NOTICE

While reasonable efforts have been made to assure the accuracy of this document, Telit assumes no liability resulting from any inaccuracies or omissions in this document, or from use of the information obtained herein. The information in this document has been carefully checked and is believed to be reliable. However, no responsibility is assumed for inaccuracies or omissions. Telit reserves the right to make changes to any products described herein and reserves the right to revise this document and to make changes from time to time in content hereof with no obligation to notify any person of revisions or changes. Telit does not assume any liability arising out of the application or use of any product, software, or circuit described herein; neither does it convey license under its patent rights or the rights of others.

It is possible that this publication may contain references to, or information about Telit products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that Telit intends to announce such Telit products, programming, or services in your country.

COPYRIGHTS

This instruction manual and the Telit products described in this instruction manual may be, include or describe copyrighted Telit material, such as computer programs stored in semiconductor memories or other media. Laws in the Italy and other countries preserve for Telit and its licensors certain exclusive rights for copyrighted material, including the exclusive right to copy, reproduce in any form, distribute and make derivative works of the copyrighted material. Accordingly, any copyrighted material of Telit and its licensors contained herein or in the Telit products described in this instruction manual may not be copied, reproduced, distributed, merged or modified in any manner without the express written permission of Telit. Furthermore, the purchase of Telit products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents or patent applications of Telit, as arises by operation of law in the sale of a product.

COMPUTER SOFTWARE COPYRIGHTS

The Telit and 3rd Party supplied Software (SW) products described in this instruction manual may include copyrighted Telit and other 3rd Party supplied computer programs stored in semiconductor memories or other media. Laws in the Italy and other countries preserve for Telit and other 3rd Party supplied SW certain exclusive rights for copyrighted computer programs, including the exclusive right to copy or reproduce in any form the copyrighted computer program. Accordingly, any copyrighted Telit or other 3rd Party supplied SW computer programs contained in the Telit products described in this instruction manual may not be copied (reverse engineered) or reproduced in any manner without the express written permission of Telit or the 3rd Party SW supplier. Furthermore, the purchase of Telit products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents or patent applications of Telit or other 3rd Party supplied SW, except for the normal non-exclusive, royalty free license to use that arises by operation of law in the sale of a product.
USAGE AND DISCLOSURE RESTRICTIONS

I. License Agreements

The software described in this document is the property of Telit and its licensors. It is furnished by express license agreement only and may be used only in accordance with the terms of such an agreement.

II. Copyrighted Materials

Software and documentation are copyrighted materials. Making unauthorized copies is prohibited by law. No part of the software or documentation may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, without prior written permission of Telit.

III. High Risk Materials

Components, units, or third-party products used in the product described herein are NOT fault-tolerant and are NOT designed, manufactured, or intended for use as on-line control equipment in the following hazardous environments requiring fail-safe controls: the operation of Nuclear Facilities, Aircraft Navigation or Aircraft Communication Systems, Air Traffic Control, Life Support, or Weapons Systems (High Risk Activities*). Telit and its supplier(s) specifically disclaim any expressed or implied warranty of fitness for such High Risk Activities.

IV. Trademarks

TELIT and the Stylized T Logo are registered in Trademark Office. All other product or service names are the property of their respective owners.

V. Third Party Rights

The software may include Third Party Right software. In this case you agree to comply with all terms and conditions imposed on you in respect of such separate software. In addition to Third Party Terms, the disclaimer of warranty and limitation of liability provisions in this License shall apply to the Third Party Right software.

TELIT HEREBY DISCLAIMS ANY AND ALL WARRANTIES EXPRESS OR IMPLIED FROM ANY THIRD PARTIES REGARDING ANY SEPARATE FILES, ANY THIRD PARTY MATERIALS INCLUDED IN THE SOFTWARE, ANY THIRD PARTY MATERIALS FROM WHICH THE SOFTWARE IS DERIVED (COLLECTIVELY “OTHER CODE”), AND THE USE OF ANY OR ALL THE OTHER CODE IN CONNECTION WITH THE SOFTWARE, INCLUDING (WITHOUT LIMITATION) ANY WARRANTIES OF SATISFACTORY QUALITY OR FITNESS FOR A PARTICULAR PURPOSE.

NO THIRD PARTY LICENSORS OF OTHER CODE SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND WHETHER MADE UNDER CONTRACT, TORT OR OTHER LEGAL THEORY, ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE OTHER CODE OR THE EXERCISE OF ANY RIGHTS GRANTED UNDER EITHER OR BOTH THIS LICENSE AND THE LEGAL TERMS APPLICABLE TO ANY SEPARATE FILES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
### APPLICABILITY TABLE

<table>
<thead>
<tr>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>GS2K based Modules</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SW Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3 x onwards</td>
</tr>
</tbody>
</table>

**Note:** The features described in the present document are provided by the products equipped with the software versions equal or higher than the versions shown in the table. See also the Revision History chapter.
## REVISION HISTORY

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>June 2016</td>
<td>Initial release</td>
</tr>
<tr>
<td>2.0</td>
<td>November 2016</td>
<td>Added following sections:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Case 1: Periodic Data Upload to Cloud (Standby Mode),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Case 2: Event Notification to Cloud (Hibernate Mode) and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Case 3: Always Cloud Connected (PS POLL Applet Mode)</td>
</tr>
<tr>
<td>2.1</td>
<td>May 2017</td>
<td>Added Chapter 6 Power Measurement and Chapter 7 Profiling.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Chapter 1, Chapter 2, Chapter 3, Chapter 5 and Appendix</td>
</tr>
<tr>
<td>2.2</td>
<td>Dec 2017</td>
<td>Added section 3.2.1.1 Configuration Settings for Longer Standby Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Standby Mode)</td>
</tr>
<tr>
<td>2.3</td>
<td>May 2018</td>
<td>Added a note on IAR IDE 6.50.5 installation, see section - 5.1Build</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and Compilation Process</td>
</tr>
</tbody>
</table>
Table of Contents

NOTICE .................................................................................................................. 2
COPYRIGHTS ........................................................................................................... 2
COMPUTER SOFTWARE COPYRIGHTS ............................................................... 2
USAGE AND DISCLOSURE RESTRICTIONS ......................................................... 3
APPLICABILITY TABLE ......................................................................................... 4
REVISION HISTORY ............................................................................................... 5

CHAPTER 1. INTRODUCTION .............................................................................. 9
  1.1 SCOPE ........................................................................................................... 9
  1.2 OVERVIEW ................................................................................................... 9
  1.3 TERMINOLOGY ............................................................................................ 10
  1.4 STANDARDS AND CONCEPTS .................................................................. 11

CHAPTER 2. FEATURES AND CAPABILITIES ....................................................... 12

CHAPTER 3. THEORY OF OPERATION ................................................................ 14
  3.1 LIMITED-AP MODE .................................................................................... 14
  3.2 STATION MODE .......................................................................................... 15
  3.2.1 Case 1: Periodic Data Upload to Cloud (Standby Mode) ...................... 15
  3.2.2 Case 2: Event Notification to Cloud (Hibernate Mode) ..................... 20
  3.2.3 Case 3: Always Cloud Connected (PS POLL Applet Mode) ............. 21

CHAPTER 4. ARCHITECTURE ............................................................................. 23
  4.1 COMPONENTS ............................................................................................ 25
  4.1.1 Hardware ................................................................................................. 25
  4.1.2 Software .................................................................................................. 26
  4.2 TASKS ......................................................................................................... 27

CHAPTER 5. EMBEDDED APPLICATION EXECUTION SEQUENCE ..................... 29
  5.1 BUILD AND COMPILATION PROCESS ....................................................... 29
  5.2 TLS LP APPLICATION CODE FLOW ......................................................... 34
  5.2.1 Basic Init Thread .................................................................................... 34
  5.2.2 Applet Loader Thread ............................................................................ 35
  USE CASE 1: PERIODIC DATA UPLOAD TO CLOUD (STANDBY MODE) .......... 40
  5.2.3 Using UDP Server .................................................................................. 40
  5.2.4 Using HTTPS Server ............................................................................. 52
  USE CASE 2 EVENT NOTIFICATION TO CLOUD (HIBERNATE MODE) ......... 64
  5.2.5 Using UDP Server .................................................................................. 64
  5.2.6 Using HTTPS Server ............................................................................. 73
  USE CASE 3 ALWAYS CLOUD CONNECTED (PS POLL APPLET MODE) ....... 84
  5.2.7 Using SSL over TCP .............................................................................. 84

CHAPTER 6. POWER MEASUREMENT ................................................................. 94
  USE CASE 1: PERIODIC DATA UPLOAD TO CLOUD (STANDBY MODE) ......... 94
  USE CASE 2: EVENT NOTIFICATION TO CLOUD (HIBERNATE MODE) ........ 96
  USE CASE 3: ALWAYS CLOUD CONNECTED (PS POLL APPLET MODE) ........ 98

CHAPTER 7. PROFILING .................................................................................. 100

APPENDIX A MACRO’S INFORMATION .............................................................. 103
List of Figures

Figure 1: Limited AP mode ................................................................. 23
Figure 2: Client Mode ................................................................. 24
Figure 3: Components – TLS Low Power ........................................... 25
Figure 4: TLS Low Power Architecture Application Flow ......................... 27
Figure 5: TLS Low Power IAR Workbench ......................................... 29
Figure 6: Elements of the Control Block ............................................ 36
Figure 7: Control Block ................................................................. 36
Figure 8: Applet 2 Header Descriptor ............................................... 37
Figure 9: Applet 3 Header Descriptor ............................................... 38
Figure 10: SDK Builder Configuration .............................................. 41
Figure 11: Use Case 1 - Jumper settings .......................................... 94
Figure 12: Use Case 1 - PIN connection .......................................... 95
Figure 13: Use Case 1 - Multimeter showing Power Consumption ............... 95
Figure 14: Use Case 2 – Jumper Settings ........................................ 96
Figure 15: Use Case 2 – PIN connection .......................................... 97
Figure 16: Use Case 3 – Power Measurement ..................................... 97
Figure 17: Use Case 3 – Jumper Settings .......................................... 98
Figure 18: Use Case 3 – PIN Connections ....................................... 99
Figure 19: GS Node wake from Standby ......................................... 101
List of Tables

Table 1: Glossary of Terms ........................................................................................................... 10
Table 2: Hardware Components .................................................................................................25
Table 3: Software Components .................................................................................................26
Table 4: Applet Header Information .........................................................................................38
Table 5: System Events & GPIOs for the Use Cases .................................................................100
Table 6: Macro’s Information Table ..........................................................................................108
Chapter 1. Introduction

1.1 Scope

This document provides detailed information about the GS2K based Temperature Light Sensor (TLS) Low Power (LP) Application Development Kit (ADK) by describing the software and hardware architecture, functional features, API’s, application execution sequence and how it can be modified or customized per user requirement. It assumes that the reader is generally familiar with GainSpan products, Internet Protocol (IP) networks and the operation and management of 802.11 wireless devices.

Targeted applications for this platform include low power sensor applications and other embedded/IOT products which require long battery life.

1.2 Overview

TLS LP ADK comes up in provisioning Limited-AP mode or STA mode based on MACRO selected at compile time.

The provisioning Limited-AP mode is limited only to provision the GS node to an Access Point or to do Over the Air Firmware Upgrade when newer firmware versions are available. This mode doesn’t showcase any of the low power functionality. When we talk of low power, it is always from the perspective of the STA mode.

In STA mode, code is modularized into individual independent applets which are loaded based on conditions. TLS LP ADK showcases the low power capabilities of GS node such as hibernate, standby, fast bootup from hibernate standby, maintaining association with Access Point across standby, maintaining association with Cloud Server consuming very low power by going into standby across beacons. It also has dynamic keep alive feature which prevents frequent disconnection with the Access Point. All the above mentioned low power capabilities are showcased using the 3-use cases which demonstrates.

a) Periodic data upload to Cloud (Standby Mode)
b) Event Notification to Cloud (Hibernate Mode)
c) Always Cloud Connected (PS POLL Applet Mode)

NOTE – GainSpan SDK is mandatory for any development on TLS Low Power ADK.
## 1.3 Terminology

The following table lists the different terminologies used in this document.

<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>Access Point</td>
</tr>
<tr>
<td>API</td>
<td>Application Programmer’s Interface</td>
</tr>
<tr>
<td>TLS</td>
<td>Temperature Light Sensor</td>
</tr>
<tr>
<td>LP</td>
<td>Low Power</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hyper Text Transfer Protocol Secure</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>NCM</td>
<td>Network Connection Manager</td>
</tr>
<tr>
<td>GEPS</td>
<td>GainSpan Embedded Platform System</td>
</tr>
<tr>
<td>RTC</td>
<td>Real Time Clock</td>
</tr>
<tr>
<td>STA</td>
<td>Station mode or Client mode</td>
</tr>
<tr>
<td>Wdd</td>
<td>Wireless Device Driver</td>
</tr>
<tr>
<td>Limited AP</td>
<td>Limited Access Point mode</td>
</tr>
<tr>
<td>WPA</td>
<td>Wi-Fi Protected Access</td>
</tr>
</tbody>
</table>

**Table 1: Glossary of Terms**

---

NOTE –

The term ‘Task’ also means thread in this document.
1.4 Standards and Concepts

Standards

Generic

None

Specific

- UDP
- HTTPS

Concepts

TLS LP ADK uses applet mechanism to develop small independent Applications. It also showcases dynamic loading of code mechanism.

It has Dynamic Keep Alive mechanism which learns from the behavior of the associated Access Point and sends periodic Keep Alive packets to the Access Point so that AP doesn't throw out of network due to inactivity.
Chapter 2. Features and Capabilities

This section provides the features and capabilities of GainSpan TLS Low Power ADK:

Features:
- Limited-AP and station mode
- Provisioning in Limited AP mode
- OTA FU in Limited AP mode
- Periodic sending of data over UDP /HTTPS
- SNTP update
- Storing/restoring TCP and HTTP parameters.
- Standby
- Hibernate
- Wakeup from Wi-Fi
- EVB’s having no sensors can send dummy data
- Dynamic keep alive

Capabilities:
- TLS Low Power ADK demonstrates fast boot up time because only a subset of firmware is brought into RAM for execution.
- It performs an APP only boot to showcase the ability to wake up without WLAN. This results in consumption of less power.
- It showcases sensor only read cycle. This sort of application is useful in following cases:
  - The application is only reading sensors frequently and sends appended data to the network (server)
  - Indication is required when the sensor values exceed thresholds or after a periodic interval (grouping the data)
- It demonstrates dynamic loading of independent small applets based on conditions.
- It utilizes the Auto cache section where 128KB of Flash memory is mapped to 16KB of RAM and thus only a small footprint of code needs to be brought into RAM for execution.
- It demonstrates sending of sensor data either through UDP or HTTPS.
- It demonstrates sending of dummy sensor data either through UDP or HTTPS in EVB’s where temperature and light sensor aren’t present.
- It demonstrates saving of TCP and HTTP parameters before going into standby and restoring the same once GS node comes out of standby thereby removing the need of establishing new HTTPS session across standby.
- It demonstrates putting the device in Hibernate mode which is the lowest power consumption mode available.
- It demonstrates waking up from Cloud Server being in lowest power consumption mode.
- It provides an option to:
  - Configure sensor data size that needs to be sent to the server
  - Configure standby time between wakeups
  - Configure wait time for UDP response.
  - Configure wait time for HTTPS response
  - Configure SNTP timeout.
  - Configure Alarm 1 pin for device wakeup.
  - Configure the Alarm 2 pin to switch from STA mode to limited-AP mode.
  - Configure the WPS pin to switch to default mode.
- Perform over the air firmware upgrade using PUSH method in Limited-AP mode
- Configure the iteration after which sensor data is send to the configured server
- Enable accurate power measurement
- Perform power profiling using GPIO’s
- Perform link check by sending ARP to the configured server before sending sensor data. Also provide option to select the iteration in which, user wants to do link check
Chapter 3. Theory of Operation

On boot up, TLS LP ADK comes up as Limited-AP or as a STA based on MACRO configuration at compile time.

3.1 Limited-AP Mode

If the macro ‘DEFAULT_MODE’ is set to GSN_LIMITED_AP or if the mode from SDK Builder is set, (Choose ‘Provisioning Mode’ from System Configuration’ tab) GS node will come up in Limited AP Mode. This mode is used for “Provisioning” the GS node to an AP and to do “Over the Air Firmware Upgrade” when newer firmware version is available.

On bootup, ‘Basic Init Thread’ also called ‘tls lp base’ application thread will load the Applet3 which in turn creates a thread “Applet3 Thread”. This task does the following:

➢ Releases WLAN reset so that it gets initiated.
➢ Initialize supplicant and random number generation modules.
➢ Initializes the wdd message handler and network stack
➢ Concurrent interfaces are initialized but not started since verified provisioning flag is enabled.
➢ Initializes the BsdLayer, state machine (sm), Network connection manager (NCM),and then posts an event to the main task to start NCM.
➢ NCM will then go ahead and create Limited AP mode.
➢ Once Limited AP is created, start the DHCP, DNS (if enabled) and the Web Server.

Now any client can connect to the Limited AP and perform:

a. Provisioning to any AP using the native provision application or by web pages (<ipaddr>/gsprov.html).

b. OTAFU of any compatible firmware using the native OTAFU application or by webpages (<ipaddr>/otafu.html).

NOTE –

This mode is not used for demonstration of “sensor data send” as there is no power saving in this mode.
### 3.2 Station Mode

If the macro ‘DEFAULT_MODE’ is set to GSN_INFRA_CLIENT or if the default mode is set as client from SDK Builder (Choose ‘Connectivity Mode’ from System Configuration’ tab) or if the device is provisioned, then GS node will come up in client Mode connecting to the provisioned AP.

Theory of Operation in station mode can be explained based on the three use cases it supports:

#### 3.2.1 Case 1: Periodic Data Upload to Cloud (Standby Mode)

**Functionality**

On the first cold boot-up, TLS low power ADK boots up fast, loads only a subset of the firmware known as main application. The main application is also called ‘tls lp base’ application which does some basic initializations of the system and starts a ‘user configured timer’. The main application goes ahead and loads the full binary known as applet2. This initializes the full system as well as releasing the reset of WLAN. It associates with provisioned AP and gets the IP address, does a link check which is nothing but an ARP request to Access Point to check whether the connection is alive. If successful, reads the sensor and sends it over Wi-Fi. GS node then saves the configuration information for L2, L3, and L4 on RTC RAM and goes to standby.

On every wakeup from standby due to ‘user configured timer’ expiry, main application (tls lp base) boots up, increments the ‘wakeup count’ and checks whether the ‘wakeup count’ equals ‘user configured data send count’. If it is equal, loads the full binary known as Applet2. This initializes the full system as well as releasing the reset of WLAN. This time it does not do L2, L3, and L4 connection. It will restore the settings saved in RTC Memory and applies it, then it reads the sensor again, combines it with the previously iteration values stored in RTC RAM, and sends sensor data over Wi-Fi to configured Server. Before sending data, it does a link check which is nothing but sending ARP request to check AP status as before.

If ‘wakeup count’ isn’t equal to ‘user configured data send count’, main application (tls lp base) loads a small footprint binary known as Applet1 which reads sensors, stores the sensor data in RTC RAM, and goes to standby without starting the network stack, file system, supplicant, and WLAN.

A feature called ‘dynamic keep Alive’ makes sure that connection with the AP is always maintained. It works as follows:

A pre-condition for this feature to work is: ‘user configured timer’ (on its expiry, send data to configured server) should be >1 minute and <= 5 minutes. On wakeup from standby, when GS node wants to send sensor data, it first does link check which is nothing but sending of ARP packet to check whether AP is still connected. Some AP’s might have thrown GS node out of the network because of inactivity and hence will send disassociation as response to ARP packet. On receiving this event, GS node will restart the association, gets connected to the AP and sends ARP request to AP which will be replied successfully. Then GS node reads sensor data and sends it to the configured Server. In parallel, GS node starts a keep alive timer for round down value of "user configured timer". (Eg.1 If “user configured timer” is 4 minutes and thirty seconds, then ‘keep alive timer’ will be rounded down to 4 minutes, Eg.2 If “user configured timer” is 2 min and 45 seconds, then ‘keep alive timer’ will be rounded down to 2 minutes). On expiry of timer, GS node wakes up, sends Grat ARP, switches on the receiver for few milliseconds in case AP sends disassociation event as response and if not goes to standby. AP on seeing this GRAT ARP will either make this entry into its ARP table or might send disassociation event. If GS node doesn’t receive disassociation event for 3 continuous Grat ARP packets, then it means that ARP table entry of the AP has been successfully updated periodically.
and won’t be throwing out GS node from the network. From the next wakeup, onwards, GS node won’t be sending link check (ARP packets) before doing sensor data send. Also, whenever keep alive packet (which is nothing but Grat ARP) is sent, it won’t be switching on the receiver thus saving power.

If AP sends disassociation event for the GRAT ARP request, GS node will again associate with the AP, get IP address, do link check. If link check is successful, it will do sensor read and send data to configured server successfully. Then it goes ahead and starts the keep alive timer but this time, timer will be configured for [around down value of “user configured timer” - 1 minute] which is 3 minutes in case of Eg.1 and 2 minutes in case of Eg.2. Then the same process will be continued. If AP again sends deauth, the keep alive timer will be reduced by 1 more minute and process repeated. If AP still throws GS node even when keep alive wakeup timer has been configured for 1 minute, then GS node goes ahead and stop the timer and just do the link check which is nothing but ARP sending every time before the sensor data packet is sent.

If periodic sending of data interval is above 5 minutes, then none of this applies and GS node will only do link check i.e. sending of ARP each time before sending data.

