The TLF35584 [PMIC](https://www.google.com/search?cs=1&sca_esv=404b834684c619cd&sxsrf=AE3TifOF5CLc1AI6RqqGDWot8iRpQgOkLQ%3A1759852318384&q=PMIC&sa=X&ved=2ahUKEwi0goDiuJKQAxW3RmwGHdMmEjwQxccNegQIAhAB&mstk=AUtExfAjliY5XvNBGzL0JekVH6jYMk1eBZh1IScgeck0cyElOwNaRnzX5PNevC4vXi1Vd86P5egdMjsP8CaiYZ9r3342Bbbqqg14lccQ_hJbL-eOCy5EFipODjCZyCgnIUxkl259xa6ztEmp2FwmNrof2E1aNED1kFX3_GK2RuxFFgvOCwg&csui=3) stays in INIT mode due to an expired window watchdog timer, a detected fault on the ERR pin (like an unconnected ERR hardware connection), under-voltage (UV) events below the BUCK converter threshold, or high microcontroller current (QUC) consumption that prevents the device from leaving INIT state. To resolve this, check the ERR pin connection, confirm the watchdog is correctly served and not expired, and ensure the microcontroller's QUC current is below the specified threshold.

Common Causes for Remaining in INIT Mode

* [**Window Watchdog**](https://www.google.com/search?cs=1&sca_esv=404b834684c619cd&sxsrf=AE3TifOF5CLc1AI6RqqGDWot8iRpQgOkLQ%3A1759852318384&q=Window+Watchdog&sa=X&ved=2ahUKEwi0goDiuJKQAxW3RmwGHdMmEjwQxccNegQIDBAB&mstk=AUtExfAjliY5XvNBGzL0JekVH6jYMk1eBZh1IScgeck0cyElOwNaRnzX5PNevC4vXi1Vd86P5egdMjsP8CaiYZ9r3342Bbbqqg14lccQ_hJbL-eOCy5EFipODjCZyCgnIUxkl259xa6ztEmp2FwmNrof2E1aNED1kFX3_GK2RuxFFgvOCwg&csui=3)**Timer Expiration:**

The watchdog timer is not being triggered correctly or in time, causing it to time out.

* [**ERR**](https://www.google.com/search?cs=1&sca_esv=404b834684c619cd&sxsrf=AE3TifOF5CLc1AI6RqqGDWot8iRpQgOkLQ%3A1759852318384&q=ERR&sa=X&ved=2ahUKEwi0goDiuJKQAxW3RmwGHdMmEjwQxccNegQIDRAB&mstk=AUtExfAjliY5XvNBGzL0JekVH6jYMk1eBZh1IScgeck0cyElOwNaRnzX5PNevC4vXi1Vd86P5egdMjsP8CaiYZ9r3342Bbbqqg14lccQ_hJbL-eOCy5EFipODjCZyCgnIUxkl259xa6ztEmp2FwmNrof2E1aNED1kFX3_GK2RuxFFgvOCwg&csui=3)**Pin Fault:**

A fault on the ERR pin, such as it being unconnected or receiving an invalid signal, will cause the PMIC to enter INIT state.

* [**Under-Voltage**](https://www.google.com/search?cs=1&sca_esv=404b834684c619cd&sxsrf=AE3TifOF5CLc1AI6RqqGDWot8iRpQgOkLQ%3A1759852318384&q=Under-Voltage&sa=X&ved=2ahUKEwi0goDiuJKQAxW3RmwGHdMmEjwQxccNegQIFRAB&mstk=AUtExfAjliY5XvNBGzL0JekVH6jYMk1eBZh1IScgeck0cyElOwNaRnzX5PNevC4vXi1Vd86P5egdMjsP8CaiYZ9r3342Bbbqqg14lccQ_hJbL-eOCy5EFipODjCZyCgnIUxkl259xa6ztEmp2FwmNrof2E1aNED1kFX3_GK2RuxFFgvOCwg&csui=3)**(UV) Events:**

The input voltage to the buck converter drops low enough to trigger a UV event but not low enough for power down, preventing the device from exiting INIT.

* [**Microcontroller**](https://www.google.com/search?cs=1&sca_esv=404b834684c619cd&sxsrf=AE3TifOF5CLc1AI6RqqGDWot8iRpQgOkLQ%3A1759852318384&q=Microcontroller&sa=X&ved=2ahUKEwi0goDiuJKQAxW3RmwGHdMmEjwQxccNegQICxAB&mstk=AUtExfAjliY5XvNBGzL0JekVH6jYMk1eBZh1IScgeck0cyElOwNaRnzX5PNevC4vXi1Vd86P5egdMjsP8CaiYZ9r3342Bbbqqg14lccQ_hJbL-eOCy5EFipODjCZyCgnIUxkl259xa6ztEmp2FwmNrof2E1aNED1kFX3_GK2RuxFFgvOCwg&csui=3)**Current Consumption:**

High current drawn by the [microcontroller](https://www.google.com/search?cs=1&sca_esv=404b834684c619cd&sxsrf=AE3TifOF5CLc1AI6RqqGDWot8iRpQgOkLQ%3A1759852318384&q=microcontroller&sa=X&ved=2ahUKEwi0goDiuJKQAxW3RmwGHdMmEjwQxccNegQIEhAB&mstk=AUtExfAjliY5XvNBGzL0JekVH6jYMk1eBZh1IScgeck0cyElOwNaRnzX5PNevC4vXi1Vd86P5egdMjsP8CaiYZ9r3342Bbbqqg14lccQ_hJbL-eOCy5EFipODjCZyCgnIUxkl259xa6ztEmp2FwmNrof2E1aNED1kFX3_GK2RuxFFgvOCwg&csui=3) from the QUC regulator can keep the device from starting up correctly.

