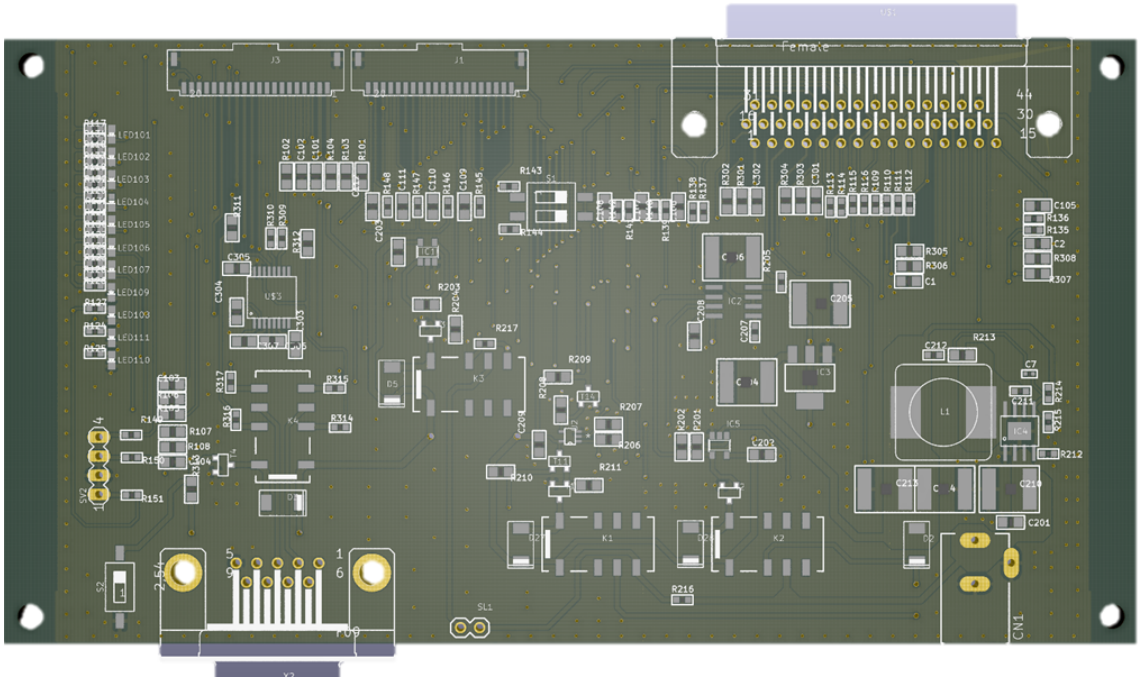


Figura C.2: Top board

C.3. Cara Bottom

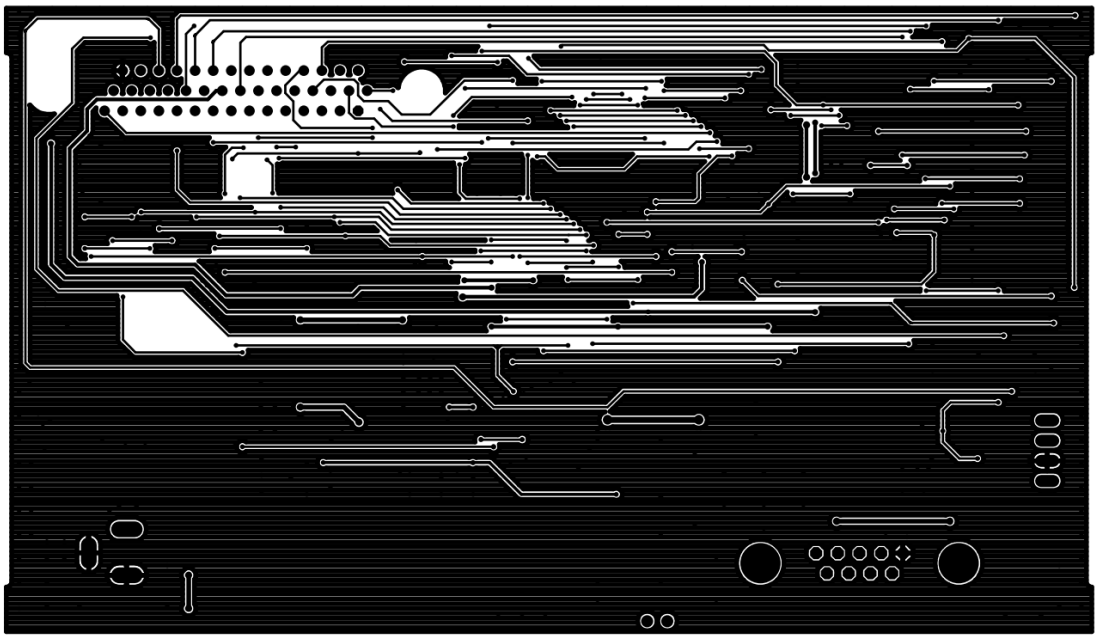


Figura C.3: Copper, solder side (sol)

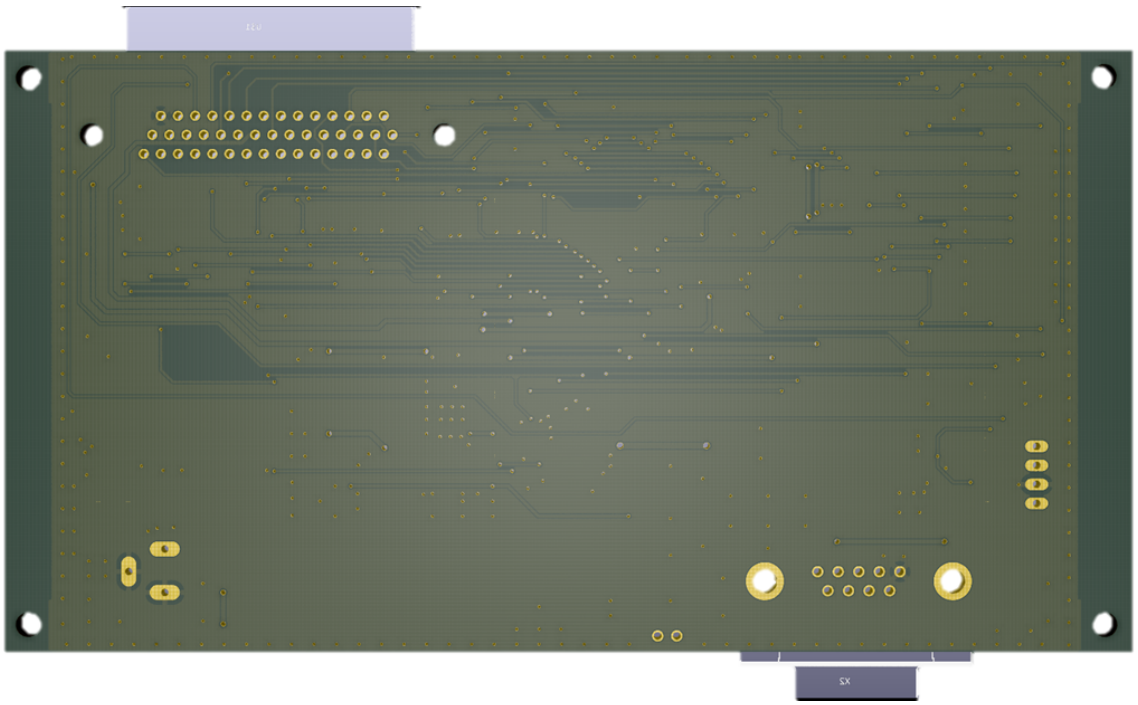


Figura C.4: Bottom board

ANEXO D:

BILL OF MATERIALS (BOM)

Device:	CEX Verification		Mariela Antillano Fernández							
Version	1.0									
Qty-1	Part References	Value	Device	REF_INT	REF_VALUE	Comercial Standard Manufacturer P/N	Int. Ref. Automotive	Automotive Standard Manufacturer P/N	Minimum Operating Temperature (ºC)	Maximum Operating Temperature (ºC)
22	R109, R110, R111, R112, R113, R114, R115, R116, R145, R146, R147, R148, R149, R150, R151, R217, R309, R310, R314, R315, R316, R317	0,0603	R_0603	CERE002	0R, 5%, 0.1W, 0603	CRCW06030000Z0EA	CERE002A	CRCW06030000Z0EA	-55	155
1	R127	1K,1%,0603	R_0603	CERE023	1K,1%,0603	CRCW06031K00FKEAC	CERE023A	ERJ-3EKF1001V	-55	155
1	R205	1M,0.1%,0603	R_0603	CERE580	1M,0.1%,0603	PCF0603R-1M0BI	CERE580A	RN73H1JTTD1004B25	-55	125
2	C211, C212	2.2nF,50V,5%,0603,C0G(NP0)	C_0603	CECO252	2.2nF,50V,5%,0603,C0G(NP0)	GRM1885C1H222JA01D	CECO252A	GCM1885C1H222JA16D	-55	125
1	R144	3.83K,0.1%,0603	R_0603	CERE579	3.83K,0.1%,0603	RT0603BRD073K83L	CERE579A	ERA-3AEB3831V	-55	155
14	R101, R102, R103, R104, R105, R106, R301, R302, R303, R304, R305, R306, R307, R308	10,5%,0805	R_0805	CERE515	10,5%,0805	CRCW080510R0JNEAC	CERE515A	RCS080510R0JNEA	-55	155
1	C207	10nF,50V,5%,0603	C_0603	CECO061	10nF,50V,5%,0603	VJ0603Y103JXACW1BC	CECO061A	GCM1887U1H103JA16D	-55	125
6	C204, C205, C206, C210, C213, C214	10uF,50V,10%,2220	C_2220K	CECO281	10uF,50V,10%,2220	C5750X5R1H106K230KA	CECO281A	CGA9N3X7R1H106K230KB	-55	85
1	R143	22.6K,0.1%,0603	R_0603	CERE578	22.6K,0.1%,0603	RT0603BRD0722K6L	CERE578A	ERA-3AEB2262V	-55	155
7	C1, C2, C101, C102, C103, C301, C302	22pF,50V,5%,0805	C_0805	CECO236	22pF,50V,5%,0805	CC0805JPNPO9BN220	CECO236A	AC0805JRNPO9BN220	-55	125
8	R135, R136, R137, R138, R139, R140, R141, R142	30K,0.1%,0603	R_0603	CERE562	30K,0.1%,0603	ERJ-PB3B3002V	CERE562A	ERA-3AEB303V	-55	155
1	C7	47pF,50V,5%,0402	C_0402	CECO086	47pF,50V,5%,0402	0402N470J500CT	CECO086A	GCM1555C1H470JA16J	-55	125
1	R212	61.9K,0.5%,0603	R_0603	CERE470	61.9K,0.5%,0603	ERJ-PB3D6192V	CERE470A	RNCF0603DTE61K9	-55	155
1	R215	100K,0.1%,0603	R_0603	CERE588	100K,0.1%,0603	APC0603B100KN	CERE588A	ERJ-PB3B1003V	-55	155
11	R201, R202, R203, R204, R206, R207, R208, R209, R210, R211, R313	100K,1%,0805	R_0805	CERE252	100K,1%,0805	CRGCQ0805F100K	CERE252A	ERJ-6ENF1003V	-55	125
10	C201, C202, C203, C208, C209, C303, C304, C305, C306, C307	100nF,100V,10%,0805	C_0805	CECO214	100nF,100V,10%,0805	C0805C104K1RAC	CECO214A	GCM21BR72A104KA37L	-55	125
10	R117, R118, R119, R120, R121, R122, R123, R124, R125, R126	330,1%,0603	R_0603	CERE577	330,1%,0603	CRCW0603330RFKEAC	CERE577A	ERJ-3EKF3300V	-55	155
1	R213	470K,1%,0805	R_0805	CERE543	470K,1%,0805	CR0805-FX-4703ELF	CERE543A	ERJ-6ENF4703V	-55	155
1	R214	487K,0.1%,0603	R_0603	CERE589	487K,0.1%,0603	CPF0603B487KE1	CERE589A	CPF0603B487KE1	-55	155
1	X2	6,18009E+11	F09HP	CECN469	618009231121 - Conector D-sub 9P hembra	618009231121	CECN469A	618009231121	-40	85
1	CN1	6,94108E+11	6,94108E+11	CECN465	DC Power JACK 5.5mm	694108301002	CECN465A	694108301002	-40	85
1	S2	A6S-1104-H	SWS001	CECN485	A6S-1104-H - Switch Slide DIP 1 pos	A6S-1104-H	CECN485A	A6S-1104-H	-20	-70
6	T1, T2, T3, T4, T11, T14	IRLML6344TRPBFSOT23	IRLML6344TRPBFSOT23	CERP083	IRLML6344TRPBF - MOSFET N-CH 30V 5A	IRLML6344TRPBF	CERP083A	IRLML6344TRPBF	-55	155
1	U$1	L77HDB44SD1CH4F	HF44H	CECN484	L77HDB44SD1CH4F - Conector 44 pin THT Horizontal	L77HDB44SD1CH4F	CECN484A	L77HDB44SD1CH4F	-55	125
1	IC4	LM5164	LM5164	CEIC286	LM5164 - IC DCDC 100V 1A SO-PowerPad-8	LM5164DDAR	CEIC286A	LM5164QDDATQ1	-40	150
1	U$3	MAX3232EP	MAX3232EP	CEIC042	MAX3232 - RS-232	MAX3232EIDBR	CEIC042A	MAX3232MDBREP	-40	85
1	IC3	NCV1117ST50T3G	NCV1117ST50T3G	CEIC280	NCV1117ST50T3G - LDO 5V 1A SOT-223-3	NCV1117ST50T3G	CEIC280A	NCV1117ST50T3G	-40	125
2	R311, R312	NO POP	R_0805	NA	*	*	*	*		
1	C104	NO POP	C_0805	CECO236	22pF,50V,5%,0805	CC0805JPNPO9BN220	CECO236A	AC0805JRNPO9BN220	-55	125
2	R107, R108	NO POP	R_0805	CERE515	10,5%,0805	CRCW080510R0JNEAC	CERE515A	RCS080510R0JNEA	-55	155
1	SL1	NOPOP	M02PTH	NA	*	*	*	*		
1	SV2	NOPOP	MA04-1	NA	*	*	*	*		
8	C105, C106, C107, C108, C109, C110, C111, C112	NOPOP	C_0805	CECO214	100nF,100V,10%,0805	C0805C104K1RAC	CECO214A	GCM21BR72A104KA37L	-55	125
8	R128, R129, R130, R131, R132, R133, R134, R216	NOPOP	R_0603	CERE002	0R, 5%, 0.1W, 0603	CRCW06030000Z0EA	CERE002A	CRCW06030000Z0EA	-55	155
5	D2, D3, D5, D27, D28	RFN2LAM4STFTR	RFN2LAM4STFTR	CERP281	RFN2LAM4STFTR - Diodo propósito general - 400V	RFN2LAM4STFTR	CERP281A	RFN2LAM4STFTR	-55	150
10	LED101, LED102, LED103, LED104, LED105, LED106, LED107, LED108, LED109, LED111	SML-D12,0603,GREEN	LED0603	CERP010	LED 560nm GREEN 0603	SML-D12P8WT86	CERP010A	SML-D12P8WT86C	-40	85
1	LED110	SML-D12,0603,RED	LED0603	CERP011	LED 620/630nm RED 0603	SML-D12U8WT86	CERP011A	SML-D12V8WT86C	-40	85
1	U2	SN74LVC1G08QDCKRQ1	SN74LVC1G08QDCKRQ1	CEIC238	2-IN AND Gate - SN74LVC1G08QDCKRQ1	SN74LVC1G08QDCKRQ1	CEIC238A	SN74LVC1G08QDCKRQ1	-40	125
2	IC1, IC5	SN74LVC1G80DBVR	74LVC1G79DBV	CEIC291	SN74LVC1G80DBVR - FlipFlop - SOT-23-5 - 5V	SN74LVC1G80DBVR	CEIC291A	SN74LVC1G80DBVR	-40	125
1	L1	SRR1208-470YL	SRR1208	CERP287	SRR1208-470YL - Inductor 47uH 15% SMD 1208	SRR1208-470YL	CERP287A	SRR1208-470YL	-40	125
1	S1	SWS002	SWS002	CECN464	A6S-2104-H - Switch Slide DIP 2 pos	A6S-2104-H	CECN464A	A6S-2104-H	-20	70
2	J1, J3	T1M-20-T-SH-L	T1M-20-X-SH-L	CECN245	T1M-20-T-SH-L - Conector 20 pines macho	T1M-20-F-SH-L-K	CECN245A	T1M-20-F-SH-L-K	-55	85
1	IC2	TLC555-Q1	TLC555-Q1	CEIC283	Timing IC - TLC555-Q1	*	CEIC283A	TLC555-Q1	-40	125
4	K1, K2, K3, K4	TX2SS-5V	TX2SS-5V	CERP280	TX2SS-5V-Z - Relay Coil 5V-28.1mA / Contact 30VDC-2A	TX2SS-5V-Z	CERP280A	TX2SS-5V-Z	-40	85

ANEXO E:

ORDEN DE PRODUCCIÓN

ORDER

We thank you for your order and are pleased to confirm it to you as follows 19 August 2020

Administrative details

Your references

Order nr.	E1229725	Purchase reference	
Service	PCB proto	Project reference	
Board name	CEX Verificator	Article number	

Invoicing & delivery details

Invoice to:
Embention Sistemas Inteligentes
David Benavente
Portugal 23
03003 Alicante
Spain
34965 115 421
purchases@embention.com

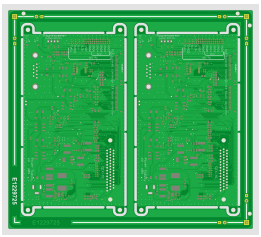
Delivered to:
Embention Sistemas Inteligentes
David Benavente
C/ Chelin 16
03114 Alicante
Spain
34965 115 421
purchases@embention.com

PCB

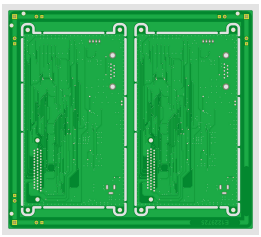
PCB Visualizer

PCB images

Top view :



Bottom view :



Buildup & Mechanical plan

Board buildup :

Technology & options

Board definition

Number of layers	2	Delivery format	Eurocircuits, according to standard panel rules
PCB width (X)	150 mm	PCB height (Y)	80 mm
eC-registration compatible	Yes		

Panel definition

Repeat in X	1	Repeat in Y	2
Panel width	174 mm	Panel height	194 mm
PCBs per panel	2	Panel border	10 mm
PCB separation method	Breakrouting	PCB spacing	10 mm
Panel without cross outs	No	Panel outline	Routed

Board definition

ORDER

Top soldermask	Green	Bottom soldermask	Green
Top legend	White	Bottom legend	No
Surface finish	Any lead free finish		
Bare board testing	Yes		

Board technology

Pattern class	6	Drill class	Drill C
Outer layer trackwidth (OL-TW)	0.250 mm	Hole density	<1000/dm2
Outer layer isolation distance (OL-TT-TP-PP)	0.150 mm	Holes <= may be reduced	0.45 mm
Outer layer annular ring (OAR)	0.125 mm		

Material definition

Board thickness	1.55 mm	Board buildup	Standard buildup
Base material	FR4IMP	Material Tg	145-150°C
Outer layer copper foil	18µm(End-Cu +/-35µm)	Inner layer copper foil	0
Extra PTH runs	0	Extra press cycles	0
Reversed buildup	No	Inner layer core thickness	Standard

Advanced options

Milling	No	Peelable mask	No
Edge connector gold surface mm²	0	Edge connector bevelling	No
Copper up to board edge	No	Plated holes on the board edge	No
Carbon contacts	No	Viafill	No
Top heatsink paste	No	Bottom heatsink paste	No
Specific tolerances	No	Specific marking	No
Press-fit holes	No	Depth routing	No
Round-edge plating	No	Chamfered mechanical holes	No
Cross section report required	No		

Pricing

Printed circuits

Order nr.	Shipment date	Quantity	Unit price	Transport price	Transport mode	Total price	VAT	Gross
E1229725	24 August 2020	5	40.06 EUR	5.37 EUR	Express	205.67 EUR	0.0 %	205.67 EUR

ASSEMBLY

Technology & options

Special assembly requirements No

	Supplied by Assembler		Supplied by Customer	
Type	**Unique parts**	**Component placed**	**Unique parts**	**Component placed**
Thru-hole	3	3	0	0
Mechanical	0	0	0	0
SMD	34	138	0	0
SMD fine pitch	0	0	0	0
Mixed	0	0	0	0
BGA	0	0	0	0
BGA FinePitch	0	0	0	0
QFN	0	0	0	0
QFN FinePitch	0	0	0	0

LGA	0	0	0	0
Unknown	0	0	0	0
Edge-Mount	0	0	0	0
Total	**37**	**141**	**0**	**0**

No info (NI) : Our online tools were unable to get price and/or availability information from our suppliers. Our engineers are informed and try to update the information manually.

Pricing

Order nr.	Estimated shipment date*	Components price	Assembly price	VAT	Gross
E1229725-A	27 August 2020	570.50 EUR	420 EUR	0.0 %	990.50 EUR

The "estimated shipment date" of the assembled PCB can only be guaranteed if all components are available for our assembly production on the confirmed shipment date of the bare PCB.

Payment terms & conditions

The payment term is 30 days from invoice date.

This quotation is valid for 30 days. All our deliveries are according to our general terms and conditions of delivery. These are available on the website , and agreed upon between us during the registration procedure. All above mentioned prices are an indication on the basis of the information at our disposal on the moment of quotation. These prices may be reviewed at the moment of order on the basis of the final documentation and conditions. The final quantity to be delivered can vary up to 5% of the ordered quantity. Delivery terms start counting upon receipt of the complete documentation and firm order.

