

CC3000 - CC3100 SimpleLink™ Wi-Fi® API Porting

Alon Srednizki

ABSTRACT

This application report serves as a guide to port CC3000-compatible code to the CC3100.

Contents

1	Introduction	1
2	Getting Started - First Steps	2
3	Application Programming Interfaces (APIs)	3
4	References	10

List of Tables

1	Easy Adaptation APIs	3
2	Medium Adaptation APIs	4

1 Introduction

The new CC3100 host driver adds greater support for different system environments. However, the driver APIs remains similar (not identical) to the ones used by the CC3000 driver.

NOTE: It is assumed that the CC3100 host driver porting (if not using one of the reference solutions provided in the [CC3100 Software Development Kit \(SDK\)](#) was already completed as described in the [CC3100 Programmers Guide wiki](#).

This application report outlines the changes needed to adapt application code to fit the CC3100, CC3200 Wi-Fi subsystem driver. It also reviews all APIs that require adaptation and divides them into sections by the level of effort required. Each API section contains a list of CC3000 APIs and their corresponding CC31xx APIs and examples on how the adaptation might be implemented.

2 Getting Started - First Steps

Before adapting the new APIs to the CC3100 application code, four steps are necessary to make sure the project is ready. It is recommended to use one of the existing CC31xx demo applications to help frame the steps below.

2.1 Reassign Include's

From the CC3000 to the CC31xx host driver, the header file definitions and configurations have changed with the intent of making things simpler. The application code needs to be pointed to the new header files to support the CC31xx.

The only header file needed by the CC31xx host driver is:


- “simplelink.h” – This header file includes everything necessary to support the CC31xx APIs. This header file replaces the CC3000’s “wlan.h”, “nvmmem.h”, “socket.h”, and “netapp.h”.

The resulting set of included header files for the CC3100 application code should be shorter and simpler than the set used with the CC3000.

```

#include "wlan.h"
#include "evnt_handler.h"
#include "nvmmem.h"
#include "socket.h"
#include "common.h"
#include "netapp.h"
#include "spi.h"

```



```

#include "simplelink.h"

```

2.2 Include “user.h”

The “user.h” file needs to be included within an application project. This file is the only file managing all platform and OS dependent items, therefore, it is the key file for doing driver porting to different MCU and OSs.

For more information, see the [CC3100 Programmers Guide wiki](#).

2.3 Verify Physical Layer

The CC31xx Host Driver utilizes the host platform’s physical interface (serial peripheral interface (SPI) or universal asynchronous receiver/transmitter (UART)) through user-configured functions assigned in the “user.h” file. These functions serve as the physical interface driver and are specific to the host platform. The function implementations are necessary for the following commands:

- Open
- Close
- Read
- Write

Unlike in CC3000, the CC3100 is using standard SPI protocol. Thus, platforms using the CC3000 need to modify their SPI driver to act as a standard SPI driver instead of a proprietary one. For further details, see the [CC3100 Host Interface User's Guide wiki](#).

2.4 Provide Callback Handler Routines for Asynchronous Events

Similar to the CC3000, the CC3100 generates asynchronous events in different situations. These asynchronous events must be caught through the use of the four event groups that can utilize or mask through user.h. The four event groups are:

- WLAN events
- Network Application events
- Socket events
- General device events

For further information on the CC3100 asynchronous events, see the [SimpleLink Wi-Fi Programmers User's Guide wiki](#).

In CC3000 applications, asynchronous events were handled via the `CC3000_UsynchCallback()` function. The callback functions that ties to the four handlers above need to replace `CC3000_UsynchCallback()` within the application code.

3 Application Programming Interfaces (APIs)

3.1 'Easy' Adaptation APIs

APIs that fall into the 'easy' adaptation category require little effort to port. The differences between these APIs are limited to function names, function return typecasts, and function parameter typecasts; the functionality of these APIs remains the same. There are 30 APIs that fall into the 'easy' adaptation category.