Troubleshooting Steps

1. **Check the Watchdog:**
   * **Ensure proper triggering:** Verify that the window watchdog is being served correctly and triggered within the allowed window.
   * **Confirm configuration:** Check the window watchdog settings, such as the open and closed window periods, using the correct SPI unlocking sequence.
2. **Verify the ERR Pin:**
   * **Check hardware connection:** Ensure the ERR pin is correctly connected to its associated hardware (e.g., SMU\_FSP).
   * **Monitor the signal:** Check for any faults or invalid input signals on the ERR pin.
3. **Inspect Power Supply:**
   * **Monitor UV events:** Look for any voltage drops at the VSx pins that might be causing a UV event, even if they are not below the power down threshold.
4. **Reduce Microcontroller Power:**
   * **Monitor QUC current:** Ensure the current drawn from the QUC regulator by the microcontroller is below the ILDO\_µC, att threshold.
   * **Optimize microcontroller power:** Put the microcontroller in sleep mode or disable unused peripherals to reduce its current consumption.

To resolve an Infineon TLF35584 staying in the INIT state,

the host microcontroller must properly initialize and service the device within the 600ms INIT timer window. Common causes for being stuck in the INIT state include a slow power-up ramp, incorrect watchdog handling, or communication errors.

Top causes and solutions

1. Failure to service the watchdog and ERR monitoring

Upon entering the INIT state, the TLF35584 automatically enables its window watchdog (WWD) and ERR monitoring. If the host microcontroller does not properly configure and service these features within 600ms, the device will trigger a fault. After three consecutive initialization timer failures, it will enter the FAILSAFE mode.

**Solution:**

* Implement a full initialization sequence over SPI within the 600ms window.
* The sequence should include clearing existing error flags, unlocking the device registers, and configuring the watchdog and ERR pin monitoring settings.
* Provide a valid watchdog trigger within the "Long Open Window" time period.
* Request a state transition to NORMAL mode.

2. Slow input voltage ramp-up

If the input voltage (



𝑉𝑆𝑥

) ramps up too slowly during power-on, the buck pre-regulator may fail to regulate its output. This triggers an undervoltage (UV) fault, forcing the device back into the INIT state.

**Solution:**

* Increase the slew rate of the input voltage during power-up.
* Momentarily apply a valid wake-up signal (WAK pin) or enable signal (ENA pin) to move the device out of the INIT state.

3. SPI communication issues

Problems with the Serial Peripheral Interface (SPI) can prevent the microcontroller from properly communicating with and initializing the TLF35584. Common errors include improper configuration or physical shorts.

**Solution:**

* **Check timing:** Ensure that the SPI communication is correctly framed. An SPI frame must begin with a chip select (SCS) low and end with a SCS high after 16 clock cycles. An SCS held low for over 2ms will cause a frame duration error (SPISF.DURE).
* **Verify wiring:** A short-to-ground on the Serial Data Input (SDI) pin will block SPI communication and cause a FAILSAFE transition.
* **Inspect flags:** Check the SPI status flag register (SPISF) for parity errors (PARE) or length errors (LENE).

4. Error monitoring settings conflict

Enabling both the error pin monitoring and the functional watchdog simultaneously with improper timing can cause conflicts that lock the device in INIT mode.

**Solution:**

* Initialize the device by disabling both the error pin monitor and the watchdog. If this successfully allows a transition to NORMAL mode, you can then enable and configure them one by one.

General troubleshooting steps

1. **Check error flags:** Read the status registers (like IF, MONSFx, SYSSF) via SPI to pinpoint the specific fault. This is the most direct way to diagnose the problem.
2. **Verify the power supply:** Ensure the battery voltage (



𝑉𝐵𝐴𝑇

) is stable and within the specified range, as dips can trigger an undervoltage fault.

1. **Confirm the initialization sequence:** Follow the step-by-step procedure outlined in the TLF35584 datasheet to clear status flags, unlock registers, configure peripherals, and request a state change.
2. **Use evaluation tools:** If available, use the Infineon GUI and evaluation board to confirm the correct initialization process and compare it with your application's behavior.

Other conditions that can cause the TLF35584 to remain in the INIT state relate to specific fault conditions, hardware connections, or improper configuration of safety mechanisms. A single-point failure in the initialization process can prevent the device from transitioning to a normal operating state.

Here are additional causes for the TLF35584 to stay in INIT mode, beyond the most common reasons:

**Faults during initialization**

The TLF35584 performs self-monitoring during the INIT phase. If it detects a fault before the microcontroller can complete its initialization, the device will remain in INIT and report the error via its status registers.

* **Voltage regulator failures**: The device's integrated voltage regulators, such as the core supply (QUC) or voltage reference (QVR), are activated during the startup sequence.
  + **Failed ramp-up**: If a regulator's output fails to reach its target voltage within the specified timing, the TLF35584 can trigger an undervoltage (UV) fault and stay in INIT. For example, if the QVR fails to enable correctly, the PMIC will not proceed.
  + **Excessive load**: If the load on a regulator, particularly the QUC, is too high during startup, it may fail to regulate properly, causing a UV fault.
* **Safety monitoring failures**: The TLF35584 has dedicated safety monitoring features that are enabled by default in INIT mode.
  + **ERR pin monitoring timeout**: If the microcontroller's safety monitoring unit (SMU) does not provide a valid toggling signal to the ERR pin in time, the ERR monitor will time out, causing an error.
  + **Functional watchdog (FWD) timeout**: The FWD, if configured, must be serviced correctly during the init sequence. A configuration or service error can prevent a state transition.

**Improper hardware configuration**

* **ERR pin connection**: A missing hardware connection between the microcontroller's SMU\_FSP (Fault Signal Path) output and the TLF35584's ERR pin can cause a failure in the ERR monitoring function. A constant low or high signal on the ERR pin is treated as a fault.
* **MPS pin state**: The Multi-Purpose Signal (MPS) pin can be configured for different functionalities. If the pin is not held at a valid logic high during initialization for the desired configuration, it can prevent a transition to NORMAL mode.
* **External component issues**: Problems with external components, such as a short to ground on a supply line, can trigger protective features and keep the device in INIT.