ANEXO F:

CÓDIGO DE PETT

El SW PETT del CEX se puede encontrar también en el siguiente enlace: *CEX PETT*

F.1. main.cpp

```
#include <Memmgr.h>
#include <System.h>
#include <Flashif.h>
#include <Hsys.h>
#include <Timer.h>
#include <Resetif.h>
#include <Logger.h>

#include <Halsuite.h>
#include <SerialTestCtrl.h>
#include <CEX_base.h>
#include <TestADC.h>
#include <TestSCI.h>
#include <TestPWM.h>
#include <TestMMC5883MA.h>
#include <TestAddr.h>
#include <TestJETI.h>
#include <AppWruAddress.h>
#include <Flash_wr.h>
#include <Address0.h>
#include <Address.h>
#include <Sysaddr.h>
#include <LTTest.h>

```

```
25#include <File.h>
26#include <Vstring.h>
27#include <Logger.h>
28#include <Delay.h>
29#include <TestCAN.h>
30
31using namespace Base;
32using namespace Dsp28335_ent;
33using namespace PETT;
34
35// Memory Manager init starts here
36/// Stack size
37const Uint16 mem_stack_sz = 0x400;
38/// System memory size (globals, statics, etc...)
39const Uint16 mem_sys_sz  = 0x800 + 0x100 + 4; // ebss.1 + ebss.2 + esysmem
40/// Internal allocator memory size
41const Uint16 mem_int_sz  = 0x20;
42/// External allocator memory size
43const Uint16 mem_ext_sz  = 0x08000 - mem_stack_sz - mem_sys_sz - mem_int_sz;
44
45#pragma DATA_SECTION("MEMMGR_INT")
46Uint16 memmgr_int_buf[mem_int_sz];
47
48#pragma DATA_SECTION("MEMMGR_EXT")
49Uint16 memmgr_ext_buf[mem_ext_sz];
50// memory manager init ends here
51
52namespace CEX
53{
54    static const Uint16 cex_addr[1] = {};
55    static const Uint16 burnin_addr[1] = {};
56
57}
58
59extern "C"
60{
61// Pre initialization. If there is external RAM to be used during initialization,
    it must be setup here.
62int _system_pre_init(void)  // PRQA S 5047 #Exception Rule 47
```

```
{
    Dsp28335_ent::System::init0();
    return 1;
}

// Post copy initialization. Here the ramfuncs have been already copied to RAM.
int _system_post_cinit(void)    // PRQA S 5047 #Exception Rule 47
{
    // Call Flash Initialization to setup flash waitstates
    // This function must reside in RAM
    Dsp28335_ent::Flash_if::init();
    return 1;
}
}

// Interrupt domain ------
interrupt void handle_isr(void)
{
    static CEX::Halsuite& hal = CEX::Halsuite::get_instance(); // HAL instance
    Bsp::Htime::get_time(); // This call is needed to avoid overflow
    hal.poll();
}

int main()
{
    CEX::Halsuite& hal = CEX::Halsuite::get_instance();

    Dsp28335_ent::Pwmdev::init_all();


    //Configure interruptions
    hal.tmr2.start_task(1000, handle_isr);
    hal.enable_global_isr();

    //Start of PETT

    SerialTestCtrl& ctrl = SerialTestCtrl::get_instance();

    //Check uaddress
```

```
102     Uint16 eff_addr = CEX::cex_addr[0]; // Address to use
103
104     // Store the system address
105     Dsp28335_ent::Hsys::set_sysaddr(Address(eff_addr));
106
107     AppWruAddress::set_addr(CEX::cex_addr[0]);
108     SerialTestCtrl::set_addr(CEX::cex_addr[0]);
109     TestAddr::get_addr(CEX::cex_addr[0]);
110
111
112     //Set PWM as output low
113     Dsp28335_ent::GPIO pwm1(Dsp28335_ent::GPIOname::gpio_pwm_01);
114     pwm1.dir(true); // true -> output
115     pwm1.fun_mux(0);
116     pwm1.clear(); // output low
117     Dsp28335_ent::GPIO pwm2(Dsp28335_ent::GPIOname::gpio_pwm_02);
118     pwm2.dir(true);
119     pwm2.fun_mux(0);
120     pwm2.clear();
121     Dsp28335_ent::GPIO pwm3(Dsp28335_ent::GPIOname::gpio_pwm_03);
122     pwm3.dir(true);
123     pwm3.fun_mux(0);
124     pwm3.clear();
125     Dsp28335_ent::GPIO pwm4(Dsp28335_ent::GPIOname::gpio_pwm_04);
126     pwm4.dir(true);
127     pwm4.fun_mux(0);
128     pwm4.clear();
129     Dsp28335_ent::GPIO pwm5(Dsp28335_ent::GPIOname::gpio_pwm_05);
130     pwm5.dir(true);
131     pwm5.fun_mux(0);
132     pwm5.clear();
133     Dsp28335_ent::GPIO pwm6(Dsp28335_ent::GPIOname::gpio_pwm_06);
134     pwm6.dir(true);
135     pwm6.fun_mux(0);
136     pwm6.clear();
137     Dsp28335_ent::GPIO pwm7(Dsp28335_ent::GPIOname::gpio_pwm_07);
138     pwm7.dir(true);
139     pwm7.fun_mux(0);
140     pwm7.clear();
```

```
141     Dsp28335_ent::GPIO pwm8(Dsp28335_ent::GPIOname::gpio_pwm_08);
142     pwm8.dir(true);
143     pwm8.fun_mux(0);
144     pwm8.clear();
145
146
147     //read the execution mode established by the user
148     ctrl.get_execution_mode();
149
150     ctrl.register_test(TestAddr::test_Address, "Test Address");
151
152     if(ctrl.mode == ctrl.burnin_mode )
153     {
154         ctrl.power_switch_ok = TestADC::check_power_switch();
155     }
156     else
157     {
158         ctrl.register_test(TestADC::test_power_an<ADC::adc08, 100, 19000>, "
    V1_SEN"); // (1000+18000)/1000 *1000
159         ctrl.register_test(TestADC::test_power_an<ADC::adc09, 100, 19000>, "
    V2_SEN");
160     }
161
162     ctrl.register_test(TestADC::test_power_an<ADC::adc00, 130, 2200>,
163             "AN1(3.3V) and 3.3V Regulator  (needs 3.3V to AN1(3.3V)"); //
    2*[(910+9100)/9100 *1000] -> voltage divider in CEX pcb * voltage divider
    verification PCB
164     ctrl.register_test(TestADC::test_power_an<ADC::adc01, 130, 2200>,
165             "AN2(3.3V) and 3.3V Regulator  (needs 3.3V to AN2(3.3V)");
166
167     ctrl.register_test(TestADC::test_power_an<ADC::adc02, 215, 3258>, // should
    be 2.15V
168             "AN3(5V) and 5V Regulator  (needs 5V to AN3(5V)"); // 2*[(6200+3900)
    /6200 *1000]
169     ctrl.register_test(TestADC::test_power_an<ADC::adc03, 0, 3258>,
170             "AN4(5V) and 5V Regulator  (needs 0.7V|2.15V to AN4(5V))");
171
172     ctrl.register_test(TestADC::test_power_an<ADC::adc04, 500, 4125>,
```

```
173             "AN5(12V) and 5V Regulator  (needs 5V to AN5(12V)"); // (2400+7500)
    /2400 *1000
174     ctrl.register_test(TestADC::test_power_an<ADC::adc05, 500, 4125>,
175             "AN6(12V) and 5V Regulator  (needs 5V to AN6(12V)");
176     ctrl.register_test(TestADC::test_power_an<ADC::adc06, 500, 12000>,
177             "AN7(36V) and 5V Regulator  (needs 5V to AN7(36V)"); // (1000+11000)
    /1000 *1000
178     ctrl.register_test(TestADC::test_power_an<ADC::adc07, 500, 12000>,
179             "AN8(36V) and 5V Regulator  (needs 5V to AN8(36V)");
180
181     if(ctrl.mode != ctrl.rs485_comm )
182     {
183         ctrl.register_test(TestSCI::testUARTB,  "UART B loopback"); //UART B (SCI
     B)
184     }
185
186     if(ctrl.mode == ctrl.rs485_comm )
187     {
188         ctrl.register_test(TestJETI::test,  "Test JETI");
189     }
190
191     ctrl.register_test(TestCAN::test,   "CAN A-B loopback"); //CAN A - CAN B
    loopback
192     ctrl.register_test(TestMMC5883MA::test, "MMC5883MA External"); //To verify
    I2C
193
194     ctrl.register_test(TestPWM::test_pwm_ecap<Pwmdev::epwm1, Pwmdev::epwm2, ECAP
    ::id_ecap1>,  "PWM1 and PWM2 to ECAP1");
195     ctrl.register_test(TestPWM::test_pwm_ecap<Pwmdev::epwm3, Pwmdev::epwm4, ECAP
    ::id_ecap2>,  "PWM3 and PWM4 to ECAP2");
196     ctrl.register_test(TestPWM::test_pwm_ecap<Pwmdev::epwm5, Pwmdev::epwm6, ECAP
    ::id_ecap3>,  "PWM5 and PWM6 to ECAP3");
197     ctrl.register_test(TestPWM::test_pwm_ecap<Pwmdev::epwm7, Pwmdev::epwm8, ECAP
    ::id_ecap4>,  "PWM7 and PWM8 to ECAP4");
198
199     ctrl.register_test(TestADC::test_temp<ADC::adc10>, "Temperature sensor test")
    ;
200
201
```

```
202     ctrl.register_op(AppWruAddress::run, 'w', "Write uaddress");
203     ctrl.register_op(LTTest::run, 'l', "Long Term Test");
204     ctrl.register_op(TestSCI::testUARTC, 'c', "UART C loopback"); //UART C (SCI C
    )
205
206     Uint16 burnin_runs = CEX::burnin_addr[0];
207
208     Base::Chrono chrono(true);
209     Real t = 0;
210     chrono.tic();
211
212     if (ctrl.mode == ctrl.burnin_mode)
213     {
214         Uint16 total_runs = 30;
215         {
216             Base::Mutex m(true);
217             Bsp::Flash_wr::write_sector(CEX::burnin_addr, &burnin_runs, sizeof(
    Base::Address0));
218         }
219         // Store the system address
220         Dsp28335_ent::Hsys::set_sysaddr(Address(burnin_runs));
221
222         //Wait for user to activate Burn In mode
223         Uint8 rx_buffer = 0;
224         log(Kstring("Press any key to start Burn In:\r\n"));
225         while (rx_buffer == 0)
226         {
227             CEX::Halsuite::get_instance().scia.read(rx_buffer);
228             log(Kstring("Press any key to start Burn In:\r\n"));
229             Delay::ms(60);
230
231         }
232
233         ctrl.run_all();
234         burnin_runs++;
235         {
236             Base::Mutex m(true);
237             Bsp::Flash_wr::write_sector(CEX::burnin_addr, &burnin_runs, sizeof(
    Base::Address0));
```

```
238         }
239
240         // Store the system address
241         Dsp28335_ent::Hsys::set_sysaddr(Address(burnin_runs));
242
243
244         //Print number of runs
245         static Base::Vstring str(48, Memmgr::external);
246         str.clear();
247         str.append_kstr(Kstring("\033[35m  >>> Number of Runs: "));
248         str.append_uint(burnin_runs+1);
249         str.append_kstr(Kstring(" <<<\033[0m"));
250         lognl(str);
251
252         if (burnin_runs < total_runs-1)
253         {
254             if (ctrl.failed == 0)
255             {
256                 t = 0;
257                 static Base::Vstring str(48, Memmgr::external);
258                 str.append_kstr(Kstring("\r\n\r\n"));
259                 str.append_kstr(Kstring("RESTARTING...\r\n"));
260                 str.append_kstr(Kstring("     ...\r\n"));
261                 str.append_kstr(Kstring("...\r\n"));
262                 log(str);
263
264                 Delay::ms(10);
265
266                 //Restart CEX
267
268                 pwm3.dir(true); // true -> output
269                 pwm3.fun_mux(0);
270
271                 pwm6.dir(true); // true -> output
272                 pwm6.fun_mux(0);
273
274                 pwm3.set();
275                 pwm6.set();
276
```

```
277                 t = chrono.toc();
278
279                 if(t > 120)
280                 {
281                     pwm4.dir(true); // true -> output
282                     pwm4.fun_mux(0);
283                     pwm5.dir(true); // true -> output
284                     pwm5.fun_mux(0);
285
286                     pwm4.set();
287                     pwm5.set();
288
289                 }
290                 while (1)
291                     ;
292
293             }
294             else
295             {
296                 burnin_runs = (unsigned int)(-1);
297                 {
298                     Base::Mutex m(true);
299                     Bsp::Flash_wr::write_sector(CEX::burnin_addr, &burnin_runs,
    sizeof(Base::Address0));
300                 }
301
302                 // Store the system address
303                 Dsp28335_ent::Hsys::set_sysaddr(Address(burnin_runs));
304
305                 log(Kstring("A test has failed! \r\n"));
306             }
307         }
308         else
309         {
310             Uint8 cont = 0;
311             while (cont < 10)
312             {
313                 log(Kstring("All runs completed! \r\n"));
314                 Delay::ms(10);
```

```
315                 cont++;
316             }
317 
318             burnin_runs = (unsigned int)(-1);
319             {
320                 Base::Mutex m(true);
321                 Bsp::Flash_wr::write_sector(CEX::burnin_addr, &burnin_runs,
    sizeof(Base::Address0));
322             }
323 
324             // Store the system address
325             Dsp28335_ent::Hsys::set_sysaddr(Address(burnin_runs));
326         }
327     }
328     else
329     {
330         //If slow_mode or fast_mode, delete burn-in file.
331         burnin_runs = (unsigned int)(-1);
332         {
333             Base::Mutex m(true);
334             Bsp::Flash_wr::write_sector(CEX::burnin_addr, &burnin_runs, sizeof(
    Base::Address0));
335         }
336 
337        // Store the system address
338        Dsp28335_ent::Hsys::set_sysaddr(Address(burnin_runs));
339 
340     }
341 
342 
343     for (;;)
344     {
345         ctrl.step();
346     }
347 }
```

Listing F.1: main.cpp

F.2. SerialTestCtrl

F.2.1. SerialTestCtrl.h

```
#ifndef INCLUDE_SERIALTESTCTRL_H_
#define INCLUDE_SERIALTESTCTRL_H_

#include <entypes.h>
#include <Array.h>
#include <Istep.h>
#include <Vstring.h>
#include <Chrono.h>
#include <Itport.h>
#include <Arrayu8.h>

namespace Base
{
    class Vstring;
}

namespace PETT
{
    typedef bool (*Test)();
    typedef bool (*Operation)();

    class SerialTestCtrl : public Base::Istep
    {
    public:
        enum Mode
        {
            rs485_comm,
            slow_mode,
            burnin_mode
        };
        Mode mode;

        int32 ran;
        int32 failed;
```

```
35
36          bool power_switch_ok;
37
38          static SerialTestCtrl& get_instance();
39
40          static void set_addr(const Uint16& addr);
41
42          void step();
43
44          void get_execution_mode();
45
46          void register_test(Test test, const char* descr);
47
48          bool register_op(Operation op, char c, const char* descr); // return
    false if selection char was already used
49
50          void run_all();
51
52          bool read(Uint8& rx_buffer);
53          static bool write(Uint8 rx_buffer);
54          static bool wr_available();
55
56          void print_kstring(const Base::Kstring& str);
57
58      private:
59          static bool run_tests();
60
61          static Uint16* addr;
62
63          SerialTestCtrl();
64
65          enum State
66          {
67              s_disconnected,
68              s_connected,
69              s_waiting,          //Connected and waiting for input
70              s_data_available,
71              s_processing,
72              s_auto_test_finished
```

```
73          };
74
75
76          static const Uint16 max_tests = 50;
77          static const Uint16 max_ops = 16;
78
79          //SCI comms
80          Base::Arrayu8 rx_buffer;
81
82          State state;
83
84          struct Testdata
85          {
86              enum Test_state
87              {
88                  pass,
89                  fail,
90                  unexecuted
91              };
92
93              Test func;
94              const char* descr;
95              Test_state test_state;
96              Testdata() : func(0), descr("")
97              {}
98              Testdata(Test test0, const char* descr0, Test_state test_state0) :
    func(test0), descr(descr0), test_state(test_state0)
99              {}
100         };
101         Testdata test_dt;
102         Base::Array<Testdata> tests_slow;
103
104         bool tests_ok;
105
106         struct Opdata
107         {
108             Operation func;
109             char sel_char;
110             const char* descr;
```

```
111             Opdata() : func(0), sel_char('0'), descr("")
112             {}
113             Opdata(Operation op0, char c0, const char* descr0) : func(op0),
    sel_char(c0), descr(descr0)
114             {}
115         };
116         Base::Array<Opdata> ops;
117 
118         Base::Chrono cr;
119 
120         bool get_state();
121         Uint16 get_size();
122         Testdata& get_test(Uint16 idx);
123 
124         bool run();
125         void print_ops();
126         void print_main();
127         void print_results(Uint16 idx, bool res);
128         void print_results_total(Uint32 ran, Uint32 failed);
129         bool process(Uint16 idx);
130         bool run_op(int16 idx);
131 
132 
133         int16 op_ind(char c);   // Returns the operation index for given char or
    -1 if not found
134 
135         char to_lower(char c);
136         bool is_num(char c);
137         Uint16 atoi(Base::Array<Uint8>& str);
138 
139     };
140 }
141 #endif
```

Listing F.2: SerialTestCtrl.h

F.2.2. SerialTestCtrl.cpp

```
1 #include <Memmgr.h>
2 #include <Kstring.h>
```

```cpp
#include <Halsuite.h>
#include <Arrayu8.h>
#include <Logger.h>
#include <Address0.h>
#include <Warning.h>
#include <GPIO.h>
#include <SerialTestCtrl.h>
#include <Delay.h>
#include <ADC.h>

using namespace Dsp28335_ent;

#define MAJOR "1"
#define MINOR "0"

namespace PETT
{

    Uint16* SerialTestCtrl::addr;


    void SerialTestCtrl::set_addr(const Uint16& addr0)
    {
        addr = const_cast<Uint16*>(&addr0);
    }

    SerialTestCtrl& SerialTestCtrl::get_instance()
    {
        static SerialTestCtrl testctrl;
        return testctrl;
    }

    bool SerialTestCtrl::read(Uint8& rx_buffer)
    {
        Base::Itport_u8* sci = &CEX::Halsuite::get_instance().scia;
        if(SerialTestCtrl::get_instance().mode == SerialTestCtrl::rs485_comm)
        {
            sci = &CEX::Halsuite::get_instance().scib;
        }
```

```
42          return sci->read(rx_buffer);
43      }
44
45      bool SerialTestCtrl::write(Uint8 rx_buffer)
46      {
47          Base::Itport_u8* sci = &CEX::Halsuite::get_instance().scia;
48          if(SerialTestCtrl::get_instance().mode == SerialTestCtrl::rs485_comm)
49          {
50              sci = &CEX::Halsuite::get_instance().scib;
51          }
52          return sci->write(rx_buffer);
53      }
54
55      bool SerialTestCtrl::wr_available()
56      {
57          Base::Itport_u8* sci = &CEX::Halsuite::get_instance().scia;
58          if(SerialTestCtrl::get_instance().mode == SerialTestCtrl::rs485_comm)
59          {
60              sci = &CEX::Halsuite::get_instance().scib;
61          }
62          return sci->wr_available();
63      }
64
65      bool SerialTestCtrl::run_tests()
66      {
67          return SerialTestCtrl::get_instance().run();
68      }
69          static const Uint16 sz8 = 300;
70
71      SerialTestCtrl::SerialTestCtrl() :
72          rx_buffer(sz8, Memmgr::external),
73          state(s_connected),
74          tests_slow(max_tests, Memmgr::external),
75          tests_ok(false),
76          ops(max_ops, Memmgr::external),
77          cr(true),
78          ran(0),
79          failed(0)
80      {
```

```
81          tests_slow.resize(0);
82          ops.resize(0);
83          rx_buffer.resize(0);
84
85          register_test(run_tests, "Run all tests");
86
87          cr.tic();
88      }
89
90      void SerialTestCtrl::get_execution_mode()
91      {
92          Real factor = 2.0F*(10.1F/6.2F); //Voltage divider in CEX PCB * voltage
    divider verification PCB
93          Dsp28335_ent::ADC::ADCchannel adc_id = Dsp28335_ent::ADC::adc03; //CEX
    AN4(5V) -> adc03
94
95          CEX::Halsuite& hal = CEX::Halsuite::get_instance();
96          Dsp28335_ent::ADC& adc = hal.adc;
97
98          adc.init(Dsp28335_ent::ADC::external_2048);//external 2V...
99
100         Base::Chrono chrono(true);
101         Base::Chrono cr_aux(true);
102
103         Real t = 0;
104         Uint32 sampled = 0;
105
106         Dsp28335_ent::Delay::ms(500); // Give time to stabilize ADC reading
107
108         cr_aux.tic();
109         chrono.tic();
110         Real mtot = 0;
111         Real mtot_conv = 0;
112
113         while(t<0.5)
114         {
115             if(cr_aux.toc()>0.005)
116             {
117                 cr_aux.tic();
```

```
118                 sampled++;
119 
120                 Real measured = (Real)adc.get_sample_volts(adc_id); //To voltage
    (4095 -> 3 V)
121                 mtot += measured;
122             }
123             t = chrono.toc();
124         }
125 
126         if(sampled>0)
127         {
128             mtot /= sampled;
129             mtot_conv = mtot*factor; // Voltage after voltage divider
130         }
131         else
132         {
133             mtot = 0;
134             mtot_conv = 0;
135         }
136 
137         logn1("\033[37m
    ------------------------------------------------------------");
138         log("|      \033[36mMeasured: \033[0m");
139         log("\033[34m");
140         log(mtot_conv, 2);
141         log("\033[34mV        \033[37m|        Mode ->");
142 
143         //Set mode
144         if(fabs(mtot_conv-2.15) < 0.25)
145         {
146             mode = rs485_comm; //2.15V -> both switch off
147             log("\033[34mRS485Comm           \033[37m|");
148         }
149         else if(fabs(mtot_conv-1.60) < 0.25)
150         {
151             mode = slow_mode; //1.6V -> 1 ON 2 OFF
152             log("\033[34mSlow           \033[37m|");
153 
154         }
```

```
155         else
156         {
157             mode = burnin_mode;//0.5V -> 2 ON 1 OFF and both ON
158             log("\033[34mBurn In  \033[37m      |");
159         }
160         lognl();
161         log("\033[37m
    ----------------------------------------------------------------");
162         lognl();
163     }
164
165     void SerialTestCtrl::register_test(Test test, const char* descr)
166     {
167         if(get_size() >= max_tests)
168         {
169             Bsp::warning();
170             return;
171         }
172         else
173         {
174             tests_slow.append(Testdata(test, descr, Testdata::unexecuted));
175         }
176     }
177
178         bool SerialTestCtrl::register_op(Operation op, char c, const char* descr)
179     {
180         if(!ops.append(Opdata(op, c, descr)))
181         {
182             Bsp::warning();
183         }
184         return true;
185     }
186
187     char SerialTestCtrl::to_lower(char c)
188     {
189         const char tr = 'a'-'A';
190         if(c>='A' && c<='Z')
191         {
192             return c+tr;
```

```
193         }
194         return c;
195     }
196 
197     int16 SerialTestCtrl::op_ind(char c)
198     {
199         Uint32 sz = ops.size();
200         c = to_lower(c);
201         for(Uint32 i=0; i<sz; i++)
202         {
203             if(c == ops[i].sel_char)
204             {
205                 return i;
206             }
207         }
208         return -1;
209     }
210 
211     void SerialTestCtrl::print_ops()
212     {
213         Uint32 sz = ops.size();
214         for(Uint32 i=0; i<sz; i++)
215         {
216             log("\033[37m ");
217             log(ops[i].sel_char);
218             log("\033[37m) ");
219             log(ops[i].descr);
220             lognl();
221 
222         }
223     }
224 
225     void SerialTestCtrl::print_kstring(const Base::Kstring& str)
226     {
227         CEX::Halsuite& hal = CEX::Halsuite::get_instance();
228         for(Uint16 i = 0; i < str.length(); i++)
229         {
230             while(!wr_available())
231             {
```

```
232                 hal.poll();
233             }
234             write(str.char_at(i));
235         }
236     }
237
238     void SerialTestCtrl::print_main()
239     {
240         //To show uaddres
241         Uint32 *eff_addr = reinterpret_cast<Uint32*>(addr);
242         if( (unsigned(*eff_addr) < Base::Address0::addr_cex_first) || (unsigned(*
    eff_addr) > Base::Address0::addr_cex_last) )
243         {
244             log("\033[37mNo uaddress. ");
245             logn();
246         }
247         else
248         {
249             log("\033[37mUaddress: ");
250             log(unsigned(*eff_addr));
251             logn();
252         }
253
254         log("\033[36mCEX Verification v");
255         log(MAJOR);
256         log(".");
257         log(MINOR);
258         logn("\033[0m");
259         print_ops();
260         Uint32 sz = get_size();
261         for(Uint32 i=0; i<sz; i++)
262         {
263             if(i<10)
264             {
265                 log(" ");
266             }
267             log(i);
268             log(") ");
269
```

```
270             if(get_test(i).test_state == test_dt.pass)
271             {
272                 log("\033[32m PASS \033[0m");
273             }
274             else if(get_test(i).test_state == test_dt.fail)
275             {
276                 log("\033[31m FAIL \033[0m");
277             }
278             else
279             {
280                 log("\033[36m UN-EXE \033[0m");
281             }
282 
283             log("\033[37m - ");
284             log(get_test(i).descr);
285             lognl();
286         }
287         lognl();
288     }
289 
290     void SerialTestCtrl::print_results(Uint16 idx, bool res)
291     {
292 
293         log(" Test #");
294         log(idx);
295         lognl();
296         if(res)
297         {
298            log("\033[32m PASS \033[0m");
299         }
300         else
301         {
302             log("\033[31m FAIL \033[0m");
303         }
304         log("\033[37m (");
305         log(get_test(idx).descr);
306         log("\033[37m) ");
307         lognl();
308     }
```

```
309
310     void SerialTestCtrl::print_results_total(Uint32 ran, Uint32 failed)
311     {
312
313         log("\033[37m Test Ran: ");
314         log(ran);
315         log("\033[37m Failed: ");
316         log(failed);
317         log("\033[37m (");
318         log(get_test(0).descr);
319         log("\033[37m)");
320         logn1();
321     }
322
323     bool SerialTestCtrl::get_state()
324     {
325         bool ret = true;
326         Uint32 sz = get_size();
327         for(int i = 0; i < sz && ret == true; i++)
328         {
329             if(get_test(i).test_state == test_dt.fail)
330             {
331                 ret = false;
332             }
333         }
334         return ret;
335     }
336
337     Uint16 SerialTestCtrl::get_size()
338     {
339         return tests_slow.size();
340     }
341
342     SerialTestCtrl::Testdata& SerialTestCtrl::get_test(Uint16 idx)
343     {
344         if (idx > get_size())
345         {
346             Bsp::warning();
347         }
```

```
348         return tests_slow[idx];
349 
350     }
351 
352     bool SerialTestCtrl::run()
353     {
354         CEX::Halsuite& hal = CEX::Halsuite::get_instance();
355         hal.led_red.set();
356         hal.led_green.clear();
357         Uint32 sz = get_size();
358         for(Uint8 idx=1; idx<sz; idx++)
359         {
360             bool res = process(idx);
361             print_results(idx, res);
362 
363             hal.led_red.toggle();
364             hal.led_green.toggle();
365         }
366         return (failed == 0);
367     }
368 
369     void SerialTestCtrl::run_all()
370     {
371         ran = 0;
372         failed = 0;
373         run(); //Run all tests
374         print_results_total(ran, failed);
375     }
376 
377     bool SerialTestCtrl::process(Uint16 idx)
378     {
379         //initialize GPIOs as inputs before executing test
380         SerialTestCtrl::Testdata& td = get_test(idx);
381         Dsp28335_ent::GPIO pwm1(Dsp28335_ent::GPIOname::gpio_pwm_01);
382         pwm1.dir(false);
383         pwm1.fun_mux(0);
384         Dsp28335_ent::GPIO pwm8(Dsp28335_ent::GPIOname::gpio_pwm_08);
385         pwm8.dir(false);
386         pwm8.fun_mux(0);
```

```
387
388         bool res = td.func();
389
390         if(res)
391         {
392             td.test_state = test_dt.pass;
393         }
394         else
395         {
396             td.test_state = test_dt.fail;
397         }
398
399         if(idx!=0)
400         {
401             ran++;
402             failed += !res;
403         }
404
405         if(res && get_state())
406         {
407             //GREEN LED
408            pwm8.set();
409            pwm1.clear();
410         }
411         else
412         {
413             //RED LED
414             pwm1.set();
415             pwm8.clear();
416         }
417
418         Dsp28335_ent::Delay::ms(200);
419
420         return res;
421     }
422
423     bool SerialTestCtrl::run_op(int16 idx)
424     {
425         if(idx>=ops.size() || ops[idx].func==0)
```

```
426         {
427             Bsp::warning();
428             return false;
429         }
430         bool res = ops[idx].func();
431         if(idx!=0)
432         {
433             ran++;
434             failed += !res;
435         }
436         return res;
437     }
438 
439     bool SerialTestCtrl::is_num(char c)
440     {
441         return (c>='0' && c<='9');
442     }
443 
444     Uint16 SerialTestCtrl::atoi(Base::Array<Uint8>& str)
445     {
446         Uint16 ret = 0;
447 
448         if(!is_num(str[0]))
449         {
450             ret = 0xFFFF;   //Return not valid value
451         }
452         else
453         {
454             for(Uint16 i = 0; i < str.size(); i++)
455             {
456                 if(is_num(str[i]))
457                 {
458                     ret *= 10;
459                     ret += str[i] - '0';
460                 }
461                 else
462                 {
463                     break;
464                 }
```

```
465             }
466         }
467 
468         return ret;
469     }
470 
471 
472     void SerialTestCtrl::step()
473     {
474         static CEX::Halsuite& hal = CEX::Halsuite::get_instance();
475         hal.poll();
476         switch(state)
477         {
478         case s_disconnected:
479             break;
480         case s_connected:
481             cr.cancel();
482             print_main();
483             state = s_waiting;
484             break;
485         case s_waiting:
486             if(read(rx_buffer[0]))
487             {
488                 state = s_data_available;
489                 rx_buffer.resize(1);
490             }
491             break;
492         case s_data_available:
493             if(!cr.ongoing())
494             {
495                 cr.tic();
496             }
497             if(cr.toc()>0.3)    //Wait a little for numbers with more than one
    char
498             {
499                 cr.cancel();
500                 Uint16 i = 1;
501                 while(read(rx_buffer[i]))
502                 {
```

```
503                     if(i > rx_buffer.size_max())
504                     {
505                         Bsp::warning();
506                         break;
507                     }
508                     i++;
509                     rx_buffer.resize(i);
510 
511                 }
512                 if(rx_buffer.size()>0)
513                 {
514                     state = s_processing;
515                     int16 idx = 0xFFFF;
516                     if(rx_buffer.size()==1 && (idx=op_ind(rx_buffer[0]))!=-1)
517                     {
518                         run_op(idx);
519                         lognl("\033[37mApp ended");
520                     }
521                     else if(is_num(rx_buffer[0]))
522                     {
523                         idx = atoi(rx_buffer);
524                         if(idx < get_size())
525                         {
526                             log((Uint32)idx);
527                             lognl();
528                             ran = 0;
529                             failed = 0;
530                             tests_ok = process(idx);
531                             if(idx==0)
532                             {
533                                 print_results_total(ran, failed);
534                             }
535                             else
536                             {
537                                 print_results(idx, tests_ok);
538                             }
539                         }
540                     }
541                     else
```

```
                    {
                        print_main();
                    }
                }
            }
            break;
        case s_processing:
            if(wr_available())
            {
                //Ignore pending input
                state = s_waiting;
            }
            break;
        case s_auto_test_finished:
            {
                if(!cr.ongoing())
                {
                    cr.tic();
                }

                if(cr.toc()>0.1)
                {
                    cr.tic();
                    if(tests_ok)
                    {
                        hal.led_red.set();
                        hal.led_green.toggle();
                    }
                    else
                    {
                        hal.led_red.toggle();
                        hal.led_green.set();
                    }
                }
            }
            break;
        }
    }
```

```
580 }
```