Operation:

If macro DEFAULT MODE is set to GSN_INFRA_CLIENT, GS node comes up as STA. ‘Basic Init Thread’ also called as TLS_LP base application thread is started which performs the following:

- Initializes clock and gets boot reason
- Initializes basic hardware, soft timer, GPIO, alarm, and UART
- Initializes persistent timer, restores persistent timer in case of normal boot. If it is cold boot, starts the user configured (default is 5 seconds) persistent timer.
- Initializes file system and loads default configuration from flash file system if it is first time boot.
- If system boot-up is because of Alarm2, changes the network mode to Limited-AP, and resets the system.
- If wakeup is from standby, restores the data from RTC RAM.
- Powers up the WLAN Core.
- Initializes the lower system modules such as exception handler, HI layer, sys ctl, and configures the clock source.
- If boot-up is because of wakeup from standby, calls API App_TriggerAppletLoad which posts a message to the Applet loader thread to load either Applet1 or Applet 2 based on RTC variable numDataTfrIttern.

If RTC variable numDataTfrIttern is equal to “data send iteration count”, Applet2 is dynamically loaded by the TLS LP base application which performs the following:

- Initializes the sensors and starts 100 ms timer for sensor stabilization.
- Releases WLAN reset so that it gets initiated.
- Reset the numDataTfrIttern count to 1.
- Initializes the supplicant and random number generator.
- Initializes the remaining system (wdd msg handler, network stack, bsd,sm,NCM).
- Initializes the interface, bsd, state machine (sm) and Network Connection Manager(NCM).
- Starts NCM to connect to the predefined SSID.
- Once successfully connected to SSID (L3 done), reads sensor data, appends the already existing sensor reading in RTC.
• Based on keep alive status, Send ARP to check whether connection with AP is still active.
• Send data over UDP/HTTPS to pre-configured server.
• If connection with the AP needs to be maintained, start the keep alive timer and on expiry send GRAT ARP.
• Save L2, L3, L4 data and goes to standby.

If the RTC variable `numDataTfrTern` is not equal to "data send iteration count", Applet1 is dynamically loaded by the TLS LP base application which performs the following:

• Initializes the sensors and starts 100 ms timer for sensor stabilization.
• Increment the `numDataTfrTern` count.
• On expiry of sensor stabilization timer, reads sensor data and stores in the RTC variable.
• Makes the `numDataTfrTern` to 0.
• Goes to standby.

This process of TLS LP main application coming up on every standby and dynamically loading of Applet1 and Applet2 based on 'wakeup count' and 'data send count' variables goes on forever. Network, supplicant and other App components as well as WLAN are loaded only when Applet2 is running since this is needed to send sensor data to configured server.
3.2.1.1 Configuration Settings for Longer Standby Time (Standby Mode)

The steps required to achieve a long standby time (up to 15 sec) in TLS Low Power (standby mode) are as follows:

**Step 1.** Enable the below flags in preIncludeFile.txt file

```plaintext
ADK_SNTP_ENABLED
APP_SEND_DATA_OVER_HTTP
APP_LINK_CHECK_USING_ARP
```

**Step 2.** In App_defaul_cfg.h, change default mode to client mode, and then edit the `standby_time` value as per the requirement (in milliseconds).

```plaintext
#define GSM_INFRA_CLIENT 0
#define GSM_INFRA_AP 0
#define GSM_INFRA_CLIENT/ (0x0, 0x0, 0x0, 0x0)
#define GSM_INFRA_AP/ (0x0, 0x0, 0x0, 0x0)
```

**Step 3.** In Applet2_main.c - `App_StoreStdbyData()`, disable `httpConnValidFlag` to store the flag value as 0 in RTC memory.
Step 4. In `App_http_data_send.c` -> `AppHttp_ConnOpen()`, disable `httpConnValidFlag` to start a new HTTP session.
3.2.2 Case 2: Event Notification to Cloud (Hibernate Mode)

**Functionality:**

In this case, GS node comes up in client mode associates with provisioned or configured Access Point. It reads the sensor data and sends it to the configured server over UDP/HTTPS. It stores the scan entries and IP Address in its RTC latch memory and goes into hibernate mode which is the lowest power consuming state of GS node (about 260nA of current is drawn in this mode). Wakeup is only through Alarm1 or Alarm2 which are RTC I/O pins. On wakeup, which happens only on Alarm press, GS node re-associates. This time it does not scan since entry is already stored and DHCP procedure will be lean where GS node will request for the same IP address it was given during the first association. Once successful, it reads and sends sensor data to configured server over UDP/HTTPS.

As mentioned previously, wakeup from Hibernate Mode is only through 2 RTC Alarms. When Alarm1 is pressed, sensor data is sent to configured server and when Alarm2 is pressed, operation mode is changed to Limited AP.

**Operation:**

If macro DEFAULT MODE is set to GSN_INFRA_CLIENT and macro APP_HIBERNATE_ENABLE is enabled, GS node comes up as STA. ‘Basic Init Thread’ also called as TLS_LP base application is started which performs the following:

- Initializes clock and gets boot reason
- Initializes basic hardware, soft timer, GPIO, alarm, and UART
- Initializes file system and loads default configuration from flash file system if it is first time boot.
- If system boot-up is because of Alarm2, changes the network mode to Limited-AP, and resets the system.
- If wakeup is from Hibernate boot, restores the hibernate data from RTC RAM.
- Powers up the WLAN Core.
- Initializes the lower system modules such as exception handler, HI layer, sys ctl, and configures the clock source.
- Calls API App_TriggerAppletLoad which posts a message to the Applet loader thread to load Applet 2 always.

Applet2 is dynamically loaded every time by the TLS LP base application whenever Alarm is pressed which performs the following:

- Initializes the sensors and starts 100 ms timer for sensor stabilization.
- Releases WLAN reset so that it gets initiated.
- Initializes the supplicant and random number generator.
- Initializes the remaining system (wdd msg handler, network stack, bsd,sm,NCM)
- Initializes the interface, bsd, state machine (sm) and Network Connection Manager(NCM)
- Starts NCM to connect to the predefined SSID. If Hibernate mode, then it will take the scan entries and goes ahead with Association. Also, it will be a lean DHCP where it will request for the last saved IP address.
- Once successfully connected to SSID (L3 done), reads sensor data and sends it over UDP/HTTPS to pre-configured server.
- Saves scan entry, IP address, PSK in RTC latch memory and goes to Hibernate to be woken up only by the RTC Alarm pins.
3.2.3 Case 3: Always Cloud Connected (PS POLL Applet Mode)

**Functionality:**

This mode also known as ‘PS POLL Applet mode’ where GS node will be always in low power associated state waiting for Asynchronous events from the Cloud Server on the network. Low Power is achieved by going into standby between the configured beacons interval. The beacon interval is customizable and has a default wakeup on 5 beacons. So, GS node will wake up only after 5 beacons. Only Unicast data will be buffered on the AP side which will be received by GS node after 5 beacons by using PS poll mechanism. If broadcast data comes during this time, it will be missed.

The 'static keep Alive’ features wherein GS node sends Grat ARP to the AP every 60 seconds. This way the connection with the AP is always maintained.

**Operation:**

If macro DEFAULT MODE is set to GSN_INFRA_CLIENT and macro APP_OPT_PS_IN_PSPOLL is enabled, GS node comes up as STA. ‘Basic Init Thread’ also called as TLS_LP base application thread is started which performs the following:

- Initializes clock and gets boot reason
- Initializes basic hardware, soft timer, GPIO, alarm, UART and persistent timer.
- Initializes file system and loads default configuration from flash file system if it is first time boot.
- If system boot-up is because of Alarm2, changes the network mode to Limited-AP, and resets the system.
- Initializes the lower system modules such as exception handler, HI layer, and sys ctl.
- Releases the reset of the WLAN and sends boot-up request informing WLAN that the APP CPU is operating in PS POLL Applet Mode
- Registers for various events such as DAB Ready (Data After Beacon), DAB Done, Disassociation event (Sync loss event, beacon timer reset event) with wireless device driver (wdd).
- Also, indicates to WLAN to wake up on configured beacon interval.
- WLAN wakes up and checks the boot-up message, on seeing the App CPU is functioning in PS poll Applet Mode, it too will load a small footprint binary which is capable of handling only a subset of events (DAB Ready, DAB Done, Disassociation event, sync loss event, beacon timer reset). It will check the beacon to see if any data is present for the node. Accordingly, it will send the event to the APP CPU.
APP CPU will take the following action:

**If data is present:**

- It receives an event DabReady event from WLAN CPU.
- It will load the Applet2 code dynamically which does the full initialization of the System, starts NCM, initializes the network interface, and be ready to receive data packets.
- It calls and sends an Interface Resume event to WLAN.
- WLAN on seeing this event will load the STA relevant binary, fetches data, and sends it to APP CPU.
- APP CPU will receive the buffer data from the AP and might send a response back.
- WLAN CPU on seeing no more data from router will send DabDone indication to APP CPU.
- APP CPU on receiving this event will stop NCM, saves the parameters in RTC, and puts the system in standby.

**If data is not present:**

- APP CPU receives an event DabDone event from WLAN CPU.
- APP CPU on receiving this event will put the system into standby.
Chapter 4. Architecture

The TLS Low Power ADK architecture consists of GS node working in limited AP/STA mode. Limited-AP mode is used only for provisioning the module and for performing Over the Air Firmware Upgrade. It is in STA mode that GS node showcases low power functionality wherein it reads sensor data and sends it over UDP or HTTPS to the configured server.

Limited AP Mode

Figure 1: Limited AP mode

On powering up the EVB board, if DEFAULT_MODE MACRO is set to GSN_LIMTED AP, the GainSpan Wi-Fi module establishes itself as a Limited AP (SSID: GS_PROV_TLSLP_xxxxxx) with WPA2 security (password being GS_xxxxxx, where "xxxxxx" is last 6 digit of Limited-AP MAC address which is 1 more than STA interface MAC address mentioned on the module). GS node acts as an HTTP server, which is a TCP based protocol that establishes the communication channel. Users browse through the list of available wireless networks on their smartphones or laptop which is the client and connects to the SSID GainSpanDemo_IOT. The TLS mobile application leverages discovery provided by the TLS embedded firmware application to enable automatic discovery of available profiles and services. Client launches the TLS application to initiate HTTP communication and acts as an HTTP client in this process.
Client Mode

Once the GS node is provisioned or if DEFAULT_MODE MACRO is set to GSN_INFRA_CLIENT at compile time, GS node comes up in client mode and connects to the provisioned SSID or SSID information provided during compile time and sends sensor data to configured server over UDP/HTTPS.

The TLS LP ADK demo is possible with any GS2K modules since the provision to send dummy data on modules which does not have sensors connected on their EVB’s (E.g.: GS2100 Modules). GS2011 EVB’s have an in-built temperature and light sensors which communicate with the APP CPU through the 12-bit SAR ADC.

Note: GS2100 based modules are not suitable for battery applications since they do not have the DC-DC control pins, and hence the power consumed in standby state will be much higher (~50µA) when compared to GS2011 numbers (~6µA).

Figure 2: Client Mode
4.1 COMPONENTS

4.1.1 Hardware

GainSpan TLS LP ADK board consists of following hardware components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GainSpan Wi-Fi Module</td>
<td>GS2K module reads and sends sensor data over Wi-Fi.</td>
</tr>
<tr>
<td>LEDs</td>
<td>Timer expiry LED (GPIO31).</td>
</tr>
</tbody>
</table>

Table 2: Hardware Components
4.1.2 Software

GainSpan TLS LP ADK uses the following software components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS Low Power Embedded Firmware Application</td>
<td>Compilation results in 4 binaries and a lib:</td>
</tr>
<tr>
<td></td>
<td>a. app bin</td>
</tr>
<tr>
<td></td>
<td>b. applet_1.bin</td>
</tr>
<tr>
<td></td>
<td>c. applet_2.bin</td>
</tr>
<tr>
<td></td>
<td>d. applet_3.bin</td>
</tr>
<tr>
<td></td>
<td>e. tls_lp_main.a</td>
</tr>
</tbody>
</table>

All the above binaries and lib are combined to generate a single app binary tls_lp_app.bin

Table 3: Software Components
4.2 Tasks

The TLS LP architecture contains different tasks being executed and they are shown in the following schematic representation:

**Figure 4: TLS Low Power Architecture Application Flow**

From an implementation perspective, TLS Low Power ADK comprises of four small applications described as follows:

- **TLS LP main application:** Also called as the base application is responsible for creating a task/thread AppletLoaderTask which in turn is responsible for loading applets dynamically. TLS LP main is executed in context of boot task itself. The base application creates another thread App_BasicInitThreadEntry which performs basic initializations such as clock control, soft timer, GPIO, alarm, UART, file system, and loads the default configuration. This thread checks the reason for boot up and acts accordingly. If wakeup
A
Rev. 2.3
Page 28 of 109

reason is Alarm2, then it sets the network mode to Limited-AP, resets the System, and in next boot up, applet_3 is loaded where GS node come up as Limited-AP. If cold boot is the boot reason, it starts the 5 second sensor read timer. On expiry of the sensor read timer, based on RTC variable 'numDataTfrItem', applet_1 or applet_2 is loaded dynamically.

**Applet_1 application:** Applet_1 is an independent application which is loaded dynamically on alternative boots by the TLS LP main App when the RTC member numDataTfrItem is 1. This applet shows the ability to wake up without WLAN. In this cycle of boot up, applet_1 firmware performs timer component initialization, ADC component initialization (which is required for sensor reading). It also starts a timer for sensor stabilization. Once the sensors are stabilized, which takes around 100ms, the GS module reads the temperature and light sensors through 12-bit SAR ADC which is connected internally to the APP CPU on the EVB and stores the values in RTC RAM. In the next step applet_1 firmware sets the RTC variable 'numDataTfrItem' to 1 and puts the system to standby for 5 seconds. Applet_1 is executed in the context of newly created thread “Applet1 Thread”.

- **Applet_2 application:** Applet_2 is an independent application which is loaded dynamically on alternative boots by the TLS LP main App when the RTC member numDataTfrItem is 0. "Applet2 Thread" is created which initializes the complete system including network stacks, WPA supplicant, and file system. NCM (Network Connection Manager) is also initialized and started. Applet_2 releases the reset of WLAN to boot up and initializes on its own. NCM module takes data from configuration file, associates to the access point, and sends sensor data (2 sets of Temperature and Light reading data) to the pre-configured Server.
  - The sending of data is showcased using
    - UDP
    - HTTPS
  - Once data is sent, module goes back to standby.

if macro APP_OPT_PS_IN_PSPOLL, then GS node will come up in PS poll applet mode ready to receive asynchronous data. Low power is achieved by going into standby between DTIM intervals.

S node wakeups from standby every 5 seconds loads the default TLS LP main binary which dynamically loads Applet_1 and Applet_2 alternatively. This keeps on repeating infinitely.

- **Applet_3 application:** Applet_3 is an independent application which is loaded dynamically on cold boot when default mode is set to Limited-AP at compile time or when Alarm2 is pressed. “Applet3 Thread” is created which initializes the complete system including network stacks, WPA supplicant, file system and NCM (Network Connection Manager). Applet_3 releases the reset of WLAN to boot up and initializes on its own. NCM module takes data from configuration file and creates a limited-AP.

GS node wakeups from standby every 5 seconds, loads the default TLS LP main binary which dynamically loads Applet_1 and Applet_2 alternatively. This keeps on repeating infinitely.
Chapter 5. Embedded Application Execution Sequence

This section describes the complete TLS Low Power ADK application flow in detail along with code snippets and sniffer logs.

5.1 BUILD AND COMPIATION PROCESS

1. Download the SDK package from the SDK Builder. Customer needs to have relevant permission to download the complete SDK package. Unzip the package and open the workspace “tls_lp.eww” from path \Embedded\adk\tls_low_power\build\. All the four applications tls_lp main, applet_1 and applet_2, and applet_3 are built as three separate binaries (linked later to form a single application image tls_lp_app.bin). The following snapshot shows all the four projects.

   ![Figure 5: TLS Low Power IAR Workbench](image)

2. Press key F8 and following screen appears on the IAR. Click on Rebuild All for compilation of all the projects.
3. Compilation results in generation of three separate binaries for each of the project app.bin, applet_1.bin and applet_2.bin, applet_3.bin along with library tls_lp_main.a.

4. Following are details of each binary and library generated with information such as which project files are needed to generate it, what .c files it comprises, and what is the size of the generated binary.

1. **Tls_lp main:**
   - This code performs basic initializations such as clock control, soft timer, UART initialization, and so on. This application starts the “Applet Loader” task and “Basic Init” task. Following is the list of files used to build this binary:
     - tls_lp_main.eww => source file listing as shown in the following snapshot
     - tls_lp_main.icf => configuration file
     - tls_lp_main.stf => steering file
     - PostBuild_tls_lp_main.bat
   - Consider the map file, the code is split as follows:
     - Total Code size in module flash : ~ 161KB (Read only code and Read only data considered).
     - GEPS code in Auto Cache (flash) : ~122 KB
     - App Code in RAM : ~ 10 KB
     - Driver code in RAM : ~ 10 KB
     - GEPS code in RAM : ~ 12 KB
     - rom patch fpt in RAM : ~ 7 KB
   - Only ~39KB of code is loaded into RAM.

2. **Applet_1:**
   - This code is loaded and runs at every odd wakeup cycle without waking up the WLAN. Following is the list of files used to build this binary:
     - applet_1.ewp
     - applet_1.icf
   - Consider the map file, the code is split as follows:
     - Total Code size : ~ 3KB (Read only code & Read only data considered).
   - Only ~3KB of code is loaded into RAM as part of applet_1.bin.

3. **Applet_2:**
   - This code is loaded and ran at every even wakeup cycle. Following is the list of files used to build this binary:
     - applet_2 ewp
     - applet_2.icf
Consider the map file, the code is split as follows:

- **Total Code size**: ~ 15KB (Read only code & Read only data considered)
- Only ~15KB of code is loaded into RAM as part of applet_2.bin.

### 4. Applet_3:

- This code is loaded and ran when GS node is configured as Limited-AP at compile time or when Alarm2 is pressed. This kicks up a thread "Applet 3 thread" which will create a Limited-AP. Following is the list of files used to build this binary:
  - applet_3.ewp
  - applet_3.icf

Consider the map file, the code is split and details are as follows:

- **Total Code size**: ~ 27KB (Readonly code & Readonly data considered).
- Only ~27KB of code is loaded into RAM as part of applet_3.bin.

### 5. tls_lp_app.a:

- Any binary can refer the APIs and global data of the binaries built earlier to it directly.
  - For example, Applet_1 can call APIs of tls_lp_main. Applet_2 can call APIs from tls_lp_main and applet_1 (Provided both binaries are up and running at the time of calling their respective API).
- This is a library generated by tls_lp application binary during its post build procedure which exports the symbols to be exposed to next compiled binaries. The symbols are exported through an elf file (tls_lp_main.a). This post build procedure takes steering file and the map file as inputs and creates a "tls_lp_main.a" file. The steering file (tls_lp_main.stf) present in the build folder contains the list of API's which are exposed to both the applets. The steering file content (tls_lp_main.stf) is as follows:

```
/*-----------------------------------------*/
hide *
```

---

**Location:** 80560NT11589A Rev. 2.3

**Date:** 2018-05-04

**Page:** 31 of 109
show auto_arp_entry_disable
show arp_resolve_notify_handler
show arp_resolve_notify_handler_ctx
show arp_notify_ip_address
show nx_packet_tx_confirm_cb
show nx_packet_tx_confirm_cb_ctx
show nx_bsd_socket_extn_array
show nx_bsd_set_error_code_fnptr
show x509_cert_validation_skip
show globalGsnPwm
show PAdcHandle
show i2cGlobalHandle
show HttpEncodeCb
show HttpRecvCb
show mDNSock6
show sdadcadc_avdden
show ptr_cal_otp_data
show cal_otp_data
show gsn_temp_sen
show globalGsnSdio
show cis0Buffer
show cis1Buffer
show csaMemory
show aes_256_ctr_decrypt
show aes_256_ctr_encrypt
show GsnSysCtl_BootReq_Patch
show GsnSysCtl_Init_Patch
show _nxe_arp_entry_delete
show GsnSntp_RadioModeSet
show App_WlanBootReq
show AppConfig_OTPMacRead
show App_WifiInit
show APP_ssl_cert_table

/********************************************/
Similarly, we have applet_1_SL.stf file which exposed couple of API which can be called by Applet2.

NOTE – “hide *” means to hide all the symbols except the selective list of symbols (functions and global variables) which are mentioned in the file. This list will be added to the newly created tls_lp_prefetch.a library. The post build action during tls_lp_main (tls_lp_main.ewp) project will invoke PostBuild_tls_lp_main.bat batch file which creates the tls_lp_prefetch.a as shown below.
5.2 TLS LP APPLICATION CODE FLOW

On bootup, The TLS LP Application binary also called the base binary is loaded using the same mechanism as any other application: boot rom -> app startup code -> tls lp code load, remap, call app_main(). app_main is executed in boot ROM context and it performs the following:

1. Re-initializes the Flash access and do the boot process.
2. Initializes power management module and sets the deep sleep state to active so that System does not go into deep sleep in idle loop.
3. Calls App_BasicInitEntry to create a new thread “Basic Init Thread” which performs basic initialization
4. Starts an idle loop which is the lowest priority task. This task will put the system in sleep/Deep sleep whenever possible (When no other thread is running, i.e. no activity is being done in the system)

INT32 App_Main(VOID)
{
    GsnFlashif_FlashAccessReinit();
    App_BootProcess();
    GsnPwrMgmt_Init(&appCtx.pwrMgmtCtx);
    GsnPwrMgmt_DpSlpActivitycbSet(App_PwrMgmtDpSlpActivityCb);
    GsnPwrMgmt_DpSlpActivitySet(&appCtx.pwrMgmtCtx, (GSN_DP_ACTIVITY_BIT_MAP_T)APP_DPSLP_ACTIVITY_BASIC_INIT_INPROGRESS);
    App_BasicInitEntry();
    while(1)
    {
        __WFI();
    }
}

5.2.1 Basic Init Thread

App_BasicInitEntry is called from app_main in bootrom context will result in creation of new thread “Basic Init Thread” which has a priority of 18. This has an entry function App_BasicInitThreadEntry which in turn calls App_BasicInit which does the following:

- Initializes clock, gets boot reason
- Initializes basic hardware, soft timer, GPIO, alarm, and UART
- Initializes persistent timer, restores persistent timer in case of normal boot and if it is cold boot,
- Starts the user configured (default is 5 seconds) persistent timer.
- Initializes file system and loads default configuration from flash file system if it is first time boot.
- If system bootup is because of Alarm2, change the network mode to Limited-AP, and resets the system.
- If wakeup is from standby, restore the data from RTC RAM.
- Power up the WLAN Core.
- Initializes the lower system modules like exception handler, hi layer, sys ctl and configure the clock source.
- Calls App_AppletLoaderInit to create ‘Applet Loader thread’.
• Calls App_TriggerAppletLoad which posts a message to the Applet loader thread queue (appletLoaderMsgQueue) to load either Applet1, Applet2 or Applet3 based on some condition.