**Incorrect initialization sequence**

Even if the basic communication is correct, the order and timing of commands are critical for the TLF35584.

* **Protected register access**: The sequence for unlocking and locking protected registers must be strictly followed. Forgetting to unlock registers before writing to them will prevent critical configurations from being applied.
* **Startup time delay**: The datasheet specifies timing requirements, such as a nominal 60 µs delay after requesting a state transition, that must be observed for the state change to take effect.
* **Clearing previous faults**: In case of a soft reset or warm reset, it's essential to read and clear any lingering error flags from the status registers *before* attempting to transition to NORMAL mode. Failing to do so means the fault persists, and the PMIC will stay in INIT.

**Software or firmware issues**

* **Unoptimized code**: The microcontroller software may take too long to perform its startup routines, exceeding the TLF35584's 600 ms INIT timer. This is common with complex firmware that initializes many peripherals.
* **Wrong drivers or silicon step**: Using the wrong communication drivers (e.g., QSPI) or firmware not suited for the specific silicon step (A, B, or C) can lead to compatibility problems and communication errors.
* **Race conditions**: Race conditions in the firmware, where one part of the code conflicts with the expected timing of another, can cause intermittent initialization failures. For example, if a state change is requested while the ERR monitor is still in an unstable state, a fault can be triggered.

To find the root cause of a TLF35584 staying in the INIT state, you must systematically check the device's status registers via the SPI interface. The registers provide detailed flags for system faults, power supply issues, and SPI communication problems.

**Key status registers to check**

**IF (Interrupt Flags)**

This is the primary register to read, as it indicates which area of the device has an active fault.

* **IF.SYS**: A set bit indicates an error in the system's core. You must then read the SYSSF register for more details.
* **IF.MON**: This bit indicates a voltage or thermal monitoring error. Read the MONSFx registers for specifics.
* **IF.SPI**: This bit signals a potential SPI communication error. Check the SPISF register for the exact fault.
* **IF.WD**: Indicates a watchdog-related fault, either during the INIT phase or normal operation. Check the WWDSTAT or FWDSTAT registers.

**SYSSF (System Status Flags)**

This register provides detailed information when the IF.SYS flag is set.

* **SYSSF.CFGE**: A double-bit error occurred on a protected configuration register. This can happen during an UNLOCK-LOCK sequence if the power supply is unstable or a temporary event occurs.
* **SYSSF.STBYSG**: Indicates a short to ground on the standby regulator (QST) output.
* **SYSSF.UCSG**: Indicates a short to ground on the buck pre-regulator (QUC) output.

**MONSFx (Monitoring Status Flags)**

These registers report faults from the voltage and thermal monitoring blocks.

* **MONSF1**: Contains flags for overvoltage (OV) events. An OV on any regulator can put the device in FAILSAFE.
* **MONSF2**: Contains flags for undervoltage (UV) events. A UV fault on the core supplies (VEXT, VEVRSB, QUC) is a common reason for a warm reset that cycles back to the INIT state.
* **OTFAIL**: Details which specific thermal sensor detected an overtemperature (OT) event that caused a shutdown.

**INITERR (Initialization Error Flags)**

This register is specifically important for diagnosing why the device fails to leave the INIT state.

* **INITERR.WWDF**: A failure occurred during the servicing of the window watchdog (WWD) during the INIT phase.
* **INITERR.FWDF**: A failure occurred during the servicing of the functional watchdog (FWD) during the INIT phase.
* **INITERR.ERRF**: Indicates a timeout or incorrect signal on the ERR pin during initialization.

**SPISF (SPI Status Flags)**

Check this register if the IF.SPI flag is set to identify communication protocol errors.

* **SPISF.PARE**: A parity error was detected, indicating a single-bit flip on the SPI bus.
* **SPISF.LENE**: A length error was detected, meaning the received SPI frame was not the correct length.
* **SPISF.DURE**: A duration error occurred, typically due to the chip select (SCS) being held low for too long.

**Recommended action sequence**

1. **Read the IF register:** After power-on, read the IF register first. If any of the main flag bits are set (IF.SYS, IF.MON, IF.SPI, or IF.WD), proceed to the corresponding status registers.
2. **Read detailed status registers:** Based on the IF register, read the appropriate SYSSF, MONSFx, INITERR, or SPISF registers to pinpoint the specific fault.
3. **Correct the root cause:** Use the information from the detailed status flags to identify and fix the underlying issue. For example, if MONSF2.QUC\_UV is set, check the input power and load on the buck regulator.
4. **Clear the flags:** After addressing the issue, write a 1 to the corresponding flag bits in the IF register and any associated status registers to clear them.
5. **Re-attempt initialization:** Re-execute the initialization sequence to see if the device can now transition to the NORMAL state.

If the ERR monitoring fails and causes the TLF35584 to stay in the INIT state, the relevant flag is in the **INITERR** register.

Specifically, you need to check the **INITERR.ERRF** flag.

**INITERR register (0x1A)**

The INITERR register logs faults that occur during the initialization (INIT) phase, including ERR monitoring failures.

* **INITERR.ERRF (ERR Pin Failure Flag):** This bit is set if the ERR pin monitoring failed during the INIT state. This typically happens if the microcontroller does not send a valid toggling signal to the ERR pin within the 600 ms INIT timer period. A constant high, constant low, or improperly timed signal will trigger this fault.
* **INITERR.WWDF (Window Watchdog Failure Flag):** While not related to ERR monitoring, this bit indicates a window watchdog failure during initialization and can also keep the device in INIT mode.
* **INITERR.FWDF (Functional Watchdog Failure Flag):** Similar to the WWDF, this bit indicates a functional watchdog failure during initialization.