Listing F.3: SerialTestCtrl.cpp

F.3. TestAddr

F.3.1. TestAddr.h

```
1  #ifndef INCLUDE_TESTADDR_H_
2  #define INCLUDE_TESTADDR_H_
3
4  namespace PETT
5  {
6      class TestAddr
7      {
8      public:
9          TestAddr();
10
11         static void get_addr(const Uint16& addr);
12
13         static bool test_Address();
14
15     private:
16         static Uint16* addr;
17
18         bool run();
19
20     };
21
22 }
23
24 #endif /* INCLUDE_TESTADDR_H_ */
```

Listing F.4: TestAddr.h

F.3.2. TestAddr.cpp

```
1 #include <Memmgr.h>
2 #include <Kstring.h>
3 #include <Halsuite.h>
```

```
#include <Arrayu8.h>
#include <Logger.h>
#include <Strutil.h>
#include <Address0.h>
#include <TestAddr.h>
#include <Delay.h>


namespace PETT
{

    Uint16* TestAddr::addr;
    void TestAddr::get_addr(const Uint16& addr0)
    {
        addr = const_cast<Uint16*>(&addr0);
    }

    bool TestAddr::test_Address()
    {
        TestAddr inst;
        return inst.run();
    }

    TestAddr::TestAddr()
    {
    }

    bool TestAddr::run()
    {
        bool ret = false;
        Uint32 *eff_addr = reinterpret_cast<Uint32*>(addr);

        if( (unsigned(*eff_addr) < Base::Address0::addr_cex_first) || (unsigned(*
    eff_addr) > Base::Address0::addr_cex_last) )
        {
            log("No uaddress. ");
            *eff_addr = Base::Address0::addr_uav_unknown; // If address is not
    valid, use "unknown"
            lognl();
```

```
41          }
42          else
43          {
44              log("  Address found: ");
45              log(unsigned(*eff_addr));
46              lognl();
47              ret = true;
48          }
49
50          return ret;
51
52      }
53  }
```

Listing F.5: TestAddr.cpp

F.4. TestADC

F.4.1. TestADC.h

```
1  #ifndef TestADC_H_
2  #define TestADC_H_
3
4  #include <entypes.h>
5  #include <GPIOsuite.h>
6  #include <ADC.h>
7  #include <Pwmdev.h>
8  #include <ADCsuite.h>
9
10 namespace PETT
11 {
12     class TestADC
13     {
14     public:
15         TestADC();
16
17         template<Dsp28335_ent::ADC::ADCchannel ADC, Uint16 IN, Uint16 FACTOR>
18         static bool test_power_an();
19         template<Dsp28335_ent::ADC::ADCchannel ADC>
```

```
20          static bool test_temp();
21          static bool check_power_switch();
22
23      private:
24
25          Real get_error(Real& value, Real mtot_conv);
26          bool run(Dsp28335_ent::ADC::ADCchannel adc_id, Real value, Real factor);
27          bool runtemp(Dsp28335_ent::ADC::ADCchannel adc_id);
28      };
29
30      template<Dsp28335_ent::ADC::ADCchannel ADC, Uint16 IN, Uint16 FACTOR>
31      bool TestADC::test_power_an()
32      {
33          Real in_real;
34          in_real = static_cast<Real>(IN)*0.01F;
35          Real factor;
36          factor = static_cast<Real>(FACTOR)*0.001F;
37
38          TestADC inst;
39          bool res = inst.run(ADC, in_real, factor);
40          return res;
41      }
42
43      template<Dsp28335_ent::ADC::ADCchannel ADC>
44      bool TestADC::test_temp()
45      {
46          TestADC inst;
47          bool res = inst.runtemp(ADC);
48          return res;
49      }
50
51  }
52  #endif
```

Listing F.6: TestADC.h

F.4.2. TestADC.cpp

```
1  #include <TestADC.h>
2  #include <Halsuite.h>
```

```
#include <ADC.h>
#include <Chrono.h>
#include <Delay.h>
#include <Logger.h>
#include <Math.h>
#include <SerialTestCtrl.h>

using namespace Dsp28335_ent;

namespace PETT
{
    TestADC::TestADC()
    {
    }


    Real TestADC::get_error(Real& value, Real mtot_conv)
    {
        Real percent_error = 0;

        if(value == 0) //Variable AN4(5V)
        {
            if(SerialTestCtrl::get_instance().mode == SerialTestCtrl::
   get_instance().rs485_comm)
            {
                value = 2.15;

            }
            else if(SerialTestCtrl::get_instance().mode == SerialTestCtrl::
   get_instance().burnin_mode)
            {
                value = 0.70;
            }
            else
            {
                value = 1.60;
            }
            percent_error = fabs(value-mtot_conv)/mtot_conv;
        }
```

```
40          else if(value == 1) //Variable Power Supply
41          {
42              if(fabs(mtot_conv - 7) < 1.5)
43              {
44                  value = 7;
45                  percent_error = fabs(value-mtot_conv)/mtot_conv;
46              }
47              else if(fabs(mtot_conv - 12) < 1.5)
48              {
49                  value = 12;
50                  percent_error = fabs(value-mtot_conv)/mtot_conv;
51              }
52              else if(fabs(mtot_conv - 36) < 1.5)
53              {
54                  value = 36;
55                  percent_error = fabs(value-mtot_conv)/mtot_conv;
56              }
57              else
58              {
59                  log("  Read value is not 7V | 12V | 36V");
60                  lognl();
61                  value = 0;
62                  percent_error = 1;
63              }
64          }
65
66          return percent_error;
67      }
68
69
70      bool TestADC::run(Dsp28335_ent::ADC::ADCchannel adc_id, Real value, Real
    factor)
71      {
72          bool ret = false;
73          CEX::Halsuite& hal = CEX::Halsuite::get_instance();
74          Dsp28335_ent::ADC& adc = hal.adc;
75
76          adc.init(Dsp28335_ent::ADC::external_2048);
77
```

```
78          Base::Chrono chrono(true);
79          Base::Chrono cr_aux(true);
80
81          Real t = 0;
82          Uint32 sampled = 0;
83
84          Delay::ms(500); // Give time to stabilize ADC reading
85
86          cr_aux.tic();
87          chrono.tic();
88          Real mtot = 0;
89          Real mtot_conv = 0;
90
91          while(t<0.5)
92          {
93              if(cr_aux.toc()>0.005)
94              {
95                  cr_aux.tic();
96                  sampled++;
97
98                  Real measured = (Real)adc.get_sample_volts(adc_id);
99                  mtot += measured;
100             }
101             t = chrono.toc();
102         }
103
104         if(sampled>0)
105         {
106             mtot /= sampled;
107             mtot_conv = mtot*factor;
108         }
109         else
110         {
111             mtot = 0;
112             mtot_conv = 0;
113         }
114
115         Real percent_error = fabs(value-mtot_conv)/mtot_conv;
116
```

```
117         if(value == 0 || value == 1)
118         {
119             percent_error = get_error(value, mtot_conv); //If expected value is
    variable
120         }
121 
122         log("  Value expected: ");
123         log(value, 2);
124         log("  |  Measured: ");
125         log(mtot_conv, 2);
126         log("  |  % error: ");
127         log(100.0F*percent_error, 2);
128         log("%");
129         lognl();
130 
131 
132         if(sampled == 0 ||  percent_error > 0.15F)
133         {
134             ret = false;
135         }
136 
137         else
138         {
139             ret = true;
140         }
141 
142 
143         return ret;
144     }
145 
146 
147     bool TestADC::runtemp(Dsp28335_ent::ADC::ADCchannel adc_id)
148     {
149         bool ret = false;
150         CEX::ADCsuite adc;
151 
152         Real CEXtemperature = adc.get_value(adc_id);
153         log("  Measured: ");
154         log(CEXtemperature, 2);
```

```
155         log(" K - ");
156         log(CEXtemperature-  Const::C2KELVIN, 2);
157         log(" C");
158         lognl();
159 
160         if (CEXtemperature<370 && CEXtemperature>290)
161         {
162             ret = true;
163         }
164         return ret;
165     }
166 
167 
168     bool TestADC::check_power_switch()
169     {
170         Uint16 ok = 0;
171         lognl("Checking Vin: ");
172         log("V1: ");
173         ok = test_power_an<ADC::adc08, 100, 19000>() ? ok+1 : ok;
174         log("V2: ");
175         ok = test_power_an<ADC::adc09, 100, 19000>() ? ok+1 : ok;
176 
177 
178         //In Burn In mode, switch between V1 and V2
179         return ok >= 1;
180     }
181 }
```

Listing F.7: TestADC.cpp

F.5. TestSCI

F.5.1. TestSCI.h

```
1 #ifndef INCLUDE_TestSCI_H_
2 #define INCLUDE_TestSCI_H_
3 
4 #include <entypes.h>
5 #include <Halsuite.h>
```

```
#include <Arrayu8.h>

namespace PETT
{
    class TestSCI
    {
    public:
        explicit TestSCI(Base::Itport_u8& sci0);

        static bool testUARTB();
        static bool testUARTC();

    private:
        TestSCI(const TestSCI& obj);

        Base::Itport_u8& sci;
        static const Uint16 tx_sz = 255;
        Base::Arrayu8 tx_buffer;
        Base::Arrayu8 rx_buffer;

        bool run();
    };

}

#endif /* INCLUDE_TestSCI_H_ */
```

Listing F.8: TestSCI.h

F.5.2. TestSCI.cpp

```
#include <TestSCI.h>
#include <Itport.h>
#include <Halsuite.h>
#include <SCI.h>
#include <SCIcfg.h>
#include <Chrono.h>
#include <Logger.h>
#include <Vstring.h>
#include <Strutil.h>
```

```
#include <GPIOmux4.h>
#include <GPIOsuite.h>
#include <Delay.h>

using namespace Dsp28335_ent;
using namespace Base;
using Base::Kstring;

namespace PETT
{

    bool TestSCI::testUARTB()
    {
        static TestSCI inst(CEX::Halsuite::get_instance().scib);
        CEX::Halsuite::get_instance().scib_hw.config(Base::SCIcfg::build(115200UL
    , Base::SCIcfg::len_8, Base::SCIcfg::stp_1, Base::SCIcfg::par_dis, 0));
        return inst.run();
    }

    bool TestSCI::testUARTC()
    {
        Dsp28335_ent::GPIO gpio_sda(Dsp28335_ent::GPIOname::gpio_i2ca_sda);
        gpio_sda.dir(false); // false -> input
        Dsp28335_ent::GPIO gpio_scl(Dsp28335_ent::GPIOname::gpio_i2ca_scl);
        gpio_scl.dir(false);
        gpio_sda.fun_mux(0); //Pin as GPIO
        gpio_scl.fun_mux(0);


        static TestSCI inst(CEX::Halsuite::get_instance().scic);
        CEX::Halsuite::get_instance().scic_hw.config(Base::SCIcfg::build(115200UL
    , Base::SCIcfg::len_8, Base::SCIcfg::stp_1, Base::SCIcfg::par_dis, 0));
        return inst.run();

    }

    TestSCI::TestSCI(Base::Itport_u8& sci0) :
        sci(sci0),
        tx_buffer(tx_sz, Memmgr::external),
```

```
47          rx_buffer(tx_sz, Memmgr::external)
48      {
49          for(Uint16 i=0;i<tx_sz;i++)
50          {
51              tx_buffer[i] = i;
52          }
53      }
54
55
56      bool TestSCI::run()
57      {
58          Base::Chrono chrono(true);
59          Uint8 v;
60
61          CEX::Halsuite& hal = CEX::Halsuite::get_instance();
62
63          //Clear possible data in buffers
64          while(sci.read(v));
65
66
67          bool ok = true;
68
69          chrono.tic();
70          Real t = chrono.toc();
71
72          Uint16 read = 0;
73          Uint16 read_ok = 0;
74          static Real tout = 0.5F;
75          for(Uint8 i=0;i<tx_sz;i++)
76          {
77              while(!sci.wr_available());
78              ok &= sci.write(tx_buffer[i]);
79              hal.poll();
80              while(!sci.read(rx_buffer[i]) && t < tout)
81                  t = chrono.toc();
82              if(tx_buffer[i] == rx_buffer[i])
83              {
84                  read_ok++;
85              }
```

```
86              else
87              {
88                  ok = false;
89              }
90              read++;
91          }
92
93          if(!ok)
94          {
95              lognl("Unable to send data");
96              return false;
97          }
98
99          if(!ok || read != tx_sz)
100         {
101             log("Read: ");
102             lognl(read);
103             log(" Read OK: ");
104             lognl(read_ok);
105
106             return false;
107         }
108         else
109         {
110             log("Read: ");
111             log(read);
112             lognl();
113             log("Read OK:");
114             log(read_ok);
115             lognl();
116             return true;
117         }
118     }
119
120 }
```

Listing F.9: TestSCI.cpp

F.6. TestCAN

F.6.1. TestCAN.h

```
#ifndef TESTCAN_VAL_H_
#define TESTCAN_VAL_H_

#include <Vstring.h>
#include <CANframe.h>

namespace PETT
{
    class TestCAN
    {
    public:
        TestCAN();

        static bool test();

    private:
        void print_canframe(const Base::CANframe& fr);

        bool run();
    };

}

#endif // TESTCAN_H_
```

Listing F.10: TestCAN.h

F.6.2. TestCAN.cpp

```
#include <CANcfg.h>
#include <CAN.h>
#include <Halsuite.h>
#include <Chrono.h>
#include <Fifospsc.h>
#include <Ttransfer.h>
```

```
#include <Logger.h>
#include <Strutil.h>
#include <TestCAN.h>

using namespace Dsp28335_ent;

namespace PETT
{
    using Base::CANid;
    using Base::CANframe;

    bool TestCAN::test()
    {
        TestCAN inst;
        return inst.run();
    }

    TestCAN::TestCAN()
    {
    }


    void TestCAN::print_canframe(const CANframe& fr)
    {
        for(Uint16 i=0; i<fr.data.length;i++)
        {
            log(Uint8(fr.data[i]));
            log_c(' ');
        }
    }

    bool TestCAN::run()
    {
        static const Uint32 id_a2b = 1234;
        static const Uint32 id_b2a = 1235;
        CEX::Halsuite& hal = CEX::Halsuite::get_instance();

        CANcfg cfg_a;
        cfg_a.br=1000000;
```

```
46          cfg_a.rx.resize(1);
47          cfg_a.rx[0].sz=16;
48          cfg_a.rx[0].flt.id.extended=false;
49          cfg_a.rx[0].flt.id.id=id_b2a;
50          cfg_a.rx[0].flt.msk = 0xFFFFFFFF;
51          hal.cana.config(cfg_a);
52
53          CANcfg cfg_b;
54          cfg_b.br=1000000;
55          cfg_b.rx.resize(1);
56          cfg_b.rx[0].sz=16;
57          cfg_b.rx[0].flt.id.extended=false;
58          cfg_b.rx[0].flt.id.id=id_a2b;
59          cfg_b.rx[0].flt.msk = 0xFFFFFFFF;
60          hal.canb.config(cfg_b);
61
62          CANframe frame_atx = {{false, id_a2b},{8,{10,11,12,13,14,15,16,17}}}; //
    {CAN id},{CAN data}. Can id: bool extended, id. CAN data: length, data
63          CANframe frame_btx = {{false, id_b2a},{8,{20,21,22,23,24,25,26,27}}};
64
65          CANframe frame_arx1 = {{false, 0},{0,{0,0,0,0,0,0,0,0}}};
66          CANframe frame_brx1 = frame_arx1;
67          CANframe frame_arx2 = frame_arx1;
68          CANframe frame_brx2 = frame_arx1;
69
70          //clear buffers
71          while(hal.cana.read(frame_arx1))
72              ;
73          while(hal.canb.read(frame_brx1))
74              ;
75
76          static const Real period = 0.25;
77          Base::Chrono clk(true);
78          clk.tic();
79
80          hal.cana.write(frame_atx);
81          hal.canb.write(frame_btx);
82
83          while(clk.toc()<period)
```

```
84          {
85          }
86
87          bool ok = true;
88
89          if(!hal.cana.read(frame_arx1)               // must read 1 from a
90                  || !hal.canb.read(frame_brx1)       // must read 1 from b
91                  || hal.cana.read(frame_arx2)        // must not read 2nd from a
92                  || hal.canb.read(frame_brx2)        // must not read 2nd from b
93                  || !frame_atx.equals(frame_brx1)     // txa must match rxb
94                  || !frame_btx.equals(frame_arx1))    // txb must match rxa
95          {
96              ok = false;
97          }
98
99          log("  CAN-A read: ");
100         print_canframe(frame_arx1);
101         lognl();
102
103         log("  CAN-B read: ");
104         print_canframe(frame_brx1);
105         lognl();
106
107         return ok;
108     }
109 }
```

Listing F.11: TestCAN.cpp

F.7. TestMMC5883MA (Test I2C)

F.7.1. TestMMC5883MA.h

```
1 #ifndef TESTMMC5883MA_H_
2 #define TESTMMC5883MA_H_
3
4 #include <Mmc5883ma.h>
5
6 namespace PETT
```

```
{
    class TestMMC5883MA
    {
    public:
        TestMMC5883MA();
        static bool test();

    private:

        Maverick::Rvector3      raw_v;
        Base::Hmeas3            m_mag;
        Base::Tmeas3D           m_mag4; ///< Sensor calibration for magnetometer
    4.
        volatile bool              bit;//
        Devices::Rawmea3        rawmag4;   ///< Raw measurements from
    magnetometer 4
        volatile Real           mag4temp;    ///< Temperature from magnetometer 4
        Devices::Simdev3D    mag4_int;
        Base::Xcal3D::Type_podsync cal_mag;
        volatile Real              x;
        volatile Real              y;
        volatile Real              z;
        Devices::Mmc5883ma dev;

        bool run();
    };
}
#endif
```

Listing F.12: TestMMC5883MA.h

F.7.2. TestMMC5883MA.cpp

```
#include <TestMMC5883MA.h>
#include <Halsuite.h>
#include <Chrono.h>
#include <Vstring.h>
#include <Delay.h>
#include <Logger.h>
#include <Warning.h>
```

```
#include <Meastypes.h>
#include <GPIOid_cex.h>
#include <GPIOsuite.h>

using namespace Dsp28335_ent;

namespace PETT
{
    bool TestMMC5883MA::test()
    {
        TestMMC5883MA inst;
        return inst.run();
    }

    TestMMC5883MA::TestMMC5883MA() :
        m_mag4(m_mag, cal_mag),
        rawmag4( x,y,z),
        mag4_int(0),
        dev(CEX::Halsuite::get_instance().i2ca, Devices::Mmc5883ma::i2caddr,
            bit,
            m_mag4, rawmag4, mag4temp,
            mag4_int)
    {
    }

    bool TestMMC5883MA::run()
    {
        Dsp28335_ent::GPIO scitx_c(Dsp28335_ent::GPIOname::gpio_scic_tx);
        scitx_c.dir(false); // true -> input
        scitx_c.fun_mux(0);
        Dsp28335_ent::GPIO scirx_c(Dsp28335_ent::GPIOname::gpio_scic_rx);
        scirx_c.dir(false); // true -> input
        scirx_c.fun_mux(0);

        CEX::Halsuite::get_instance().i2ca.init();

        bool ret = true;

        const Uint16 mpu_rate = dev.get_desired_rate();
```

```
47
48          Base::Chrono chrono(true);
49          Base::Chrono cr_aux(true);
50
51          bool sample_ok = false;
52          bool zero_value = false;
53
54          Base::Xcal3D::Type_podsync::Wrsafe wrmag(cal_mag);
55          wrmag.data.cal_data.nentries = 0;
56
57          chrono.tic();
58          cr_aux.tic();
59
60          m_mag4.write(0, 0, 0);
61          Base::Hmeas3::Rdsafe rd(m_mag);
62
63          Real t = chrono.toc();
64          Uint32 range_fails = 0;
65          Uint32 sample_read_ok = 0;
66          static const Real test_time = 5.0F;
67
68          Base::Rv3 read;
69          Maverick::Rvector3 v;
70
71          static Real min_sample_time = 1.0F/mpu_rate;
72
73          while((t < test_time) && (range_fails != 2))
74          {
75              if(cr_aux.toc()>min_sample_time)
76              {
77                  cr_aux.tic();
78                  dev.step();
79
80                  if(rd.is_new()) // Read data if available. Return True if measure
    changed since last time get_value was called
81                  {
82                      read = rd.get_updated_value();
83                      v.copy(read.v);
84                      v.scale(Const::TESLA2GAUSS);
```

```
85
86                  Real module = 0;
87                  module = v.norm2();
88                  log("\r");
89                  log("  Mag.(G): ");
90                  log_rvector3(v, 4);
91                  log(" G");
92                  log(" | Module: ", module, 4, false);
93                  log(" G");
94                  log("  Temp: ", mag4temp - Const::C2KELVIN, 2, false);
95                  log(" C");
96
97                  zero_value = (Rmath::fabsr(v[0]) == 0) || (Rmath::fabsr(v[1])
    == 0) || (Rmath::fabsr(v[2]) == 0);
98
99                  if(zero_value || (module < 0.01) || ((mag4temp - Const::
   C2KELVIN) > -40 && (mag4temp - Const::C2KELVIN) < 85))
100                 {
101                     lognl("-- Range fail");
102                     lognl("Module range: [0.01, +inf] G");
103                     lognl("Temperature range: [-40, 85] C");
104                     range_fails++;
105                     if(range_fails < 2)
106                     {
107                         lognl("-- Taking new sample...\n");
108                         Delay::ms(2000);
109                     }
110                 }
111                 else
112                 {
113                     log("-- Sample in range!");
114                     sample_ok = true;
115                     sample_read_ok++;
116                 }
117             }
118         }
119         t = chrono.toc();
120     }
121
```

```
122         lognl();
123         log("Samples read in range: ");
124         log(sample_read_ok);
125         lognl();
126 
127         if(!sample_ok && range_fails == 0)
128         {
129             log("    Test failed because no data has been read\r\n");
130             ret = false;
131         }
132         if(range_fails == 2)
133         {
134             log("    Test failed because there were range fails\r\n");
135             ret = false;
136         }
137 
138         return ret;
139     }
140 }
```