5.2.2 Applet Loader Thread

This thread is created from app_main by calling App_AppletLoaderInit. This newly created thread has a priority of 25. This thread has an entry function App_AppletLoaderThreadEntry which waits on the queue appletLoaderMsgQueue infinitely. When a message is posted to the queue for loading Applet, it calls the App_AppletLoadAndRun with AppletId and flag as parameter which loads the Applet to the SRAM dynamically.

```c
 App_AppletLoaderInit(VOID)
{
    /*Create app loader thread*/
    UINT32 msgSizeInWords;
    msgSizeInWords = GsnOsal_QueueMsgSizeInWords(sizeof(APPLET_LOAD_MSG_T));
    /*Creating the Mailbox to receive messages*/
    GsnOsal_QueueCreate( &appletLoaderMsgQueue, msgSizeInWords,
        ( UINT8* )appletLoaderMsgQueueBuf,
        sizeof( appletLoaderMsgQueueBuf ));

    GsnOsal_ThreadCreate(App_AppletLoaderThreadEntry, NULL,
        &appletLoaderThreadCtx,
        "Applet Loader Thread", 25,
        (UINT8 *)appletLoaderThreadStack,
        sizeof(appletLoaderThreadStack),
        GSN_OSAL_THREAD_INITIAL_READY);
}
```

```c
App_AppletLoaderThreadEntry(UINT32 ctx)
{
    ...
    while(1)
    {
        /*wait on the message queue to get an event to load or unload the requested applet*/
        GsnOsal_QueueGet(&appletLoaderMsgQueue, (UINT8 *)&msg, GSN_OSAL_WAIT_FOREVER);
        /*Load and run the requested Applet*/
        if(msg.flags & APPLETLOAD_FLAG_LOAD)
        {
            App_AppletLoadAndRun(msg.appletId, msg.flags);
        }
        else if(msg.flags & APPLETLOAD_FLAG_UNLOAD)
        {
        }
        else if(msg.flags & APPLETLOAD_FLAG_ERASE_CONFIG_FILE)
        {
            AppConfig_Erase();
            GsnSys_Reset(GSN_RESET_REASON_APP_SW_RESET);
        }
    }
}
```

The function App_AppletLoadAndRun loads either Applet1, Applet2, or Applet3 based on AppletId. How this is achieved is explained in the following section:

1. Applet information is part of the control block which in turn is part of single image (gs2000_SingleImage.bin) where the control block is located at address 0x00.
Following are the elements of the control block structure.

![Control Block Diagram](image)

**Figure 6: Elements of the Control Block**

2. The control block of 5.3.0 GA indicates the size of control block as 0x8C or 140 bytes which implies that there are 7 Block Descriptors (each 20 bytes). First three are for Application, WLAN, and web Image. The fourth, fifth, and sixth Block descriptors are for Applet1, Applet2, and Applet3 as seen in the following diagram:

![Control Block Diagram](image)

**Figure 7: Control Block**
3. Following diagram shows the Block descriptor information of both the Applets. Only the offset address along with owner of the block is a valid value. For Applets, the owner of the block will have unique Id’s (1001, 1002, 1003).

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Applet 1 Block Descriptor</th>
<th>Applet 2 Block Descriptor</th>
<th>Applet 3 Block Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset (4 Bytes)</td>
<td>00075c72</td>
<td>00076aa2</td>
<td>0007a3fe</td>
</tr>
<tr>
<td>Size (4 Bytes)</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>Program Address (4 Bytes)</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>SRAM Address (4 Bytes)</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
<tr>
<td>Owner of Block (2 Bytes)</td>
<td>1001</td>
<td>1002</td>
<td>1003</td>
</tr>
<tr>
<td>Flag (2 Bytes)</td>
<td>xxxx</td>
<td>xxxx</td>
<td>xxxx</td>
</tr>
</tbody>
</table>

4. The offset address specifies where the Applet block descriptor of individual Applet would be present. This would have all the relevant information pertaining to that Applet. The generic structure of Applet block descriptor header will have the following elements.

typedef struct GSN_APPLET_HDR_S
{
    uint8_t alok[4];            /* signature “alok” */
    uint32_t rev;               /* revision of this structure */
    uint32_t size;              /* size of the applet in flash including applet header */
    uint32_t progRamEntryAddr;  /* virtual entry address of the applet */
    uint32_t codeSectionBegin;  /* virtual address of code section */
    uint32_t codeSectionEnd;
    uint32_t fptSectionBigin;
    uint32_t fptSectionEnd;
    uint32_t dataInitSectionBegin;
    uint32_t dataInitSectionEnd;
    uint32_t dataSectionStart;
    uint32_t dataSectionEnd;
    int32_t appletSramAddr;
} GSN_APPLET_HDR_T;

5. From the previous diagram of the control block, Applet 1 is at offset 0x00075c72, Applet 2 is at offset 0x00076aa2 and applet 3 is at offset 0x0007a3fe. The applet header information which is 60 bytes is found in single image.

![Figure 8: Applet 2 Header Descriptor](image-url)
When a call is made to load a Applet, `App_AppletLoad` is the API called along with Applet ID and flags as parameters as shown:
App_AppletLoad(3, flags)

This will put a message in appletLoaderMsgQueue. App_AppletLoaderThreadEntry will process messages from this queue and calls App_AppletLoadAndRun to load the particular applet.

App_AppletLoadAndRun(msg.appletId, msg.flags);

In this function, API App_FindAppletBlockDesc will first find the block descriptor for the given applet number

which provides address of the applet code entry function. Once the offset is found, it is added to romFwImageFlashAddr to get the absolute address of the Applet.

/* Get applet Firmware flash address. */

appletFwImageFlashAddr = romFwImageFlashAddr + appletBlkDesc.offset;

Copy the 60 byte Applet header which has relevant information of various code section pertaining to that Applet. Using that, the Applet will be loaded dynamically.
USE CASE 1: PERIODIC DATA UPLOAD TO CLOUD (STANDBY MODE)

5.2.3 Using UDP Server

This use case demonstrates sending of sensor data on every `APP_DATA_SEND_ITERATION` to a configured UDP server. GS node wakes up every `APP_DATA_READ_INTERVAL` increments the RTC wakeup count variable numDataTfrItem. If numDataTfrItem is equal to `APP_DATA_SEND_ITERATION`, then Applet2 will be loaded. This initializes the full system as well as releasing the reset of WLAN. It then associates with provisioned AP gets IP address, does a link check which is nothing but a ARP request to Access Point to check whether the connection is alive. If successful, reads the sensor sends it over WiFi. GS node then saves the configuration information for L2, L3, and L4 on RTC RAM and goes to standby.

If numDataTfrItem is not equal to `APP_DATA_SEND_ITERATION`, then Applet1 will be loaded which initializes and reads sensor data, stores in RTC RAM, and goes to standby.

5.2.3.1 Macros

Following are the macros required to be enabled or modified for sending sensor data using UDP (app_default_cfg.h and preIncludeFile.txt):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_MODE</td>
<td>This macro needs to be set to GSN_INFRA_CLIENT.</td>
</tr>
<tr>
<td>DEFAULT_CLI_SSID</td>
<td>This macro specifies the SSID of the Access Point to which GS node will be connected.</td>
</tr>
<tr>
<td>APP_DATA_READ_INTERVAL</td>
<td>This macro specifies the duration for which GS node will be in standby between boot-ups. By default, it is 5 seconds.</td>
</tr>
<tr>
<td>APP_DATA_SEND_ITERATION</td>
<td>This macro specifies the iteration number based on which sensor data is sent to the configured number. If iteration number is not matched, GS node only reads sensor data and saves it in RTC RAM. The maximum entries that can be saved in RTC is 32.</td>
</tr>
<tr>
<td>DEFAULT_CLI_CHNL</td>
<td>This macro specifies the channel on which SSID is operating.</td>
</tr>
<tr>
<td>DEFAULT_CLI_SECURITY</td>
<td>This macro specifies the type of security used on the Access Point.</td>
</tr>
<tr>
<td>DEFAULT_CLI_PASSPHRASE</td>
<td>This macro specifies the passphrase of the Access Point.</td>
</tr>
<tr>
<td>UDP_SERVER_ADDRESS</td>
<td>This macro specifies the IP Address of the Server receiving the sensor data.</td>
</tr>
<tr>
<td>UDP_SERVER_PORT</td>
<td>This macro specifies the port number of the UDP Server.</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>APP_UDP_DATA_RECEIVE_ENABLE</td>
<td>This macro needs to be enabled when response from UDP Server is expected.</td>
</tr>
<tr>
<td>UDP_RECEIVE_TIMEOUT</td>
<td>This macro specifies the wait time for response from UDP Server. Default is 1 second.</td>
</tr>
<tr>
<td>APP_MAC_LEVEL_CONFIRM_REQUIRED</td>
<td>This macro is applicable only for data sent over UDP. MAC level confirmation for the UDP packet sent is checked after the transmission.</td>
</tr>
<tr>
<td>APP_WIFI_LINK_KEEPALIVE</td>
<td>Enable this macro along with APP_LINK_CHECK_USING_ARP to send keep alive messages dynamically so that the connection with AP is maintained across standby.</td>
</tr>
<tr>
<td>APP_LINK_CHECK_USING_ARP</td>
<td>If this macro is enabled, ARP is done to gateway after every N wakeup cycles to check if the L2 connection is intact.</td>
</tr>
</tbody>
</table>

On SDK Builder, customer is required to populate “Frequency of data upload” and “No of times product shall read sensor per data upload” which will be equivalent to MACRO’s APP_DATA_READ_INTERVAL and APP_DATA_SEND_ITERATION.

APP_DATA_READ_INTERVAL = Frequency of data upload/No of times product shall read sensor per data upload.

APP_DATA_SEND_ITERATION = No of times product shall read sensor per data upload.

Figure 10: SDK Builder Configuration
5.2.3.2 Code Flow

The code flow for the use case is as follows:

On boot-up, the TLS LP Application binary also called as the base binary is loaded using the same mechanism as any other application:

Boot ROM -> App startup code -> TLS LP code load, remap, call app_main().

‘app_main’ is executed in boot ROM context and it performs the following:

1. Re-initializes the Flash access and performs the boot process.
2. Initializes power management module and sets the deep sleep state to active so that System does not go into deep sleep in idle loop.
3. Calls App_BasicInitEntry to create a new thread “Basic Init Thread” which performs basic initialization
4. Starts an idle loop which is the lowest priority task. This task will put the system in sleep/Deep sleep whenever possible (When no other thread is running, i.e. no activity is being done in the system)

INT32 App_Main(VOID)
{
   GsnFlashif_FlashAccessReinit();
   App_BootProcess();
   GsnPwrMgmt_Init(&appCtx.pwrMgmtCtx);
   GsnPwrMgmt_DpSlpActivitycbSet(App_PwrMgmtDpSlpActivityCb);
   GsnPwrMgmt_DpSlpActivitySet(&appCtx.pwrMgmtCtx, (GSN_DP_ACTIVITY_BIT_MAP_T)APP_DPSLP_ACTIVITY_BASIC_INIT_INPROGRESS);
   App_BasicInitEntry();
   while(1)
   {
      __WFI();
   }
}

App_BasicInitEntry

App_BasicInitEntry is called from app_main in bootrom context which results in creation of new thread called "Basic Init Thread" with a priority of 18. This has an entry function App_BasicInitThreadEntry which in turn calls App_BasicInit and performs the following:

- Initializes clock and gets boot reason
- Initializes basic hardware, soft timer, GPIO, alarm, and UART
- Initializes persistent timer, restores persistent timer in case of normal boot. If it is cold boot, starts the user configured (default is 5 seconds) persistent timer.
- Initializes file system and loads default configuration from flash file system if it is first time boot.
- If system boot-up is because of Alarm2, changes the network mode to Limited-AP, and resets the system.
- If wakeup is from standby, restores the data from RTC RAM.
- Powers up the WLAN core.
- Initializes the lower system modules such as exception handler, HI layer, sys ctl, and configures the clock source.
• Calls API App_TriggerAppletLoad which posts a message to the Applet loader thread queue (appletLoaderMsgQueue) to load either Applet1 or Applet 2 based on RTC variable numDataTfrItem.

******************************************************************************

VOID App_BasicInit(VOID)
{
    APP_RTC_SECNDRY_MEM_INFO_T *pAppRtcSecMemInfo = App_RtcSecMemInfoPtrGet();
    ........................................
    GsnClkCtl_Init(&pAppCtx->clkCtlCtx);
    bootReason = GSN_BOOT_REASON_GET();
    if(bootReason != GSN_WIF_SYS_BOOT_NORMAL_BOOT)
    {
        GsnRtc_AlarmConfig(GSN_RTC_ALARM_2, 0, FALSE);
        GsnRtc_AlarmConfig(GSN_RTC_ALARM_3, 0, FALSE);
        App_BasicHwInit();
    }
    /*Initialize Soft Timer */
    App_SoftTimerInit(pAppCtx);
    App_GpioInit();
    /* Initialize alarm */
    App_AlarmsInit(pAppCtx);
    #ifndef APP_POWER_MEASUREMENT_ENABLE
    AppDbg_UartInit();
    #endif
    AppDbg_Printf("r
r\n[TLS LP] Boot Reason = %d\n", GSN_BOOT_REASON_GET());
    #ifndef APP_HIBERNATE_ENABLE
    App_PersistentTmrInit(pAppCtx); if(bootReason != GSN_WIF_SYS_BOOT_NORMAL_BOOT)
    { #ifndef APP_OPT_PS_IN_PSPOLL
        App_SensorReadTmrStart(); #else
        /* If use case 1 and 3 both are enabled start sensor read timer */
        #ifdef APP_STATUS_UPDATE_ENABLE
        App_SensorReadTmrStart();
        #endif
        #endif
    }#endif
    else
    { /* In case of use case 3, restore persistent timer after giving boot request to WLAN. Else restore it now. */
        GsnPersistTmr_Restore();
    }
    #endif
    if(bootReason != GSN_WIF_SYS_BOOT_NORMAL_BOOT)
    {
        App_RtcDump();
    }
    /* This mutex in used AppFs_Init function. IT is also required when WLAN writes callibration data */
    GsnOsal_MutexCreate(&fsInitLock, GSN_OSA_PRI_NO_INHERIT);
}

80560NT11589A Rev. 2.3  Page 43 of 109  2018-05-04
if (1 == pAppRtcLtcMemInfo->alarm2Press)
{
    pAppRtcLtcMemInfo->alarm2Press = 0;
    GsnSys_Reset(GSN_RESET_REASON_APP_SW_RESET);
}
}

//Todo- Check whether to do it here.
#define APP_HIBERNATE_ENABLE
else if(APP_IS_HIBERNATE_BOOT() == TRUE)
{
    App_RestoreHibernateData(pAppCtx);
}
#endif
else
{
    App_ReStoreStdbyData(pAppCtx);
}

/* Check if WLAN boot is required */
if(App_IsWlanBootReq() == FALSE)
{
    goto APP_TRIGGER_APPLET_LOAD;
}
#endif

if (pAppCtx->config[APP_DFLT_CFG_INDEX].networkConfig.networkMode ==
    (GSN_NETWORK_MODE_T)GSN_WIF_MAC_IF_TYPE_WLAN_AP)
{
    GsnPwrMgmt_WlanCorePowerUp();
}
else
{
#ifdef APP_NO_SENSOR
#ifdef APP_OPT_PS_IN_PSPOLL
    GsnPwrMgmt_WlanCorePowerUp();
#endif
#endif
}

GsnExcpHdlr_Init();
App_HiInit(pAppCtx);

/*Initialise wif System Control message Handler */
App_SysCtlInit(pAppCtx);
App_IfBasicInit(pAppCtx, &pAppCtx->appNwIfc[APP_DFLT_CFG_INDEX]);

APP_TRIGGER_APPLET_LOAD:
App_AppletLoaderInit();
App_Cm3ClkCfg(&pAppCtx->cm3ClkCfg);
App_TriggerAppletLoad();

App_TriggerAppletLoad(VOID)
{
    ......
    if((GSN_LIMITED_AP == appCtx.config.networkConfig.networkMode))
    {
        /*Load web prov applet (applet 3)*/
        flags = (APPLETLOAD_FLAGS_T)(APPLETLOAD_FLAG_LOAD |
                    APPLETLOAD_FLAG_CALL_ENTRY_FUNC);
        App_AppletLoad(3, flags);
    }
    if(0 == pAppRtcSecMemInfo->pAppRtcLtchMemInfo
       || bootInfo == GSN_WIF_SYS_BOOTINFO_NORMAL_BOOT_ALARM)
    {
        /* in case of alarm wake up dont disturb the cycle*/
        if(bootInfo != GSN_WIF_SYS_BOOTINFO_NORMAL_BOOT_ALARM)
        {
            pAppRtcSecMemInfo->pAppRtcLtchMemInfo = 1;
        }
    }
    /*Load Remaining Part of the code*/
    flags = (APPLETLOAD_FLAGS_T)(APPLETLOAD_FLAG_LOAD);
    App_AppletLoad(1, flags);
    flags = (APPLETLOAD_FLAGS_T)(APPLETLOAD_FLAG_LOAD |
                    APPLETLOAD_FLAG_CALL_ENTRY_FUNC);
    App_AppletLoad(2, flags);
}
else
{
    flags = (APPLETLOAD_FLAGS_T)(APPLETLOAD_FLAG_LOAD |
                    APPLETLOAD_FLAG_CALL_ENTRY_FUNC);
    App_AppletLoad(1, flags);
}
}                      

On first boot, Applet2 will be loaded as RTC variable pAppRtcSecMemInfo->pAppRtcLtchMemInfo is 0.

On every wakeup from standby, whenever RTC member variable numDataTfritern is equal to
APP_DATA_SEND_ITERATION, TLS LP/base firmware dynamically loads Applet_2 onto RAM
location 0x20050000 as mentioned in the Applet header descriptor field appletSramAddr using
the function App_TriggerAppletLoad. Once code is loaded, the entry function in Applet 2 code
App_Applet2Entry is called as mentioned in SingleImageBuilderConfig.txt present in build
g folder.

************************************************************************
App_CreateSensorStablizeLock();
GSN_PWR_MGMT_CM3_SLEEP_DEEP_EN();

appletId=2
appletBin=./Debug\Exe\applet_2.bin
appletMap=./Debug\List\applet_2.map
appletEntryFunc=App_Applet2Entry
appletSramAddr=20050000

************************************************************************

The applet_2 entry function App_Applet2Entry performs the following:

Creates Application thread “Applet2 Thread” with priority 17. This thread has entry function
App_Applet2ThreadEntry and performs the following:

1. Starts a 100ms timer for the sensors to stabilize, only then data can be read. This is
   performed in the Applet2_main.c file.

************************************************************************
App_CreateSensorStablizeLock();
GSN_PWR_MGMT_CM3_SLEEP_DEEP_EN();

80560NT11589A Rev. 2.3 Page 45 of 109 2018-05-04
sensorStablizeTmrHndl = GsnSoftTmr_Start(&sensorStablizeTmr,
GSN_SOFT_TMR_ONESHOT, 0,
GSN_SOFT_TMR_MILLISECONDS_TO_TICKS(90),
App_SensorStabilizeTmrCb,
NULL);
(void)(sensorStablizeTmrHndl); /*Compilation Warning suppression*/
App_WaitOnSensorStablizeLock();

************************************************************************
2. Sends boot request for WLAN using the API App_WlanBootReq.
3. Initializes the supplicant and random number generator.
4. Initializes wdd msg handler, network stack, interface, BsdLayer, state machine (sm),
    Network connection manager (ncm), and then posts an event
    (APP_EVENT_NW_START_NCM) to the main task to start NCM using API
    App_applet2TaskNotify(APP_EVENT_NW_START_NCM, 0).
5. Applet2 thread infinitely waits on messages on the queue applet2ThreadMsgQueue.

************************************************************************
6. NCM takes the configuration information from the app_default_config.h and connects to
    the SSID. Once GS Node is successfully connected and IP address has been allocated
    which is received from the AP, it sends out APP_EVENT_NW_CONNECT_DONE to the
    sm.
7. Applet2 main task processes this event and checks whether we need to send the sensor
data. If sensor need not be sent, Applet2 main thread breaks from the case statement.
If sensor data needs to be sent, then Applet2 proceeds with the next event
    APP_EVENT_DATA_SEND which is triggered periodically (every 5 seconds)
    from the TLS LP code using the API App_applet2TaskNotify (APP_EVENT_DATA_SEND, 0).
8. As part of APP_EVENT_DATA_SEND event handling, Applet2’s sm thread checks
    whether number of data transfers has equaled DEFAULT_LINK_CHECK_ITERATION.
    If so, then to check whether GS Node is associated with the Access Point, Applet2 main
    task invalidates and deletes the ARP entry. When UDP data is sent the next time, the
    lower layer of network stack triggers an ARP for the AP. The return value informs
    whether ARP was successful or not. If ARP fails, Applet2 main task triggers a L2 sync
    loss which restarts the NCM process. This step is done as recovery mechanism for the
    Application in case the AP on the other side has been switched off.
9. Calls AppSm_DataSend to read sensors.
10. Calls AppTls_SensorRead to initialize the ADC component and read the temperature
    and light sensor values into the RTC array.
11. Increments UdpSeqNo and stores it in RTC.
12. Calls API App_DataSend to send data. Check for ARP entry of the gateway and calls
    API App_DataSend. Based on the MACRO defined, send data via UDP using call
    AppUdp_DataSend (pAppCtx)
13. In function AppUdp_DataSend, if intended to receive MAC level confirmation for the
    UDP packet sent, enable the macro APP_MAC_LEVEL_CONFIRM_REQUIRED. Once
    the macro is enabled, the Applet2 thread registers for callback with the WLAN
component so that WLAN sends status of the packet informing whether it was successfully acknowledged by the gateway or not using the API App_PacketTxCnfrmCbReg(App_UdpTxConfirmCb, pAppCtx).