**How to use this information**

1. **Read the IF register:** After a system reset, read the IF register. If the IF.SYS bit is set, a system error occurred.
2. **Read the INITERR register:** If IF.SYS is set, read the INITERR register at SPI address 0x1A to see if the ERRF bit is high. This confirms that ERR monitoring was the cause of the INIT failure.
3. **Correct the issue:** If INITERR.ERRF is set, verify the connection and timing of the signal from your microcontroller's safety management unit to the TLF35584's ERR pin. Ensure a valid toggling signal is sent quickly enough during the INIT phase.
4. **Clear the flag:** After addressing the underlying cause, clear the INITERR register by writing a 1 to the ERRF bit. You should also write 1 to the corresponding IF.SYS bit to clear the interrupt flag.
5. **Re-attempt initialization:** Re-execute the initialization sequence, including the transition to NORMAL mode.

Even if you have explicitly disabled the Functional Watchdog (FWD) in the TLF35584's configuration, you still need to handle its flag in the INITERR register. This is a crucial step to ensure the device transitions out of the INIT state and doesn't get stuck in a failsafe loop.

Here's why and what you need to do:

**Why you must handle the FWD flag**

* **Persistent flags:** Fault flags in the

TLF35584

 are sticky and persist across some reset conditions. Even if the FWD is configured as disabled, a failure from a previous initialization attempt may have set the INITERR.FWDF bit. If this flag is not explicitly cleared, the device's logic can interpret the persistent flag as an active fault and prevent a transition to NORMAL mode.

* **Initialization window:** During the 600ms INIT window, the

TLF35584

 performs self-monitoring and requires the microcontroller to complete its setup sequence. While the FWD is disabled, a failure in another part of the initialization—such as the Window Watchdog (WWD) or ERR monitoring—could trigger a state change back to INIT. If the INITERR.FWDF flag from a prior failure is still set, it could contribute to a failsafe transition.

* **Initial state behavior:** On a cold power-on, the device enters the INIT state. Even if your intended configuration is to disable the FWD, a race condition or a temporary event during power-up could potentially cause an unintended trigger or an internal event that sets the FWDF bit. The safest practice is to always read and clear this flag as part of your standard initialization routine.

**What you need to do**

After a reset or power-on, your microcontroller's initialization sequence for the

TLF35584

 should include these steps:

1. **Perform the UNLOCK sequence** to enable write access to protected registers.
2. **Explicitly configure the FWD as disabled** to ensure the correct setting is applied.
3. **Read the INITERR register (0x1A)**.
4. **Clear the INITERR.FWDF flag** by writing a 1 to that specific bit position. This ensures any previously set fault from an FWD failure is erased.
5. **Perform the LOCK sequence** to protect the configuration registers from being accidentally overwritten.
6. **Request a state transition to NORMAL** by writing the appropriate command to the DEVCTRL.STATEREQ register.
7. **Monitor the INITERR register** again to confirm the transition was successful and no new initialization errors occurred.

To properly serve the ERR monitoring on the Infineon TLF35584, your host microcontroller must generate a continuous toggling signal on the ERR pin within the specified timing parameters. This "alive" signal is monitored by the PMIC during the INIT state and throughout normal operation. Failure to provide a valid signal within the 600ms INIT timer window will cause the PMIC to detect a fault and remain in INIT mode.

**The ERR monitoring signal: "Alive" supervision**

The TLF35584's ERR pin is a functional safety feature used to supervise the correct operation of the microcontroller's internal Safety Management Unit (SMU). A valid, non-faulty operation is indicated by a toggling signal, while a constant low or high signal is interpreted as an error.

**Steps for proper ERR monitoring service**

**1. Microcontroller configuration**

* **Generate a periodic toggling signal:** Configure one of your microcontroller's General-Purpose Input/Output (GPIO) pins to serve as the ERR signal output. For Infineon's AURIX devices, this is typically done using the SMU Fault Signal Path (FSP) interface.
* **Establish the correct timing:** The signal must toggle within the minimum and maximum detection periods specified in the TLF35584 datasheet. The frequency and duty cycle can vary, but the duration of both the high and low states (t\_High\_ERR and t\_Low\_ERR) must be maintained within the valid range.
* **Connect the output to the ERR pin:** Connect the configured GPIO pin from your microcontroller directly to the ERR pin of the TLF35584. No external pull-up or pull-down devices are required.

**2. ERR servicing during the INIT state**

* **Start toggling immediately:** As soon as the microcontroller starts up, you must begin generating the toggling ERR signal. The ERR monitoring function is enabled by default in the TLF35584's INIT state.
* **Provide sufficient periods:** Before requesting the transition to the NORMAL state via SPI, you must provide a minimum number of valid ERR signal periods. The datasheet specifies that at least **three valid periods** are required for a successful INIT-to-NORMAL transition.
* **Do not delay:** The entire initialization sequence, including the ERR servicing, must be completed within the 600ms INIT timer. If the microcontroller is too slow to start generating the ERR signal, the PMIC will time out and remain in INIT.

**3. Handling errors**

* **Read the INITERR register:** If the device gets stuck in INIT, read the INITERR register (address 0x1A) via SPI. The INITERR.ERRF bit will be set if the ERR monitoring failed during initialization.
* **Diagnose the fault:**
  + **No toggling:** A constant high or low signal on the ERR pin will trigger the fault. Check the microcontroller's configuration.
  + **Incorrect timing:** If the high or low period is outside the specified range, the PMIC will register an error.
* **Clear the fault and proceed:**
  + Fix the root cause on the microcontroller side (e.g., ensure the toggling signal is correct).
  + Clear the INITERR.ERRF flag via SPI by writing a 1 to that bit.
  + Restart the initialization sequence to allow a state transition.

**4. ERR monitoring during normal operation**

* **Continuous service:** Once in the NORMAL state, the microcontroller must continue to provide the valid toggling signal to the ERR pin to prove its "aliveness."
* **Fault indication:** If the microcontroller's SMU detects an internal failure, it should stop toggling and hold the ERR pin low to signal the fault to the TLF35584. The PMIC will then transition to a defined safe state.