Table 1. Easy Adaptation APIs

Number	CC3000 API	CC31xx API
1	int socket (long domain, long type, long protocol)	_i16 sl_Socket(_i16 Domain, _i16 Type, _i16 Protocol)
2	long closesocket (long sd)	_i16 sl_Close(_i16 sd)
3	long accept (long sd, sockaddr *addr, socklen_t *addrlen)	_i16 sl_Accept(_i16 sd, SISockAddr_t *addr, SISocklen_t *addrlen)
4	long bind (long sd, const sockaddr *addr, long addrlen)	_i16 sl_Bind(_i16 sd, const SISockAddr_t *addr, _i16 addrlen)
5	long listen (long sd, long backlog)	_i16 sl_Listen(_i16 sd, _i16 backlog)
6	long connect (long sd, const sockaddr *addr, long addrlen)	_i16 sl_Connect(_i16 sd, const SISockAddr_t *addr, _i16 addrlen)
7	int select (long nfds, fd_set *readsds, fd_set *writesds, fd_set *exceptsds, struct timeval *timeout)	_i16 sl_Select(_i16 nfds, SIFdSet_t *readsds, SIFdSet_t *writesds, SIFdSet_t *exceptsds, struct STimeval_t *timeout)
8	int setsockopt (long sd, long level, long optname, const void *optval, socklen_t optlen)	_i16 sl_SetSockOpt(_i16 sd, _i16 level, _i16 optname, const void *optval, SISocklen_t optlen)
9	int getsockopt (long sd, long level, long optname, void *optval, socklen_t *optlen)	_i16 sl_GetSockOpt(_i16 sd, _i16 level, _i16 optname, void *optval, SISocklen_t *optlen)
10	int recv (long sd, void *buf, long len, long flags)	_i16 sl_Recv(_i16 sd, void *buf, _i16 Len, _i16 flags)
11	int recvfrom (long sd, void *buf, long len, long flags, sockaddr *from, socklen_t *fromlen)	_i16 sl_RecvFrom(_i16 sd, void *buf, _i16 Len, _i16 flags, SISockAddr_t *from, SISocklen_t *fromlen)
12	int send (long sd, const void *buf, long len, long flags)	_i16 sl_Send(_i16 sd, const void *buf, _i16 Len, _i16 flags)
13	int sendto (long sd, const void *buf, long len, long flags, const sockaddr *to, socklen_t tolen)	_i16 sl_SendTo(_i16 sd, const void *buf, _i16 Len, _i16 flags, const SISockAddr_t *to, SISocklen_t tolen)
14	int gethostbyname (char *hostname, unsigned short usNameLen, unsigned long *out_ip_addr)	_i16 sl_NetAppDnsGetHostByName(_i8 * hostname, _u16 usNameLen, _u32* out_ip_addr, _u8 family)
15	void wlan_stop (void)	_i16 sl_Stop(_u16 timeout)
16	long wlan_disconnect (void)	_i16 sl_WlanDisconnect(void)
17	long wlan_ioctl_del_profile (unsigned long ullIndex)	_i16 sl_WlanProfileDel(_i16 Index)

You might adapt the 'easy' APIs by simply replacing the old functions and their parameter typecasts with the new implementations. One way to do this is to use inline functions as a means to replace functions with their updated equivalents.

Below is an example of using an inline function to replace `socket()` with the new `sl_Socket()`. This inline function could sit in a header file, and could reference the CC31xx 'SimpleLink' header file in which `sl_Socket()` is defined (simplelink.h). Any new parameter types could be handled with a typecast.

```
#include "simplelink.h"

inline long socket(long domain, long type, long protocol)
{
    return (int)sl_Socket((int)domain, (int)type, (int)protocol);
}
```

NOTE: For all BSD socket APIs, the driver already has backward compatibility for standard BSD socket API names (as used in CC3000), by just defining "SL_INC_STD_BSD_API_NAMING" compilation flag.

3.2 'Medium' Adaptation APIs

APIs that fall into the 'medium' adaptation category require a little more effort to port. These APIs may differ in input parameters or name, but overall functionality remains the same.