14. If intended to receive data from UDP Server, enable the macro APP_UDP_DATA_RECEIVE_ENABLE. Once the macro is enabled, Applet2 thread calls api AppUdp_DataRecv to wait for the configured time to receive response back from the UDP Server.

15. Calls api App_GoToLowPowerState to go into standby using App_GotoStandby.

NOTE – Code flow which is under Keep Alive MACRO “APP_WIFI_LINK_KEEPALIVE” cannot be seen.

### 5.2.3.3 Demo

Perform the following steps for the demo:

**Step 5.** The macros mentioned should be enabled or populated with proper values in app_default_config.h, and corresponding flags needs to be enabled from preIncludeFile.txt.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_MODE</td>
<td>GSN_INFRA_CLIENT</td>
</tr>
<tr>
<td>DEFAULT_CLI_SSID</td>
<td>&quot;ASUS&quot;</td>
</tr>
<tr>
<td>APP_DATA_READ_INTERVAL</td>
<td>5000 // standby is for 5 second</td>
</tr>
<tr>
<td>APP_DATA_SEND_ITERATION</td>
<td>2 // on every even iteration it will send data</td>
</tr>
<tr>
<td>DEFAULT_CLI_CHNL</td>
<td>11</td>
</tr>
<tr>
<td>DEFAULT_CLI_SECURITY</td>
<td>GSN_WLAN_WPA_PERSONAL</td>
</tr>
<tr>
<td>DEFAULT_CLI_PASSPHRASE</td>
<td>“xxxxxxxx” //xxxxxxx will be replaced with AP pwd</td>
</tr>
<tr>
<td>UDP_SERVER_ADDRESS</td>
<td>192.168.2.195</td>
</tr>
<tr>
<td>UDP_SERVER_PORT</td>
<td>5000</td>
</tr>
<tr>
<td>APP_UDP_DATA_RECEIVE_ENABLE</td>
<td>This is defined in preIncludeFile.txt, remove the __ to enable</td>
</tr>
<tr>
<td>UDP_RECEIVE_TIMEOUT</td>
<td>6000 // default is 1000</td>
</tr>
<tr>
<td>APP_MAC_LEVEL_CONFIRM_REQUIRED</td>
<td>This is enabled by default in file preIncludeFile.txt</td>
</tr>
</tbody>
</table>
### Macro Description

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_WIFI_LINK_KEEPALIVE</td>
<td>Enable this in file preIncludeFile.txt</td>
</tr>
<tr>
<td>APP_LINK_CHECK_USING_ARP</td>
<td>Enable this macro.</td>
</tr>
</tbody>
</table>

**Step 6.** Make build by pressing F8 and selecting MAKE. This generates the bin file. Program the binary to GS node.

**Step 7.** Follow the steps mentioned in “GS2K TLS Low Power Server Setup User Guide.pdf” and start the UDP Server by executing the commands on the command prompt which is connected to the same Access Point as the GS node. Here 5000 is the UDP Port of the Server which is the same as given in the macro configuration file (app_default_cfg.h)

```
D:}\>node tls-lp_udpServer.js 5000
```

UDP Server listening on 0.0.0.0:5000

**Step 8.** Start the GS Node. Verify from Tera Term that it powers up, associates with the AP configured, and performs the following steps:

1. GS node performs a cold boot
2. Loads Applet2
3. Connects to configure SSID and the channel information provided by NCM.
4. Once connected, it resolves ARP, sends the sensor data, and waits for MAC level ack.
5. Waits for UDP receive.
6. Server sends 512 bytes as response all the time.
7. Receives the response and prints the length only.
8. Saves L2 and L3 information in RTC memory
9. Goes to standby for 5 seconds.

****************************************************

```
[TLS LP] Boot Reason = 1
[TLS LP] Boot Reason = 0
[TLS LP] Boot Info:
    Boot Reason : 0
    Boot Info : 0
    Reset Type: 0
    Reset Reason : 0
    Reset Info Size: 0
    Reset Addl Info: 0x0
    ResetInfo:
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
```
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000

[TLS LP] Loading Applet, ID = 2
[TLS LP] Data Send Applet
[TLS LP] TLS_LP_GS2K-5.3.0.26 [Nov 10 2016, 15:03:44]
[TLS LP] GEPS VERSION 5.3.0.27
[TLS LP] WLAN VERSION 5.3.0.52
[TLS LP] Ip Address = 0
[TLS LP] Device Configuration Info:
  MAC Addr     : 0:1d:c9:17:88:92
  SSID         : ASUS
  Channel      : 11
  DHCP     : TRUE
  Device Name  : GS_178892

[TLS LP] Starting NCM (STA Mode):
[TLS LP] Nw Connect Done
[TLS LP] APP_EVENT_DATA_SEND Event: Connected. Send DataLux: RawVal = 2618, ConvVal = 620.129272
temp: RawVal = 1932, ConvVal = 27.746796
[TLS LP] Resolving ARP entry for IP = c0a801c3
[TLS LP] ARP entry Resolve Cb status= 1

[TLS LP] MAC ACK Success
[TLS LP] UDP Data Receiving
[TLS LP] UDP Data Received: Num Bytes = 512
[TLS_LP] Ip Address ý
GS node performs the following steps when the five seconds expire:

1. GS node wakes up with reason 1 which is GSN_WIF_SYS_BOOT_NORMAL_BOOT
2. Loads Applet1
3. Initializes ADC and Read Sensors.
4. Stores Sensor data in RTC
5. Goes to 5 second standby.

[TLS LP] Boot Reason = 1
[TLS LP] Loading Applet, ID = 1
[TLS LP] Sensor Read Applet
Lux: RawVal = 2496, ConvVal = 669.286743
temp: RawVal = 1937, ConvVal = 27.63320

GS node performs the following steps when the next five seconds expire:

1. Loads Applet2
2. Restores L2, L3 connections from configuration file and applies it.
3. Reads sensor data, sends it to the configured UDP server, and waits for MAC level ack.
5. Server sends 512 bytes as response all the time.
6. Receives the response and prints the length only.
7. Goes to standby for 5 seconds.

[TLS LP] Boot Reason = 1
[TLS LP] Loading Applet, ID = 2
[TLS LP] Data Send Applet
[TLS LP] Date and time Thu, 10 Nov 2016 12:41:57 GMT
[TLS LP] Ip Address = c0a8010e
[TLS LP] dhcpRqstCurrIpAddr = 1, Ip Address = c0a8010e
[TLS LP] Device Configuration Info:
  MAC Addr : 0:1d:c9:17:88:92
  SSID : ASUS
  Channel : 11
  DHCP : TRUE
  Device Name : GS_178892
[TLS LP] Starting NCM (STA Mode):
[TLS LP] Nw Connect Done
[TLS LP] APP_EVENT_DATA_SEND Event: Connected. Send DataLux: RawVal = 2474, ConvVal = 678.151184
temp: RawVal = 1936, ConvVal = 27.655914
[TLS LP] MAC ACK Success
[TLS LP] UDP Data Receiving
[TLS LP] UDP Data Received: Num Bytes = 512
[TLS LP] Ip Address ý
Step 9. Start the GS Node. Verify from command prompt and the sensor data from the UDP client is as shown below:

![UDP Server receiving sensor data from client every 10 second.](image)
5.2.4 Using HTTPS Server

This use case demonstrates sending of sensor data on every 'APP_DATA_SEND_ITERATION' to a configured HTTPS server. GS node wakes up every 'APP_DATA_READ_INTERVAL' increments the RTC wakeup count variable numDataTfrItern. If numDataTfrItern is equal to APP_DATA_SEND_ITERATION, then Applet2 will be loaded. This initializes the full system as well as releasing the reset of WLAN. It then associates with provisioned AP gets IP address, does a link check which is nothing but a ARP request to Access Point to check whether the connection is alive. If successful, reads the sensor sends it over WiFi to the preconfigured HTTPS Server. SNTP is done before sending data. All the HTTPS and TCP configuration information is saved in RTC RAM in between standby so that on next wakeup, GS node does not have to perform L2, L3, L4 or reconnect to the HTTPS Server thus saving power.

If numDataTfrItern is not equal to APP_DATA_SEND_ITERATION, then Applet1 will be loaded which initializes and reads sensor data, stores in RTC RAM, and goes to standby.

5.2.4.1 Macros

Following are the macros required to be enabled or modified for sending sensor data using HTTPS (app_default_cfg.h and preIncludeFile.txt):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_MODE</td>
<td>This needs to be set to GSN_INFRA_CLIENT</td>
</tr>
<tr>
<td>DEFAULT_CLI_SSID</td>
<td>The SSID of the Access Point to which GS node will be connected.</td>
</tr>
<tr>
<td>APP_DATA_READ_INTERVAL</td>
<td>This is the duration for which GS node will be in standby between bootup. By default, it is 5 seconds.</td>
</tr>
<tr>
<td>APP_DATA_SEND_ITERATION</td>
<td>Based on the iteration number, sensor data will be sent to configured number. If iteration number is not matched, GS node will read only sensor data and save in RTC ram. The maximum entries that can be saved in RTC is 32</td>
</tr>
<tr>
<td>DEFAULT_CLI_CHNL</td>
<td>This specifies the channel on which SSID is operating.</td>
</tr>
<tr>
<td>DEFAULT_CLI_SECURITY</td>
<td>Type of security used on the Access Point.</td>
</tr>
<tr>
<td>DEFAULT_CLI_PASSPHRASE</td>
<td>Passphrase of the Access Point.</td>
</tr>
<tr>
<td>APP_SEND_DATA_OVER_HTTP</td>
<td>To enable sending sensor data via HTTPS, this macro need to enable this in file preIncludeFile.txt</td>
</tr>
<tr>
<td>HTTP_SERVER_ADDRESS</td>
<td>IP Address of the HTTPS Server receiving the sensor data.</td>
</tr>
<tr>
<td>HTTP_SERVER_PORT</td>
<td>Port number of the HTTPS Server.</td>
</tr>
<tr>
<td>APP_SNTP_SVR_IP</td>
<td>IP Address of the global SNTP Server.</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ADK_SNTP_ENABLED</td>
<td>To enable doing time sync with NTP server, this macro need to enable this in file preIncludeFile.txt</td>
</tr>
<tr>
<td>SNTP_SERVER_DOMAIN_NAME</td>
<td>Domain name of NTP Server //either IP address of DNS is needed</td>
</tr>
<tr>
<td>APP_SNTP_TIMEOUT</td>
<td>How long will GS node wait after sending the SNTP request? It is 1 second by default.</td>
</tr>
<tr>
<td>HTTP_SERVER_URL</td>
<td>Script to be run on the Server side for the data received.</td>
</tr>
<tr>
<td>APP_WIFI_LINK_KEEPALIVE</td>
<td>Enable this macro along with APP_LINK_CHECK_USING_ARP to send keep alive messages dynamically so that the connection with AP is maintained across standby.</td>
</tr>
<tr>
<td>APP_LINK_CHECK_USING_ARP</td>
<td>If this macro is enabled, ARP is done to gateway after every N wakeup cycles to check if the L2 connection is intact.</td>
</tr>
</tbody>
</table>

On SDK Builder, Customer needs to populate “Frequency of data upload” and “No of times product shall read sensor per data upload” which will be equivalent to MACRO’s APP_DATA_READ_INTERVAL and APP_DATA_SEND_ITERATION.

APP_DATA_READ_INTERVAL = Frequency of data upload / No of times product shall read sensor per data upload.

APP_DATA_SEND_ITERATION = No of times product shall read sensor per data upload.

5.2.4.2 Code Flow

The code flow for the use case is as follows:

On boot-up, the TLS LP Application binary also called as the base binary is loaded using the same mechanism as any other application:

Boot ROM -> app startup code -> TLS LP code load, remap, call app_main()

'app_main' is executed in boot ROM context and it performs the following:

1. Re-initializes the Flash access and do the boot process.
2. Initializes power management module and sets the deep sleep state to active so that System does not go into deep sleep in idle loop.
3. Calls App_BasicInitEntry to create a new thread “Basic Init Thread” which performs basic initialization
4. Starts an idle loop which is the lowest priority task. This task will put the system in sleep/Deep sleep whenever possible (When no other thread is running, i.e. no activity is being done in the system)

******************************************************************

INT32 App_Main(VOID)
{  
GsnFlashf_FlashAccessReinit();  
App_BootProcess();  
GsnPwrMgmt_Init(&appCtx.pwrMgmtCtx);  
GsnPwrMgmt_DpSlpActivitycbSet(App_PwrMgmtDpSlpActivityCb);  
GsnPwrMgmt_DpSlpActivitySet(&appCtx.pwrMgmtCtx, (GSN_DP_ACTIVITY_BIT_MAP_T)APP_DPSLP_ACTIVITY_BASIC_INIT_INPROGRESS);  
App_BasicInitEntry();  
while(1)  
{    
   __WFI();  
}  
}  

App_BasicInitEntry

This thread is created from app_main by calling App_BasicInitEntry. This newly created thread has a priority of 18. This has an entry function App_BasicInitThreadEntry which calls App_BasicInit that performs the following:

- Initializes clock, gets boot reason
- Initializes basic hardware, soft timer, GPIO, alarm, and UART
- Initializes persistent timer, restores persistent timer in case of normal boot. If it is cold boot, it starts the user configured (default is 5 seconds) persistent timer.
- Initializes file system and loads default configuration from flash file system if it is first time boot.
- If system boot-up is because of Alarm2, changes the network mode to Limited-AP, and resets the system.
- If wakeup is from standby, restores the data from RTC RAM.
- Powers up the WLAN Core.
- Initializes the lower system modules such as exception handler, HI layer, sys ctl, and configures the clock source.
- Calls API App_TriggerAppletLoad which posts a message to the Applet loader thread queue (appletLoaderMsgQueue) to load either Applet1 or Applet 2 based on RTC variable numDataTfrltern.

VOID App_BasicInit(VOID)
{
   ........................................
   GsnClkCtl_Init(&pAppCtx>clkCtlCtx);
   bootReason = GSN_BOOT_REASON_GET();
   if(bootReason != GSN_WIF_SYS_BOOT_NORMAL_BOOT)  
   {    
      GsnRtc_AlarmConfig(GSN_RTC_ALARM_2, 0, FALSE);
      GsnRtc_AlarmConfig(GSN_RTC_ALARM_3, 0, FALSE);
      App_BasicHwInit();  
   }  
}
/* Initialize Soft Timer */
App_SoftTimerInit(pAppCtx);

App_GpioInit();
/* Initialize alarm */
App_AlarmsInit(pAppCtx);

AppDbg_UartInit();
App_PersistentTmrInit(pAppCtx);

if(bootReason != GSN_WIF_SYS_BOOT_NORMAL_BOOT)
{
    App_SensorReadTmrStart();
    /* If use case 1 and 3 both are enabled start sensor read timer */
}
else
{
    /* In case of use case 3, restore persistent timer after giving
       boot request to WLAN. Else restore it now. */
    GsnPersistTmr_Restore();
}

if(bootReason != GSN_WIF_SYS_BOOT_NORMAL_BOOT)
{
    App_RtcDump();
}

/* This mutex in used AppFs_Init function. It is also required when WLAN writes callibration data */
GsnOsal_MutexCreate(&fsInitLock, GSN_OSAL_PRI_NO_INHERIT);

......................................

if (1 == pAppRtcLtchMemInfo->alarm2Press)
{
    pAppRtcLtchMemInfo->alarm2Press = 0;
    GsnSys_Reset(GSN_RESET_REASON_APP_SW_RESET);
}
else
{
    App_ReStoreStdbyData(pAppCtx);
}

if(App_IsWlanBootReq() == FALSE)
{
    goto APP_TRIGGER_APPLET_LOAD;
}

GsnExcpHdlr_Init();
App_HilInit(pAppCtx);

/* Initialise wif System Control message Handler */
App_SysCtlInit(pAppCtx);

App_IfBasicInit(pAppCtx, &pAppCtx->appNwIfc[APP_DFLT_CFG_INDEX]);

......................

APP_TRIGGER_APPLET_LOAD:
App_AppletLoaderInit();
App_Cm3ClkCfg(&pAppCtx->cm3ClkCfg);
App_TriggerAppletLoad();

}
App_TriggerAppletLoad(VOID)
{
    ........
    else if(0 == pAppRtcLtchMemInfo->numDataTfrtern ||
            APP_DATA_SEND_ITERATION <= pAppRtcLtchMemInfo->numDataTfrtern ||
            bootInfo == GSN_WIF_SYS_BOOTINFO_NORMALBOOT_ALARM)
    {
        /* in case of alarm wake up don't disturb the cycle*/
        if(bootInfo != GSN_WIF_SYS_BOOTINFO_NORMALBOOT_ALARM)
        {
            if(APP_DATA_SEND_ITERATION <= pAppRtcLtchMemInfo->numDataTfrtern)
            {
                /* In case of data send iteration reset the iteration count to 1 */
                pAppRtcLtchMemInfo->numDataTfrtern = 1;
            }
            else
            {
                pAppRtcLtchMemInfo->numDataTfrtern++;
            }
        }
        /* When applet 2 is loaded for first time, applet2Load_ColdBoot value is set to 5 */
        if(5 != pAppRtcLtchMemInfo->applet2Load_ColdBootFlag)
        {
            /* Value 1: Tells applet2 is yet to load for the first time */
            pAppRtcLtchMemInfo->applet2Load_ColdBootFlag = 1;
        }
        /*Load Remaining Part of the code*/
        flags = (APPLETLOAD_FLAGS_T)(APPLETLOAD_FLAG_LOAD |
                                   APPLETLOAD_FLAG_CALL_ENTRY_FUNC);
        App_AppletLoad(2, flags);
    }
    else
    {
        #ifndef APP_NO_SENSOR
        /*Applet 1 only does reading and storing sensor data. This appelt should be loaded only if
          TLS sensor is present on the board */
        flags = (APPLETLOAD_FLAGS_T)(APPLETLOAD_FLAG_LOAD |
                                   APPLETLOAD_FLAG_CALL_ENTRY_FUNC);
        App_AppletLoad(1, flags);
        #else
        flags = (APPLETLOAD_FLAGS_T)(APPLETLOAD_FLAG_LOAD |
                                   APPLETLOAD_FLAG_CALL_ENTRY_FUNC);
        App_AppletLoad(2, flags);
        #endif
        pAppRtcLtchMemInfo->numDataTfrtern++;
    }
}*******************************************************

On first boot, since RTC variable pAppRtcSecMemInfo-> numDataTfrtern is 0, Applet2 will be
loaded.

On every wakeup from standby, whenever RTC member variable numDataTfrtern is 0, TLS LP
or base firmware dynamically loads Applet_2 onto RAM location 0x20050000 as mentioned in
the Applet header descriptor field appletSramAddr using the function App_TriggerAppletLoad.
Once code is loaded, the entry function in Applet 2 code App_Applet2Entry is called as
mentioned in SingleImageBuilderConfig.txt present in build folder.
The applet_2 entry function App_Applet2Entry performs the following:

Creates Application thread “Applet2 Thread” with priority 17. This thread has entry function App_Applet2ThreadEntry which performs the following:

1. Starts a 100ms timer for the sensors to stabilize, only then data can be read. This is performed in the Applet2_main.c file.

```c
App_CreateSensorStablizeLock();
GSN_PWR_MGMT_CM3_SLEEP_DEEP_EN();
sensorStablizeTmrHndl = GsnSoftTmr_Start(&sensorStablizeTmr,
GSN_SOFT_TMR_ONESHOT, 0,
GSN_SOFT_TMR_MILLISECONDS_TO_TICKS(90),
App_SensorStabilizeTmrCb,
NULL);
(void)(sensorStablizeTmrHndl); /*Compilation Warning suppression*/
App_WaitOnSensorStablizeLock();
```

2. Sends boot request for WLAN using the API App_WlanBootReq.
3. Initializes the supplicant and random number generator.
4. Initializes wdd msg handler, network stack, interface, BsdLayer, state machine (sm), Network connection manager (NCM), and posts an event (APP_EVENT_NW_START_NCM) to the main task to start NCM using API App_applet2TaskNotify (APP_EVENT_NW_START_NCM, 0).
5. Applet2 thread infinitely waits for messages on the queue applet2ThreadMsgQueue.

```c
while( TRUE )
{
    /**< wait on message queue */
    GsnOsal_QueueGet( &applet2ThreadMsgQueue, ( UINT8* )&msg,
        GSN_OSAL_WAIT_FOREVER );
    /**< Process the message */
    GsnSq_TaskMonitorStart(APP_CFG_SQ_MAIN_TASK_ID, msg.msgId);
    AppSm_MsgProcess(pAppCtx, msg);
    GsnSq_TaskMonitorStop(APP_CFG_SQ_MAIN_TASK_ID);
}
```

6. NCM takes the configuration information from the app_default_config.h and connects to the SSID. Once GS Node is successfully connected and IP address has been allocated which is received from the AP, it sends out APP_EVENT_NW_CONNECT_DONE to the sm.
7. Applet2 main task (app_sm.c) processes this event and checks whether the sensor data needs to be sent. If sensor need not be sent, Applet2 main thread breaks from the case statement. If sensor data needs to be sent, then Applet2 proceeds with the next event APP_EVENT_DATA_SEND which is triggered periodically (every 5 second) from the TLS LP code using the API App_applet2TaskNotify (APP_EVENT_DATA_SEND, 0).
8. As part of APP_EVENT_DATA_SEND event handling, Applet2’s sm thread checks whether number of data transfers has equaled DEFAULT_LINK_CHECK_ITERATION. If so, then to check whether GS Node is associated with the Access Point, Applet2 main task calls AppSm_LinkCheck api where it invalidates and deletes the ARP entry and
triggers a new ARP request to the AP. Once the AP responds, it saves it in its ARP cache. The return value informs whether ARP was successful or not. If ARP fails, Applet2 main task triggers a L2 sync loss which restarts the NCM process. This step is done as recovery mechanism for the Application in case the AP on the other side has been switched off.