Listing F.13: TestMMC5883MA.cpp

F.8. TestPWM

F.8.1. TestPWM.h

```
1 #ifndef TestPWM_H_
2 #define TestPWM_H_
3 
4 #include <entypes.h>
5 #include <GPIOsuite.h>
6 #include <Pwmdev.h>
7 #include <ECAP.h>
8 
9 namespace PETT
10 {
11     class TestPWM
12     {
13     public:
```

```
14          TestPWM();
15
16          template<Dsp28335_ent::Pwmdev::PWMid PWMID, Dsp28335_ent::Pwmdev::PWMid
    PWMID2, Dsp28335_ent::ECAP::Id ECAPID>
17          static bool test_pwm_ecap();
18
19          bool run(Dsp28335_ent::Pwmdev::PWMid pwm_id, Dsp28335_ent::Pwmdev::PWMid
    pwm_id2, Dsp28335_ent::ECAP::Id ecap_num);
20      };
21
22      template<Dsp28335_ent::Pwmdev::PWMid PWMID, Dsp28335_ent::Pwmdev::PWMid
    PWMID2, Dsp28335_ent::ECAP::Id ECAPID>
23      bool TestPWM::test_pwm_ecap()
24      {
25          TestPWM inst;
26          bool res = inst.run(PWMID, PWMID2, ECAPID);
27          return res;
28      }
29  }
30
31  #endif /* INCLUDE_TestPWM_H_ */
```

Listing F.14: TestPWM.h

F.8.2. TestPWM.cpp

```
1  #include <TestPWM.h>
2  #include <Halsuite.h>
3  #include <Pwmsuite.h>
4  #include <ECAPcfg.h>
5  #include <Chrono.h>
6  #include <Delay.h>
7  #include <Math.h>
8  #include <GPIOsuite.h>
9  #include <Pwmdev.h>
10 #include <GPIOid_cex.h>
11
12 using namespace Dsp28335_ent;
13
14 namespace PETT
```

```
{

    TestPWM::TestPWM()
    {
    }


    bool TestPWM::run(Dsp28335_ent::Pwmdev::PWMid pwm_id, Dsp28335_ent::Pwmdev::
    PWMid pwm_id2, Dsp28335_ent::ECAP::Id ecap_num)
    {

        Dsp28335_ent::Pwmdev pwms(pwm_id);
        Dsp28335_ent::Pwmdev pwms2(pwm_id2);
        Dsp28335_ent::ECAP ecap(ecap_num);
        Base::Capdata data[2];
        Base::Capdata data2[2];

        static const Dsp28335_ent::GPIOid GPIOidECAP1 =  Dsp28335_ent::GPIOname::
    gpio_ecap1;

        Dsp28335_ent::ECAPcfg cfg = {true, GPIOidECAP1+ecap_num, ECAPcfg::wrap2,
    {ECAPcfg::trig_rising, ECAPcfg::trig_falling, ECAPcfg::trig_rising, ECAPcfg::
    trig_falling}};

        ecap.config(cfg);

        GPIO gpio_cex(GPIOid(Dsp28335_ent::GPIOname::gpio_pwm_01+pwm_id));
        GPIO gpio_cex2(GPIOid(Dsp28335_ent::GPIOname::gpio_pwm_01+pwm_id2));

        gpio_cex.init(GPIOtun::build(GPIOtun::dir_output,
                                     GPIOtun::pu_dis,
                                     Base::GPIOmux4::mux_1, //mux1 -> PWM
                                     GPIOtun::qsel_sync));


        gpio_cex2.init(GPIOtun::build(GPIOtun::dir_output,
                                     GPIOtun::pu_dis,
                                     Base::GPIOmux4::mux_0, //mux0 -> GPIO
                                     GPIOtun::qsel_sync));
```

```
50          gpio_cex2.clear();
51
52
53          Pwmdev::Config pwm_cfg;
54          pwm_cfg.active_high = true;
55          pwm_cfg.rmod = Pwmdev::duty;
56          pwm_cfg.cfg_low  = 0;
57          pwm_cfg.cfg_high = 1;
58
59          static const Uint16 freq = 5000;
60          pwms.set_enabled(false);
61          pwms.set_module_freq(freq); //5Khz
62          pwms.config(pwm_cfg);
63          pwms.set_enabled(true);
64          pwms.set(0.5); // Needed enabled = true to modify duty cycle
65
66          Uint32 sampled = 0;
67          Uint32 range_fails = 0;
68
69
70          Base::Chrono cr(true);
71
72          //Wait for PWM stabilization
73          Delay::ms(400);
74
75          Uint16 ind = 0;
76          cr.tic();
77
78          while(cr.toc()<0.2)
79          {
80              if(ecap.read(data[ind]))
81              {
82                  ind++;
83                  if(data[0].type != Base::Capdata::t_hgh)
84                  {
85                      ind = 0;
86                  }
87                  else if(ind>1)
88                  {
```

```
                    if(data[1].type == Base::Capdata::t_low)
                    {
                        sampled++;
                        Real t = Bsp::Kclk::get_sysclkperiod() * (data[0].tics +
    data[1].tics);
                        Real f = 1.0F/t;
                        if(fabs(freq-f) > freq*0.05)  //Allow 5 percent
                        {
                            range_fails++;
                        }
                    }
                    ind = 0;
                }
            }
        }

        pwms.set_enabled(false);



        gpio_cex.init(GPIOtun::build(GPIOtun::dir_output,
                                     GPIOtun::pu_dis,
                                     Base::GPIOmux4::mux_gpio,
                                     GPIOtun::qsel_sync));

        gpio_cex.clear();

        Delay::ms(400);

        gpio_cex2.init(GPIOtun::build(GPIOtun::dir_output,
                                     GPIOtun::pu_dis,
                                     Base::GPIOmux4::mux_1, //mux1 -> PWM
                                     GPIOtun::qsel_sync));


       gpio_cex.init(GPIOtun::build(GPIOtun::dir_output,
                                     GPIOtun::pu_dis,
                                     Base::GPIOmux4::mux_0, //mux0 -> GPIO
                                     GPIOtun::qsel_sync));
```

```
127         gpio_cex.clear();
128 
129 
130         Pwmdev::Config pwm_cfg2;
131         pwm_cfg2.active_high = true;
132         pwm_cfg2.rmod = Pwmdev::duty;
133         pwm_cfg2.cfg_low  = 0;
134         pwm_cfg2.cfg_high = 1;
135 
136 
137         pwms2.set_enabled(false);
138         pwms2.set_module_freq(freq); //5Khz
139         pwms2.config(pwm_cfg2);
140         pwms2.set_enabled(true);
141         pwms2.set(0.5);
142 
143         Uint32 sampled2 = 0;
144         Uint32 range_fails2 = 0;
145 
146 
147         Base::Chrono cr2(true);
148 
149         //Wait for PWM stabilization
150         Delay::ms(400);
151 
152         ind = 0;
153         cr2.tic();
154 
155         while(cr2.toc()<0.2)
156         {
157             if(ecap.read(data2[ind]))
158             {
159                 ind++;
160                 if(data2[0].type != Base::Capdata::t_hgh)
161                 {
162                     ind = 0;
163                 }
164                 else if(ind>1)
165                 {
```

```
166                     if(data2[1].type == Base::Capdata::t_low)
167                     {
168                         sampled2++;
169                         Real t2 = Bsp::Kclk::get_sysclkperiod() * (data2[0].tics +
    data2[1].tics);
170                         Real f2 = 1.0F/t2;
171                         if(fabs(freq-f2) > freq*0.05)  //Allow 5 percent
172                         {
173                             range_fails2++;
174                         }
175                     }
176                     ind = 0;
177                 }
178             }
179         }
180 
181         pwms2.set_enabled(false);
182 
183 
184 
185         gpio_cex2.init(GPIOtun::build(GPIOtun::dir_output,
186                                       GPIOtun::pu_dis,
187                                       Base::GPIOmux4::mux_gpio,
188                                       GPIOtun::qsel_sync));
189 
190         gpio_cex2.clear();
191 
192         cfg.en = false;
193         ecap.config(cfg);  // Disable eCAP
194 
195          if((sampled == 0 || range_fails>5) || (sampled2 == 0 || range_fails2>5))
196          {
197              return false;
198          }
199          else
200          {
201              return true;
202          }
203     }
```

```
}
```

Listing F.15: TestPWM.cpp

F.9. Operación Write Address (AppWruAddress)

F.9.1. AppWruAddress.h

```
#ifndef INCLUDE_APPWRUADDRESS_H_
#define INCLUDE_APPWRUADDRESS_H_

#include <Itport.h>
#include <Array.h>
#include <Arrayu8.h>

namespace PETT
{
    class AppWruAddress
    {
    public:
        AppWruAddress();

        static bool run();
        static void set_addr(const Uint16& addr);

    private:

        bool run_op();

        static Uint16* addr;
        //SCI comms
        Base::Itport_u8& sci;
        Base::Arrayu8 rx_buffer;

    };
}

#endif /* INCLUDE_APPWRUADDRESS_H_ */
```

Listing F.16: AppWruAddress.h

F.9.2. AppWruAddress.cpp

```
#include <Flash_wr.h>
#include <AppWruAddress.h>
#include <Address0.h>
#include <Address.h>
#include <Hsys.h>
#include <Logger.h>
#include <Mutex.h>
#include <Sysaddr.h>
#include <Array.h>
#include <Arrayu8.h>
#include <Halsuite.h>
#include <Chrono.h>
#include <Strutil.h>


using Rmath::powr;
using namespace CEX;
using namespace Dsp28335_ent;

namespace PETT
{
    Uint16* AppWruAddress::addr;
    static const Uint16 sz8 = 300;

    bool AppWruAddress::run()
    {
        AppWruAddress inst;
        bool res = inst.run_op();
        return res;
    }

    void AppWruAddress::set_addr(const Uint16& addr0)
    {
        addr = const_cast<Uint16*>(&addr0);
    }

    AppWruAddress::AppWruAddress() :
```

```
38          sci(CEX::Halsuite::get_instance().scia),
39          rx_buffer(sz8, Memmgr::external)
40      {
41      }
42
43      bool AppWruAddress::run_op()
44      {
45          bool ret = true;
46
47          Base::Chrono chrono(true);
48          CEX::Halsuite& hal = CEX::Halsuite::get_instance();
49
50          chrono.tic();
51          Real t = chrono.toc();
52
53          log("Uaddress: ");
54          lognl();
55
56          Uint16 uaddr = 0;
57          Uint32 aux = 0;
58          Uint32 aux2 = 0;
59          Uint16 i = 0;
60
61          for(i=0; i<5; i++)
62          {
63              while(!sci.read(rx_buffer[0]));
64              Strutil::parse_uint(rx_buffer, aux, aux2);
65              log(aux2);
66              uaddr += aux2 * powr(10, 4-i);
67              aux = 0;
68              aux2 = 0;
69          }
70
71          const void* ptr_dst = reinterpret_cast<const void*>(addr);
72          {
73              // Disable interrupts to not to try execution of interrupt code which
    is in flash
74              Base::Mutex m(true);
75              ret = Bsp::Flash_wr::write_sector(ptr_dst, &uaddr, sizeof(Base::
```

```
    Address0));
76          }
77
78          // If address is not valid, use "unknown" ??
79          if(!ret || (uaddr < Base::Address0::addr_cex_first) || (uaddr > Base::
    Address0::addr_cex_last))
80          {
81              lognl();
82              log("\033[37mWrong uaddress. ");
83              ret = false;
84          }
85          else
86          {
87              // Store the system address
88              Dsp28335_ent::Hsys::set_sysaddr(Address(uaddr));
89
90              lognl();
91              log("\033[37mUaddress setted. ");
92          }
93          return ret;
94      }
95  }
```

Listing F.17: AppWruAddress.cpp

F.10. Operacións LTT

F.10.1. LTTest.h

```
1  #ifndef INCLUDE_LTTEST_H_
2  #define INCLUDE_LTTEST_H_
3
4
5  namespace PETT
6  {
7      class LTTest
8      {
9      public:
10         LTTest();
```

```
11          static bool run();
12
13      private:
14
15          bool run_op();
16
17      };
18  }
19
20
21  #endif /* INCLUDE_LTTEST_H_ */
```

Listing F.18: LTTest.h

F.10.2. LTTest.cpp

```
1  #include <LTTest.h>
2  #include <SerialTestCtrl.h>
3  #include <Chrono.h>
4  #include <Delay.h>
5  #include <Logger.h>
6  #include <Resetif.h>
7
8
9  using namespace Dsp28335_ent;
10 using namespace Base;
11
12 namespace PETT
13 {
14
15     bool LTTest::run()
16     {
17         LTTest inst;
18         bool res = inst.run_op();
19         return res;
20     }
21
22     LTTest::LTTest()
23     {
24     }
```

```
25
26    bool LTTest::run_op()
27    {
28        bool ret = true;
29        SerialTestCtrl& ctrl = SerialTestCtrl::get_instance();
30
31        Base::Chrono chrono(true);
32
33        Uint8 cont = 1;
34        Real t = 0;
35
36        Delay::ms(500);
37
38        chrono.tic();
39
40        while(t<3600 && ctrl.failed == 0)
41        {
42            ctrl.run_all();
43            t = chrono.toc();
44            lognl("Run ");
45            log(cont);
46            lognl(" ");
47            lognl("Time ");
48            log(t);
49            lognl(" ");
50            cont++;
51            if (ctrl.failed != 0)
52            {
53                ret = false;
54            }
55            Dsp28335_ent::Reset_if::get_instance();   // Force initialization of
    reset singleton
56        }
57
58
59        return ret;
60    }
```

```
}
```

Listing F.19: LTTest.cpp

F.11. tiiw.cpp

```
///    C++ Template instantiation issues workarrounds.
///    \see http://processors.wiki.ti.com/index.php/C%2B%2B_Template_Instantiation_Issues

#include <tiiw.h>

// using workarround 2
#include <Autopilot.h>
#include <Arbitration_cfg.h>
#include <Deadmanchecker_fw.h>
#include <CAN.h>
#include <CANids.h>
#include <CANin_suite.h>
#include <CANout_suite.h>
#include <CANmb.h>
#include <CANsc_suite.h>
#include <CEX_pwms.h>
#include <Commgr.h>
#include <ECAPcfg_cex.h>
#include <Find.h>
#include <Fmsgconsumer.h>
#include <Fmsgproducer.h>
#include <GPIO.h>
#include <GPIOid_cex.h>
#include <GPIOsuitetun.h>
#include <GPIOtun.h>
#include <Hbvar.h>
#include <Icompress.h>
#include <Jetibox_fmsg0.h>
#include <Jetibox_fmsg_fw.h>
#include <Portmgr.h>
#include <Pwmdev.h>
#include <Stepmgr.h>
```

```
#include <Stickrawtrans.h>
#include <Sticktypes.h>
#include <Strutil.h>
#include <Telemetry.h>
#include <Tnarray.h>
#include <Tportnull.h>
#include <Tunnelwr.h>
#include <Varmgr.h>
#include <Vars_cex.h>
#include <Vpktport.h>
#include <Xcfgcomport.h>
#include <Xfmsg8_cex_fw.h>
#include <Xpccansuite.h>
#include <Xpccantrait_cex.h>
#include <Xpccanutils.h>
#include <Xpcecapsuite_cex.h>
#include <Xpcu8trait.h>
#include <Xtunnelwr.h>

// using workarround 3
namespace // anonymous namespace, ensure content is never used
{

    static void never_used()
    {
        using namespace Base;
        using namespace CEX;

        build_arrayext_1param<Dsp28335_ent::CANmb,  Mblock<Dsp28335_ent::CANmb::
    Build_params> >();
        build_arrayext_1param<Arb::Autopilot,       Array<Arb::Deadmanchecker>
    >();
        build_arrayext_1param<Dsp28335_ent::GPIO,   Mblock<const Dsp28335_ent::
    GPIOid> >();
        build_arrayext_1param<Dsp28335_ent::Pwmdev, Base::Rngit<Dsp28335_ent::
    Pwmdev::PWMid> >();
        build_arrayext_1param<CEX_pwm,              Base::Rngit<Dsp28335_ent::
    Pwmdev::PWMid> >();
        build_arrayext_1param<Fmsgconsumer,         Mblock<const Fmsgconsumer::
```

```
    Config> >();
67      build_arrayext_1param<Fmsgproducer,         Mblock<const Fmsgproducer::
    Config> >();
68      build_arrayext_1param<Bsp::Hbvar,           Rngit<Bvar> >();
69      build_arrayext_1param<Ver::Cap_pps,         Array<volatile Real> >();
70      build_arrayext_1param<Ver::Cappulse,        Mblock<const Ver::Cappulse::
    Buildparams> >();
71      build_arrayext_1param<CANserial,            Uint16>();
72      build_arrayext_1param<CANin_p,              Uint16>();
73      build_arrayext_1param<CANout_c,             Uint16>();
74      build_arrayext_1param<Chrono,               bool>();
75      build_arrayext_1param<Ver::Cap_pps,         Array<float> >();
76
77      build_arrayext_1param<CEX::Stickppm_cex,    Mblock<const Uvar> >();
78
79
80
81      Array<Istep*>(0U, Base::Memmgr::external);
82
83      build_arrayext_2param<Vpktport,             Uint16,     Base::Memmgr::
    Type > ();
84
85      {
86          using namespace Base::Tuntraits;
87
88          tuntrait_array<Tundefault<Arb::Autopilot_cfg>,      Array<Arb::
    Autopilot_cfg> >();
89          tuntrait_array_ref<Tundefault<CANin_p>,             Array<CANin_p>
    >();
90          tuntrait_array_ref<Tundefault<CANout_c>,            Array<CANout_c>
    >();
91          tuntrait_array_ref<Tundefault<SerialCAN>,           Array<SerialCAN>
    >();
92
93          tuntrait_array<Tundefault<CANid>,               CANids<Arb::
    Arbitration_cfg::nmaxautopilots>::Tcanids>();
94          tuntrait_array<Cxet<CANcfg::Rxcfg>,                 CANcfg::Trx_array
                >();
95          tuntrait_array<CEX::CEX_pwms::Pwmstun,              CEX_pwms::TPwms>
```

```
();
96          tuntrait_array<Tuntraits::Cxet<CEX::Jetibox_fmsg0>, Tnarrayresz<CEX::
Jetibox_fmsg0, jetibox_nvecs> >();
97          tuntrait_array<Xpc<Xpccantrait_cex>::Tuntrait0,     Xpc_can_cex::
Type_map> ();
98          tuntrait_array<Xpc<Xpcu8_trait>::Tuntrait0,         Xpc_u8::Type_map>
 ();
99          tuntrait_array<Xpc<Xpcecap_trait>::Tuntrait0,       Xpc_ecap::
Type_map> ();
100         tuntrait_array<Arraytun_size16<Cxet<Xfmsg0>,        Xfmsg0array >,
Xfmsg8_pod >();
101
102         tuntrait_array<Tundefault<Base::Xfmsgcanp0>,        CEX::Telemetry::
Type_tx_ini> ();
103         tuntrait_array<Tundefault<Base::Xfmsgcanp0>,        CEX::Telemetry::
Type_tx> ();
104
105         tuntrait_array<Tundefault<Xtunnelwr_array>,         Xtunnelwr_array>
();
106         tuntrait_array<Tundefault<Dsp28335_ent::ECAPcfg>,   ECAPcfg_cex> ();
107         tuntrait_array<Tundefault<Xcappps0>,                Xcappps_array_cex
> ();
108
109         tuntrait_array<Tundefault<GPIOsuitetun>,            GPIOarray_tun> ()
;
110     }
111
112     build_tunarrayext_1param<Portmgr::Port_cfg, Array<Ivpktport*>*> ();
113     Tport_null<Uint8,Uint8>::get_instance();
114
115     {
116         using Dsp28335_ent::GPIOtun;
117         GPIOtun::build(GPIOtun::dir_input, GPIOtun::pu_dis, GPIOmux4::mux_0,
GPIOtun::qsel_async);
118     }
119
120     Declval<Base::Tunnelwr>::ref().parse<Base::RTCM3parser>(Declval<Base::
RTCM3parser>::ref(),0U);
121     Declval<Base::Tunnelwr>::ref().parse<Base::CANserial_parser>(Declval<Base
```

```
::CANserial_parser>::ref(),0U);

     Base::Tuntraits::Arraytun_nosize<Base::Tuntraits::Cxet<Base::
Stickrawtrans>,
          Base::Tnarray<Base::Stickrawtrans, 16U> >::str2elem(
                  Declval<Base::Tnarray<Base::Stickrawtrans, 16U> >::ref(),
                  Declval<Base::Lossy>::ref());
     Declval<Base::Tunarray<Base::Stickout> >::ref().cset(Declval<Base::Lossy
>::ref());

     Declval< Xcfgcomport::Type_tun >::ref().cset(Declval<Lossy>::ref());
     Declval< Xcfgcomport::Type_tun >::ref().cget(Declval<Lossy>::ref());

     Maverick::Find::sorted(Declval<Mblock<const Uint32> >::ref(),
                            0,
                            Declval<Uint32>::ref(),
                            Declval<Uint32>::ref());

     Declval<Xpc_ecap>::ref().add_consumer( Declval<Xpcecap_trait::XC::Idx>::
ref(),
                                            Declval<Xpcecap_trait::XC::Idx>::
ref(),
                                            Declval<Array<Ver::Cap_pps>   >::
ref());
     Declval<Xpc_ecap>::ref().add_consumer( Declval<Xpcecap_trait::XC::Idx>::
ref(),
                                            Declval<Xpcecap_trait::XC::Idx>::
ref(),
                                            Declval<Array<Stickppm_cex>  >::
ref());
     Declval<Xpc_ecap>::ref().add_consumer( Declval<Xpcecap_trait::XC::Idx>::
ref(),
                                            Declval<Xpcecap_trait::XC::Idx>::
ref(),
                                            Declval<Array<Ver::Cappulse>  >::
ref());

     Tarraycfg<Array<CANin_p>&,   CANin_suite_cex > (Declval<Array<CANin_p>
>::ref());
```

```
148         Tarraycfg<Array<CANout_c>&,  CANout_suite_cex> (Declval<Array<CANout_c>
    >::ref());
149         Tarraycfg<Array<SerialCAN>&, CANsc_suite_cex > (Declval<Array<SerialCAN>
    >::ref());
150
151
152         Xpccanutils::check_can_in(Declval<Base::Array<Base::CANin_p> >::ref(),
153                                   Declval<Base::Xpc<Base::Xpccantrait_cex> >::ref
    (),
154                                   0,
155                                   Declval<Xpccantrait_cex::XP::Idx>::ref(),
156                                   Declval<Xpccantrait_cex::XP::Idx>::ref());
157
158         Fifospsc<Capdata>(0, Base::Memmgr::external);
159
160         Fifospsc<Uint8>(0, Base::Memmgr::external);
161
162         Strutil::parse_real<Real>(Declval<Lossy>::ref(), 0, 0, ' ', Declval<Real
    >::ref());
163     }
164}
```