There are two primary methods for managing the ERR monitoring on the TLF35584 to enable a successful transition from the INIT to NORMAL state: properly servicing the ERR signal or explicitly disabling the ERR monitor.

**Method 1: Proper ERR monitoring servicing (recommended for functional safety)**

Servicing the ERR monitor involves your microcontroller generating a periodic, toggling signal that acts as an "alive" supervision signal for the PMIC.

**Hardware setup**

* Connect an output pin from your microcontroller's safety management unit (SMU) to the

TLF35584

's ERR pin. For Infineon AURIX devices, this is typically the Fault Signal Path (FSP) pin.

* No external pull-up or pull-down components are needed. The

TLF35584

 detects a constant high or low signal as a failure indication.

**Software procedure**

1. **Start toggling immediately:** Your microcontroller must begin generating the toggling signal on its SMU output as soon as it powers on, and certainly before attempting any state transitions via SPI.
2. **Toggle within timing constraints:** The signal must toggle within the minimum and maximum period specifications defined in the TLF35584 datasheet.
3. **Provide sufficient cycles:** You must provide a minimum of **three valid toggling periods** before requesting the state change from INIT to NORMAL. This confirms the microcontroller is alive and correctly executing its safety routines.
4. **Issue the state transition command:** After the ERR signal has been properly toggled at least three times, send the SPI command to transition the TLF35584 from INIT to NORMAL mode.
5. **Continue toggling in NORMAL mode:** Once in the NORMAL state, your microcontroller must continue to provide the toggling ERR signal as part of the ongoing functional safety monitoring.

**Method 2: Disabling the ERR monitor**

Disabling the ERR monitor is an alternative approach, especially if the functional safety provided by the ERR pin is not required for your specific application or if a simpler start-up sequence is preferred.

**Software procedure**

1. **Read and clear flags:** As part of your initialization routine, read the INITERR register at address 0x1A and clear any pending ERRF flags by writing a 1 to that bit position. This removes any lingering faults from previous startup attempts.
2. **Unlock protected registers:** Execute the special 32-bit SPI unlock sequence to enable writing to protected configuration registers.
3. **Disable the ERR monitor:** Write to the SYSPCFG1 register (or the relevant ERR monitor configuration register) and set the ERREN bit to 0D (Disabled).
4. **Lock protected registers:** Execute the SPI lock sequence to protect the new configuration from being accidentally changed.
5. **Issue the state transition command:** After successfully disabling the ERR monitor, request the transition from INIT to NORMAL state. The TLF35584 will no longer require the ERR signal to be serviced for this transition.

Your service routine serviceErrMonitorSignal() is designed to provide a toggling signal, which is the correct approach. However, the use of a simple busy-wait loop with delay\_MS\_STM(10) is fundamentally flawed for meeting the strict timing requirements of the TLF35584's ERR pin monitoring.

**Issues with your current approach**

* **Too slow and inaccurate:** Your code generates a cycle with a 20ms period (10ms high, 10ms low). The TLF35584's timing specification for a valid ERR toggle requires a much faster signal, typically in the microseconds range (e.g., 55-100 µs), not milliseconds. The PMIC will detect your 10ms pulse as an "invalid ERR signal timing" and move to a failsafe state, or at the very least prevent the INIT-to-NORMAL transition.
* **Busy-wait blocks the CPU:** Relying on a busy-wait loop (delay\_MS\_STM) is not suitable for an embedded system. It consumes processor time and prevents other critical startup tasks from running. Your ERR servicing code and your SPI initialization code are fighting for CPU time, which increases the likelihood of a timing-related failure within the 600ms INIT window.
* **Fixed number of cycles:** Your for loop provides exactly six toggles and then stops. The ERR monitoring in NORMAL mode is a continuous supervision mechanism. The signal must keep toggling indefinitely until a safety-related event requires it to be held low.

**Correct approach for AURIX ERR servicing**

For proper ERR monitoring, especially in safety-critical applications with AURIX, you should use the microcontroller's dedicated peripherals to generate the signal reliably and accurately.

**1. Use the Safety Management Unit (SMU)**

The most robust and correct way to generate the ERR signal is through the AURIX's internal SMU Fault Signal Path (FSP). The SMU is a functional safety block designed for this exact purpose.

* **Hardware:** Connect the AURIX's SMU\_FSP pin (e.g., P33.8) directly to the TLF35584's ERR pin.
* **Software:**
  + **Configure the SMU:** Use the AURIX's SMU configuration registers to set up the FSP output for a periodic toggling signal.
  + **Tie to a safety alarm:** The SMU should be configured so that if one of its internal safety alarms is triggered, it automatically forces the FSP pin to a constant low state.
  + **Example (conceptual):**

c

*// Example: Enable SMU FSP control over P33.8*

IfxPort\_setPinMode(IfxPort\_P33\_8.port, IfxPort\_P33\_8.pinIndex, IfxPort\_Mode\_output\_pushPull);

*// Configure SMU\_PCTL register*

SMU\_PCTL.B.PCS = 1; *// Give control of the pin to the SMU*

*// Configure SMU for periodic toggling*

*// ... (specific SMU registers depend on AURIX series)*

*// Trigger a software alarm to simulate a fault*

*// SMU\_AGxCFy\_reg.Bx = 1;*

Use code with caution.

**2. Use a dedicated timer module**

If you are not using the SMU, a hardware timer is the next best option for accurate timing.

* **Configure a timer module:** Set up a timer (e.g., GTM, CCU6) to generate a periodic interrupt or toggle an output pin with a precise frequency.
* **Select a valid frequency:** Choose a frequency within the valid range specified by the TLF35584 datasheet for t\_High\_ERR and t\_Low\_ERR. For example, 100 µs high and 100 µs low would give a 5 kHz signal, well within the typically supported range.
* **Example (using AURIX's GTM):**

c

*// Conceptual: GTM setup for 5 kHz square wave on P33.8*

*// Configure GTM module*

*// Configure GTM timer output on P33.8*

*// Start the timer to generate the PWM signal*

Use code with caution.