9. Calls AppSm_DataSend to read sensors.

10. Calls AppTls_SensorRead to initialize the ADC component and read the temperature and light sensor values into the RTC array.

11. Increments UdpSeqNo and stores it in RTC.

12. If http flag is enabled, calls AppHttp_DataSend to send data over https.
   a. Prepares the data to be sent.
   b. Loads certificate
   c. Gets the HTTP Server address from app_default_cfg.h
   d. Opens the https connection on the given port.
   e. Calls setsock option to set the Tx retry rate to 2 secs, number of transmissions to 3, and sets the priority to voice.
   f. Populates the http headers.
   g. Sets the timeout for receiving response.
   h. Posts data to the server.

13. Calls api App_GoToLowPowerState to save L2, L3, L4 configuration details in RTC RAM and then goes into standby using App_GotoStandby.

NOTE – Code flow which is under Keep Alive MACRO “APP_WIFI_LINK_KEEPALIVE” cannot be seen.

5.2.4.3 Demo

Perform the following steps for the demo:

Step 10. Below are the macro’s that will be needed to be enabled or modified for sending sensor data using HTTPS (app_default_cfg.h and preIncludeFile.txt):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_MODE</td>
<td>GSN_INFRA_CLIENT</td>
</tr>
<tr>
<td>DEFAULT_CLI_SSID</td>
<td>&quot;ASUS&quot;</td>
</tr>
<tr>
<td>APP_DATA_READ_INTERVAL</td>
<td>5000 // standby is for 5 second</td>
</tr>
<tr>
<td>APP_DATA_SEND_ITERATION</td>
<td>2 // on every even iteration it will send data</td>
</tr>
<tr>
<td>DEFAULT_CLI_CHNL</td>
<td>11</td>
</tr>
<tr>
<td>DEFAULT_CLI_SECURITY</td>
<td>GSN_WLAN_WPA_PERSONAL</td>
</tr>
</tbody>
</table>
Macro | Description
--- | ---
DEFAULT_CLI_PASSPHRASE | “xxxxxxxx” //xxxxxxxx will be replaced with AP pwd
APP_SEND_DATA_OVER_HTTP | To enable sending sensor data via HTTPS, this macro needs to be enabled in file preIncludeFile.txt (remove ___)
HTTP_SERVER_ADDRESS | 192.168.2.223
HTTP_SERVER_PORT | 443
ADK_SNTP_ENABLED | To enable time, sync with NTP server, this macro needs to be enabled in file preIncludeFile.txt (remove ___)
APP_SNTP_SVR_IP | -
SNTP_SEVER_DOMAIN_NAME | pool.ntp.org //either IP address or DNS is needed
APP_SNTP_TIMEOUT | 3000
HTTP_SERVER_URL | "/cgi-bin/get1.pl?10"
APP_WIFI_LINK_KEEPALIVE | Enable this in file preIncludeFile.txt.
APP_LINK_CHECK_USING_ARP | Enable this macro.

**Step 11.** Make build by pressing F8 and selecting MAKE. This generates the bin file. Program the binary to GS node.

**Step 12.** Follow the steps mentioned in GS2K TLS Low Power Server Setup User Guide.pdf and start the HTTPS Server (section 2.1 HTTPS Server) using the commands mentioned. Apache server can be started with the Laptop which is connected to the same Access Point as that of GS node. Here 443 is the HTTPS Port of the Server which is same as given in the macro configuration file (app_default_cfg.h).

**Step 13.** Start the Node, verify from Tera Term that GS node powers up, associates with the AP configured, and performs the following:
1. GS node performs a cold boot
2. Loads Applet2
3. Connects to configure sync with the NTP Server done.
4. Performs time sync with the NTP server.
5. Reads sensor data.
7. Sends Post request to the Server.
8. Server will send 200 OK and 512 bytes of data as response all the time.
9. Receives the response and prints the data only.
10. Saves L2, L3, L4 info in RTC memory
11. Goes to standby for 5 seconds.
[TLS LP] Boot Reason = 1
[TLS LP] Boot Reason = 0
[TLS LP] Boot Info:
  Boot Reason : 0
  Boot Info : 0
  Reset Type: 0
  Reset Reason : 0
  Reset Info Size: 0
  Reset Addl Info: 0x0
  ResetInfo:
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000
  0x00000000

[TLS LP] Loading Applet, ID = 2
[TLS LP] Data Send Applet
[TLS LP] TLS_LP_GS2K-5.3.0.26 [Nov 10 2016, 18:09:14]
[TLS LP] GEPS VERSION 5.3.0.27
[TLS LP] WLAN VERSION 5.3.0.52
[TLS LP] Ip Address = 0
[TLS LP] Device Configuration Info:
  MAC Addr     : 0:1d:c9:17:88:92
  SSID         : ASUS
  Channel      : 11
  DHCP         : TRUE
  Device Name  : GS_178892
[TLS LP] Starting NCM (STA Mode):
[TLS LP] Nw Connect Done
[TLS LP] IP address Changed so Closing HTTP
  ****SNTP time sync*****[TLS_LP]AppSm_TimeSyncDoneCb
[TLS LP] APP_EVENT_DATA_SEND Event: Connected. Send DataLux: RawVal = 3870, ConvVal = 115.660896
temp: RawVal = 2001, ConvVal = 26.189178
[TLS LP] Sending Data Over HTTP
[TLS LP] GsnHttp_Start SUCCESSFUL
[TLS LP] HTTP connection established successfully
[TLS LP] Http Open Success
[TLS LP] datalen 56 httpTimeout = 1
[TLS LP] Sending POST request to Server
[TLS LP] HTTP POST response size = 520 & return Val = 0

32 30 30 20 4f 4b 0d 0a 53 57 41 4e 44 20 44 41
54 41 20 55 50 54 4c 4f 4e 44 20 44 41
53 3b 4e 4f 20 43 4f 4e 46 49 47 55 52 41 54 49
4f 4e 20 50 45 4e 44 49 4e 47 20 44 45
56 49 43 45 3b 50 4c 45 41 45 20 44 45 20 47 4f 20 42
41 43 4b 20 50 44 4f 4e 46 49 47 55 52 41 54 49
20 48 49 50 20 48 4f 4f 4f 52 41 59 2e 53 57 41 4e 44 49 47 55
4e 20 44 41 54 41 20 55 50 54 50 4c 4f 4e 47 20 44 45
46 49 47 55 52 41 45 49 4f 4e 20 50 45 4e 44 49 4e 47 20 45
4f 20 44 45 56 49 43 45 3b 50 4c 45 45 20 45 41 54 49 20
47 4f 20 42 41 43 4b 20 54 4f 20 53 4c 45 45 20 45 41 54 49 20
2e 48 49 50 20 48 4f 4f 4f 52 41 59 2e 53 57 41 4e 44 49 47 55
53 57 41 4e 4e 20 44 41 54 41 20 55 50 54 50 4c 4f 4e 47 20 44 45
44 20 53 55 43 45 43 45 53 53 3b 4e 4f 20 43 4f 4e 47 20 45
46 49 47 55 52 41 45 49 4f 4e 20 50 45 4e 44 49 4e 47 20 45
4e 47 20 54 4f 20 44 45 56 49 43 45 3b 50 4c 45 45 20 45 41 54 49 20
41 53 45 20 47 4f 20 42 41 43 4b 20 54 4f 20 53 4c 45 45 20 45 41 54 49 20
4c 45 45 50 2e 48 49 50 20 48 4f 4f 4f 52 41 59 2e 53 57 41 4e 44 49 47 55
52 41 59 2e 53 57 41 4e 4e 20 44 41 54 41 20 55 50 54 50 4c 4f 4e 47 20 44 45
50 4c 4f 4e 41 44 20 53 55 43 43 45 53 53 3b 4e 4f 47 20 44 45
20 43 4f 4e 46 49 47 55 52 41 54 49 4f 4e 20 50 45
45 4e 44 49 4e 47 20 54 4f 20 44 45 56 49 43 45 3b 50 4c 45 45 20 45 41 54 49 20
3b 50 4c 45 41 53 45 20 47 4f 20 42 41 43 4b 20 54 4f 20 53 4c 45 45 20 45 41 54 49 20
54 4f 20 53 4c 45 45 50 2e 48 49 50 20 48 4f 4f 4f 52 41 59 2e 53 57 41 4e 44 49 47 55
20 48 4f 4f 52 41 59 2e 53 57 41 4e 4e 20 44 41 54 41 20 55 50 54 50 4c 4f 4e 47 20 44 45
54 41 20 55 50 54 4c 4f 4e 44 49 47 55 52 41 54 49 4f 4e 20 50 45
53 3b 4e 4f 20 43 4f 4e 46 49 47 55 52 41 54 49 4f 4e 20 50 45
56 49 43 45 3b 50 4c 45 41 53 45 20 47 4f 20 42 41 43 4b 20 54 4f 20 53 4c 45 45 20 45 41 54 49 20
41 43 4b 20 54 4f 20 53 4c 45 45 50 2e 48 49 50 20 48 4f 4f 4f 52 41 59 2e 53 57 41 4e 44 49 47 55
20 48 49 50 20 48 4f 4f 52 41 59 2e 3a 29 3a 29 3a 29
3a 29 3a 29 3a 29 3a 29
[TLS LP] HTTP POST SUCCESS

GS node performs the following steps when the five seconds expire:

1. GS node wakes up with reason 1 which is GSN_WIF_SYS_BOOT_NORMAL_BOOT
2. Loads Applet1
3. Initializes ADC and Read Sensors.
4. Stores Sensor data in RTC
5. Goes to 5 second standby.

[TLS LP] Boot Reason = 1
[TLS LP] Loading Applet, ID = 1
[TLS LP] Sensor Read Applet
Lux: RawVal = 3931, ConvVal = 79.298599
temp: RawVal = 2010, ConvVal = 25.98745
GS node performs the following steps when the next five seconds (10 seconds from boot) expire:

1. Loads Applet2.
2. Restores L2, L3, and L4 connection from configuration file and applies it.
3. Reads sensor data.
4. Resumes HTTPS connection.
5. Sends Post request to the Server.
6. Server sends 200 OK and 512 bytes of data as response all the time.
7. Receives a response and prints the data only.
8. Saves L2, L3, L4 information in RTC memory.
9. Goes to standby for 5 seconds.
[TLS LP] HTTP POST SUCCESS
USE CASE 2 EVENT NOTIFICATION TO CLOUD (HIBERNATE MODE)

This use case demonstrates sending sensor data when asynchronous event (Alarm1 on the EVB) is triggered. When an asynchronous event is triggered GS node which is in Hibernation mode wakes up, initializes the full system, gets L2, L3, and L4 configuration information from configuration file stored in the flash and RTC RAM, and associates with the provisioned AP. It also initializes and reads sensor, sends it over UDP/HTTPS to the configured Server and goes into Hibernate State only to be woken up by Alarm1. It stores the scan entries so that next time when it is woken, it need not scan for association and performs a lean DHCP wherein GS node starts with requesting the stored IP address from the previous session.

5.2.5 Using UDP Server

This use case demonstrates sending sensor data when asynchronous event (Alarm1 on the EVB) is triggered. Because of the trigger, GS node which is in Hibernation mode wakes up, initializes the full system, gets L2, L3, L4 configuration information from configuration file stored in the flash and RTC RAM, and associates with the provisioned AP. It also initializes and reads sensor, sends it over UDP to the configured Server and goes into Hibernate State only to be woken up by Alarm1.

5.2.5.1 Macros

Following are the macros required to be enabled or modified for sending sensor data using UDP (app_default_cfg.h and preIncludeFile.txt):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_MODE</td>
<td>This needs to be set to GSN_INFRA_CLIENT</td>
</tr>
<tr>
<td>DEFAULT_CLI_SSID</td>
<td>The SSID of the Access Point to which GS node will be connected.</td>
</tr>
<tr>
<td>APP_DATA_READ_INTERVAL</td>
<td>This is the duration for which GS node will be in standby between boot-up. By default, it is 5 seconds.</td>
</tr>
<tr>
<td>APP_DATA_SEND_ITERATION</td>
<td>Based on the iteration number, sensor data will be sent to configured number. If iteration number is not matched, GS node will read only sensor data and save in RTC ram. The maximum entries that can be saved in RTC is 32</td>
</tr>
<tr>
<td>DEFAULT_CLI_CHNL</td>
<td>This specifies the channel on which SSID is operating.</td>
</tr>
<tr>
<td>DEFAULT_CLI_SECURITY</td>
<td>Type of security used on the Access Point.</td>
</tr>
<tr>
<td>DEFAULT_CLI_PASSPHRASE</td>
<td>Passphrase of the Access Point.</td>
</tr>
<tr>
<td>UDP_SERVER_ADDRESS</td>
<td>IP Address of the Server receiving the sensor data.</td>
</tr>
<tr>
<td>UDP_SERVER_PORT</td>
<td>Port number of the UDP Server.</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>APP_UDP_DATA_RECEIVE_ENABLE</td>
<td>If response from UDP Server is expected, this macro needs to be enabled</td>
</tr>
<tr>
<td>UDP_RECEIVE_TIMEOUT</td>
<td>Wait time for response from UDP Server. Default is 1 second</td>
</tr>
<tr>
<td>APP_MAC_LEVEL_CONFIRM_REQUIRED</td>
<td>This is applicable only for data sent over UDP. Mac level confirmation for the UDP packet sent is checked after the transmission</td>
</tr>
<tr>
<td>APP_HIBERNATE_ENABLE</td>
<td>To go into Hibernate after every data sent, this macro needs to be enabled in file preIncludeFile.txt</td>
</tr>
</tbody>
</table>

### 5.2.5.2 Code Flow

The code flow for the use case is as below:

On boot-up, the TLS LP Application binary also called as the base binary is loaded using the same mechanism as any other application:

Boot ROM -> app startup code -> TLS LP code load, remap, call app_main()

`app_main` is executed in boot ROM context and it performs the following:

1. Re-initializes the Flash access and performs the boot process.
2. Initializes power management module and sets the deep sleep state to active so that System does not go into deep sleep in idle loop.
3. Calls App_BasicInitEntry to create a new thread “Basic Init Thread” which performs basic initialization
4. Starts an idle loop which is the lowest priority task. This task will put the system in sleep/Deep sleep whenever possible (When no other thread is running, i.e. no activity is being done in the system)

```c
INT32 App_Main(VOID)
{
    GsnFlashif_FlashAccessReinit();
    App_BootProcess();
    GsnPwrMgmt_Init(&appCtx.pwrMgmtCtx);
    GsnPwrMgmt_DpSlpActivitycbSet(App_PwrMgmtDpSlpActivityCb);
    GsnPwrMgmt_DpSlpActivitySet(&appCtx.pwrMgmtCtx,
    (GSN_DP_ACTIVITY_BIT_MAP_T)APP_DPSLP_ACTIVITY_BASIC_INIT_INPROGRESS);
    App_BasicInitEntry();
    while(1)
    {
        __WFI();
    }
}
```

******************************************************************
INT32 App_Main(VOID)
{}}
**App_BasicInitEntry**

This thread is created from `app_main` by calling `App_BasicInitEntry`. This newly created thread has a priority of 18. This thread contains an entry function `App_BasicInitThreadEntry` which calls `App_BasicInit`.

`App_BasicInit` performs the following:

- Initializes clock and gets boot reason
- Initializes basic hardware, soft timer, GPIO, alarm, and UART
- Initializes file system and loads default configuration from flash file system if it is first time boot.
- If system boot-up is because of Alarm2, changes the network mode to Limited-AP, and resets the system.
- If wakeup is from alarm1, restores hibernate data.
- Powers up the WLAN Core.
- Calls API `App_TriggerAppletLoad` which posts a message to the Applet loader thread queue (`appletLoaderMsgQueue`) which always load Applet2.

```c
VOID App_BasicInit(VOID)
{
    .........................................
    GsnClkCtl_Init(&pAppCtx->clkCtlCtx);
    bootReason = GSN_BOOT_REASON_GET();
    if(bootReason != GSN_WIF_SYS_BOOT_NORMAL_BOOT) {
        GsnRtc_AlarmConfig(GSN_RTC_ALARM_2, 0, FALSE);
        GsnRtc_AlarmConfig(GSN_RTC_ALARM_3, 0, FALSE);
        App_BasicHwInit();
    }
    /*Initialise Soft Timer */
    App_SoftTimerInit(pAppCtx);
    App_GpioInit();
    /* Initialize alarm */
    App_AlarmsInit(pAppCtx);
    AppDbg_UartInit();
    App_RestoreHibernateData(pAppCtx);
    if((App_IsWlanBootReq() == FALSE)) {
        goto APP_TRIGGER_APPLET_LOAD;
    }
    GsnExcpHdlr_Init();
    App_HilInit(pAppCtx);
    /* Initialise wif System Control message Handler */
    App_SysCtlInit(pAppCtx);
    App İşBasicInit(pAppCtx, &pAppCtx->appNwIfc[APP_DFLT_CFG_INDEX]);
    .........................................
    APP_TRIGGER_APPLET_LOAD:
    App_AppletLoaderInit();
    App_Cm3ClkCfg(&pAppCtx->cm3ClkCfg);
    App_TriggerAppletLoad();
}
```

This thread contains an entry function `App_BasicInitThreadEntry` which calls `App_BasicInit`.
App_TriggerAppletLoad(VOID)
{
    ........
    else if(0 == pAppRtcLtchMemInfo->numDataTfrItem ||
            APP_DATA_SEND_ITERATION <= pAppRtcLtchMemInfo->numDataTfrItem ||
            bootInfo == GSN_WIF_SYS_BOOTINFO_NORMALBOOT_ALARM)
    {
        /* in case of alarm wake up don't disturb the cycle*/
        if(bootInfo != GSN_WIF_SYS_BOOTINFO_NORMALBOOT_ALARM)
        {
            if(APP_DATA_SEND_ITERATION <= pAppRtcLtchMemInfo->numDataTfrItem)
            {
                /* In case of data send iteration reset the iteration count to 1 */
                pAppRtcLtchMemInfo->numDataTfrItem = 1;
            }
            else
            {
                pAppRtcLtchMemInfo->numDataTfrItem++;
            }
        }
    }

    /* When applet 2 is loaded for first time, applet2Load_ColdBoot value is set to 5 */
    if(5 != pAppRtcLtchMemInfo->applet2Load_ColdBootFlag)
    {
        /* Value 1: Tells applet 2 is yet to load for the first time */
        pAppRtcLtchMemInfo->applet2Load_ColdBootFlag = 1;
    }

    /*Load Remaining Part of the code*/
    flags = (APPLETLOAD_FLAGS_T)(APPLETLOAD_FLAG_LOAD | APPLETLOAD_FLAG_CALL_ENTRY_FUNC);
    App_AppletLoad(2, flags);
**********************************************************************

During first boot, since RTC variable pAppRtcSecMemInfo->pAppRtcLtchMemInfo is 0, Applet2 will be loaded.

On every wakeup from standby, whenever RTC member variable numDataTfrItem is 0, TLS LP or base firmware dynamically loads Applet_2 onto RAM location 0x20050000 as mentioned in the Applet header descriptor field appletSramAddr using the function App_TriggerAppletLoad. Once code is loaded, the entry function in Applet 2 code App_Applet2Entry is called as mentioned in SingleImageBuilderConfig.txt present in build folder.

*************************************************************************
appletId=2
appletBin=\Debug\Exe\applet_2.bin
appletMap=\Debug\List\applet_2.map
appletEntryFunc=App_Applet2Entry
appletSramAddr=20050000
*************************************************************************

The applet_2 entry function App_Applet2Entry performs the following:

Creates Application thread "Applet2 Thread" with priority 17. This thread has entry function App_Applet2ThreadEntry which performs the following:
1. Starts a 100ms timer for the sensors to stabilize, only then data can be read. This is performed in the Applet2_main.c file.

```c
App_CreateSensorStablizeLock();
GSN_PWR_MGMT_CM3_SLEEP_DEEP_EN();
sensorStablizeTmrHndl = GsnSoftTmr_Start(&sensorStablizeTmr,
GSN_SOFT_TMR_ONESHOT, 0,
GSN_SOFT_TMR_MILLISECONDS_TO_TICKS(90),
App_SensorStabilizeTmrCb,
NULL);
(void)(sensorStablizeTmrHndl); /* Compilation Warning suppression*/
App_WaitOnSensorStablizeLock();
```

2. Sends boot request for WLAN using the API App_WlanBootReq.
3. Initializes the supplicant and random number generator.
4. Initializes wdd msg handler, network stack, interface, BsdLayer, state machine (sm), Network connection manager (NCM), and posts an event (APP_EVENT_NW_START_NCM) to the main task to start NCM using API App_applet2TaskNotify (APP_EVENT_NW_START_NCM, 0).
5. Applet2 thread infinitely waits on messages on the queue applet2ThreadMsgQueue.

```c
while( TRUE ) {
   /**< wait on message queue */
   GsnOsal_QueueGet( &applet2ThreadMsgQueue, ( UINT8* )&msg,
   GSN_OSAL_WAIT_FOREVER );
   /**< Process the message */
   GsnSq_TaskMonitorStart(APP_CFG_SQ_MAIN_TASK_ID, msg.msgId);
   AppSm_MsgProcess(pAppCtx, msg);
   GsnSq_TaskMonitorStop(APP_CFG_SQ_MAIN_TASK_ID);
}
```

6. NCM takes the configuration information from the app_default_config.h and connects to the SSID. Once GS Node is successfully connected and IP address has been allocated which is received from the AP, it sends out APP_EVENT_NW_CONNECT_DONE to the sm.
7. Applet2 main task processes this event and checks whether the sensor data needs to be sent. If sensor need not be sent, Applet2 main thread breaks from the case statement. If sensor data needs to be sent, then Applet2 proceeds with the next event APP_EVENT_DATA_SEND which is triggered periodically (every 5 second) from the TLS LP code using the API App_applet2TaskNotify (APP_EVENT_DATA_SEND, 0).
8. As part of APP_EVENT_DATA_SEND event handling, Applet2’s sm thread Calls AppSm_DataSend to read sensors.
9. Calls AppSm_DataSend to read sensors.
10. Calls AppTls_SensorRead to initialize the ADC component and read the temperature and light sensor values into the RTC array.
11. Increments UdpSeqNo and stores it in RTC.
12. Calls API App_DataSend to send data. Checks for ARP entry of the gateway and calls API App_DataSend. Based on the macro defined, send data via UDP using call AppUdp_DataSend (pAppCtx)
13. In function AppUdp_DataSend, if intended to receive MAC level confirmation for the UDP packet sent, enable the macro APP_MAC_LEVEL_CONFIRM_REQUIRED. Once the macro is enabled, the Applet2 thread registers for callback with the WLAN component so that WLAN sends status of the packet informing whether it was successfully acknowledged by the gateway or not using the API App_PacketTxCnfrmCbReg(App_UdpTxConfirmCb, pAppCtx).