Listing F.20: tiiw.cpp

F.12. Logger.h

```
1#ifndef _LOGGER_H__
2#define _LOGGER_H__
3
4
5#include <Vstring.h>
6#include <Array.h>
7#include <Rvector3.h>
8#include <SerialTestCtrl.h>
9
10using namespace Base;
11
12namespace PETT
13{
```

```
14
15      static void log(const Kstring& str, bool newline=false)
16      {
17          SerialTestCtrl::get_instance().print_kstring(str);
18          if(newline)
19          {
20              SerialTestCtrl::get_instance().print_kstring(Kstring("\r\n"));
21          }
22      }
23
24      static inline void lognl(const Kstring& str)
25      {
26          log(str, true);
27      }
28
29      static void log(const char* str, bool newline=false)
30      {
31          Kstring kstr(str);
32          log(kstr, newline);
33      }
34
35      static inline void lognl(const char* str)
36      {
37          log(str, true);
38      }
39
40      static inline void lognl()
41      {
42          log("\r\n");
43      }
44
45      static void log(Uint32 v)
46      {
47          static Vstring buf(32, Memmgr::external);
48
49          buf.clear();
50          buf.append_uint(v);
51          log(buf);
52      }
```

```
53
54      static void log(Uint16 v)
55      {
56          log(static_cast<Uint32>(v));
57      }
58
59      static void log(char v)
60      {
61          static Vstring buf(32, Memmgr::external);
62          buf.clear();
63          buf.append_char(v);
64          log(buf);
65      }
66
67      static inline void log_c(char v)
68      {
69          log(v);
70      }
71
72      static void lognl(char v)
73      {
74          log(v);
75          lognl();
76      }
77
78      static void log(Uint8 v)
79      {
80          static Vstring buf(32, Memmgr::external);
81
82          buf.clear();
83          buf.append_uint(v);
84          log(buf);
85      }
86
87      static void log(Real v, Uint8 digits=3)
88      {
89          static Vstring buf(32, Memmgr::external);
90
91          buf.clear();
```

```
92          buf.append_real(v, digits);
93          log(buf);
94      }
95
96      static void log(const char* str, Real v, Uint16 ndigits, bool nl = true)
97      {
98          log(str);
99          log(v, ndigits);
100         if(nl)
101         {
102             lognl();
103         }
104     }
105
106     static inline void log_rvector3(const Maverick::Rvector3& v, Uint16 nFrac)
107     {
108         log_c('(');
109         log(v[0], nFrac);
110         log(", ");
111         log(v[1], nFrac);
112         log(", ");
113         log(v[2], nFrac);
114         log_c(')');
115     }
116
117 }
118 #endif
```

Listing F.21: Logger.h

F.13. TestJETI

F.13.1. TestJETI.h

```
1 #ifndef INCLUDE_TESTJETI_H_
2 #define INCLUDE_TESTJETI_H_
3
4 #include <GPIO.h>
5
```

```
using namespace Dsp28335_ent;

namespace PETT
{
    class TestJETI
    {
    public:
        TestJETI();
        static bool test();
        float time_counter(bool cont);

    private:

        bool run();
        Dsp28335_ent::GPIO scia_tx;
        Dsp28335_ent::GPIO scia_rx;


    };

} /* namespace PETT */


#endif /* INCLUDE_TESTJETI_H_ */
```

Listing F.22: TestJETI.h

F.13.2. TestJETI.cpp

```
#include <TestJETI.h>
#include <Halsuite.h>
#include <GPIO.h>
#include <Delay.h>
#include <Logger.h>


using namespace Dsp28335_ent;
using namespace Base;

```

```
11 namespace PETT
12 {
13
14     bool TestJETI::test()
15     {
16         TestJETI inst;
17         return inst.run();
18     }
19
20
21     TestJETI::TestJETI() :
22         scia_tx(Dsp28335_ent::GPIOname::gpio_scia_tx),
23         scia_rx(Dsp28335_ent::GPIOname::gpio_scia_rx)
24     {
25
26     }
27
28     float TestJETI::time_counter(bool cont)
29     {
30         Base::Chrono chrono(true);
31         Real t = 0;
32         if(scia_rx.get() && cont)
33         {
34             chrono.tic(); // Start time count when Arduino sends 1
35             while(scia_rx.get()); //Wait for Arduino to send 1
36             while(!scia_rx.get()); //Wait for the Arduino to send 0
37             if(scia_rx.get()) //Wait for the Arduino to send 1 again to measure
    the time between rising edges
38             {
39                 t = chrono.toc();
40                 log("  Time: ");
41                 log(t);
42                 lognl("");
43             }
44         }
45         else if(!scia_rx.get()) //Wait for Arduino to send 0
46         {
47             while(!scia_rx.get());
48             chrono.tic(); // Start time count when Arduino sends 1
```

```
49          while(scia_rx.get()); //Wait for Arduino to send 1
50          while(!scia_rx.get()); //Wait for the Arduino to send 0
51          if(scia_rx.get()) //Wait for the Arduino to send 1 again to measure
    the time between rising edges
52          {
53              t = chrono.toc();
54              log("  Time: ");
55              log(t);
56              lognl("");
57          }
58      }
59      return t;
60  }
61
62  bool TestJETI::run()
63  {
64
65      bool res = false;
66      Real t = 0;
67
68      scia_rx.dir(false); // false -> input
69      scia_rx.fun_mux(0);
70      scia_rx.clear(); // output low
71
72      scia_tx.dir(false); // false -> input
73      scia_tx.fun_mux(0);
74      scia_tx.clear(); // output low
75
76      scia_tx.dir(true); // true -> output
77
78      scia_tx.clear(); // output low
79      Delay::ms(1);
80
81      scia_tx.set(); // output high
82      Delay::ms(1);
83
84      scia_tx.clear(); // output low
85      Delay::ms(1);
86
```

```
87          scia_tx.set();
88          Delay::ms(1);
89
90          scia_tx.clear();
91          Delay::ms(1);
92
93          scia_tx.set();
94          Delay::ms(1);
95
96          scia_tx.clear();
97          Delay::ms(1);
98
99          scia_tx.set();
100         Delay::ms(5);
101
102         scia_tx.clear();
103         Delay::ms(5);
104
105         scia_tx.dir(false); // false -> input
106
107     t = time_counter(res);
108     if(t>0.0045 && t<0.007)
109     {
110         lognl("1 ");
111         res = true;
112     }
113
114     t = time_counter(res);
115     if(t>0.0045 && t<0.007)
116     {
117         lognl("2 ");
118         res &= true;
119     }
120
121     return res;
122 }
123
```

```
}
```

Listing F.23: TestJETI.cpp

F.14. Arduino

```
byte digital_pin = PD2;
volatile int digital_value;

void setup() {

  Serial.begin(115200);
  pinMode(PD2, INPUT);
  Serial.println("Start test:");
  attachInterrupt(digitalPinToInterrupt(PD2), read_data, FALLING);

}

void loop() // run over an+ over
{

}

void read_data() {
  Serial.println("Byte");
  digital_value = pulseIn(digital_pin, HIGH);
  Serial.println("digital_value");
  Serial.println(digital_value);
  rx_data(digital_value);
}

void rx_data(int signal_rx)
{
  Serial.println("Rx");
  Serial.println(signal_rx);

  if(signal_rx > 1110 && signal_rx < 1170)
  {
    Serial.println("Signal OK");
```

```
34      detachInterrupt(digitalPinToInterrupt(PD2));
35      delay(1000);
36      tx_data();
37    }
38    else if(signal_rx != 0)
39    {
40      Serial.println("Fail");
41      detachInterrupt(digitalPinToInterrupt(PD2));
42    }
43  }
44
45  void tx_data()
46  {
47      delay(1000);
48      pinMode(PD2, OUTPUT);
49      delay(1000);
50      digitalWrite(PD2, LOW);
51      delay(1000);
52      digitalWrite(PD2, HIGH);
53      delay(500);
54      digitalWrite(PD2, LOW);
55      delay(500);
56      digitalWrite(PD2, HIGH);
57      delay(500);
58      digitalWrite(PD2, LOW);
59      delay(500);
60      digitalWrite(PD2, HIGH);
61      delay(500);
62      digitalWrite(PD2, LOW);
63      delay(500);
64      pinMode(PD2, INPUT);
65  }
```

Listing F.24: ReadSerial.ino

ANEXO G:

ACCEPTANCE TEST PROCEDURE (ATP)

G.1. CEX ATP

CAN EXPANDER ATP v2.5

Embention

03 de junio de 2021

Índice general

II

CAPÍTULO 1

CEX Verification 2.5

1.1 Objeto y Alcance

El presente procedimiento describe las actividades a llevar a cabo para la verificación de un CAN Expander (CEX). Este documento abarca las tareas de verificación desde la llegada de la PCB a las instalaciones de Embention hasta justo antes del envío al cliente.

El objetivo de este procedimiento es la detección precoz de CAN Expanders que puedan venir defectuosos por parte del montador, evitando así la entrega al cliente de productos defectuosos.

Para la realización de la Verificación se necesita una PCB de Verificación 1.0 Modo Slow, una PCB de Verificación 1.0 Modo BurnIn y el sw PETT del CEX.

La CEXVerificationPCB 1.0 que se puede encontrar en: https://github.com/embention/CEX/releases/tag/release%2FCEXVerificationPCB%2F1.0

1.2 Documentos Referenciados

Name	Version	Author	Year
Producción Manual ATP	v1.2.7	Embention	2021

1.3 Descripción del Procedimiento

El procedimiento se activa tras el pedido de un cliente. Este procedimiento requiere que la siguiente información sea proporcionada:

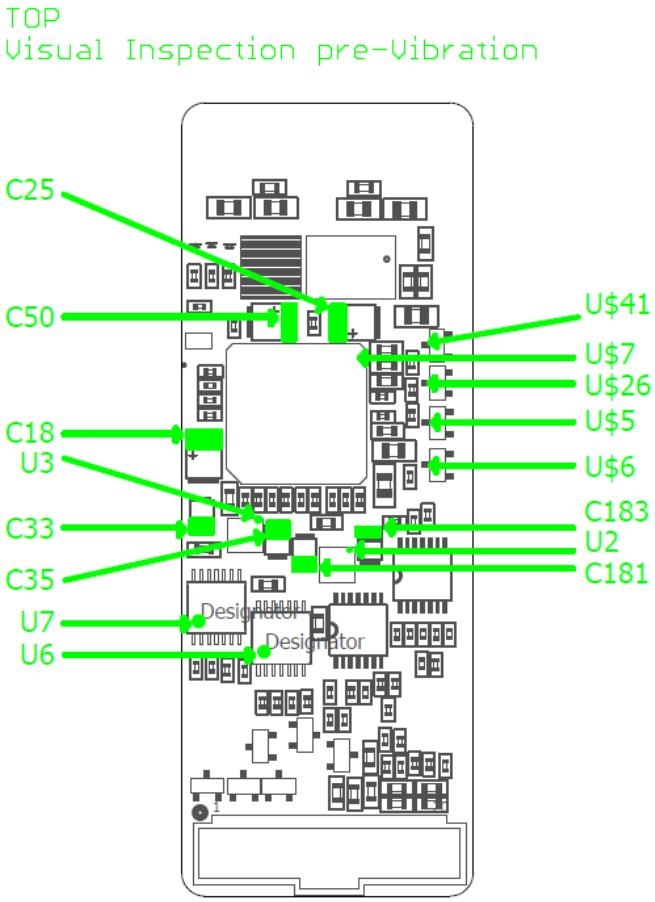
- Production Order: orden de producción del CEX.
- Part Number (PN): part number del CEX.
- Serial Number (SN): número de serie del CEX.
- SW version: versión de software con la que sale el CEX.
- Date: fecha en la cual se empieza la verificación.
- Laboratory Temperature (ºC): temperatura del laboratorio en ºC cuando empieza la verificación.
- Laboratory Humidity (%): humedad del laboratorio en % cuando empieza la verificación.
- **Producto a Verificar: seleccionar el producto a verificar entre:**
    - Veronte CAN Expander v1.2
    - CAN Expander MC Unit (RS232 & RS485)v1.2
    - CAN Expander MC Unit (UART & RS485)v1.2
    - CAN Expander MC Unit (UART HALF DUPLEX& RS485)v1.2

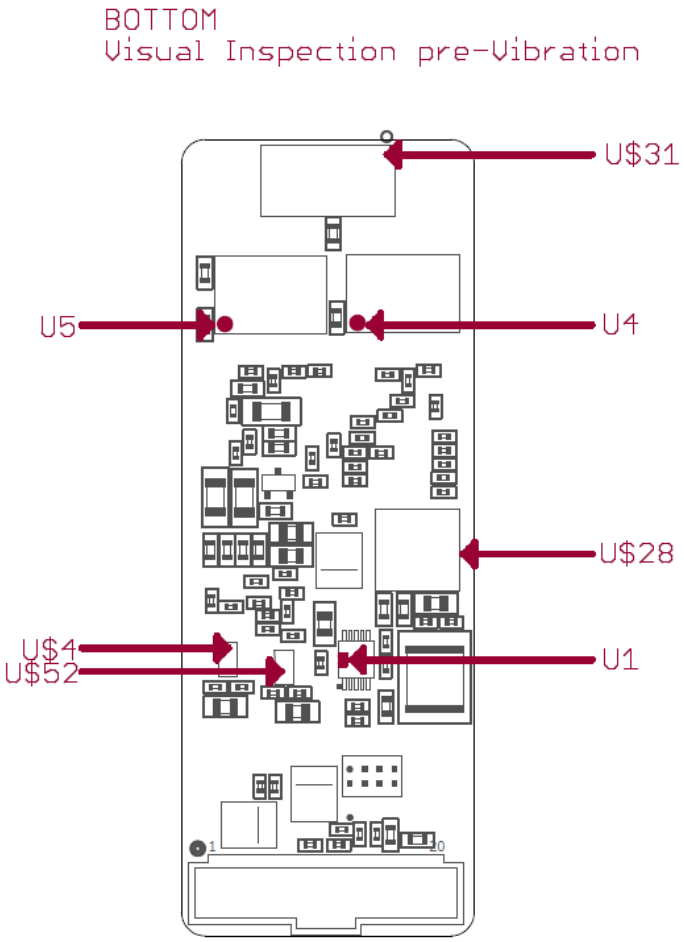
Estos datos se tienen que introducir en el apartado «INFO» del Acceptance Test Report (ATR) o también llamado Production Process Control Sheet.

1.4 Inspección Visual pre Vibrado

Realizar la inspección visual como se describe en Producción Manual ATP v1.2.7, sección 1.2. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Inspecci%C3%B3n%20Visual/index.html

En las siguientes figuras se describen los componentes y las orientaciones a inspeccionar.


TOP
Visual Inspection pre-Vibration
C25
C50
C18
U3
C33
C35
U7
U6
U$41
U$7
U$26
U$5
U$6
C183
U2
C181
Designator
Designator



1.5 Shaking Test

Realizar el shaking test como se describe en Producción Manual ATP v1.2.7, sección 1.3. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Shaking%20test/index.html

Fijar el panel de CEX al marco.

Configurar los DIP switch en modo vibración 1.

A continuación se representan las curvas de vibración de los ejes.

La frecuencia predomina en los ejes X e Y, alrededor de 30 Hz. Debido a que es la misma mesa de vibración que utiliza 1xVeronte y 4xVeronte, para evitar estresar sus flex durante su proceso de verificación y ensamblaje, la vibración en el eje Z es menor.