**Recommended initialization sequence**

Your ERR serving code should be integrated into a complete initialization sequence:

1. **Start SMU/Timer for ERR:** Start the ERR toggling immediately upon microcontroller startup.
2. **Wait for minimum periods:** Before proceeding, ensure at least three valid ERR cycles have been completed.
3. **Perform SPI initialization:** Use SPI to clear any existing flags (e.g., INITERR.ERRF), unlock the PMIC, configure the watchdog, and set up the voltage monitors.
4. **Request INIT to NORMAL:** Send the state transition command via SPI.
5. **Wait for confirmation:** Read the status register to confirm the transition to NORMAL was successful.
6. **Continue monitoring:** Keep the ERR signal toggling continuously throughout the application's runtime.

Generating the ERR signal for the TLF35584 using a timer module instead of the SMU on an AURIX microcontroller is a viable option, especially if the full functional safety features of the SMU are not being utilized for the ERR pin. The key is to achieve the precise and continuous toggling required by the PMIC. AURIX offers powerful timer modules like the Generic Timer Module (GTM) and the Capture/Compare Unit 6 (CCU6) which can be used for this purpose.

**Using the GTM Timer Output Module (TOM)**

The GTM is highly flexible and capable of generating accurate PWM signals.

**Configuration steps (conceptual with iLLD functions)**

1. **Enable GTM Clocks:** Ensure the necessary GTM clocks are enabled.

c

IfxGtm\_Cmu\_enableClocks(&IfxGtm\_g\_gtmCmuConfig); *// Example function call*

Use code with caution.

1. **Configure GTM TOM Channel:**
   * Initialize the configuration structure for the GTM TOM PWM.
   * Specify the GTM module and TOM channel you want to use.
   * Set the **frequency** for the toggling signal. You will need to calculate the timer period based on the GTM clock frequency and the desired ERR signal frequency. Refer to the TLF35584 datasheet for the valid timing range of the ERR signal's high and low periods (often in the microsecond range, e.g., 55 µs to 100 µs).
   * Set the **duty cycle** to 50% for a clean toggling signal.
   * Configure the output pin (e.g., P33.8) to be controlled by the GTM TOM channel.
   * Initialize the GTM TOM channel with your configured settings.
2. **Start the Timer:** Run the GTM TOM channel to begin generating the PWM signal on the output pin.

**Example (GTM TOM conceptual)**

c

#include "IfxGtm\_Tom\_Pwm.h"

#include "IfxGtm\_Cmu.h"

#include "IfxPort.h" *// For configuring the GPIO pin*

*// Define the pin connected to TLF35584 ERR*

#define ERR\_PIN IfxPort\_P33\_8

*// TLF35584 ERR timing (check datasheet for exact values)*

*// Let's assume a period of 200 us (100 us high, 100 us low), giving a frequency of 5 kHz*

#define ERR\_SIGNAL\_FREQUENCY\_HZ 5000

#define GTM\_CLOCK\_FREQUENCY\_HZ 100000000 *// Example GTM clock frequency*

*// Global variables for GTM TOM configuration*

IfxGtm\_Tom\_Pwm\_Config g\_gtmPwmConfig;

IfxGtm\_Tom\_Pwm\_Driver g\_gtmPwmDriver;

void initErrTimer(void)

{

*// Enable GTM clocks*

IfxGtm\_Cmu\_enableClocks(&IfxGtm\_g\_gtmCmuConfig); *// Need to define g\_gtmCmuConfig*

*// Initialize the PWM configuration structure with default values*

IfxGtm\_Tom\_Pwm\_initConfig(&g\_gtmPwmConfig, &MODULE\_GTM); *// Example for TOM0*

*// Set the output pin*

g\_gtmPwmConfig.pin.outputPin = &ERR\_PIN;

g\_gtmPwmConfig.pin.outputMode = IfxPort\_OutputMode\_pushPull;

g\_gtmPwmConfig.pin.padDriver = IfxPort\_PadDriver\_cmosAutomotiveSpeed1;

*// Calculate period based on desired frequency and GTM clock*

*// Period = (GTM\_CLOCK\_FREQUENCY / ERR\_SIGNAL\_FREQUENCY\_HZ)*

g\_gtmPwmConfig.timer.period = (GTM\_CLOCK\_FREQUENCY\_HZ / ERR\_SIGNAL\_FREQUENCY\_HZ);

g\_gtmPwmConfig.timer.signal = IfxGtm\_Timer\_Out\_highLow; *// Toggle high/low*

g\_gtmPwmConfig.timer.duration = g\_gtmPwmConfig.timer.period / 2; *// 50% duty cycle*

*// Initialize the GTM TOM PWM driver*

IfxGtm\_Tom\_Pwm\_init(&g\_gtmPwmDriver, &g\_gtmPwmConfig);

*// Start the PWM generation (this often happens automatically after init or with a run function)*

IfxGtm\_Tom\_Pwm\_start(&g\_gtmPwmDriver, TRUE); *// Start continuous PWM*

}

Use code with caution.

**Using the CCU6 Timer**

The CCU6 module can also generate PWM signals, though it might be simpler to configure for basic toggling.

**Configuration steps (conceptual with iLLD functions)**

1. **Enable CCU6 Module:** Enable the CCU6 module.
2. **Configure Timer:**
   * Initialize the configuration structure for the CCU6 Timer with default values.
   * Set the timer frequency (which determines the toggling frequency of the ERR pin).
   * Configure the timer mode (e.g., up-counting, continuous mode).
   * Calculate and set the period value.
   * Initialize the CCU6 timer module.
3. **Configure PWM High/Low Driver:**
   * Initialize the configuration structure for the CCU6 PWM High/Low driver.
   * Assign the previously configured timer to it.
   * Specify the output pin (P33.8) and configure its driver strength and mode.
   * Set the duty cycle to 50%.
   * Initialize the CCU6 PWM High/Low driver.
4. **Start the Timer:** Run the CCU6 timer to start generating the PWM signal.