14. If intended to receive data from UDP Server, enable the macro APP_UDP_DATA_RECEIVE_ENABLE, the Applet2 thread calls api AppUdp_DataRecv to wait for the configured time to receive response back from the UDP Server.

15. Calls API App_GoToLowPowerState to save L2, L3, and L4 configuration details in RTC latch and file system, and goes into hibernate using App_GotoHibernate.

5.2.5.3 Demo

Perform the following steps for the demo:

**Step 1.** The macros mentioned should be enabled or populated with proper values in app_default_config.h and corresponding flags needs to be enabled from preIncludeFile.txt.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_MODE</td>
<td>GSN_INFRA_CLIENT</td>
</tr>
<tr>
<td>DEFAULT_CLI_SSID</td>
<td>&quot;ASUS&quot;</td>
</tr>
<tr>
<td>APP_DATA_READ_INTERVAL</td>
<td>5000 // standby is for 5 second</td>
</tr>
<tr>
<td>APP_DATA_SEND_ITERATION</td>
<td>2 // on every even iteration it will send data</td>
</tr>
<tr>
<td>DEFAULT_CLI_CHNL</td>
<td>11</td>
</tr>
<tr>
<td>DEFAULT_CLI_SECURITY</td>
<td>GSN_WLAN_WPA_PERSONAL</td>
</tr>
<tr>
<td>DEFAULT_CLI_PASSPHRASE</td>
<td>“xxxxxxxx” //xxxxxxxx will be replaced with AP pwd</td>
</tr>
<tr>
<td>UDP_SERVER_ADDRESS</td>
<td>192.168.2.195</td>
</tr>
<tr>
<td>UDP_SERVER_PORT</td>
<td>5000</td>
</tr>
<tr>
<td>APP_UDP_DATA_RECEIVE_ENABLE</td>
<td>This is defined in preIncludeFile.txt, remove the ___ to enable.</td>
</tr>
<tr>
<td>UDP_RECEIVE_TIMEOUT</td>
<td>6000 // default is 1000</td>
</tr>
<tr>
<td>APP_MAC_LEVEL_CONFIRM_REQUIRED</td>
<td>This is enabled by default in file preIncludeFile.txt</td>
</tr>
<tr>
<td>APP_HIBERNATE_ENABLE</td>
<td>To go into Hibernate after every data sent, this macro needs to be enabled in file preIncludeFile.txt (remove __)</td>
</tr>
</tbody>
</table>
Step 2. Make build by pressing F8 and selecting MAKE. This generates the bin file. Program the binary to GS node.

Step 3. Follow the steps mentioned in GS2K TLS Low Power Server Setup User Guide.pdf and start the UDP Server by executing the commands in Laptop which is connected to the same Access Point as that of GS node. Here 5000 is the UDP Port of the Server which is same as given in the macro configuration file (app_default_cfg.h)

D:\node tls-lp_udpServer.js 5000

UDP Server listening on 0.0.0.0:5000

Step 4. Start the GS Node. Verify from Tera Term that it powers up, associates with the AP configured, and performs the following:

1. GS node performs a cold boot
2. Loads Applet2
3. Connects to configure SSID and the channel information provided using NCM.
4. Once connected, resolves ARP, sends the sensor data, and waits for MAC level ack.
5. Waits for UDP receive if flag is enabled.
6. Server sends 512 bytes as response all the time.
7. Receives response and prints the length only.
8. Saves scan entries as well as channel number of SSID, IP address to RTC latch.
9. Goes to hibernate only to be woken by Alarm1.

[TLS LP] Boot Reason = 0
[TLS LP] Boot Info:
    Boot Reason : 0
    Boot Info : 0
    Reset Type: 0
    Reset Reason : 0
    Reset Info Size: 0
    Reset Addl Info: 0x0
    ResetInfo:
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    0x00000000
    [TLS LP] Loading Applet, ID = 2
    [TLS LP] Data Send Applet
GS node performs the following when Alarm1 Button is pressed:

1. GS node wakes up with boot information as 1 which is GSN_WIF_SYS_BOOTINFO_NORMALBOOT_ALARM
2. Loads Applet2
3. Connects to SSID which is stored in RTC using NCM.
4. Performs a lean DHCP by using the saved IP Address.
5. Once connected, resolves ARP, sends the sensor data, and waits for MAC level ack.
6. Waits for UDP receive if flag is enabled.
7. Server sends 512 bytes as response all the time.
8. Receives response and prints the length only.
9. Saves scan entries as well as channel number of SSID, IP address to RTC latch.
10. Goes to hibernate only to be woken by Alarm1.
[TLS LP] Loading Applet, ID = 2
[TLS LP] Data Send Applet
[TLS LP] GEPS VERSION 5.3.0.27
[TLS LP] WLAN VERSION 5.3.0.52
[TLS LP] Ip Address = c0a8010e
[TLS LP] dhcpRqstCurrIpAddr = 1, Ip Address = c0a8010e
[TLS LP] Device Configuration Info:
  MAC Addr     : 0:1d:c9:17:88:92
  SSID         : ASUS
  Channel      : 11
  DHCP         : TRUE
  Device Name  : GS_178892

[TLS LP] Starting NCM (STA Mode):
[TLS LP] Nw Connect Done
  ****SNTP time sync*****[TLS LP]AppSm_TimeSyncDoneCb
[TLS LP] APP_EVENT_DATA_SEND Event: Connected. Send DataLux: RawVal = 0, ConvVal = 1675.000000
  temp: RawVal = 0, ConvVal = -273.000000
[TLS LP] Resolving ARP entry for IP = c0a801c3
[TLS LP] ARP entry Resolve Cb status= 1
[TLS LP] MAC ACK Success
[TLS LP] UDP Data Receiving
[TLS LP] UDP Data Received: Num Bytes = 512Scan ENtry 1[TLS LP]Ip Address = c0
5.2.6 Using HTTPS Server

This use case demonstrates sending sensor data when an asynchronous event (Alarm1 on the EVB) is triggered. Because of the trigger, GS node which is in Hibernation mode wakes up, initializes the full system, gets L2, L3, and L4 configuration information from configuration file and RTC RAM stored in the flash, and associates with the provisioned AP. It also initializes and reads sensor, sends it over HTTPS to the configured Server and goes into Hibernate State only to be woken up by Alarm1.

5.2.6.1 Macros

Following are the macros required to be enabled or modified for sending sensor data using HTTPS (app_default_cfg.h and preIncludeFile.txt):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_MODE</td>
<td>This macro needs to be set to GSN_INFRA_CLIENT.</td>
</tr>
<tr>
<td>DEFAULT_CLI_SSID</td>
<td>This macro specifies the SSID of the Access Point to which GS node will be connected.</td>
</tr>
<tr>
<td>APP_DATA_READ_INTERVAL</td>
<td>This macro specifies the duration for which GS node will be in standby between boot-up. By default, it is 5 seconds.</td>
</tr>
<tr>
<td>APP_DATA_SEND_ITERATION</td>
<td>This macro specifies the iteration number based on which the sensor data will be sent to configured number. If iteration number is not matched, GS node will only read sensor data and save in RTC RAM. The maximum entries that can be saved in RTC is 32.</td>
</tr>
<tr>
<td>DEFAULT_CLI_CHNL</td>
<td>This macro specifies the channel on which SSID is operating.</td>
</tr>
<tr>
<td>DEFAULT_CLI_SECURITY</td>
<td>This macro specifies the type of security used on the Access Point.</td>
</tr>
<tr>
<td>DEFAULT_CLI_PASSPHRASE</td>
<td>This macro specifies the passphrase of the Access Point.</td>
</tr>
<tr>
<td>APP_SEND_DATA_OVER_HTTP</td>
<td>To enable sending sensor data via HTTPS, this macro needs to be enabled in file preIncludeFile.txt.</td>
</tr>
<tr>
<td>HTTP_SERVER_ADDRESS</td>
<td>This macro specifies the IP Address of the HTTPS Server receiving the sensor data.</td>
</tr>
<tr>
<td>HTTP_SERVER_PORT</td>
<td>This macro specifies the port number of the HTTPS Server.</td>
</tr>
<tr>
<td>APP_SNTP_SVR_IP</td>
<td>This macro specifies the IP Address of the global SNTP Server.</td>
</tr>
</tbody>
</table>
Macro | Description
--- | ---
ADK_SNTP_ENABLED | To enable time sync with NTP server, this macro needs to be enabled in file preIncludeFile.txt.
SNTP_SERVER_DOMAIN_NAME | This macro specifies the domain name of NTP Server (or IP address of DNS is needed).
APP_SNTP_TIMEOUT | This macro specifies the time GS node waits after sending the SNTP request. It is 1 second by default.
APP_HIBERNATE_ENABLE | To go into Hibernate after every data send, this macro needs to be enabled in file preIncludeFile.txt.
HTTP_SERVER_URL | This macro specifies the script to be run on the Server side for the data received.

### 5.2.6.2 Code Flow

The code flow for the use case is as follows:

On boot-up, the TLS LP Application binary also called as the base binary is loaded using the same mechanism as any other application.

Boot ROM -> app startup code -> TLS LP code load, remap, call app_main().

app_main is executed in boot ROM context and it performs the following:

1. Re-initializes the Flash access and performs the boot process.
2. Initializes power management module and sets the deep sleep state to active so that System does not go into deep sleep in idle loop.
3. Calls App_BasicInitEntry to create a new thread "Basic Init Thread" which performs basic initialization
4. Starts an idle loop which is the lowest priority task. This task will put the system in sleep/Deep sleep whenever possible (When no other thread is running, i.e. no activity is being done in the system)

```c
INT32 App_Main(VOID)
{
    GsnFlashf_FlashAccessReinit();
    App_BootProcess();
    GsnPwrMgmt_Init(&appCtx.pwrMgmtCtx);
    GsnPwrMgmt_DpSlpActivitycbSet(App_PwrMgmtDpSlpActivityCb);
    GsnPwrMgmt_DpSlpActivitySet(&appCtx.pwrMgmtCtx,
      (GSN_DP_ACTIVITY_BIT_MAP_T)APP_DPSLP_ACTIVITY_BASIC_INIT_INPROGRESS);
    App_BasicInitEntry();
    while(1)
    {
        __WFI();
    }
}
```
**App_BasicInitEntry**

This thread is created from app_main by calling App_BasicInitEntry. This newly created thread has a priority of 18. This has an entry function App_BasicInitThreadEntry which calls App_BasicInit.

App_BasicInit performs the following:

- Initializes clock and gets boot reason
- Initializes basic hardware, soft timer, GPIO, alarm, and UART
- Initializes file system and loads default configuration from flash file system if it is first time boot.
- If system boot-up is because of Alarm2, changes the network mode to Limited-AP, and resets the system.
- If wakeup is from alarm1, restores hibernate data.
- Powers up the WLAN Core.
- Calls API App_TriggerAppletLoad which posts a message to the Applet loader thread queue (appletLoaderMsgQueue) that always loads Applet2.

```c
VOID App_BasicInit(VOID)
{
..........................................
GsnClkCtl_Init(&pAppCtx->clkCtlCtx);
bootReason = GSN_BOOT_REASON_GET();
if(bootReason != GSN_WIF_SYS_BOOT_NORMAL_BOOT)
{
    GsnRtc_AlarmConfig(GSN_RTC_ALARM_2, 0, FALSE);
    GsnRtc_AlarmConfig(GSN_RTC_ALARM_3, 0, FALSE);
    App_BasicHwInit();
}
/*Initialize Soft Timer */
App_SoftTimerInit(pAppCtx);
App_GpioInit(); /* Initialize alarm */
App_AlarmsInit(pAppCtx);
AppDbg_UartInit();
App_RestoreHibernateData(pAppCtx);

if(App_IsWlanBootReq() == FALSE)
{
    goto APP_TRIGGER_APPLET_LOAD;
}
GsnExcpHdr_Init();
App_HlInit(pAppCtx); /*Initialise wif System Control message Handler */
App_SysCtlInit(pAppCtx);

APP_TRIGGER_APPLET_LOAD:
App_AppletLoaderInit();
App_Cm3ClkCfg(&pAppCtx->cm3ClkCfg);
```

80560NT11589A Rev. 2.3 Page 75 of 109 2018-05-04
App_TriggerAppletLoad();

App_TriggerAppletLoad(VOID)
{
    ..........         
    else if(0 == pAppRtcLtchMemInfo->numDataTfrItern ||
        APP_DATA_SEND_ITERATION <= pAppRtcLtchMemInfo->numDataTfrItern
        || bootInfo == GSN_WIF_SYS_BOOTINFO_NORMALBOOT_ALARM)
    {
        /* in case of alarm wake up don't disturb the cycle*/
        if(bootInfo != GSN_WIF_SYS_BOOTINFO_NORMALBOOT_ALARM)
        {
            if(APP_DATA_SEND_ITERATION <= pAppRtcLtchMemInfo->numDataTfrItern)
                {
                    /* In case of data send iteration reset the iteration count to 1 */
                    pAppRtcLtchMemInfo->numDataTfrItern = 1;
                }
            else
                {
                    pAppRtcLtchMemInfo->numDataTfrItern++;
                }
        }

        /* When applet 2 is loaded for first time, applet2Load_ColdBoot value is set to 5 */
        if(5 != pAppRtcLtchMemInfo->applet2Load_ColdBootFlag)
        {
            /* Value 1: Tells applet2 is yet to load for the first time */
            pAppRtcLtchMemInfo->applet2Load_ColdBootFlag = 1;
        }

        /*Load Remaining Part of the code*/
        flags = (APPLETLOAD_FLAGS_T)(APPLETLOAD_FLAG_LOAD |
            APPLETLOAD_FLAG_CALL_ENTRY_FUNC);
        App_AppletLoad(2, flags);
       ******************************************************************************
    
    Up on first boot, since RTC variable pAppRtcSecMemInfo->numDataTfrItern is 0, Applet2 will be loaded.

    On every wakeup from standby, whenever RTC member variable numDataTfrItern is 0, TLS LP or base firmware dynamically loads Applet_2 onto RAM location 0x20050000 as mentioned in the Applet header descriptor field appletSramAddr using the function App_TriggerAppletLoad. Once code is loaded, the entry function in Applet 2 code App_Applet2Entry is called as mentioned in SingleImageBuilderConfig.txt present in build folder.

******************************************************************************
appId=2
appletBin=\Debug\Exe\applet_2.bin
appletMap=\Debug\List\applet_2.map
appletEntryFunc=App_Applet2Entry
appletSramAddr=20050000
******************************************************************************
The applet_2 entry function App_Applet2Entry does the following:

Creates Application thread "Applet2 Thread" with priority 17. This thread has entry function App_Applet2ThreadEntry which performs the following:

1. Starts a 100ms timer for the sensors to stabilize, only then data can be read. This is performed in the Applet2_main.c file.

```c
App_CreateSensorStabilizeLock();
GSN_PWR_MGMT_WCM3_SLEEP_DEEP_EN();
sensorStabilizeTmrHndl = GsnSoftTmr_Start(&sensorStabilizeTmr,
GSN_SOFT_TMR_ONESHOT, 0,
GSN_SOFT_TMR_MILLISECONDS_TO_TICKS(90),
App_SensorStabilizeTmrCb,
NULL);
(void)(sensorStablizeTmrHndl); /*Compilation Warning suppression*/
App_WAITOnSensorStabilizeLock();
```

2. Sends boot request for the WLAN using the API App_WlanBootReq.
3. Initializes the supplicant and random number generator.
4. Initializes wdd msg handler, network stack, interface, BsdLayer, state machine (sm), Network connection manager (NCM), and posts an event (APP_EVENT_NW_START_NCM) to the main task to start NCM using API App_applet2TaskNotify (APP_EVENT_NW_START_NCM, 0).
5. Applet2 thread infinitely waits on messages on the queue applet2ThreadMsgQueue.

```c
while( TRUE )
{
    /**< wait on message queue */
    GsnOsal_QueueGet( &applet2ThreadMsgQueue, ( UINT8* )&msg,
    GSN_Osal_WAIT_FOREVER );
    /**< Process the message */
    GsnSq.TaskMonitorStart(APP_CFG_SQ_MAIN_TASK_ID, msg.msgId);
    AppSm_MsgProcess(pAppCtx, msg);
    GsnSq.TaskMonitorStop(APP_CFG_SQ_MAIN_TASK_ID);
}
```

6. NCM takes the configuration information from the app_default_config.h and connects to the SSID. Once GS Node is successfully connected and IP address has been allocated which is received from the AP, it sends out APP_EVENT_NW_CONNECT_DONE to the sm.

7. Applet2 main task (app_sm.c) processes this event and checks whether the sensor data needs to be sent or not. If sensor data need not be sent, Applet2 main thread breaks from the case statement. If sensor data needs to be sent, then Applet2 proceeds with the next event APP_EVENT_DATA_SEND which is triggered periodically (every 5 second) from the TLS LP code using the API App_applet2TaskNotify (APP_EVENT_DATA_SEND, 0).

8. As part of APP_EVENT_DATA_SEND event handling, Applet2’s sm thread calls AppSm_DataSend to read sensors.
9. Calls AppSm_DataSend to read sensors.
10. Call AppTls_SensorRead to initialize the ADC component and read the temperature and light sensor values into the RTC array.
11. Increments UdpSeqNo and stores it in RTC.
12. If http flag is enabled, calls AppHttp_DataSend to send data over https. This function performs the following:
   a. Prepares the data to be sent.
   b. Loads certificate
   c. Gets the HTTP Server address from app_default_cfg.h
   d. Opens the https connection on the given port.
   e. Calls setsock option to set the Tx retry rate to 2 seconds, number of transmission to 3, and sets the priority to voice.
   f. Populates the http headers.
   g. Sets the timeout for receiving response.
   h. Posts data to the server.


5.2.6.3 Demo

Perform the following steps for the demo:

**Step 1.** The macros mentioned should be enabled or modified for sending sensor data using HTTPS (app_default_cfg.h and preIncludeFile.txt):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_MODE</td>
<td>GSN_INFRA_CLIENT</td>
</tr>
<tr>
<td>DEFAULT_CLI SSID</td>
<td>&quot;ASUS&quot;</td>
</tr>
<tr>
<td>APP_DATA_READ_INTERVAL</td>
<td>5000 // standby is for 5 second</td>
</tr>
<tr>
<td>APP_DATA_SEND_ITERATION</td>
<td>2 // on every even iteration it will send data</td>
</tr>
<tr>
<td>DEFAULT_CLI_CHNL</td>
<td>11</td>
</tr>
<tr>
<td>DEFAULT_CLI_SECURITY</td>
<td>GSN_WLAN_WPA_PERSONAL</td>
</tr>
<tr>
<td>DEFAULT_CLI_PASSPHRASE</td>
<td>“xxxxxxxx” //xxxxxxxx will be replaced with AP pwd</td>
</tr>
<tr>
<td>APP_SEND_DATA_OVER_HTTP</td>
<td>To enable sending sensor data via HTTPS, this macro needs to be enabled in file preIncludeFile.txt (remove __)</td>
</tr>
<tr>
<td>HTTP_SERVER_ADDRESS</td>
<td>192.168.2.223</td>
</tr>
<tr>
<td>HTTP_SERVER_PORT</td>
<td>443</td>
</tr>
<tr>
<td>ADK_SNTP_ENABLED</td>
<td>To enable time, sync with NTP server, this macro needs to be enabled in file preIncludeFile.txt (remove __)</td>
</tr>
<tr>
<td>APP_SNTP_SVR_IP</td>
<td>-</td>
</tr>
</tbody>
</table>
Macro | Description
---|---
SNTP_SERVER_DOMAIN_NAME | pool.ntp.org //either IP address or DNS is needed
APP_SNTP_TIMEOUT | 3000
HTTP_SERVER_URL | "/cgi-bin/get1.pl?10"
APP_HIBERNATE_ENABLE | To go into Hibernate after every data sent, this macro needs to be enabled in file preincludeFile.txt (remove __)

**Step 2.** Make build by pressing F8 and selecting MAKE. This generates the bin file. Program the binary to GS node.

**Step 3.** Follow the steps mentioned in GS2K TLS Low Power Server Setup User Guide.pdf and start the HTTPS Server (Section 2.1 HTTPS Server) by executing the commands mentioned. Apache server can be started in Laptop which is connected to the same Access Point as GS node. Here 443 is the HTTPS Port of the Server which is same as given in the macro configuration file (app_default_cfg.h).

**Step 4.** Start the GS Node. Verify from TeraTerm that it powers up, associates with the AP configured, and performs the following:
1. GS node performs a cold boot
2. Loads Applet2
3. Connects to configure SSID and the channel info provided using ncm.
4. Performs time sync with the NTP Server.
5. Reads sensor data.
7. Sends Post request to the Server.
8. Server sends 200 OK and 512 bytes of data as response all the time.
9. Receives the response and prints the data only.
10. Saves scan entry and IP Address in RTC memory
11. Goes to Hibernate state.