- Eje X

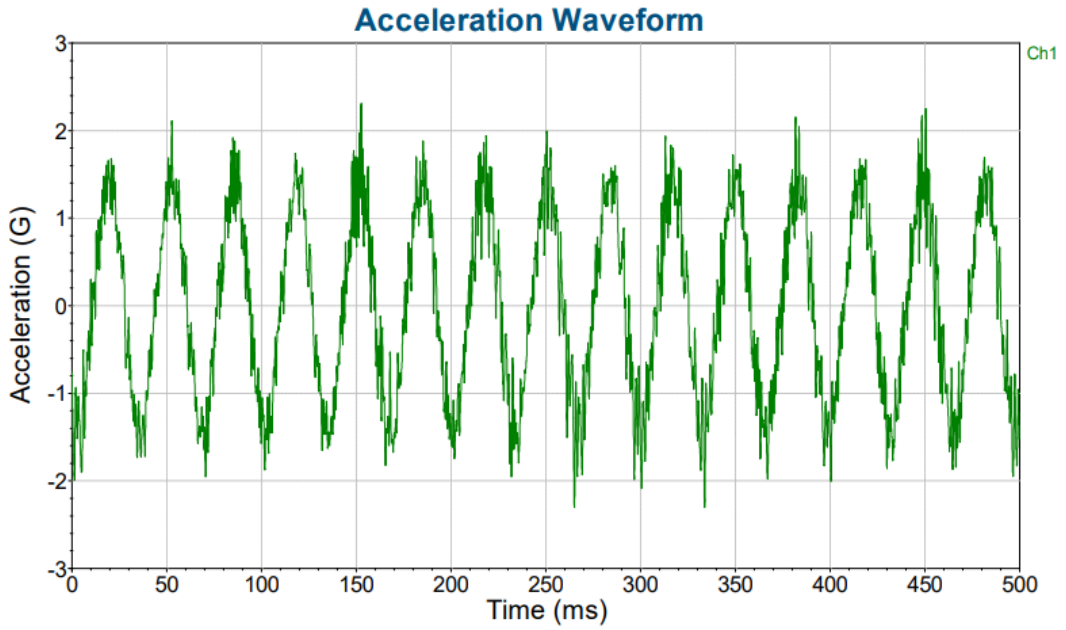


Figura 1: Aceleración eje X

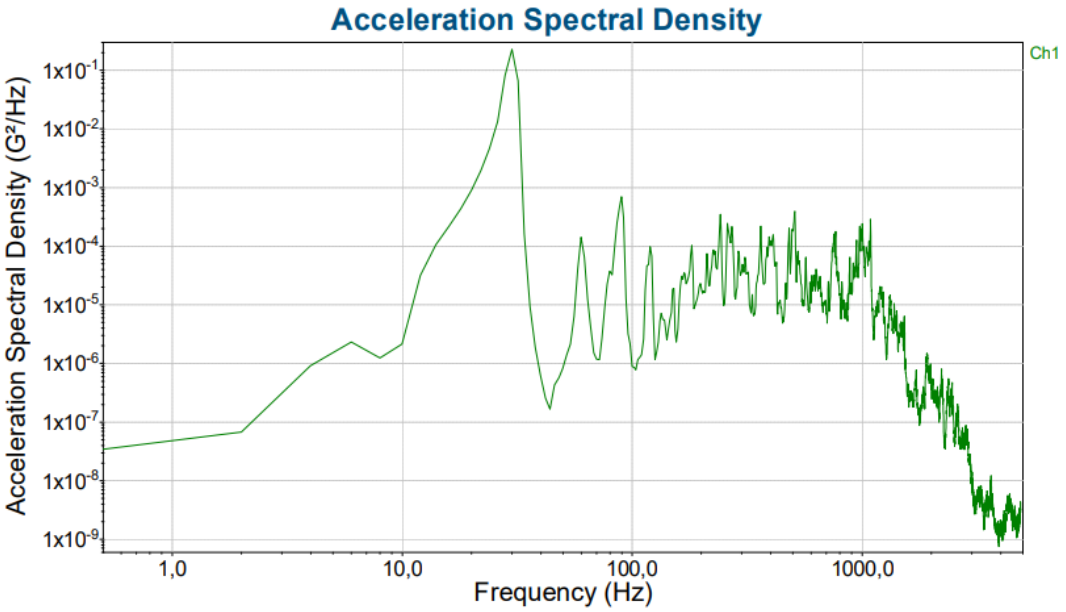


Figura 2: Densidad de Aceleración Espectral eje X

- Eje Y

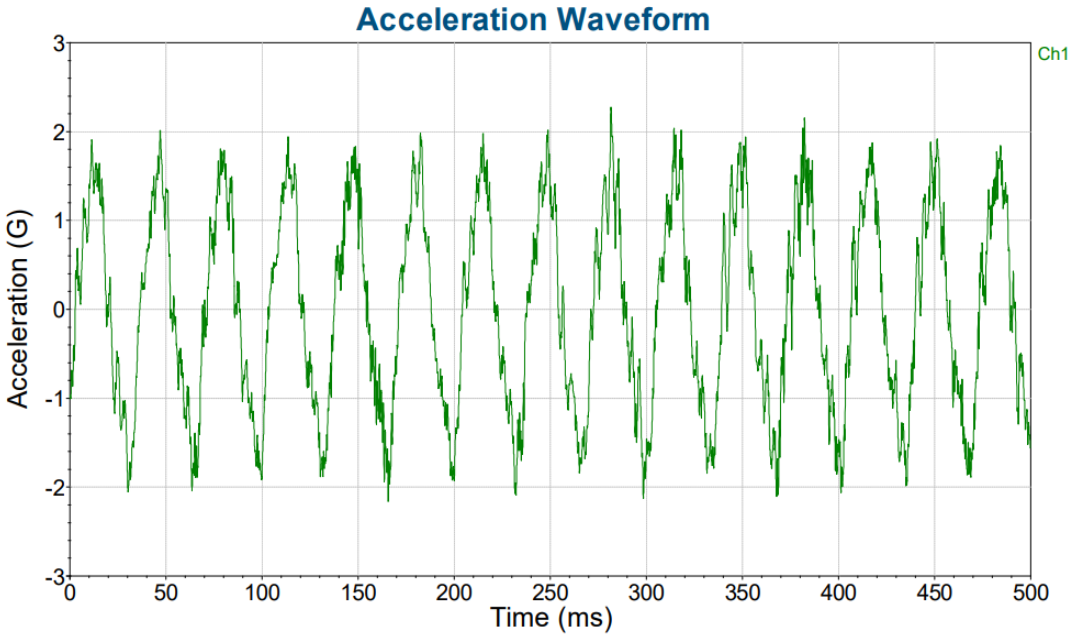


Figura 3: Aceleración eje Y

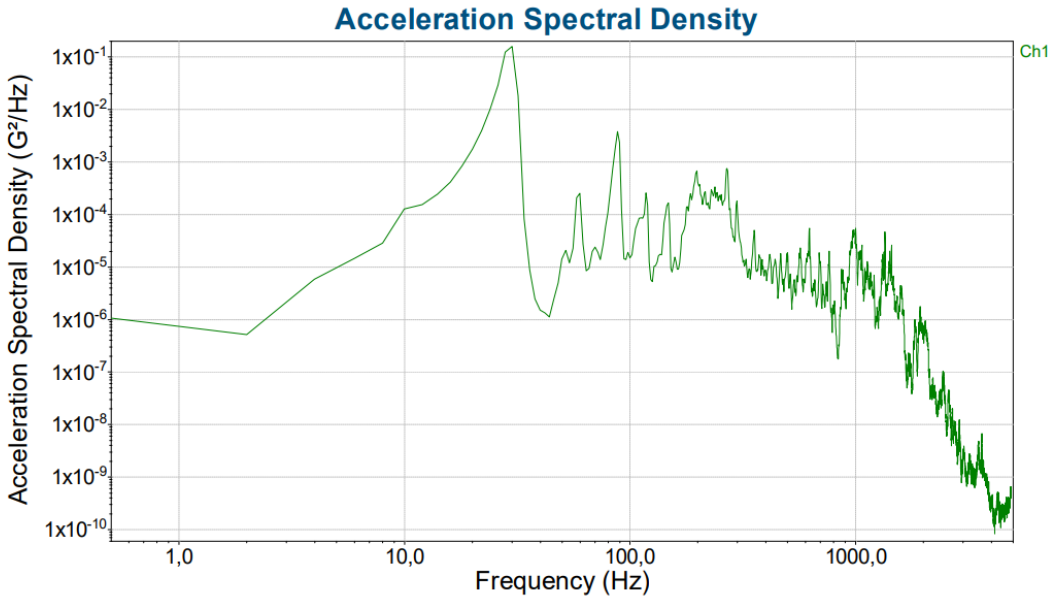


Figura 4: Densidad de Aceleración Espectral eje Y

- Eje Z

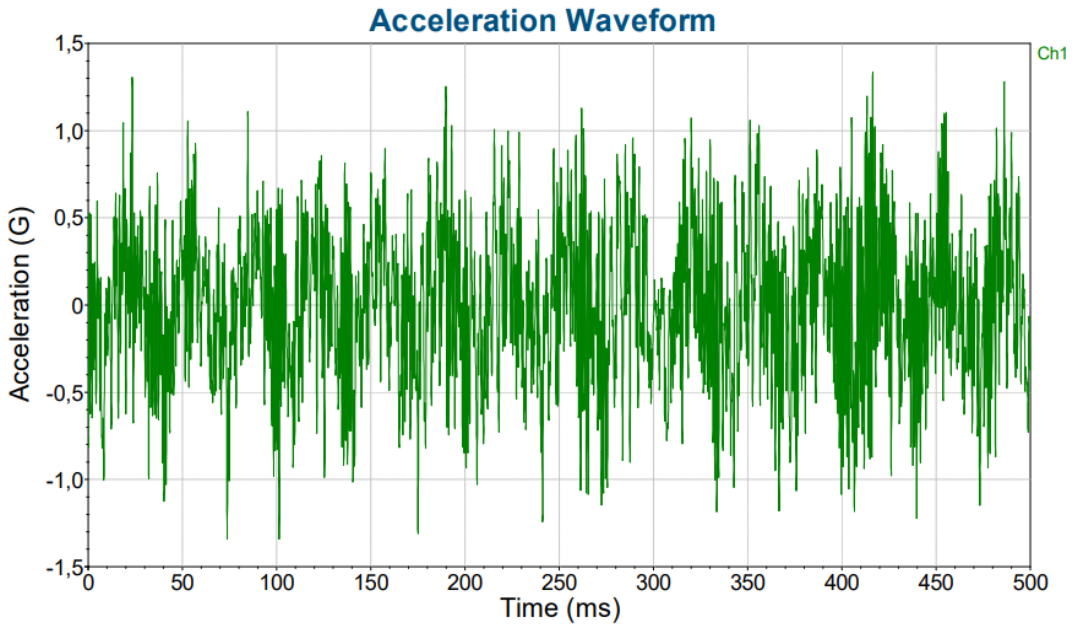


Figura 5: Aceleración eje Z

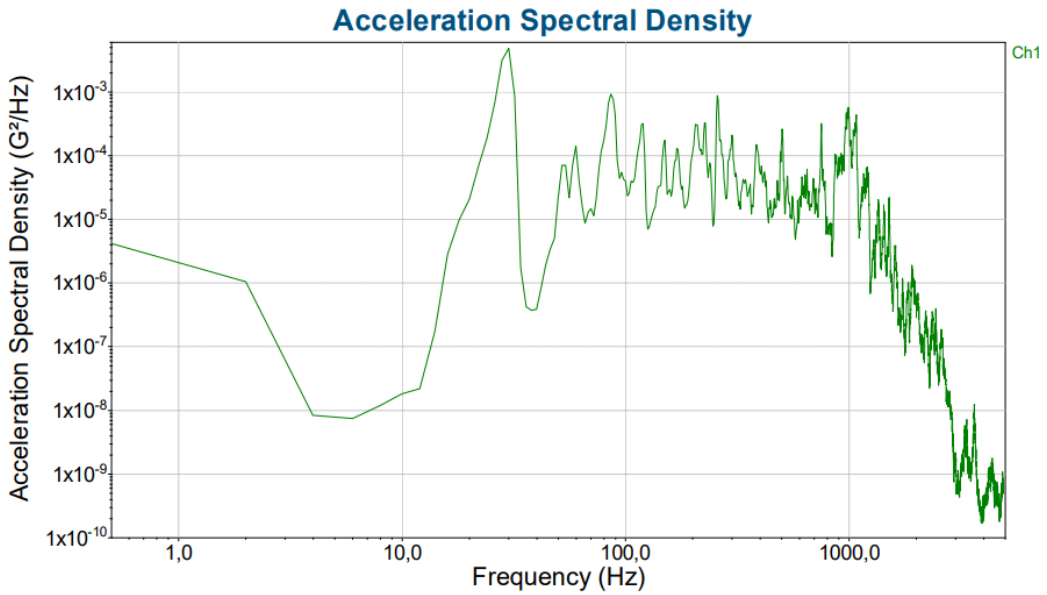


Figura 6: Densidad de Aceleración Espectral eje Z

1.6 Inspección visual post Vibrado

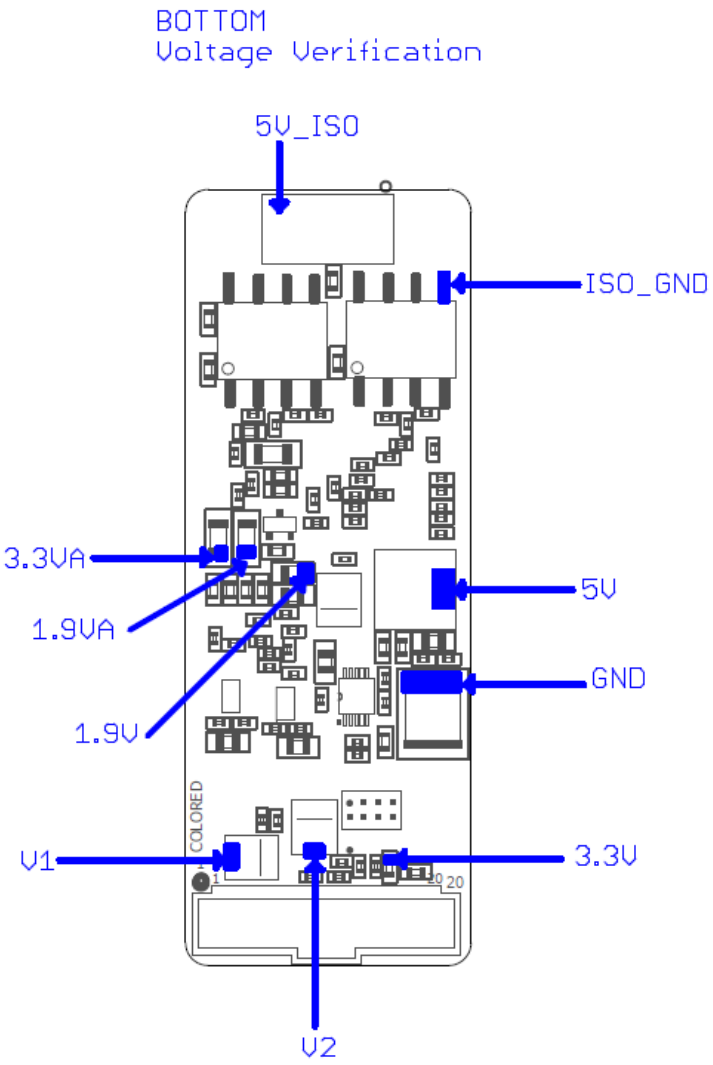
Realizar otra inspección visual como se describe en la sección 1.4 de este documento, para comprobar los componentes después de la prueba de vibración.

1.7 Verificación de Cortocircuitos

Realizar la verificación de cortocircuitos como se describe en Producción Manual ATP v1.2.7, sección 1.4. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Verificaci%C3%B3n%20de%20Cortocircuitos/index.html

Los puntos a medir se encuentran en la siguiente imagen.

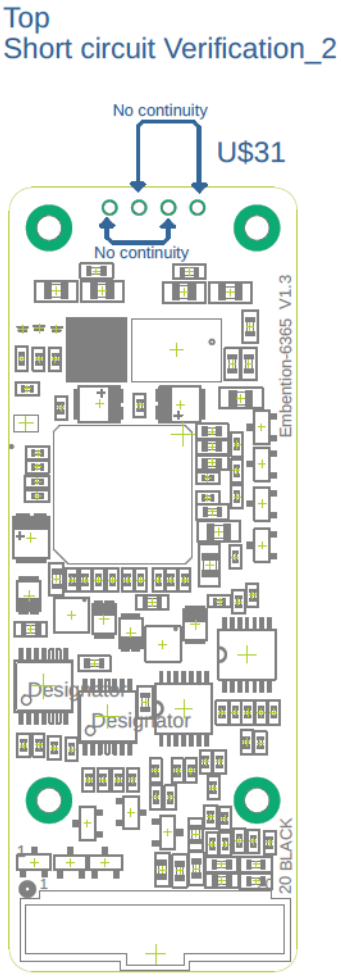
La verificación de la señal 5V_ISO debe hacerse respecto a ISO_GND las otras verificaciones respecto a GND.

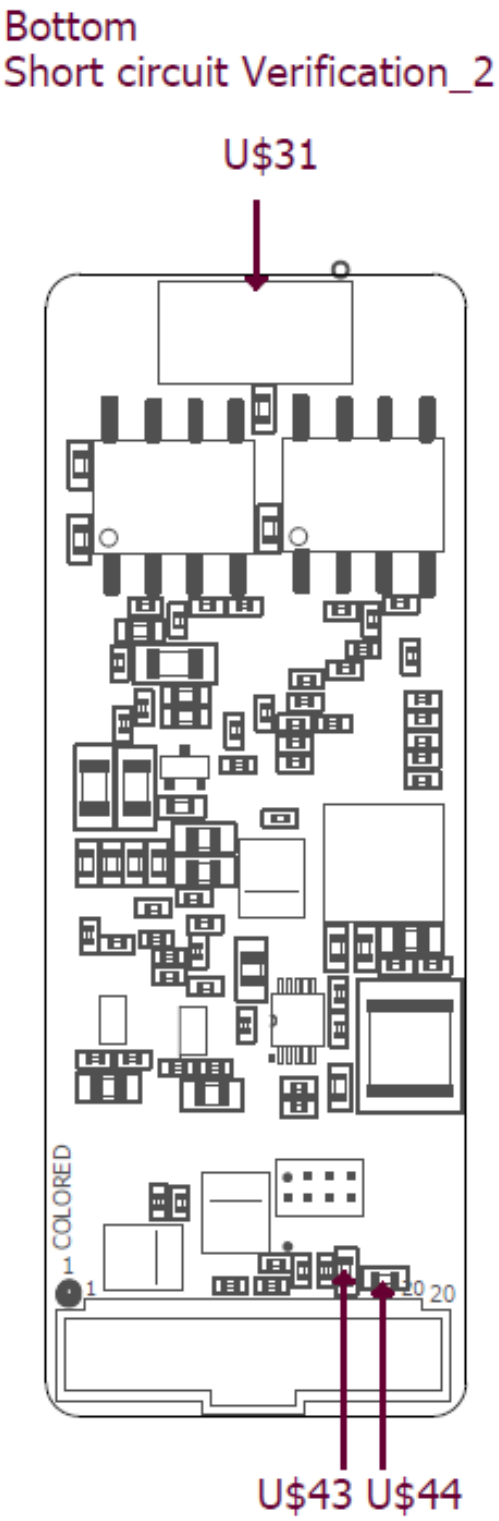


1.8 Verificación de Cortocircuitos 2

Realizar la verificación de cortocircuitos como se hizo en el punto anterior, pero esta vez se debe comprobar:

- Hay continuidad entre los pines de U$43
- Hay continuidad entre los pines de U$44
- No hay continuidad entre los pines 5V y 5V_ISO de U$31.
- No hay continuidad entre los pines GND e ISO_GND de U$31.



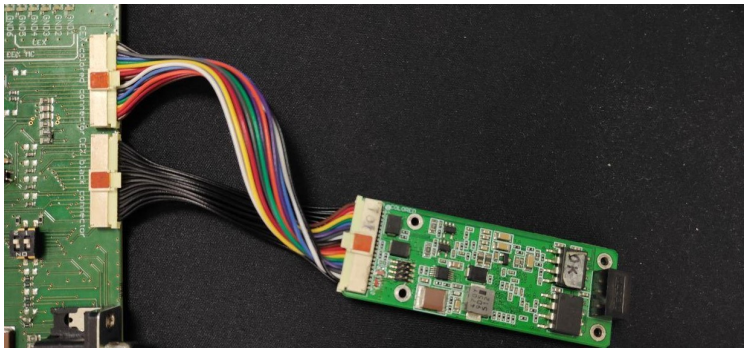

Bottom
Short circuit Verification_2
U$31
COLORED
1
1
20
20
U$43
U$44

1.9 Verificación de Corrientes y Voltajes

Realizar la verificación de corrientes y voltajes como se describe en Producción Manual ATP v1.2.7, sección 1.5. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Verificaci%C3%B3n%20de%20Corrientes%20y%20Voltajes/index.html

Se debe utilizar la PCB de Verificación Modo Slow de la siguiente manera:

- Conectar la PCB de Verificación al CAN Expander, conectando cada conector al lugar correspondiente.



- Conectar el jumper a la PCB de Verificación.
- Conectar la PCB de Verificación a una fuente de alimentación de 12 V, 24V o 36 V.
- Medir la corriente y los voltajes.
  - Los puntos a medir son los mismos que en Verificación de Cortocircuitos.
  - La señal de 5V_ISO se debe medir respecto a la señal ISO_GND.

1.10 Verificación de Programación

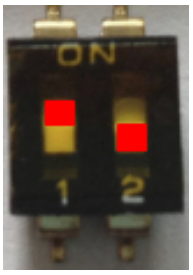
Realizar la verificación de programación como se describe en Producción Manual ATP v1.2.7, sección 1.6 para programar PETT. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Verificaci%C3%B3n%20de%20Programado/index.html

1.11 Verificación Funcional pre Burn In

La verificación funcional se tiene que realizar como se indica en Producción Manual ATP v1.2.7, sección 1.7. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Verificaci%C3%B3n%20Funcional/index.html

Se debe realizar utilizando la PCB de Verificación Modo Slow y las conexiones son las mismas que en el apartado Verificación de Corrientes y Voltajes.

Configurar los DIP switch en modo slow test:



El test pasa si todos los tests de PETT imprimen en pantalla un «Pass».

Los tests son:

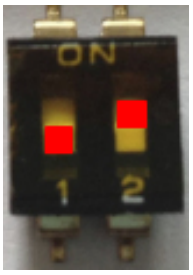
- Test Address
- V1_SEN
- V2_SEN
- Analogic 1 (3.3V)
- Analogic 2 (3.3V)
- Analogic 3 (5V)
- Analogic 4 (5V)
- Analogic 5 (12V)
- Analogic 6 (12V)
- Analogic 7 (36V)
- Analogic 8 (36V)
- UART B
- CAN A-B
- Test I2C (MMC5883MA External)
- PWM1 and PWM2 to ECAP1
- PWM3 and PWM4 to ECAP2
- PWM5 and PWM6 to ECAP3
- PWM7 and PWM8 to ECAP4
- Temperature CEX

1.12 Burn In Test

Realizar el Burn-in test como se detalla en Producción Manual ATP v1.2.7, sección 1.8. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Burn-in%20test/index.html

Se debe realizar utilizando la PCB de Verificación Modo Burn In y el cableado es el mismo que en el apartado Verificación de Corrientes y Voltajes.

Configurar los DIP switch en modo Burn-in:



El test pasa si se completan las 30 iteraciones del Burn In en Valkiria.

1.13 Verificación del Tropicalizado

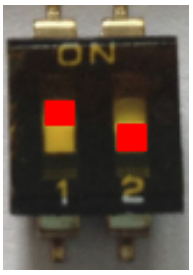
Para realizar la verificación del tropicalizado se tienen que seguir las instrucciones que se describen en Producción Manual ATP v1.2.7, sección 1.10. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Verificaci%C3%B3n%20Tropicalizado/index.html

1.14 Verificación Funcional post Tropicalizado

La verificación funcional se tiene que realizar como se indica en Producción Manual ATP v1.2.7, sección 1.7. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Verificaci%C3%B3n%20Funcional/index.html

Se debe realizar utilizando la PCB de Verificación Modo Slow y las conexiones son las mismas que en el apartado Verificación de Corrientes y Voltajes.

Configurar los DIP switch en modo slow test:



El test pasa si todos los tests de PETT imprimen en pantalla un «Pass».

Los tests son:

- Test Address
- V1_SEN
- V2_SEN
- Analogic 1 (3.3V)
- Analogic 2 (3.3V)
- Analogic 3 (5V)
- Analogic 4 (5V)

- Analogic 5 (12V)
- Analogic 6 (12V)
- Analogic 7 (36V)
- Analogic 8 (36V)
- UART B
- CAN A-B
- Test I2C (MMC5883MA External)
- PWM1 and PWM2 to ECAP1
- PWM3 and PWM4 to ECAP2
- PWM5 and PWM6 to ECAP3
- PWM7 and PWM8 to ECAP4
- Temperature CEX

1.15 Verificación Final

La verificación final se tiene que realizar como se indica en Producción Manual ATP v1.2.7, sección 1.21. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Verificaci%C3%B3n%20Final/index.html

CAPÍTULO 2

CEX Verification PCB 1.0

2.1 Objeto y Alcance

El presente procedimiento describe las actividades a llevar a cabo para las modificaciones de las PCBs de Verificación del CEX para adaptarlas a los distintos test de verificación de la línea de producción.

La CEXVerificationPCB 1.0 que se puede encontrar en: https://github.com/embention/CEX/releases/tag/release%2FCEXVerificationPCB%2F1.0

2.2 Modo Slow

El Modo Slow, por su parte, empieza a verificar cuando la persona que está verificando el producto lo indica por teclado. Como ambos modos funcionan por HW de la misma manera, la PCB de Verificación es la misma.

Para el modo Slow, los únicos cambios que se deben realizar en la PCB de verificación son:

- Modificar relay: se debe cortar la última conexión del relay (K1 en el sch) y soldar un cable entre ese pad y el segundo pad, como se indica en las siguientes imágenes.

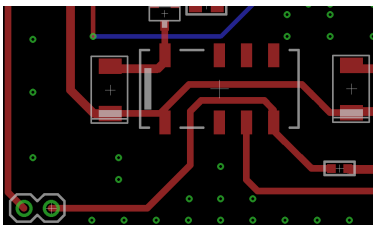


Figura 1: Relay antes de la modificación

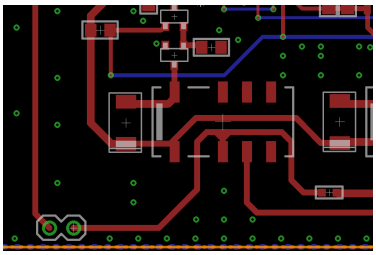


Figura 2: Relay después de la modificación

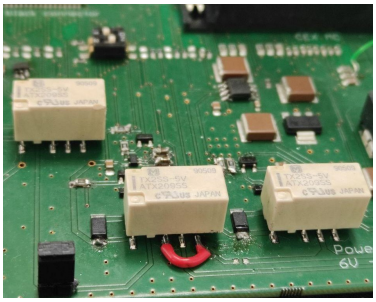


Figura 3: Relay modificado

- Cambiar alimentación del RS-232 driver: se debe cortar la conexión con la alimentación de 3.3V y conectar a la señal de 5V_SERVICE

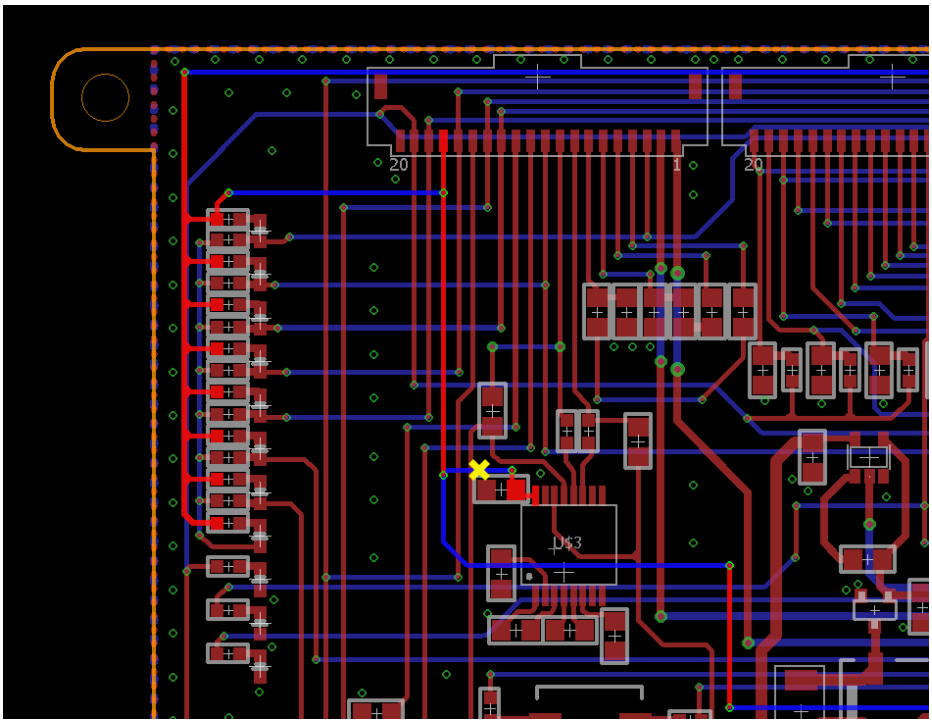


Figura 4: Cortar conexión a 3.3V en el .brd

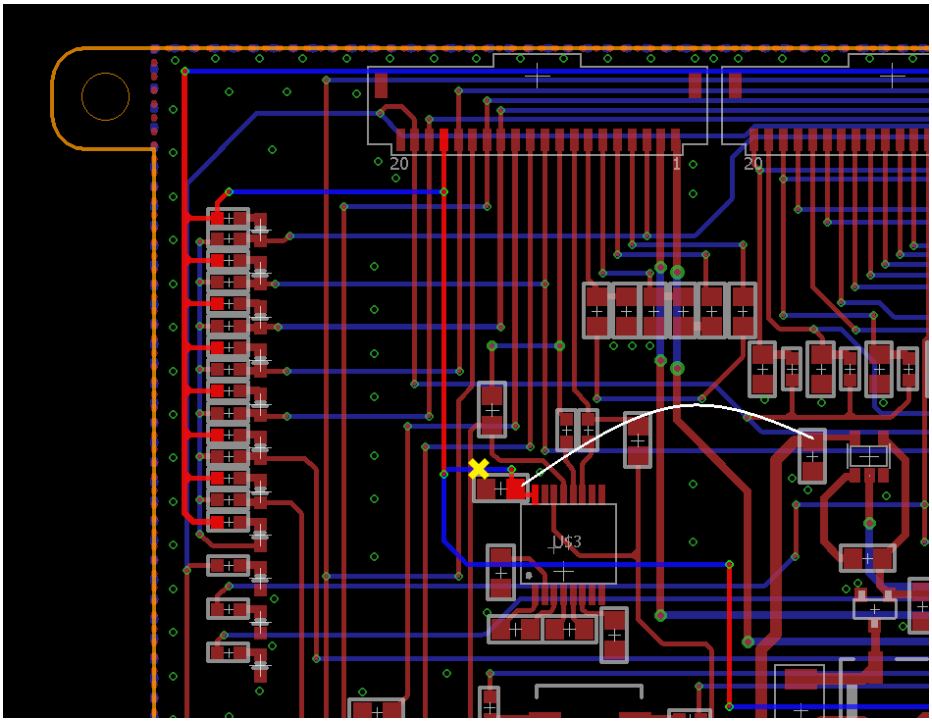


Figura 5: Conectar a 5V_SERVICE en el .brd

- Cambiar R de 0 Ohm de los PWM por R de 10 kOhm: se deben sustituir las resistencias de los PWM (R109, R110, R111, R112, R113, R114, R115 y R116 en el .sch) por resistencias de 10 kOhm.