**Example (CCU6 conceptual)**

c

#include "IfxCcu6\_Timer.h"

#include "IfxCcu6\_PwmHl.h"

#include "IfxPort.h"

#define ERR\_PIN IfxPort\_P33\_8

#define ERR\_SIGNAL\_FREQUENCY\_HZ 5000 *// Target 5 kHz*

IfxCcu6\_Timer\_Config g\_ccu6TimerConfig;

IfxCcu6\_PwmHl\_Config g\_ccu6PwmConfig;

IfxCcu6\_PwmHl\_Channel g\_ccu6PwmChannel; *// Example for a single channel*

void initErrTimer(void)

{

*// Enable CCU6 module*

*// IfxCcu6\_enableModule(ccu6Module); // This is part of initModule usually*

*// 1. Configure the CCU6 timer*

IfxCcu6\_Timer\_initModuleConfig(&g\_ccu6TimerConfig, &MODULE\_CCU60); *// Use CCU60 or CCU61*

g\_ccu6TimerConfig.timer = IfxCcu6\_TimerId\_t12; *// Use Timer 12*

g\_ccu6TimerConfig.base.t12Frequency = ERR\_SIGNAL\_FREQUENCY\_HZ \* 2; *// Timer runs at 2x signal frequency for 50% duty*

g\_ccu6TimerConfig.base.t12Period = (g\_ccu6TimerConfig.base.t12Frequency / ERR\_SIGNAL\_FREQUENCY\_HZ) - 1; *// Calculate period*

IfxCcu6\_Timer\_initModule(&g\_ccu6TimerConfig, &MODULE\_CCU60);

*// 2. Configure the PWM High/Low driver*

IfxCcu6\_PwmHl\_initConfig(&g\_ccu6PwmConfig, &MODULE\_CCU60);

g\_ccu6PwmConfig.timer = &g\_ccu6TimerConfig;

g\_ccu6PwmConfig.base.channelCount = 1;

g\_ccu6PwmConfig.channel[0].outputPin = &ERR\_PIN;

g\_ccu6PwmConfig.channel[0].outputMode = IfxPort\_OutputMode\_pushPull;

g\_ccu6PwmConfig.channel[0].padDriver = IfxPort\_PadDriver\_cmosAutomotiveSpeed1;

IfxCcu6\_PwmHl\_init(&g\_ccu6PwmChannel, &g\_ccu6PwmConfig);

*// Set 50% duty cycle*

Ifx\_TimerValue compareValue = g\_ccu6TimerConfig.base.t12Period / 2;

IfxCcu6\_PwmHl\_update(&g\_ccu6PwmChannel, &compareValue);

IfxCcu6\_Timer\_applyUpdate(&g\_ccu6TimerConfig); *// Transfer compare values*

*// 3. Start the timer*

IfxCcu6\_Timer\_run(&g\_ccu6TimerConfig);

}

Use code with caution.

**Important considerations for both timer methods**

* **Timing Accuracy:** Refer to the TLF35584 datasheet for the exact minimum and maximum periods (t\_High\_ERR, t\_Low\_ERR) for a valid ERR signal. Ensure your timer configuration meets these requirements precisely.
* **Initialization Sequence:** Regardless of the timer method chosen, the overall initialization sequence remains the same:
  1. Start the ERR signal generation (using your GTM or CCU6 timer) *immediately* after the microcontroller powers on.
  2. Ensure at least three valid ERR toggling periods have occurred.
  3. Perform the necessary SPI communications: unlock registers, clear any INITERR.ERRF flags, configure other PMIC features (watchdogs, monitors), and request the transition to NORMAL state.
  4. Verify the transition was successful by reading the DEVSTAT register.
  5. The timer must continue to generate the ERR signal continuously during the entire NORMAL operation of the PMIC.
* **Error Indication:** If your microcontroller detects a fault, it should stop the timer or configure the output pin to hold the ERR pin low to signal the fault to the TLF35584.
* **iLLD Documentation:** The examples provided are conceptual. Refer to the specific Infineon Low-Level Driver (iLLD) documentation for your AURIX microcontroller series (e.g., TC2xx or TC3xx) for the exact functions, parameter types, and register details for configuring GTM or CCU6 timers to output PWM signals on specific GPIO pins.

For ERR monitoring on an AURIX microcontroller, the most relevant code examples can be found in Infineon's official GitHub repository for code examples, but the best sources are often the application notes and expert training materials from the Infineon website. These resources demonstrate the proper use of the iLLD functions for specific peripherals like the GTM and SMU.

**1. SPI communication with TLF35584**

* **Repository:** Infineon Code Examples for AURIX™ Development Studio
* **Location:** Find the SPI\_TLF\_1 example, which is often developed for a specific board (e.g., KIT\_TC397\_TFT or KIT\_TC375\_ARD\_SB). The board-specific part of the name may differ based on your hardware.
* **Relevance:** This example demonstrates the correct use of the QSPI module to communicate with the TLF35584 via SPI, including how to handle the initialization sequence and read status flags.
* **What to extract:** Use the SPI master initialization and the functions for sending/receiving data to build your TLF35584 communication driver.

**2. GTM TOM PWM generation**

* **Repository:** Infineon Code Examples for AURIX™ Development Studio
* **Location:** Look for examples with names like GTM\_TOM\_PWM\_1 or GTM\_ATOM\_PWM\_1. The TOM submodule is most commonly used for basic PWM generation, which is sufficient for ERR signal toggling.
* **Relevance:** These examples show how to configure the GTM to generate a continuous PWM signal with a specific frequency and duty cycle. You will adapt this to meet the ERR signal timing requirements.
* **What to extract:** The core logic for initializing the GTM clock, configuring the TOM channel for PWM generation, and linking it to the correct port pin.