******************************************************************************************

[TLS LP] Boot Reason = 1
[TLS LP] Boot Reason = 0
[TLS LP] Boot Info:
  Boot Reason : 0
  Boot Info : 0
  Reset Type: 0
  Reset Reason : 0
  Reset Info Size: 0
  Reset Addl Info: 0x0
  ResetInfo: 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000

[TLS LP] Loading Applet, ID = 2
[TLS LP] Data Send Applet
[TLS LP] GEPS VERSION 5.3.0.27
[TLS LP] WLAN VERSION 5.3.0.52
[TLS LP] Ip Address = 0
[TLS LP] Device Configuration Info:
  MAC Addr    : 0:1d:c9:17:88:92
  SSID        : ASUS
  Channel     : 11
  DHCP        : TRUE
  Device Name : GS_178892

[TLS LP] Starting NCM (STA Mode):
[TLS LP] Nw Connect Done
[TLS LP] IP address Changed so Closing HTTP
  ****SNTP time sync*****[TLS LP]AppSm_TimeSyncDoneCb
[TLS LP] APP_EVENT_DATA_SEND Event: Connected. Send DataLux: RawVal = 0, ConvVal = 1675.000000
temp: RawVal = 0, ConvVal = -273.000000
[TLS LP] Sending Data Over HTTP
[TLS LP] Start http
[Tls_LP] GsnHttp_Start SUCCESSFUL
[TLS LP] HTTP connection established successfully
[TLS LP] Http Open Success

[TLS LP] datalen 50 httpTimeout = 1
[TLS LP] Sending POST request to Server
[TLS LP] HTTP POST response size = 520 & return Val = 0
32 30 30 20 4f 4b 0d 0a 53 57 41 4e 4e 20 44 41 20 44 41
54 41 20 55 50 4c 4f 41 44 20 53 55 43 43 45 53 3b 4e 4f
20 43 4f 4e 46 49 47 55 52 41 54 49 4f 4e 20 50 45 4e 44
49 47 20 54 20 44 45 56 49 43 45 3b 50 4l 45 41 53 45 20
47 4f 20 42 41 43 4b 20 54 20 53 4c 45 50

80560NT11589A Rev. 2.3  Page 80 of 109  2018-05-04
GS node performs the following when Alarm1 Button is pressed:

1. GS node wakes up with boot information as 1 which is GSN_WIF_SYS_BOOTINFO_NORMALBOOT_ALARM
2. Loads Applet2
3. Connects to SSID which is stored in RTC using NCM.
4. Performs a lean DHCP by using the saved IP Address.
5. Performs time sync with the NTP Server.
6. Reads sensor data.
7. Establishes HTTPS connection.
8. Sends Post request to the Server.
9. Server sends 200 OK and 512 bytes of data as response all the time.
10. Receives response and prints the data only.
11. Saves scan entry and IP Address in RTC memory
12. Goes to Hibernate state.

[TLS LP] Boot Reason = 0
[TLS LP] Boot Info:
   Boot Reason : 0
   Boot Info : 1
   Reset Type: 0
   Reset Reason : 0
   Reset Info Size: 0
   Reset Addl Info: 0x0
   ResetInfo: 0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000

[TLS LP] Loading Applet, ID = 2
[TLS LP] Data Send Applet
[TLS LP] GEFS VERSION 5.3.0.27
[TLS LP] WLAN VERSION 5.3.0.52
[TLS LP] Ip Address = c0a8010e
[TLS LP] dhcpRgstCurrIpAddr = 1, Ip Address = c0a8010e
[TLS LP] Device Configuration Info:
   MAC Addr     : 0:1d:c9:17:88:92
   SSID         : ASUS
   Channel      : 11
   DHCP     : TRUE
   Device Name  : GS_178892

[TLS LP] Starting NCM (STA Mode):
[TLS LP] Nw Connect Done
****SNTP time sync***** [TLS LP] AppSm_TimeSyncDoneCb
[TLS LP] APP_EVENT_DATA_SEND Event: Connected. Send DataLux: RawVal = 0, ConvVal = 1675.000000
   temp: RawVal = 0, ConvVal = -273.000000
[TLS LP] Sending Data Over HTTP
[TLS LP] Start http
[TLS LP] GsnHttp_Start SUCCESSFUL
[TLS LP] HTTP connection established successfully
[TLS LP] Http Open Success
[TLS LP] datalen 50 httpTimeout = 1
[TLS LP] Sending POST request to Server
[TLS LP] HTTP POST response size = 520 & return Val = 0

32 30 30 20 4f 4b 0d 0a 53 57 41 4e 20 4d 41 54 41 20 53 55 43 43 45 53 53 3b 4e 4f 20 43 4f 4e 46 49 47 55
30 49 50 20 48 4f 4f 52 41 59 2e 53 57 41 4e 20 44 41 54 41 20 53 55 43 43 53 53 3b 4e 4f 20 43 4f 4e 46 49 47 55
Scan ENtry 1 TLS LP Ip Address = c0a801
USE CASE 3 ALWAYS CLOUD CONNECTED (PS POLL APPLET MODE)

5.2.7 Using SSL over TCP

This use case demonstrates:

a. GS node being always cloud connected but still operating in low power mode.
b. GS node being woken up from remote server running SSL over TCP.
c. GS node being in standby between beacons on Customized wakeup interval.
d. GS node sends Keep Alive packets to the Server periodically and receiving response.

Since HTTP Server generally tends to close connection with the client due to inactivity for a certain time frame and requires Server-side configuration, SSL over TCP Server is used for demonstration.

Whenever the TCP SSL Keep Alive Timer expires (which is 300 seconds):

GS node sends TYPE 1 as data to Server.
Server sends TYPE 2 as response to GS Node.

Whenever the TCP SSL Server 60 second trigger timer elapses:

Server Sends TYPE 3 to GS node
GS node sends TYPE 4 to Server
Server sends Type 5 as ACK to GS Node.

5.2.7.1 Macros

Following are the macros required to be enabled or modified for receiving asynchronous events from cloud server (app_default_cfg.h and preIncludeFile.txt):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_MODE</td>
<td>This macro needs to be set to GSN_INFRA_CLIENT.</td>
</tr>
<tr>
<td>DEFAULT_CLI_SSID</td>
<td>This macro specifies the SSID of the Access Point to which GS node will be connected.</td>
</tr>
<tr>
<td>TCP_SSL_KEEP_ALIVE_INTERVAL</td>
<td>This macro is enabled by default where after 5 minutes, GS node will send a keep alive packet to the Server and waits for acknowledgement to know that the connection is alive.</td>
</tr>
<tr>
<td>TCP_SSL_DATA_LENGTH</td>
<td>This macro specifies the length of data to receive.</td>
</tr>
<tr>
<td>DEFAULT_CLI_CHNL</td>
<td>This macro specifies the channel on which SSID is operating.</td>
</tr>
<tr>
<td>DEFAULT_CLI_SECURITY</td>
<td>This macro specifies the type of security used on the Access Point.</td>
</tr>
</tbody>
</table>
### Macro Description

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_CLI_PASSPHRASE</td>
<td>This macro specifies the passphrase of the Access Point.</td>
</tr>
<tr>
<td>APP_SEND_DATA_OVER_HTTP</td>
<td>Disable this macro in file preIncludeFile.txt</td>
</tr>
<tr>
<td>HTTP_SERVER_ADDRESS</td>
<td>This macro specifies the IP Address of the TCP Over SSL Server receiving the sensor data.</td>
</tr>
<tr>
<td>HTTP_SERVER_PORT</td>
<td>This macro specifies the Port number of the TCP Over SSL.</td>
</tr>
<tr>
<td>APP_SNTP_SVR_IP</td>
<td>This macro specifies the IP Address of the global SNTP Server.</td>
</tr>
<tr>
<td>ADK_SNTP_ENABLED</td>
<td>To enable time sync with NTP server, this macro needs to be enabled this in file preIncludeFile.txt</td>
</tr>
<tr>
<td>SNTP_SEVER_DOMAIN_NAME</td>
<td>This macro specifies the Domain name of NTP Server //either IP address of DNS is needed.</td>
</tr>
<tr>
<td>APP_SNTP_TIMEOUT</td>
<td>This macro specifies the time GS node waits after sending the SNTP request. It is 1 second by default.</td>
</tr>
<tr>
<td>HTTP_SERVER_URL</td>
<td>This macro is not needed. Disable it.</td>
</tr>
<tr>
<td>APP_OPT_PS_IN_PSPOLL</td>
<td>It is defined if asynchronous data receive is required. In this case GS node, will establish TCP SSL connection with data server, and sends data when trigger is received from the server.</td>
</tr>
<tr>
<td>APP_NO_SENSOR</td>
<td>Enable this to receive trigger from Wi-Fi and hence this does not have any relevance in the demo.</td>
</tr>
</tbody>
</table>

### 5.2.7.2 Code Flow

The code flow for the use case is as follows:

On boot-up, the TLS LP Application binary also called as the base binary is loaded using the same mechanism as any other application.

Boot ROM -> app startup code -> TLS LP code load, remap, call app_main()

‘app_main’ is executed in boot ROM context and it performs the following:

1. Re-initializes the Flash access and performs the boot process.
2. Initializes power management module and sets the deep sleep state to active so that System does not go into deep sleep in idle loop.
3. Calls App_BasicInitEntry to create a new thread “Basic Init Thread” which performs basic initialization
4. Starts an idle loop which is the lowest priority task. This task will put the system in sleep/Deep sleep whenever possible (When no other thread is running, i.e. no activity is being done in the system)
INT32 App_Main(VOID)
{
    GsnFlashif_FlashAccessReinit();
    App_BootProcess();
    GsnPwrMgmt_Init(&appCtx.pwrMgmtCtx);
    GsnPwrMgmt_DpSlpActivitycbSet(App_PwrMgmtDpSlpActivityCb);
    GsnPwrMgmt_DpSlpActivitySet(&appCtx.pwrMgmtCtx,
        (GSN_DP_ACTIVITY_BIT_MAP_T)APP_DPSLP_ACTIVITY_BASIC_INIT_INPROGRESS);
    App_BasicInitEntry();

    while(1)
    {
        __WFI();
    }
}  

App_BasicInitEntry

This thread is created from app_main by calling App_BasicInitEntry. This newly created thread has a priority of 18. This has an entry function App_BasicInitThreadEntry which calls App_BasicInit.

App_BasicInit performs the following:

- Initializes clock, gets boot reason
- Initializes basic hardware, soft timer, GPIO, alarm, and UART
- Initializes persistent timer, file system and loads default configuration from flash file system if it is first time boot.
- If system bootup is because of Alarm2, change the network mode to Limited-AP, and resets the system.
- Initializes the lower system modules like exception handler, hi layer and sys ctl.
- If bootup is due to DHCP lease expiry, set the bootup event to APP_PS_BOOT_ACTIVITY_DHCP_RENEW.
- If bootup is due to TCP SSL keep alive timer expiry which is 300 seconds by default, set the bootup event to APP_PS_BOOT_ACTIVITY_TCPSSL_KEEPALIVE.
- If bootup is due to status update event, set the bootup event to APP_PS_BOOT_ACTIVITY_DATA_SEND.
- Release the reset of the WLAN and give bootup request informing him that the APP CPU is operating in PS POLL Applet mode
- Registers for various events such as DAB Ready (Data After Beacon), DAB Done, Disassociation event
- Sync loss event, beacon timer reset event) with wireless device driver(wdd).

**************************************************************************

if(GSN_SYS_IS_PS_BOOT_NORMAL() == TRUE)
{
    GsnWdd_AssocStatusSet(&appCtx->appNwIfc[APP_DFLT_CFG_INDEX].wddCtx);
    GsnWdd_DabDoneIndCbReg(&appCtx->appNwIfc[APP_DFLT_CFG_INDEX].wddCtx, App_PsDabDoneIndCb, &appCtx);
    GsnWdd_DabReadyIndCbReg(&appCtx->appNwIfc[APP_DFLT_CFG_INDEX].wddCtx, App_PsDabReadyIndCb, &appCtx);
    GsnWdd_DisassocIndCbReg(&appCtx->appNwIfc[APP_DFLT_CFG_INDEX].wddCtx, App_PsBootDisassocIndCb, &appCtx);
}
GsnWdd_BeaconMissCbReg(&pAppCtx->appNwIfc[APP_DFLT_CFG_INDEX].wddCtx,
App_PsBootBeaconMissIndCb, &appCtx);
GsnWdd_TimeStampRstCbReg(&pAppCtx->appNwIfc[APP_DFLT_CFG_INDEX].wddCtx,
App_PsBootTimeStampRstIndCb, &appCtx);
}

********************************************************************************

• WLAN wakes up and checks the boot-up message. If the App CPU is functioning in PS
poll applet mode, it too will load a small footprint binary which is capable of handling only
a subset of events (DAB Ready, DAB Done, Disassociation event, sync loss event,
beacon timer reset). It will check the beacon to see if any data is present for the node.
Accordingly, it will send the event to the APP CPU as no data (DAB Done) or data is
present (DAB Ready).

• If there is no data, DAB Done Indication from WLAN is received and hence we make the
dabIndFlag as TRUE. APP CPU will take the GS node in standby for the configured
beacon interval in the base thread itself:

*******************************
if ((pAppCtx->psPollCtx.dabIndFlag == 1)
&& (App_PsBootActivityGet() == 0))
{
    /* Reset the flag for next iteration */
    pAppCtx->psPollCtx.dabIndFlag = 0;

    /* Go to standby */
    App_PsGotoStandby(pAppCtx);
}
*******************************

• If there is data, GS node will start the Applet Loader Thread and call
App_TriggerAppletLoad to load Applet2.

*******************************
App_AppletLoaderInit();
App_Cm3ClkCfg(&pAppCtx->cm3ClkCfg);
App_TriggerAppletLoad();
*******************************

Up on first boot, since RTC variable pAppRtcSecMemInfo-> numDataTrfItern is 0, Applet2 will
be loaded.

On every wakeup from standby, whenever RTC member variable numDataTrfItern is 0, TLS LP
or base firmware dynamically loads Applet_2 onto RAM location 0x20050000 as mentioned in
the Applet header descriptor field appletSramAddr using the function App_TriggerAppletLoad.
Once code is loaded, the entry function in Applet 2 code App_Applet2Entry is called as
mentioned in SingleImageBuilderConfig.txt present in build folder.

******************************************************************************
appId=2
appletBin=./Debug\Exe\Applet_2.bin
appletMap=./Debug\List\Applet_2.map
appletEntryFunc=App_Applet2Entry
appletSramAddr=20050000
******************************************************************************

80560NT11589A Rev. 2.3 Page 87 of 109 2018-05-04
The applet_2 entry function App_Applet2Entry performs the following:

Creates Application thread "Applet2 Thread" with priority 17. This thread has entry function App_Applet2ThreadEntry which performs the following:

1. Initializes the supplicant and random number generator.
2. Initializes wdd msg handler, network stack, interface, BsdLayer, state machine (sm), Network connection manager (NCM), and posts an event (APP_EVENT_NW_START_NCM) to the main task to start NCM using API App_applet2TaskNotify(APP_EVENT_NW_START_NCM, 0).
3. If any of PS poll applet related events occur, make the data semdong flag as true.

```
if((GSN_WIF_SYS_BOOT_NORMAL_BOOT != GSN_BOOT_REASON_GET())
   || (GSN_WIF_SYS_BOOTINFO_NORMAL_BOOT_ALARM == GSN_BOOT_INFO_GET())
   || (GSN_SYS_IS_PS_BOOT_ANY()== TRUE))
pAppCtx->dataSendPending = 1;
```

4. Applet2 thread infinitely waits for messages on the queue applet2ThreadMsgQueue.

```
while( TRUE )
{
    /**< wait on message queue */
    GsnOsal_QueueGet( &applet2ThreadMsgQueue, ( UINT8* )&msg,
                      GSN_OSAL_WAIT_FOREVER );
    /**< Process the message */
    GsnSq_TaskMonitorStart(APP_CFG_SQ_MAIN_TASK_ID, msg.msgId);
    AppSm_MsgProcess(pAppCtx, msg);
    GsnSq_TaskMonitorStop(APP_CFG_SQ_MAIN_TASK_ID);
}
```

5. NCM takes the configuration information from the app_default_config.h and connects to the SSID. Once GS Node is successfully connected and IP address has been allocated which is received from the AP, it sends out APP_EVENT_NW_CONNECT_DONE to the sm.

6. Applet2 main task (app_sm.c) processes APP_EVENT_NW_CONNECT_DONE event and checks the reason for wakeup and takes action accordingly.
If it is normal boot, it registers callback for data indication and starts TCP over SSL client.
If it is TCP SSL keep alive boot, it posts message to Applet2 thread to handle.
If it is DHCP boot, it clears activity and registers for DabDoneIndCb event.
If it is Data send event, it notifies Applet2 of the event.

```
case APP_EVENT_NW_CONNECT_DONE:
                    
                    if ( (GSN_SYS_IS_PS_BOOT_NORMAL() == TRUE) 
                    {
                        /*Register call back for Data indication*/
                        GsnWdd_DabDoneIndCbReg(&pAppCtx->appNwIfc[APP_DFLT_CFG_INDEX].wddCtx,
                                              &AppSm_DabDoneIndCb, pAppCtx->sysCtlCtx);
                        GsnSysCtl_InterfaceResume(&pAppCtx->sysCtlCtx);
                    }
                    
```
7. On seeing no more data from router, WLAN CPU will send DabDone indication to APP CPU.
8. APP CPU on receiving this event will stop NCM, saves the parameters in RTC and puts the system in standby.

5.2.7.3 Demo

Perform the following steps for the demo:

Step 1. The macros mentioned should be enabled or modified for sending sensor data using HTTPS (app_default_cfg.h and preIncludeFile.txt):

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_MODE</td>
<td>GSN_INFRA_CLIENT</td>
</tr>
<tr>
<td>DEFAULT_CLI_SSID</td>
<td>&quot;ASUS&quot;</td>
</tr>
<tr>
<td>TCP_SSL_KEEP_ALIVE_INTERVAL</td>
<td>30</td>
</tr>
<tr>
<td>TCP_SSL_DATA_LENGTH</td>
<td>32</td>
</tr>
<tr>
<td>DEFAULT_CLI_CHNL</td>
<td>11</td>
</tr>
<tr>
<td>DEFAULT_CLI_SECURITY</td>
<td>GSN_WLAN_WPA_PERSONAL</td>
</tr>
</tbody>
</table>

Macro Table:

- **DEFAULT_MODE**: GSN_INFRA_CLIENT
- **DEFAULT_CLI_SSID**: "ASUS"
- **TCP_SSL_KEEP_ALIVE_INTERVAL**: 30
- **TCP_SSL_DATA_LENGTH**: 32
- **DEFAULT_CLI_CHNL**: 11
- **DEFAULT_CLI_SECURITY**: GSN_WLAN_WPA_PERSONAL
Macro Description

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_CLI_PASSPHRASE</td>
<td>“xxxxxxxx” //xxxxxxxx will be replaced with AP pwd</td>
</tr>
<tr>
<td>APP_SEND_DATA_OVER_HTTP</td>
<td>Disable this macro in file preIncludeFile.txt((add __))</td>
</tr>
<tr>
<td>HTTP_SERVER_ADDRESS</td>
<td>192.168.1.51</td>
</tr>
<tr>
<td>HTTP_SERVER_PORT</td>
<td>5000</td>
</tr>
<tr>
<td>APP_SNTP_SVR_IP</td>
<td>-</td>
</tr>
<tr>
<td>ADK_SNTP_ENABLED</td>
<td>To enable time sync with NTP server, this macro needs to be enabled in file preIncludeFile.txt(remove __)</td>
</tr>
<tr>
<td>SNTP_SEVER_DOMAIN_NAME</td>
<td>pool.ntp.org //either IP address of DNS is needed</td>
</tr>
<tr>
<td>APP_SNTP_TIMEOUT</td>
<td>3000</td>
</tr>
<tr>
<td>HTTP_SERVER_URL</td>
<td>-</td>
</tr>
<tr>
<td>APP_OPT_PS_IN_PSPOLL</td>
<td>Enable this in file preIncludeFile.txt (remove __)</td>
</tr>
<tr>
<td>APP_NO_SENSOR</td>
<td>Enable this in file preIncludeFile.txt (remove __)</td>
</tr>
</tbody>
</table>

Step 2. Make build by pressing F8 and selecting MAKE. This generates the bin file. Program the binary to GS node.

Step 3. Follow the steps mentioned in GS2K TLS Low Power Server Setup User Guide.pdf and start the SSL over TCP Server (2.2 TCP Server over SSL) using the commands mentioned. Node JS can be started in Laptop which is connected to the same Access Point as GS node. Here 5000 is the Port of the TCP SSL Server which is the same as given in the macro configuration file (app_default_cfg.h).

Step 4. Start the GS Node. Verify from TeraTerm that it powers up, associates with the AP configured, and performs the following:
1. GS node performs a cold boot
2. Loads Applet2
3. Connects to configure SSID and the channel info provided using NCM.
4. Opens SSL over TCP.
5. Once connected, sends TYPE 4 message to Server.
6. Receives TYPE 5 message from Server.
7. Starts the TCP SSL Keep Alive Timer of 5 minutes.
8. Goes to standby and wakes up every Custom Interval configured(500ms)
9. If there is no trigger from Server or Timer event elapses, goes to standby.