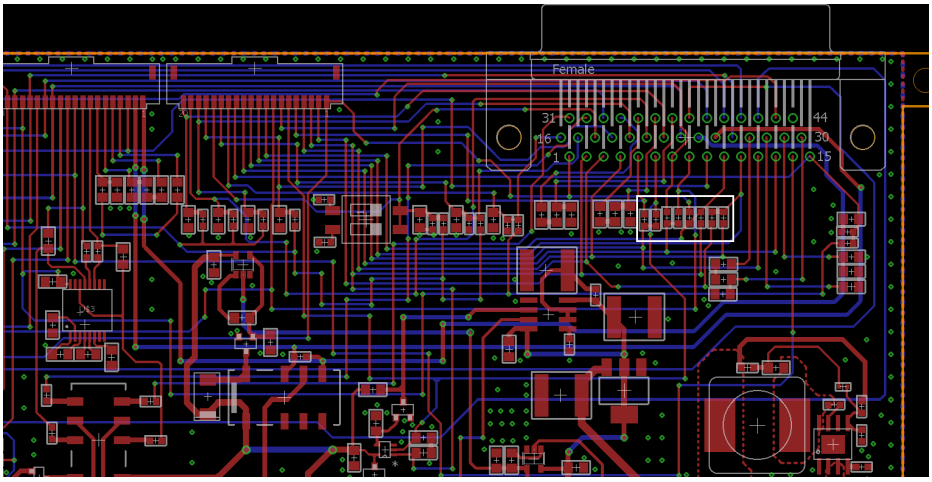


Figura 6: Resistencias de PWM en el .brd

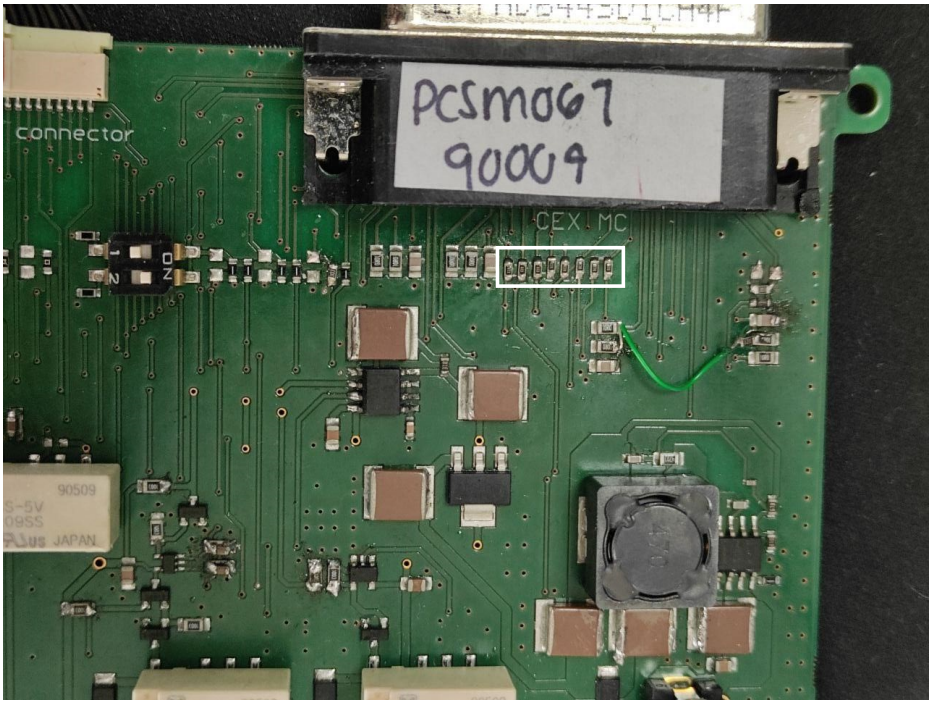


Figura 7: Resistencias de PWM en la PCB

- Cambiar las R de los LEDs de PASS y FAIL a 910 Ohm: se deben sustituir las resistencias de los LEDs de PASS y FAIL por R de 910 Ohm (o 1 kOhm)

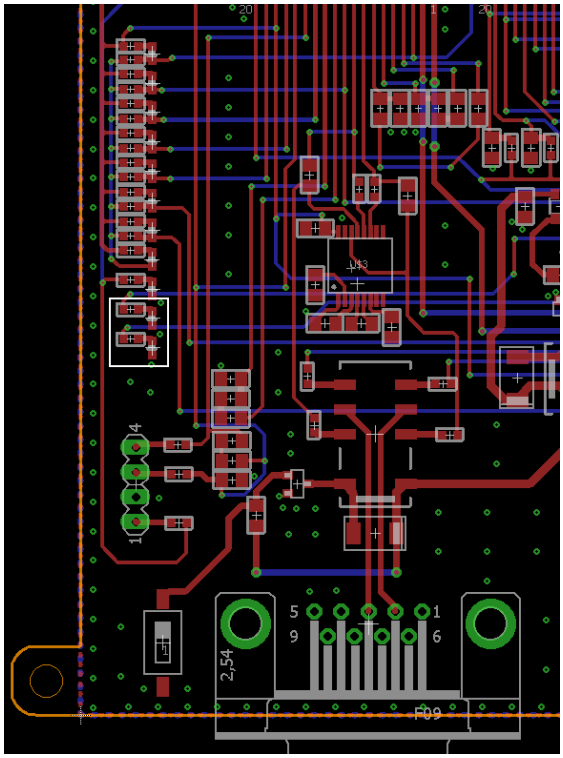


Figura 8: Resistencias de LEDs en el .brd

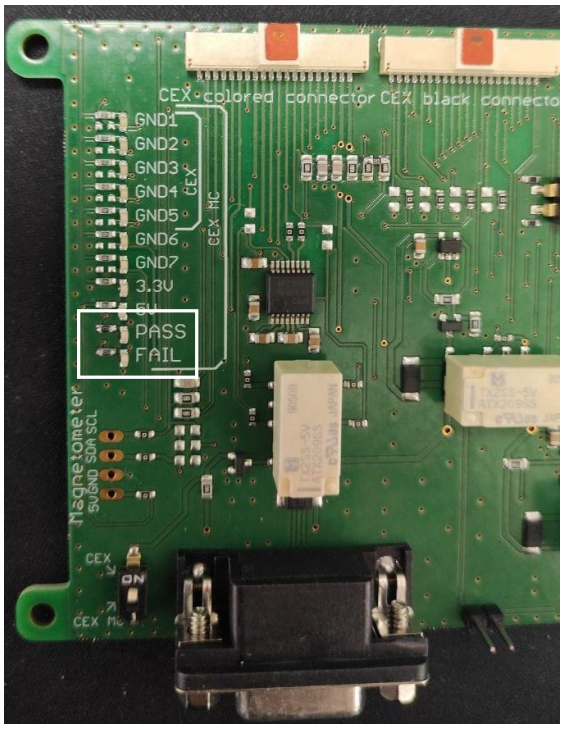


Figura 9: Resistencias de LEDs en la PCB

- Agregar R de 120 Ohm de terminación del CAN (tanto para el loopback del CEX como del CEX MC): se deben agregar resistencias de 120 Ohm entre CAN+ y CAN- del conector del CEX (R101 y R102) y del CEX MC (R305 y R307) como se indica en las siguientes imágenes.

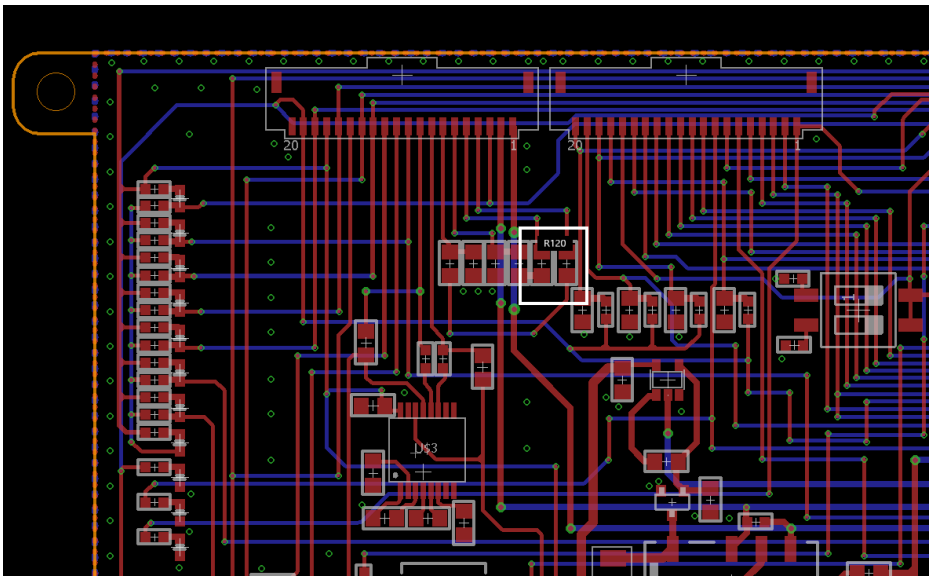


Figura 10: Resistencias de CAN del CEX en el .board

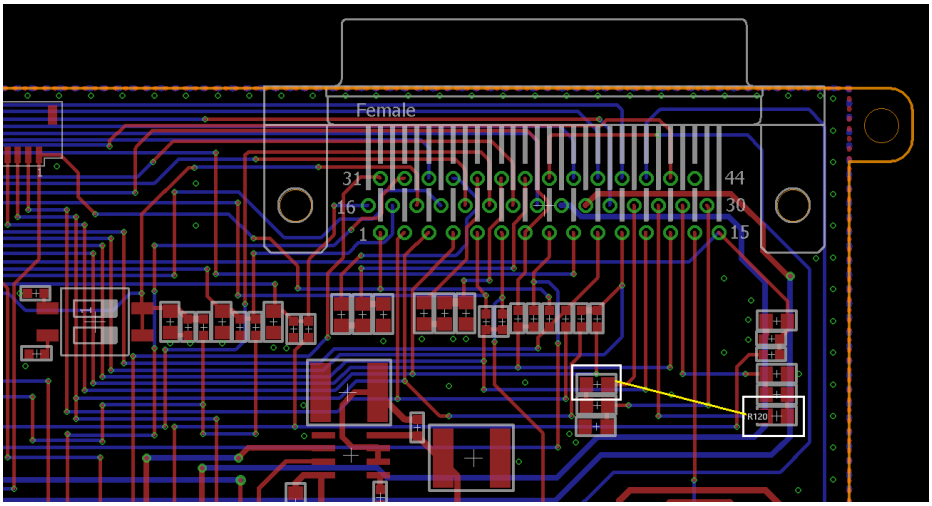


Figura 11: Resistencias de CAN del CEX MC en el .board

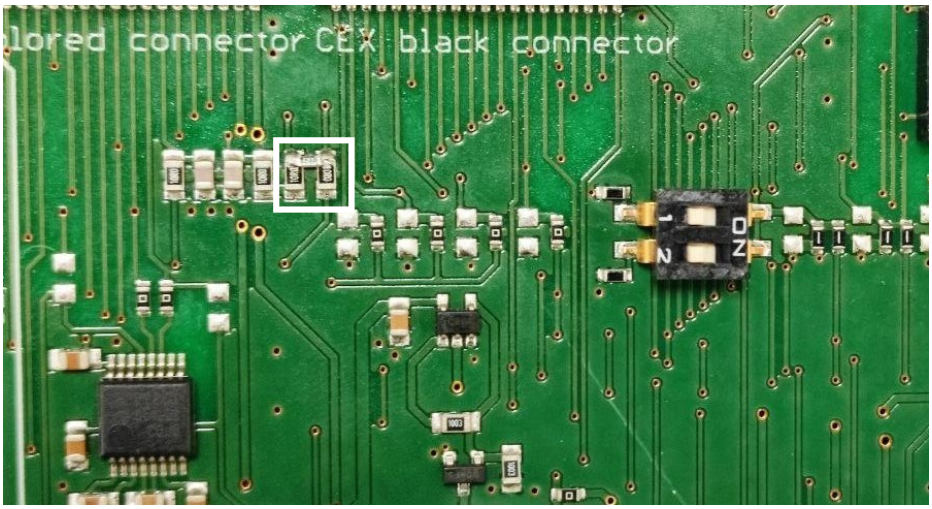


Figura 12: Resistencias de CAN del CEX en la PCB

Figura 13: Resistencias de CAN del CEX MC en la PCB

- Agregar conector para magnetómetro externo y para jumper de alimentación: se deben agregar los conectores como se indica en las siguientes imágenes.

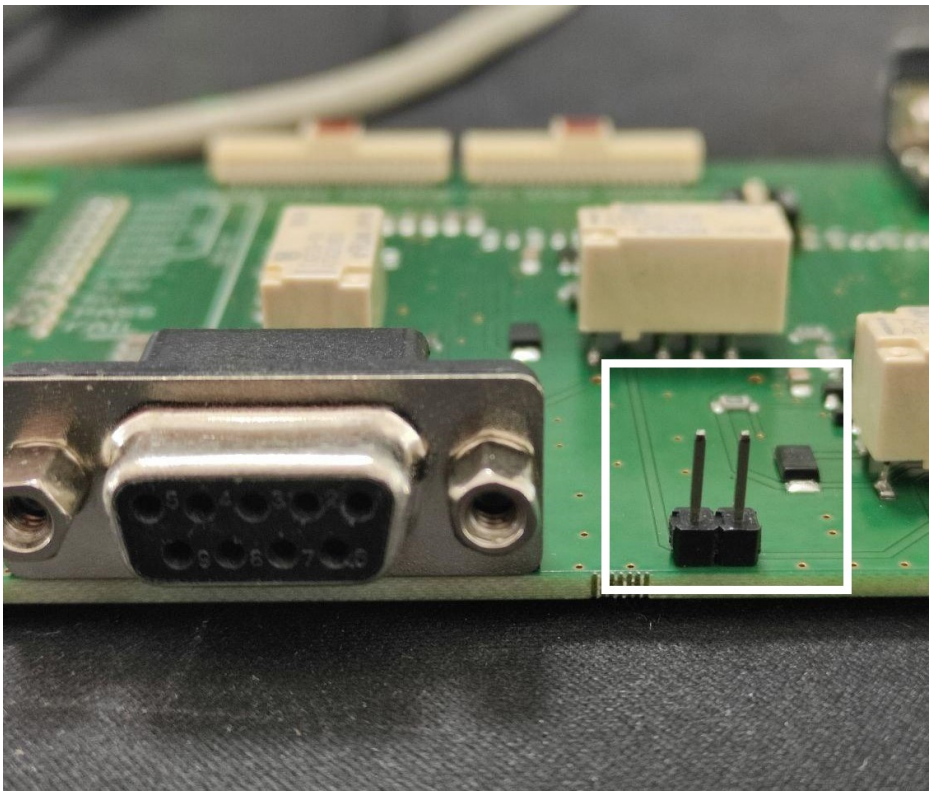


Figura 14: Conector para Jumper

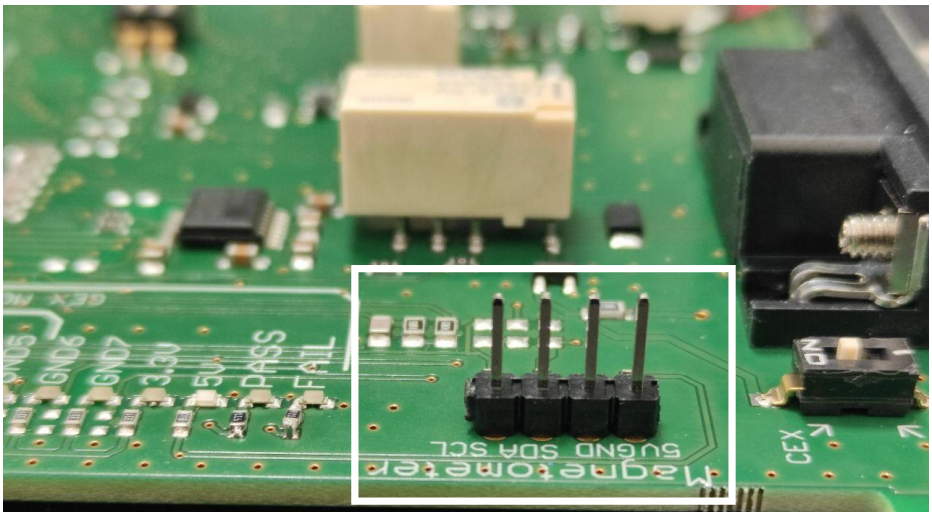


Figura 15: Conector para Magnetómetro

- Preparar el magnetómetro externo: se debe utilizar una PCB de un magnetómetro externo que contenga un

MMC5883MA y se debe conectar a la PCB de Verificación siguiendo la siguiente imagen.

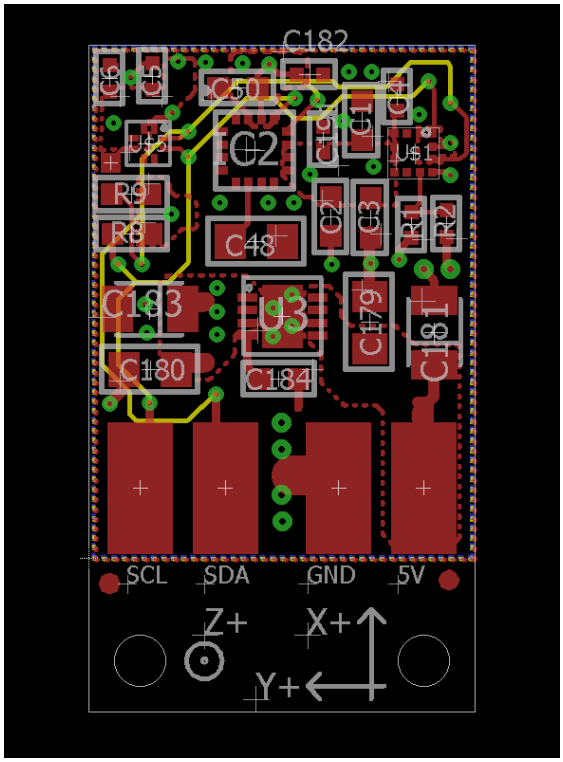


Figura 16: Board de la PCB del Magnetómetro externo

2.3 Modo BurnIn

El Modo Burn In consiste en tener la PCB de Verificación preparada para que el producto a verificar realice ciclos de encendido y apagado cambiando sus alimentaciones entre V1 y V2 y entre 7 V y 36 V. Para el modo burn in, además de los cambios anteriores se debe:

- Desoldar la resistencia de 0 Ohm que está entre V1 y V2 (R217):

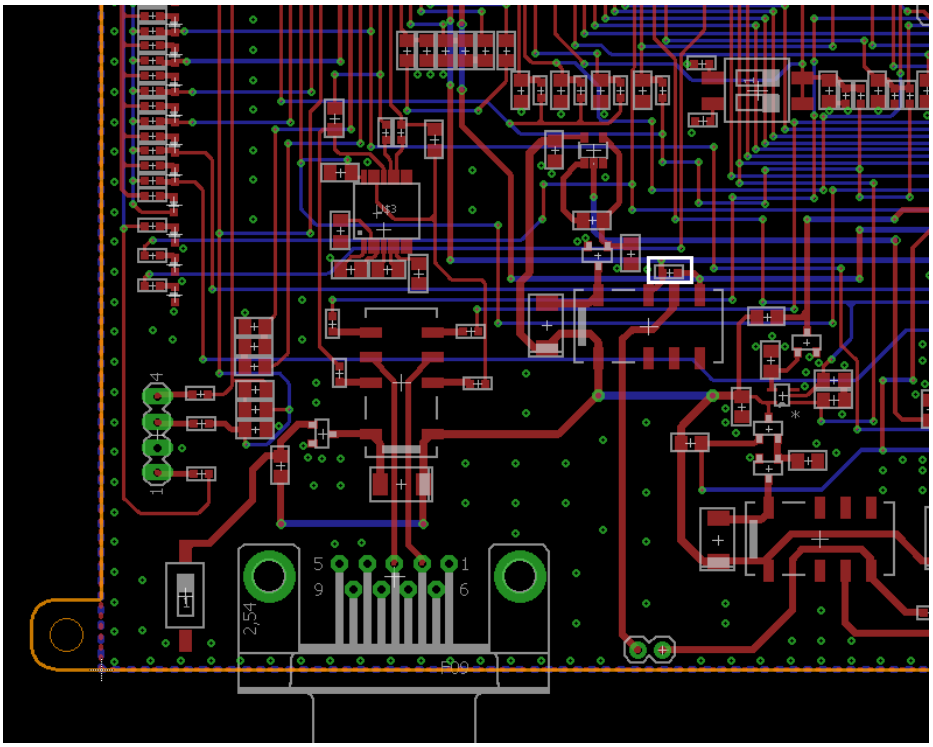


Figura 17: Resistencia a desoldar

- Agregar C de 47nF en paralelo (encima) a las resistencias de reset (R206 y R207)

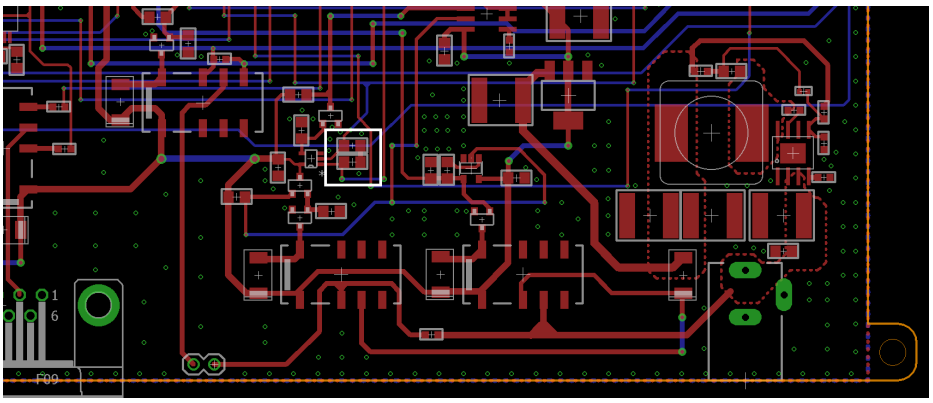


Figura 18: Resistencias de reset en el .brd

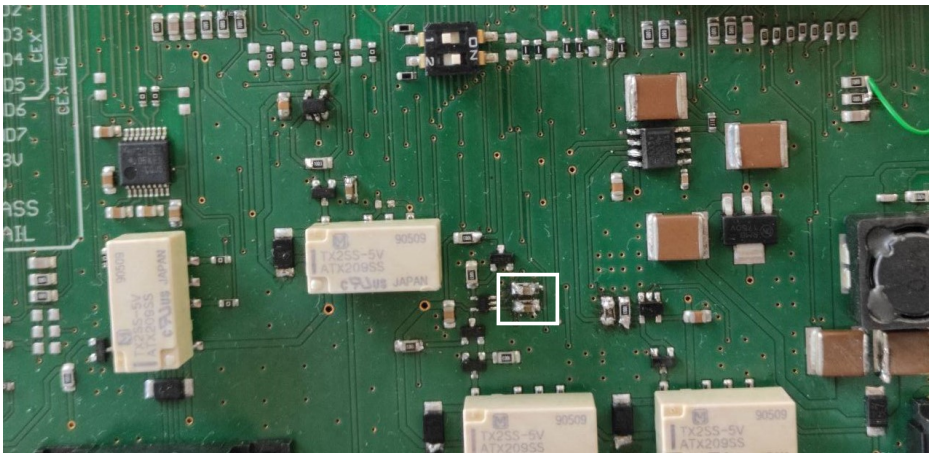


Figura 19: Resistencias de reset en la PCB

- Agregar una resistencia de 100 kOhm en serie entre la R de reset R206 y la entrada de la AND. Para ello:

  1º Cortar pista bottom

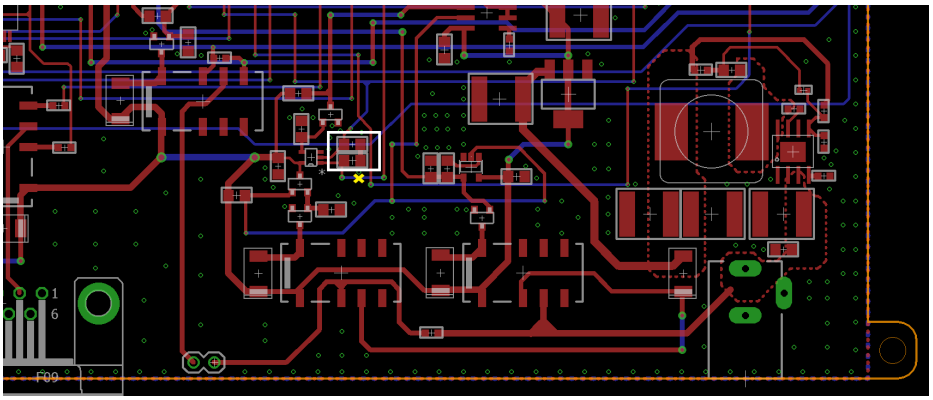


Figura 20: Pista a cortar

2º Soldar la resistencia de 100 kOhm.