**3. SMU alarm and emergency stop**

* **Repository:** Infineon Code Examples for AURIX™ Development Studio
* **Location:** Search for examples with SMU in the name, such as SMU\_Emergency\_Stop\_Alarm\_1.
* **Relevance:** This is the most robust and safety-compliant method. The examples demonstrate how to use the SMU to generate the Fault Signal Path (FSP) signal, which is the "alive" toggling signal for the ERR pin. It also shows how a safety alarm can trigger a specific reaction, such as halting the ERR signal.
* **What to extract:** The code for unlocking and locking SMU registers, configuring the FSP output, and associating an SMU alarm with a specific action.

**4. Integration application notes (crucial)**

Beyond the raw code examples, Infineon provides detailed Application Notes (AN) and training documents that show how to integrate the AURIX microcontroller with PMICs like the TLF35584. These documents often include code snippets and explain the overall system-level interaction.

* **Document:** AP32402 "Integration of TC3xx microcontrollers with TLF35584 / TLF35585 system power supply"
* **Relevance:** This application note is a highly valuable resource as it directly addresses your use case. It covers hardware integration, start-up sequences, reset handling, and safety management, including the use of the SMU\_FSP for ERR monitoring.
* **Where to find:** On the Infineon website by searching for "AP32402" or "TLF35584 application note."

**Recommendation for your workflow**

1. **Read the AP32402 application note:** Start with this to get a complete overview of the integration process and understand the safety-critical aspects.
2. **Adapt the SPI example (SPI\_TLF\_1):** Use this as a base for your TLF35584 driver.
3. **Use the SMU or GTM example:**
   * **For safety-critical applications:** Use the **SMU example** (SMU\_Emergency\_Stop\_Alarm\_1) to generate the ERR signal. It's the most correct method.
   * **For simpler applications or learning:** Use the **GTM TOM PWM example** (GTM\_TOM\_PWM\_1) to generate the toggling signal. It's easier to set up but does not have the integrated safety features of the SMU.
4. **Integrate and test:** Combine the SPI and ERR signal generation code. Ensure your initialization routine correctly clears all previous flags in the INITERR register, starts the ERR signal, and then transitions the PMIC to the NORMAL state.

Yes, the GTM\_TOM\_PWM\_1 example code can be adapted to generate the PWM signal required for ERR pin monitoring on the TLF35584. This approach uses the Generic Timer Module (GTM) to generate a precise and continuous toggling signal.

The key adaptations needed are:

1. **Pin assignment:** Modify the code to output the PWM signal on the specific pin connected to the TLF35584's ERR pin (e.g., P33.8).
2. **Timing parameters:** Configure the GTM timer to generate a 50% duty cycle signal with a period within the valid range specified by the TLF35584 datasheet (typically in the microsecond range).

**How to adapt the GTM\_TOM\_PWM\_1 example**

The GTM\_TOM\_PWM\_1 example usually focuses on driving an LED, but the underlying principle is the same. Follow these steps:

**1. Define the correct output pin**

In the example code, find where the output pin is defined and change it to your target pin (e.g., P33.8).

c

*// Original in example:*

#define LED\_PIN IfxPort\_P00\_5 *// Example pin for an LED*

*// Your modified code:*

#define ERR\_PIN IfxPort\_P33\_8 *// The pin connected to the TLF35584 ERR pin*

Use code with caution.

**2. Configure the GTM TOM PWM settings**

Within the initialization function (e.g., initGtmPwm), adjust the frequency and duty cycle parameters. The ERR monitoring signal must have a 50% duty cycle and a frequency that falls within the TLF35584's specified window.

c

*// Example from previous response adapted:*

void initErrTimer(void)

{

*// ... GTM clock initialization ...*

*// Configure PWM output on the ERR pin*

g\_gtmPwmConfig.pin.outputPin = &ERR\_PIN;

g\_gtmPwmConfig.pin.outputMode = IfxPort\_OutputMode\_pushPull;

g\_gtmPwmConfig.pin.padDriver = IfxPort\_PadDriver\_cmosAutomotiveSpeed1;

*// Set the PWM signal to a valid frequency for ERR monitoring*

#define ERR\_SIGNAL\_FREQUENCY\_HZ 5000 *// 5kHz example, verify with datasheet*

#define GTM\_CLOCK\_FREQUENCY\_HZ 100000000 *// Example GTM clock frequency*

g\_gtmPwmConfig.timer.period = (GTM\_CLOCK\_FREQUENCY\_HZ / ERR\_SIGNAL\_FREQUENCY\_HZ);

g\_gtmPwmConfig.timer.signal = IfxGtm\_Timer\_Out\_highLow; *// Toggle high/low*

g\_gtmPwmConfig.timer.duration = g\_gtmPwmConfig.timer.period / 2; *// 50% duty cycle*

*// Initialize and start the GTM PWM*

IfxGtm\_Tom\_Pwm\_init(&g\_gtmPwmDriver, &g\_gtmPwmConfig);

IfxGtm\_Tom\_Pwm\_start(&g\_gtmPwmDriver, TRUE);

}

Use code with caution.

**3. Start the timer at the correct time**

As discussed previously, you must start the ERR timer immediately after the microcontroller powers on, before you attempt any SPI communication with the PMIC. The GTM start() function should be called early in your initialization sequence.

**4. Continuously run the timer**

The GTM will run independently of the CPU once started. You do not need to call the function repeatedly in a loop. The hardware timer will take care of generating the continuous toggling signal required for the ERR monitoring in both INIT and NORMAL modes.

**Caveat: GTM vs. SMU**

While using the GTM is a valid approach, it is less tightly integrated with the microcontroller's safety features than the SMU. For a safety-critical application, the SMU is the more robust solution, as it can be configured to automatically drive the ERR pin to a low state upon detecting an internal safety violation. The GTM method would require additional software to detect faults and manually stop the PWM and hold the pin low.