*******************************************************************
[TLS LP] Boot Reason = 0
[TLS LP] Boot Info:
  Boot Reason : 0
Boot Info : 0
Reset Type: 0
Reset Reason : 0
Reset Info Size: 0
Reset Addl Info: 0x0
ResetInfo:
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
[TLS LP] Loading Applet, ID = 2
[TLS LP] Data Send Applet
[TLS LP] GEPS VERSION 5.3.0.27
[TLS LP] WLAN VERSION 5.3.0.52
[TLS LP] Ip Address = 0
[TLS LP] Device Configuration Info:
  MAC Addr     : 0:1d:c9:17:88:92
  SSID         : ASUS
  Channel      : 11
  DHCP         : TRUE
  Device Name  : GS_178892
[TLS LP] Starting NCM (STA Mode):
[TLS LP] Nw Connect Done
[TLS LP] Open TCP SSL connection , 0
[TLS LP] App_TcpSSLConnOpen on socketId 32
[TLS LP] GsnSsl_Open status 0
[TLS LP] TCP SSL KA started
[TLS LP] APP_EVENT_DATA_SEND Event: Connected. Send Data
[TLS LP] Sending Data Over TCP Ssl
[TLS LP] Send Data outDataLen 181
[TLS LP] TCP POST response size = 134, status 0
50 4f 53 54 20 2f 63 67 69 2d 62 69 6e 2f 73 77
61 6e 6e 65 74 2e 70 6c 20 48 54 54 50
33 32 0d 0a 61 61 61 61 61 61 61 61
GS node performs the following when the 60 second trigger timer on the Server expires:

1. Server sends TYPE 3 message
2. Loads Applet2 and resumes SSL over TCP.
3. Receives and process messages.
4. Sends TYPE 4 message.
5. Waits and receives TYPE 5 message from Server.
6. Goes to standby and wakes up at every Custom Interval configured(500ms)
GS2K TLS Low Power ADK Application Note

2f 31 2e 31 0d 0a 48 4f 53 54 3a 31 39 32 2e 31 36 38 2e 31 35 31 0d 0a 43 4f 4e 4e 45 43 54 49 4f 4e 3a 20 6b 65 65 70 2d 61 6c 69 76 65 0d 0a 43 4f 4e 54 45 4e 54 2d 4c 45 4e 47 48 3a 33 32 0d 0a 0d 0a 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 54 59 50 45 3d 32 0d 0a [TLS LP] byteRead 134 [TLS LP] Type 5 received [TLS LP] more data to receive [TLS LP] TCP POST response size = 0, status 80000009

[TLS LP] byteRead 134
[TLS LP] Boot Reason = 1
[TLS LP] Normal PS Poll boot
[TLS LP] Boot Reason = 1

Following steps are performed when the 300 second TCP SSL timer on the GS node expires:

1. GS node sends TYPE 1 message to Server.
2. Server sends TYPE 2 message to GS node.
3. GS node goes to standby and wakes up at every Custom Interval configured(500ms)

[TLS LP] Boot Reason = 1
[TLS LP] Boot Reason = 1
[TLS LP] TCP-SSL Keep alive boot
[TLS LP] Loading Applet, ID = 2
[TLS LP] Data Send Applet
[TLS LP] Date and time Sat, 12 Nov 2016 11:46:22 GMT
[TLS LP] Ip Address = 0
[TLS LP] Device Configuration Info:
   MAC Addr     : 0:1d:c9:17:88:92
   SSID         : ASUS
   Channel      : 11
   DHCP         : TRUE
   Device Name  : GS_178892
[TLS LP] Starting NCM (STA Mode):
[TLS LP] Nw Connect Done
[TLS LP] Open TCP SSL connection , 3232235827
[TLS LP] Resume TCP SSL connection
[TLS LP] Sending TCP SSL KA
[TLS LP] Send Data outDataLen 165
[TLS LP] TCP POST response size = 8, status 0 54 59 50 45 3d 32 0d 0a
[TLS LP] byteRead 8
[TLS LP] Type 2 received
[TLS LP] Boot Reason = 1
[TLS LP] Normal PS Poll boot
[TLS LP] Boot Reason = 1
Chapter 6. Power Measurement

This section describes how to setup the GS module with jumper setting and to perform Power Measurement in all the three use cases.

**USE CASE 1: PERIODIC DATA UPLOAD TO CLOUD (STANDBY MODE)**

**Step 1:** Make a build and for Use Case 1 as in Chapter-5 and program it on the GS2011 module.

**Step 2:** Jumper setting are as shown below:

---

**NOTE:** For more details refer, “GS2K Module Power Measurement Application Note”.

---

**Figure 11: Use Case 1 - Jumper settings**

**Step 3:** Connect the positive of 3.3V power supply to TP3 and negative to TP17. Connect J23 pins to multimeter, since current is being measured across the VRTC pins.
Step 4: Power ON the GS node and run the use case, where GS node sends sensor data to configured server and goes to standby for 5 minutes (by default). The multimeter will show the consumption as 7.4uA during this period.
**USE CASE 2: EVENT NOTIFICATION TO CLOUD (HIBERNATE MODE)**

**Step 1:** Make a build and for Use Case2 as mentioned previously and program it on the GS2011 module.

**Step 2:** Jumper setting are as shown below:

NOTE: For more details refer, “GS2K Module Power Measurement Application Note”.

![Figure 14: Use Case 2 – Jumper Settings](image)
Step 3: Connect the positive of 3.3V power supply to TP3 and negative to TP17. Connect J23 pins to Multimeter since current across the VRTC pins is measured.

![Image of multimeter and circuitboard](image1)

**Figure 15: Use Case 2 – PIN connection**

Step 4: Power ON GS node and run the use case, where GS node sends sensor data to the configured server. Whenever the Alarm is pressed and GS node goes into Hibernate mode which is the lowest power consuming state. The multimeter will show the consumption as 274 Nano Amperes during this period.

![Image of multimeter](image2)

**Figure 16: Use Case 3 – Power Measurement**
USE CASE 3: ALWAYS CLOUD CONNECTED (PS POLL APPLET MODE)

**Step 1:** Make a build and for Use Case3 as in Chapter 5 and program it on the GS2011 module.

**Step 2:** Jumper setting are as shown below:

```
NOTE: For more details refer, “GS2K Module Power Measurement Application Note”.
```

![Figure 17: Use Case 3 – Jumper Settings](image-url)
**Step 3:** Connect the positive of 3.3V power supply to TP3 and negative to TP17. Connect J23 pins to Multimeter since current across the VRTC pins is measured.

![Image of PIN Connections](image)

**Figure 18: Use Case 3 – PIN Connections**

**Step 4:** Power ON GS node and run the use case. GS node will be in deep sleep in between data processing and will be in standby between beacons thus saving power.
Chapter 7. Profiling

This section describes the time taken by each of the software components for execution.

To achieve this a mechanism is introduced, where the GPIO’s are made high when the software component starts and made low once completed. This gives the exact measure of the time taken for the software component to execute when it is captured and plotted on a CRO. This mechanism is useful for customers to calculate the battery life and exactly know which software components consumes how much Power and decide the best power saving mechanism for the application.

From the code perspective, GPIO8, GPIO9 and GPIO3 are used for profiling. The same GPIO’s are used at different level of code flow.

The following table gives the System Events and the GPIO’s assigned for the 3 use cases. It also covers both non-sensor data as well as sensor data.

<table>
<thead>
<tr>
<th>System Events</th>
<th>Use-Case: Periodic data upload to cloud over HTTPS/UDP</th>
<th>Use-Case: Periodic data upload to cloud over HTTPS/UDP</th>
<th>Use-Case: Event Notification to Cloud over HTTPS/UDP</th>
<th>Use-Case: Always Cloud Connected over HTTPS-TCP-SSL (Sub event: Wake up from standby to receive beacon)</th>
<th>Use-Case: Always Cloud Connected over HTTPS- TCP-SSL (Sub event: Wake up from standby to exchange data with cloud server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>From application CPU bootup till start of release of wlan CPU reset. Note: Reset of WLAN-CPU isn’t released during this entire event.</td>
<td>GPIO 0</td>
<td>GPIO 0</td>
<td>GPIO 0</td>
<td>GPIO 0</td>
<td>GPIO 8</td>
</tr>
<tr>
<td>From Temperature &amp; Light sensor stabilization start till data is read from the sensor.</td>
<td>NA</td>
<td>GPIO 9</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Wireless-LAN CPU bootup (Simultaneous with Wireless-LAN CPU bootup)</td>
<td>GPIO 0</td>
<td>GPIO 8</td>
<td>GPIO 0</td>
<td>GPIO 9</td>
<td>GPIO 0</td>
</tr>
<tr>
<td>Application CPU bootup, post Wireless-LAN backup completion</td>
<td>GPIO 8</td>
<td>GPIO 8</td>
<td>GPIO 8</td>
<td>GPIO 3</td>
<td>GPIO 3</td>
</tr>
<tr>
<td>Restoring wireless configurations</td>
<td>GPIO 3</td>
<td>GPIO 3</td>
<td>GPIO 3</td>
<td>GPIO 3</td>
<td>GPIO 3</td>
</tr>
<tr>
<td>Restoring IP &amp; socket configurations</td>
<td>GPIO 8</td>
<td>GPIO 8</td>
<td>GPIO 8</td>
<td>NA</td>
<td>GPIO 8</td>
</tr>
<tr>
<td>Data read from sensors</td>
<td>NA</td>
<td>GPIO 8</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Data exchange with cloud server</td>
<td>GPIO 3</td>
<td>GPIO 3</td>
<td>GPIO 8</td>
<td>NA</td>
<td>GPIO 3</td>
</tr>
<tr>
<td>Preparing the system (both Application &amp; Wireless-LAN CPU) for transition to low power mode (Standby or Hibernate).</td>
<td>GPIO 9</td>
<td>GPIO 9</td>
<td>GPIO 9</td>
<td>GPIO 9</td>
<td>GPIO 9</td>
</tr>
<tr>
<td>Preparing Wireless-LAN CPU for transitioning to low power mode (Standby or Hibernate).</td>
<td>GPIO 8</td>
<td>GPIO 8</td>
<td>NA</td>
<td>GPIO 8</td>
<td>GPIO 8</td>
</tr>
</tbody>
</table>

| Table 5: System Events & GPIOs for the Use Cases |
As the customers, may have different sensors - their stabilization times may differ. An option is provided to bypass this sensor reading from the Temperature and Light Sensor and send pre-defined dummy data.

The following are the CRO details of Use Case 1 - ‘Periodic data upload data to cloud (standby)’, where non-sensor data is sent to UDP Server.

**GPIO 8** => Blue  
**GPIO 9** => Green  
**GPIO 3** => Pink  
**Power Graph** => Yellow

The following snapshots shows, when GS node wakes up from standby, apply the settings from the RTC memory and send pre-configured data to UDP Server and goes back to standby. Link Check (Sending ARP to Access Point before sending data packet) has been disabled.

![Power Graph for wakeup from standby, send data and again back to standby](image)

**Figure 19: GS Node wake from Standby**

The numbered indication in the graph or the figure above is as explained:

1. GS Node wakes up from standby and does some basic initialization. The blue color indicates the start of initialization code and ends when boot request to WLAN is sent.
2. App CPU after some basic initialization gives boot request to WLAN. Green Color indicates the start of WLAN boot request and (4) shows the return from WLAN indicating completion of WLAN bootup.
3. APP CPU after giving boot request to WLAN goes ahead with its other initialization. Pink Color indicates the start of initialization and point (5) indicates the remaining initialization done by the APP CPU after WLAN bootup is complete.
4. Restoration of L2 Connection. The Blue line indicates L2 configuration is taken from RTC RAM and applied.
5. Restoration of L3 and sending of data. The thick Pink color line indicates restoration of L3 stored in RTC RAM and then sending of data to the configured server which ends @ (8).
6. Indicates start of standby. The Green Color line indicates the APP CPU has started the standby procedure.
10. Give standby request to WLAN. The Blue line indicates APP CPU sending an indication to WLAN informing that it wants to go to standby. WLAN responds as shown @ (11).
12. After receiving the response from WLAN, APP CPU will store relevant data into RTC RAM and goes into standby which is indicated by the green color line @ (12).
## Appendix A  Macro’s Information

Following table lists the description of macros in App_default_cfg.h and preIncludeFile.txt:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_SEND_DATA_OVER_HTTP</td>
<td>This macro (preIncludeFile.txt) is used to Enable/Disable posting of sensor data to configured server over HTTPS. By default, data is sent over UDP. This is DISABLED by default.</td>
</tr>
<tr>
<td>HTTP_SERVER_ADDRESS</td>
<td>If sensor data is sent via HTTPS, then either IP Address or domain name of the server needs be configured. If HTTPS Server has a name, then this macro (App_default_cfg.h) should be configured with the HTTPS Server IP Address. Prerequisite: APP_SEND_DATA_OVER_HTTP needs to be enabled in preInclude.txt.</td>
</tr>
<tr>
<td>HTTP_SERVER_DOMAIN_NAME</td>
<td>If sensor data is sent via HTTPS, then either IP Address or domain name of the server needs be configured. If HTTPS Server has a name, then this macro (App_default_cfg.h) should be configured with the HTTPS Server Name. Prerequisite: APP_SEND_DATA_OVER_HTTP needs to be enabled in preInclude.txt.</td>
</tr>
<tr>
<td>HTTP_SERVER_URL</td>
<td>This macro (App_default_cfg.h) is used to configure HTTPS Server URL on which POST is done.</td>
</tr>
<tr>
<td>HTTP_RESPONSE_TIMEOUT</td>
<td>This is the time for which GS node waits for receiving response from HTTPS Server before closing the connection on its side.</td>
</tr>
<tr>
<td>HTTP_SERVER_PORT</td>
<td>This is the port of HTTPS Server which is 443 by default.</td>
</tr>
<tr>
<td></td>
<td><strong>Prerequisite:</strong> APP_SEND_DATA_OVER_HTTP needs to be enabled in preInclude.txt.</td>
</tr>
</tbody>
</table>
**Macro** | **Description**
--- | ---
**APP_NO_SENSOR** | If "APP_NO_SENSOR" macro is enabled in preInclude.txt, no timer is started for sensor stabilization and dummy sensor data (APP_NO_SENSOR_DATA_LEN) is sent over UDP/HTTPS. GS2100 EVB's do not have temperature and light sensors connected and hence this option needs to be enabled so that dummy sensor data is sent to configure server.
This is DISABLED by default.

**APP_NO_SENSOR_DATA_LEN** | On EVBs which do not have sensors (GS2100), there is provision to send dummy data. The number of dummy bytes that needs to be sent can be configured with this macro (App_defaultCfg.h). Here, valid number is between 1 to 512.
**Prerequisite:** APP_NO_SENSOR needs to be enabled in preInclude.txt and default value is 512.

**UDP_SERVER_ADDRESS** | If sensor data is sent via UDP, then either IP Address or domain name of the UDP Server needs be configured. If UDP Server has a name, then this macro (App_defaultCfg.h) should be configured with the UDP Server IP Address.

**UDP_SERVER_DOMAIN_NAME** | If sensor data is sent via UDP, then either IP Address or domain name of the UDP Server needs be configured. If UDP Server has a name, then this macro (App_defaultCfg.h) should be configured with the UDP Server Name.

**UDP_SERVER_PORT** | This macro (App_defaultCfg.h) is used to configure the port of UDP Server to which sensor data is being sent.

**UDP_RECEIVE_TIMEOUT** | If there is UDP level ACK expected from UDP Server, this timeout macro (App_defaultCfg.h) needs to be configured.
**Prerequisite:** APP_UDP_DATA_RECEIVE_ENABLE needs to be enabled in preInclude.txt.

**APP_UDP_DATA_RECEIVE_ENABLE** | If this macro (preIncludeFile.txt) is defined, application waits for the UDP response packet on the same socket over which data is sent.
This is DISABLED by default and default value is 1 Sec.
<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_MAC_LEVEL_CONFIRM_REQUIRED</td>
<td>This macro (preIncludeFile.h) is applicable only for data sent over UDP. When this macro is enabled, MAC level confirmation for the UDP packet sent is checked after the transmission. This is ENABLED by default.</td>
</tr>
<tr>
<td>APP_WIFI_LINK_KEEPALIVE</td>
<td>If GS node is in standby for a longer duration and customer wants to maintain L2 level association with the Access Point, then the macro APP_WIFI_LINK_KEEPALIVE needs to be enabled in preIncludeFile.txt to start the keep alive procedure. If APP_KEEP_ALIVE_STATIC macro is also enabled, then static keep Alive is triggered every 60 seconds. If APP_LINK_CHECK_USING_ARP is enabled, then dynamic keep alive procedure will be triggered. This is DISABLED by default. <strong>Prerequisite:</strong> APP_LINK_CHECK_USING_ARP or APP_KEEP_ALIVE_STATIC needs to be enabled in preInclude.txt.</td>
</tr>
<tr>
<td>APP_KEEP_ALIVE_STATIC</td>
<td>If GS node is in standby for a longer duration and customer wants to maintain L2 level association with the Access Point, then a static keep alive timer can be configured by modifying this macro (preIncludeFile.txt) which is 60 seconds by default. GS node wakes up every 60 seconds and sends a Gratuitous ARP and goes back to standby. This is DISABLED by default. <strong>Prerequisite:</strong> APP_WIFI_LINK_KEEPALIVE needs to be enabled in preInclude.txt.</td>
</tr>
<tr>
<td>GS2200M_EVB</td>
<td>Enable this (preIncludeFile.txt) to support GS2200 module. This is DISABLED by default.</td>
</tr>
<tr>
<td>GS2200M_SKB</td>
<td>Enable this (preIncludeFile.txt) support GS2200 SKB module. This is DISABLED by default.</td>
</tr>
<tr>
<td>APP_LINK_CHECK_USING_ARP</td>
<td>If this macro (App_default_cfg.h) is enabled, ARP is done to gateway after every DEFAULT_LINK_CHECK_ITERATION wakeup cycles to check if the L2 connection is intact. Failure to resolve the ARP indicates to a possible disassociation done by</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AP while the node is in standby and hence a fresh connection is triggered. This is ENABLED by default.</td>
<td></td>
</tr>
<tr>
<td>DEFAULT_LINK_CHECK_ITERATION</td>
<td>This is a configurable macro (App_default_cfg.h) that specifies the number of iterations after which ARP is done to gateway to check if L2 connection is intact. Prerequisite: APP_LINK_CHECK_USING_ARP needs to be enabled in App_default_cfg.h</td>
</tr>
<tr>
<td>DEFAULT_FAILURE_RATE</td>
<td>This is a configurable macro (App_default_cfg.h) that specifies the number of continuous link failures after which GS node directly re-associates and will not check the link status. This way it saves power wherein, AP has disassociated GS node and there is no point in doing a link status check thus wasting unwanted packet exchanges. This issue happened with a AP where Clients were being thrown out of network very frequently because of no activity for small durations.</td>
</tr>
<tr>
<td>ADK_SNTP_ENABLED</td>
<td>This macro (preIncludeFile.txt) will enable syncing system time with network time on association with an AP. This is DISABLED by default.</td>
</tr>
<tr>
<td>APP_SNTP_PERIODIC</td>
<td>This macro (preIncludeFile.txt) will enable syncing system time with network time periodically. This is DISABLED by default.</td>
</tr>
<tr>
<td>APP_SNTP_SVR_IP</td>
<td>This macro (App_default_cfg.h) is used to synchronize the time with SNTP Server needed to establish HTTPS connection. IP Address of the SNTP Server needs to be given. Prerequisite: ADK_SNTP_ENABLED needs to be enabled in preInclude.txt.</td>
</tr>
<tr>
<td>APP_SNTP_TIMEOUT</td>
<td>This is the macro (App_default_cfg.h) for configuring wait time for SNTP response. Prerequisite: ADK_SNTP_ENABLED needs to be enabled in preInclude.txt.</td>
</tr>
<tr>
<td>APP_POWER_MEASUREMENT_ENABLE</td>
<td>This macro (preIncludeFile.txt) is enabled for power profiling. When this is enabled, all the LEDs and all</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>APP_POWER_PROFILING</td>
<td>This macro (preIncludeFile.txt) is applicable to get the time taken by various activities while sending the data. It is used for power profiling. (Example: How much time WLAN firmware takes to boot up, how much time does the L2 &amp; L3 level association takes, and so on). This can be measured using CRO using GPIOs. This is DISABLED by default.</td>
</tr>
<tr>
<td>APP_DEEPSLEEP_DISABLE</td>
<td>Enable this (preIncludeFile.txt) to disable deep sleep. This is useful while using i-jet to debug the code. This is DISABLED by default.</td>
</tr>
<tr>
<td>APP_HIBERNATE_ENABLE</td>
<td>If this macro is defined (preIncludeFile.txt), l2 connection is teared down before going to Hibernate state which is ultra-low power. Only RTC alarms can wake up the System. This is DISABLED by default.</td>
</tr>
<tr>
<td>APP_OPT_PS_IN_PSPOLL</td>
<td>If this macro is defined (preIncludeFile.txt) if wakeup from WIFI mode (PS POLL Applet Mode) needs to be enabled. The System will be in ultra-low power state, ready to receive any asynchronous events from Server. This is DISABLED by default.</td>
</tr>
<tr>
<td>APP_STATUS_UPDATE_ENABLE</td>
<td>This macro is used (preIncludeFile.txt) if periodic data is to be updated to the server along with asynchronous data receive. <strong>Prerequisite:</strong> APP_OPT_PS_IN_PSPOLL needs to be enabled in preInclude.txt . This is DISABLED by default.</td>
</tr>
<tr>
<td>APP_DEBUG_LOG_STATS</td>
<td>Enable this macro (preInclude.txt) to log application debug statistics. To log statistics, and addition block of RTC memory is enabled. This is DISABLED by default.</td>
</tr>
</tbody>
</table>
| SENSOR_THRESHOLD_HANDLE                   | This macro (preInclude.txt) is used to enable sensor threshold based interrupt. This allows posting data to
<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>the server when any sensor value crosses configured thresholds.</td>
</tr>
<tr>
<td></td>
<td>This is ENABLED by default.</td>
</tr>
<tr>
<td>FACTORY_RESTORE_ENABLE</td>
<td>This macro (preInclude.txt) is used to restore the factory setting.</td>
</tr>
<tr>
<td></td>
<td>This is ENABLED by default.</td>
</tr>
</tbody>
</table>

Table 6: Macro’s Information Table
SUPPORT INQUIRIES

Link to www.telit.com and contact our technical support team for any questions related to technical issues.

www.telit.com