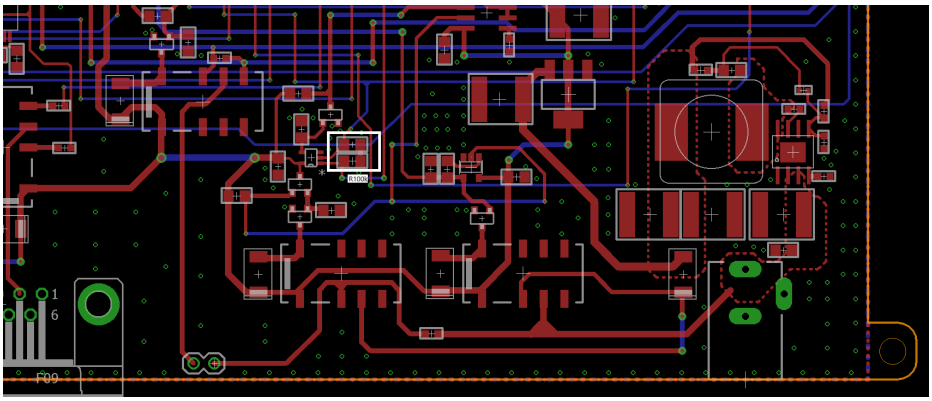


Figura 21: Soldar resistencia

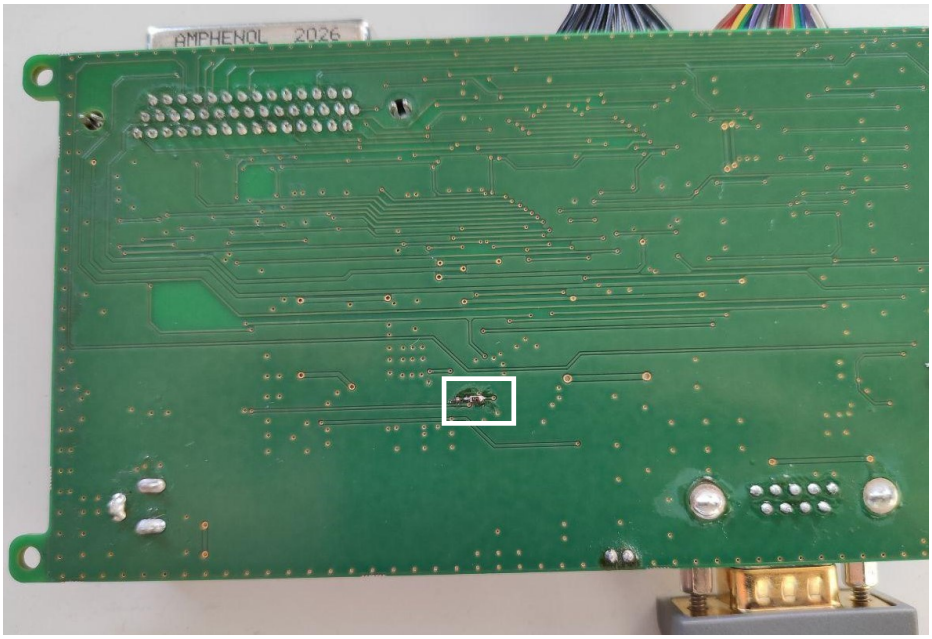


Figura 22: R de 100 kOhm soldada en serie

- Agregar una resistencia de 4.7 kOhm y un condensador de 330 nF entre el pin C (señal PWR_SWITCH) del flip-flop IC1 y GND:

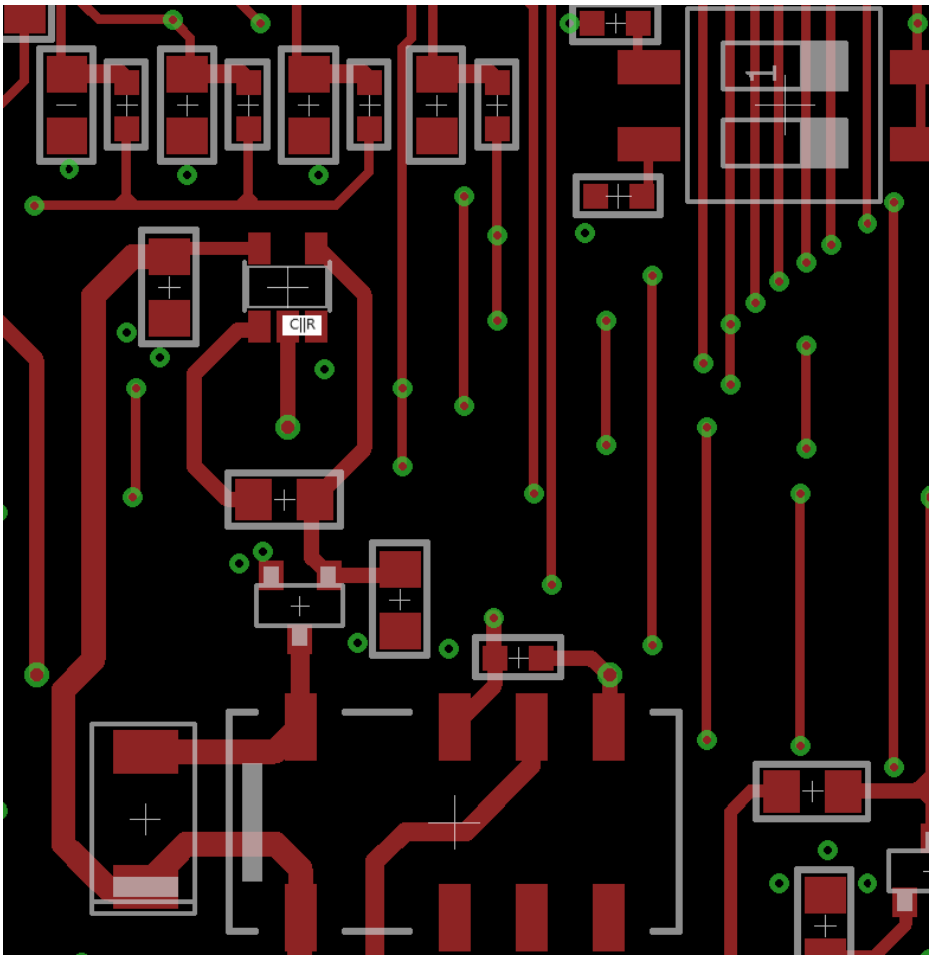


Figura 23: Agregar C (330 nF) y R (4.7 kOhm) en paralelo

- Agregar un condensador de tantalio de 4.7 uF en paralelo con el condensador de la alimentación del flip-flop de Supply Group Select (C203).

2.4 Modo Comunicación RS485

El modo Comunicación RS485 es para el caso en el que el CEX MC tiene RS485 y Jeti (PN CAN Expander MC Unit (UART HALF DUPLEX& RS485)v1.2) es necesaria una PCB de Verificación con otras modificaciones, porque en este caso, la comunicación con el ordenador es utilizando el estandar RS485.

Para ello a la PCB de Verificación se le deben hacer las siguientes modificaciones:

- Quitar el loopback de la señal RS485: para ello desoldar: R301, R302, C302, R303, R304 y C301.
- Quitar conexiones de RS232 con conector X2, desoldar: R315 y R317.
- Soldar cables para poder conectar el RS485-USB.
- Soldar cable para Jeti y conectarlo al pin digital 2 del Arduino.

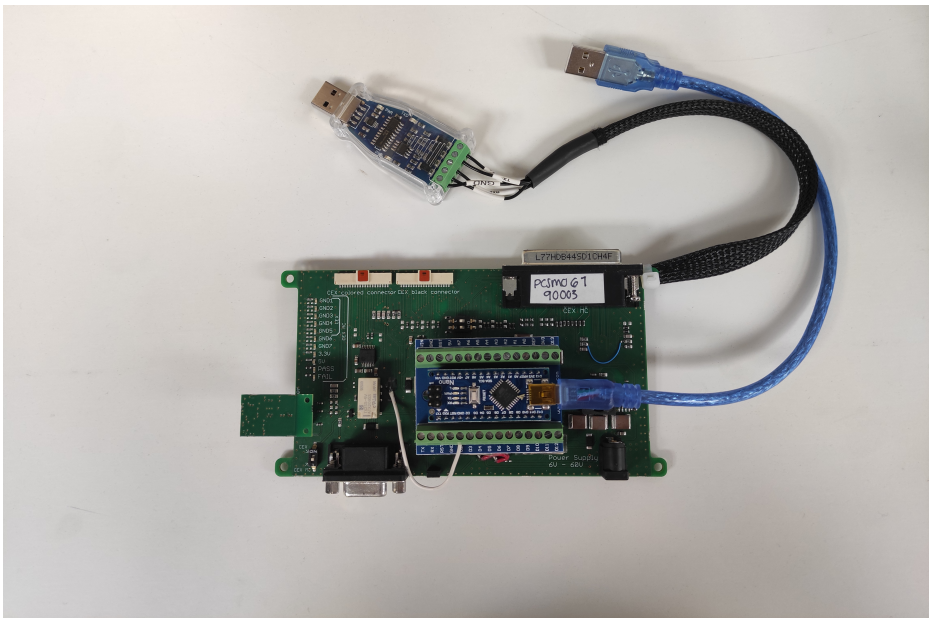


Figura 24: PCB de Verificación Modo Comunicación RS485

2.5 UART C Test

Para el test de la UART C, además de los cambios del Slow test, se debe:

- Soldar las resistencias y condensadores que están NOPOP originalmente: soldar dos resistencias de 10 Ohms y un condensador de 22 pF.

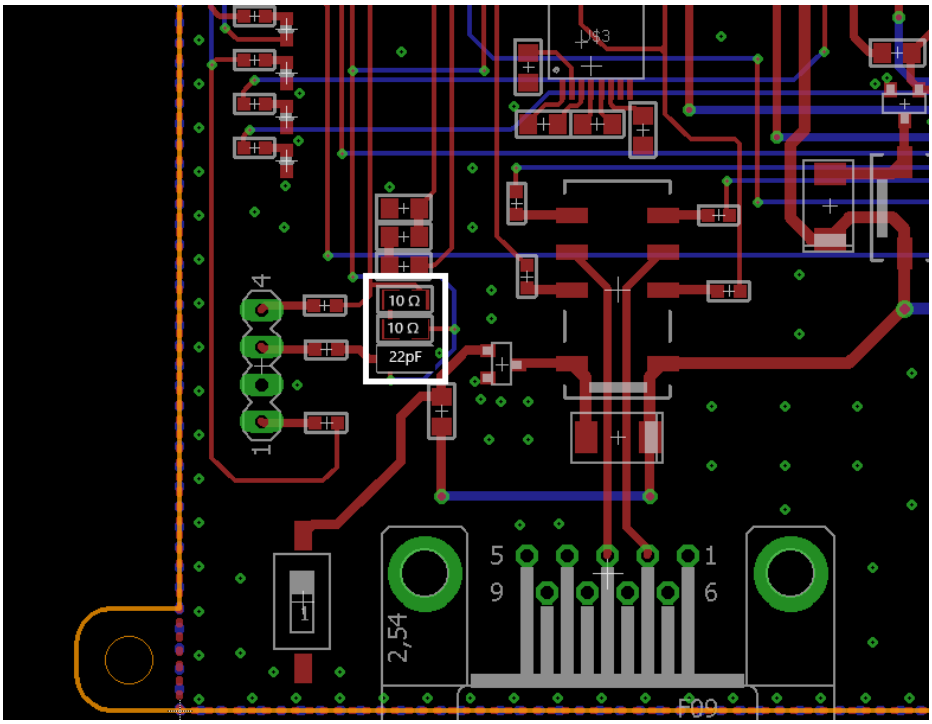


Figura 25: Resistencias y condensador a soldar en el .brd

G.2. CEX MC ATP

CEX MC ATP v2.5

Embention

03 de junio de 2021

Índice general

II

CAPÍTULO 1

CEX MC Verification 2.5

1.1 Objeto y Alcance

El presente procedimiento describe las actividades a llevar a cabo para la verificación de un CAN Expander MC (CEX MC). Este documento abarca las tareas de verificación desde la llegada de la PCB del CAN Expander a las instalaciones de Embention hasta justo antes del envío al cliente.

El objetivo de este procedimiento es la detección precoz de CAN Expanders que puedan venir defectuosos por parte del montador, evitando así la entrega al cliente de productos defectuosos.

Para la realización de la Verificación se necesita una PCB de Verificación 1.0 Modo Slow y otra Modo Comunicación RS485 y el sw PETT del CEX.

1.2 Verificación del CAN Expander

Para la verificación del CAN Expander se deben seguir los pasos que se describen en el siguiente enlace: https://documentation.embention.net/CEXATP/es/latest/CEX/index.html hasta la «Verificación Funcional post Tropicalizado».

1.3 Verificación del Montaje en Caja

Para realizar la verificación del montaje en caja se tienen que seguir las instrucciones que se describen en Producción Manual ATP v1.2.7, sección 1.12. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Verificaci%C3%B3n%20Montaje%20en%20caja/index.html

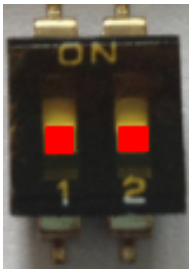
1.4 Verificación Funcional post-Montaje en Caja

La verificación funcional se tiene que realizar como se indica en Producción Manual ATP v1.2.7, sección 1.7. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Verificaci%C3%B3n%20Funcional/index.html

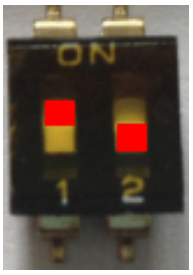
Se debe realizar utilizando la PCB de Verificación y se debe conectar el CEX MC al conector correspondiente.

A continuación, seleccionar Modo CEX MC, como se observa en la siguiente imagen.

Si el CEX a Verificar es un CAN Expander MC Unit (UART HALF DUPLEX& RS485)v1.2, configruar los DIP switch en Modo Comunicación RS485 y utilizar la PCB de Verificación de dicho modo:



Para los otros casos, configurar los DIP switch en Modo Slow Test y utilizar la PCB de Verificación correspondiente:



El test pasa si todos los tests de PETT imprimen en pantalla un «Pass».

Los tests son:

- Test Address
- V1_SEN
- V2_SEN
- Analogic 1 (3.3V)

- Analogic 2 (3.3V)
- Analogic 3 (5V)
- Analogic 4 (5V)
- Analogic 5 (12V)
- Analogic 6 (12V)
- Analogic 7 (36V)
- Analogic 8 (36V)
- UART B (Test Jeti, en el Communication RS485 Mode)
- CAN A-B
- Test I2C (MMC5883MA External)
- PWM1 and PWM2 to ECAP1
- PWM3 and PWM4 to ECAP2
- PWM5 and PWM6 to ECAP3
- PWM7 and PWM8 to ECAP4
- Temperature CEX

1.5 Verificación Final

La verificación final se tiene que realizar como se indica en Producción Manual ATP v1.2.7, sección 1.21. También se puede encontrar en: https://documentation.embention.net/produccion/es/latest/ATP/ATP/Verificaci%C3%B3n%20Final/index.html

ANEXO H:

ACCEPTANCE TEST RESULT (ATR)

CEX Autopilot Acceptance Test Report	ATR CEX - v2.5

Production Order	
PN	
SN	
Date	
Laboratory Temperature (ºC)	
Laboratory Humidity (%)	

Product	
Veronte CAN Expander v1.2	☑
CAN Expander MC Unit (RS232 & RS485)v1.2	☐
CAN Expander MC Unit (UART & RS485)v1.2	☐
CAN Expander MC Unit (UART HALF DUPLEX& RS485)v1.2	☐

Reference Documents	CEXATP 2.5

EMBENTION

VISUAL INSPECTION PRE-VIBRATION							
Component	Assembled:		Polarisation:		Aligned/ Cold welding		Result
	PASS	FAIL	PASS	FAIL	PASS	FAIL	
U$7	☑	☐	☑	☐	☑	☐	OK
U$41	☑	☐	☑	☐	☑	☐	OK
U$26	☑	☐	☑	☐	☑	☐	OK
U$5	☑	☐	☑	☐	☑	☐	OK
U$6	☑	☐	☑	☐	☑	☐	OK
C183	☑	☐	☑	☐	☑	☐	OK
U2	☑	☐	☑	☐	☑	☐	OK
C181	☑	☐	☑	☐	☑	☐	OK
C25	☑	☐	☑	☐	☑	☐	OK
C50	☑	☐	☑	☐	☑	☐	OK
C18	☑	☐	☑	☐	☑	☐	OK
U3	☑	☐	☑	☐	☑	☐	OK
C33	☑	☐	☑	☐	☑	☐	OK
C35	☑	☐	☑	☐	☑	☐	OK
U7	☑	☐	☑	☐	☑	☐	OK
U6	☑	☐	☑	☐	☑	☐	OK
U$31	☑	☐	☑	☐	☑	☐	OK
U4	☑	☐	☑	☐	☑	☐	OK
U$28	☑	☐	☑	☐	☑	☐	OK
U1	☑	☐	☑	☐	☑	☐	OK
U5	☑	☐	☑	☐	☑	☐	OK
U$4	☑	☐	☑	☐	☑	☐	OK
U$52	☑	☐	☑	☐	☑	☐	OK
Others	☑	☐	☑	☐	☑	☐	OK

Comments	

I certify that the previous steps have been performed according to the approved procedures and specifications	
Technician	
Date	
Signature	

SHAKING TEST		
Shaking Test Accomplished		Result
PASS	FAIL	
☑	☐	OK

Comments	

Technician	
Date	
Signature	

EMBENTION

VISUAL INSPECTION POST-VIBRATION			
Component	Assembled/ Aligned		Result
	PASS	FAIL	
U$7	☑	☐	OK
U$41	☑	☐	OK
U$26	☑	☐	OK
U$5	☑	☐	OK
U$6	☑	☐	OK
U2	☑	☐	OK
U3	☑	☐	OK
U7	☑	☐	OK
U6	☑	☐	OK
U$31	☑	☐	OK
U4	☑	☐	OK
U$28	☑	☐	OK
U1	☑	☐	OK
U5	☑	☐	OK
U$4	☑	☐	OK
U$52	☑	☐	OK
Others	☑	☐	OK

Comments	

Technician	
Date	
Signature	

EMBENTION

SHORT CIRCUIT VERIFICATION			
Signal	No short circuit		Result
	PASS	FAIL	
V1	☑	☐	OK
V2	☑	☐	OK
5V	☑	☐	OK
5V_ISO	☑	☐	OK
3.3V	☑	☐	OK
3.3VA	☑	☐	OK
1.9V	☑	☐	OK
1.9VA	☑	☐	OK

Tool	ID
Multimeter	ID

Comments	

Technician	
Date	
Signature	

EMBENTION

SHORT CIRCUIT VERIFICATION			
Signal	Short circuit		Result
	PASS	FAIL	
No continuity between pin 2 and 4 U$31	☑	☐	OK
No continuity between pin 1 and 3 U$31	☑	☐	OK
Continuity between pins U$43	☑	☐	OK
Continuity between pins U$44	☑	☐	OK

Tool	ID
Multimeter	ID

Comments	

Technician	
Date	
Signature	

VOLTAGES VERIFICATION				
Signal	Voltage range [V]		Measured voltage [V]	Result
	Min	Max		
V1	5,95	8,05		NOK
V2	5,95	8,05		NOK
5V	4,25	5,75		NOK
5V_ISO	4,25	5,75		NOK
3.3V	3,14	3,47		NOK
3.3VA	3,14	3,47		NOK
1.9V	1,81	2,00		NOK
1.9VA	1,81	2,00		NOK

CURRENTS VERIFICATION				
Signal	Current range [mA]		Measured current [mA]	Result
	Min	Max		
I CEX	90,00	145,00		NOK

Voltage measurement Tool	ID
Multimeter	ID

Laboratory Temperature (ºC)	
Laboratory Humidity (%)	

Current measurement Tool	ID
Multimeter	ID

Comments	

Technician	
Date	
Signature	

EMBENTION

CEX HAS BEEN PROGRAMMED			
Programmed	Test result		Result
	PASS	FAIL	
PETT	☑	☐	OK

Comments	

Technician	
Date	
Signature	

EMBENTION

FUNCTIONAL VERIFICATION PRE-BURN-IN				
Test number	Test peripheral	Test result		Result
		PASS	FAIL	
1	Test Address	☑	☐	OK
2	V1_SEN	☑	☐	OK
3	V2_SEN	☑	☐	OK
4	Analogic 1 (3.3V)	☑	☐	OK
5	Analogic 2 (3.3V)	☑	☐	OK
6	Analogic 3 (5V)	☑	☐	OK
7	Analogic 4 (5V)	☑	☐	OK
8	Analogic 5 (12V)	☑	☐	OK
9	Analogic 6 (12V)	☑	☐	OK
10	Analogic 7 (36V)	☑	☐	OK
11	Analogic 8 (36V)	☑	☐	OK
12	UART B	☑	☐	OK
13	CAN A-B	☑	☐	OK
14	Test I2C (MMC5883MA External)	☑	☐	OK
15	PWM1 and PWM2 to ECAP1	☑	☐	OK
16	PWM3 and PWM4 to ECAP2	☑	☐	OK
17	PWM5 and PWM6 to ECAP3	☑	☐	OK
18	PWM7 and PWM8 to ECAP4	☑	☐	OK
19	Temperature CEX	☑	☐	OK

Comments	

Technician	
Date	
Signature	

EMBENTION

BURN-IN TEST (65ºC)				
Test number	Test peripheral	Test result		Result
		PASS	FAIL	
1	Test Address	☑	☐	OK
2	V1_SEN	☑	☐	OK
3	V2_SEN	☑	☐	OK
4	Analogic 1 (3.3V)	☑	☐	OK
5	Analogic 2 (3.3V)	☑	☐	OK
6	Analogic 3 (5V)	☑	☐	OK
7	Analogic 4 (5V)	☑	☐	OK
8	Analogic 5 (12V)	☑	☐	OK
9	Analogic 6 (12V)	☑	☐	OK
10	Analogic 7 (36V)	☑	☐	OK
11	Analogic 8 (36V)	☑	☐	OK
12	UART B	☑	☐	OK
13	CAN A-B	☑	☐	OK
14	Test I2C (MMC5883MA External)	☑	☐	OK
15	PWM1 and PWM2 to ECAP1	☑	☐	OK
16	PWM3 and PWM4 to ECAP2	☑	☐	OK
17	PWM5 and PWM6 to ECAP3	☑	☐	OK
18	PWM7 and PWM8 to ECAP4	☑	☐	OK
19	Temperature CEX	☑	☐	OK
20	Burn in Test Accomplished (65ºC)	☑	☐	OK

Comments	

Technician	
Date	
Signature	

COATING VERIFICATION		
CEX has been coated		Result
OK	NOK	
☑	☐	OK

Comments	

Technician	
Date	
Signature	

EMBENTION

FUNCTIONAL VERIFICATION POST-COATING				
Test number	Test peripheral	Test result		Result
		PASS	FAIL	
1	Test Address	☑	☐	OK
2	V1_SEN	☑	☐	OK
3	V2_SEN	☑	☐	OK
4	Analogic 1 (3.3V)	☑	☐	OK
5	Analogic 2 (3.3V)	☑	☐	OK
6	Analogic 3 (5V)	☑	☐	OK
7	Analogic 4 (5V)	☑	☐	OK
8	Analogic 5 (12V)	☑	☐	OK
9	Analogic 6 (12V)	☑	☐	OK
10	Analogic 7 (36V)	☑	☐	OK
11	Analogic 8 (36V)	☑	☐	OK
12	UART B	☑	☐	OK
13	CAN A-B	☑	☐	OK
14	Test I2C (MMC5883MA External)	☑	☐	OK
15	PWM1 and PWM2 to ECAP1	☑	☐	OK
16	PWM3 and PWM4 to ECAP2	☑	☐	OK
17	PWM5 and PWM6 to ECAP3	☑	☐	OK
18	PWM7 and PWM8 to ECAP4	☑	☐	OK
19	Temperature CEX	☑	☐	OK

Comments	

Technician	
Date	
Signature	

ENCLOSING VERIFICATION			
Task	PASS	FAIL	Result
Fixing glue in screws	☑	☐	N/A
IP sealing	☑	☐	N/A
Damage in enclosure	☑	☐	N/A

Comments	

Technician	
Date	
Signature	

EMBENTION

FUNCTIONAL VERIFICATION POST COATING				
Test number	Test peripheral	Test result		Result
		PASS	FAIL	
1	Test Address	☑	☐	N/A
2	V1_SEN	☑	☐	N/A
3	V2_SEN	☑	☐	N/A
4	Analogic 1 (3.3V)	☑	☐	N/A
5	Analogic 2 (3.3V)	☑	☐	N/A
6	Analogic 3 (5V)	☑	☐	N/A
7	Analogic 4 (5V)	☑	☐	N/A
8	Analogic 5 (12V)	☑	☐	N/A
9	Analogic 6 (12V)	☑	☐	N/A
10	Analogic 7 (36V)	☑	☐	N/A
11	Analogic 8 (36V)	☑	☐	N/A
12	UART B	☑	☐	N/A
13	CAN A-B	☑	☐	N/A
14	Test I2C (MMC5883MA External)	☑	☐	N/A
15	PWM1 and PWM2 to ECAP1	☑	☐	N/A
16	PWM3 and PWM4 to ECAP2	☑	☐	N/A
17	PWM5 and PWM6 to ECAP3	☑	☐	N/A
18	PWM7 and PWM8 to ECAP4	☑	☐	N/A
19	Temperature CEX	☑	☐	N/A

Comments	

Technician	
Date	
Signature	

EMBENTION

FINAL CHECK			
Test peripheral	Test result		Result
	PASS	FAIL	
Previous stages inspection	☑	☐	OK
Visual inspection	☑	☐	OK
Torque Check	☑	☐	OK
Free object check	☑	☐	OK

Comments	

Technician	
Production Manager	
Date	
Signature	

EMBENTION

FINAL CHECK				
	ID		Tool	ID
Multimeter	TST001		Oscilloscope	OSC002
Multimeter	TST005			
Multimeter	TST009			
Multimeter	TST003			
Multimeter	TST004			
Multimeter	TST002			
Multimeter	TST010			
Multimeter	TST011			
Multimeter	TST014			
Multimeter	TST015			
Multimeter	TST016			

MATERIAL CONSUMPTION						
Stage	**Description**	**#INTREF**	**Batch**	**Amount**	**Date**	**Technician**

Technician	
Date	
Signature	