Table 2. Medium Adaptation APIs

Number	CC3000 API	CC31xx API
1	long netapp_config_mac_address (unsigned char *mac)	sl_NetCfgSet(SL_MAC_ADDRESS_SET,1,SL_MAC_ADDR_LEN,(_u8 *)newMacAddress)
2	unsigned char nvmem_get_mac_address (unsigned char *mac)	sl_NetCfgGet(SL_MAC_ADDRESS_GET,NULL,&macAddressLen,(_u8 *)macAddressVal)
3	long netapp_dhcp (unsigned long *aucIP, unsigned long *aucSubnetMask, unsigned long *aucDefaultGateway, unsigned long *aucDNSServer)	For Static IP settings: SINetCfgIpV4Args_t ipV4; ipV4.ipV4 = (_u32)SL_IPV4_VAL(10,1,1,201); ipV4.ipV4Mask = (_u32)SL_IPV4_VAL(255,255,255,0); ipV4.ipV4Gateway = (_u32)SL_IPV4_VAL(10,1,1,1); ipV4.ipV4DnsServer = (_u32)SL_IPV4_VAL(8,16,32,64); sl_NetCfgSet(SL_IPV4_STA_P2P_CL_STATIC_ENABLE,IPCONFIG_MODE_ENABLE_IPV4,sizeof(SINetCfgIpV4Args_t),(_u8 *)&ipV4) For DHCP settings: _u8 val = 1; sl_NetCfgSet(SL_IPV4_STA_P2P_CL_DHCP_ENABLE,IPCONFIG_MODE_ENABLE_IPV4,1,&val)
4	void netapp_ipconfig (tNetappIpconfigRetArgs *ipconfig)	For Setting IP address: SINetCfgIpV4Args_t ipV4; ipV4.ipV4 = (_u32)SL_IPV4_VAL(10,1,1,201); ipV4.ipV4Mask = (_u32)SL_IPV4_VAL(255,255,255,0); ipV4.ipV4Gateway = (_u32)SL_IPV4_VAL(10,1,1,1); ipV4.ipV4DnsServer = (_u32)SL_IPV4_VAL(8,16,32,64); sl_NetCfgSet(SL_IPV4_STA_P2P_CL_STATIC_ENABLE,IPCONFIG_MODE_ENABLE_IPV4,sizeof(SINetCfgIpV4Args_t),(_u8 *)&ipV4) For getting IP address: _u8 len = sizeof(SINetCfgIpV4Args_t); _u8 dhcpIsOn = 0; SINetCfgIpV4Args_t ipV4 = {0}; sl_NetCfgGet(SL_IPV4_STA_P2P_CL_GET_INFO,&dhcpIsOn,&len,(_u8 *)&ipV4)
5	long wlan_connect (unsigned long ulSecType, char *ssid, long ssid_len, unsigned char *bssid, unsigned char *key, long key_len)	_i16 sl_WlanConnect(_i8* pName, _i16 NameLen, _u8 *pMacAddr, SISecParams_t* pSecParams, SISecParamsExt_t* pSecExtParams)
6	long wlan_add_profile (unsigned long ulSecType, unsigned char *ucSsid, unsigned long ulSsidLen, unsigned char *ucBssid, unsigned long ulPriority, unsigned long ulPairwiseCipher_Or_Key, unsigned long ulGroupCipher_TxKeyLen, unsigned long ulKeyMgmt, unsigned char *ucPf_OrKey, unsigned long ulPassPhraseLen)	_i16 sl_WlanProfileAdd(_i8* pName, _i16 NameLen, _u8 *pMacAddr, SISecParams_t* pSecParams, SISecParamsExt_t* pSecExtParams, _u32 Priority, _u32 Options)
7	long wlan_ioctl_set_connection_policy (unsigned long should_connect_to_open_ap, unsigned long should_use_fast_connect, unsigned long ulUseProfiles)	sl_WlanPolicySet(SL_POLICY_CONNECTION,SL_CONNECTION_POLICY((a,b,c,d,e),NULL,0)

The 'medium' APIs can be adapted by simply replacing the old functions and their parameter typecasts with the new implementations. Similar to the 'easy' adaptation APIs, one method of implementation is to use inline functions as a means to replace functions with their updated equivalents. Below is an example of using an inline function to replace `wlan_add_profile()` with the new `sl_ProfileAdd()`. This inline function could sit in a new header file, and could reference the CC31xx 'Simplelink' header file in which `sl_ProfileAdd()` is defined (`simplelink.h`). The new parameter types could be handled with a typecast.

3.3 APIs That are Implemented Differently

APIs that fall into the 'implemented differently' adaptation category require the most effort to port. These APIs may differ in functionality from their previous implementations.

3.3.1 Initializing WLAN Driver

In CC3000, it is custom to call the `wlan_init()` function for initializing the host driver callbacks and other driver parameters.

In CC3100, WLAN driver initialization is no longer explicitly required from a host application; much of the functionality is handled through the definitions given within `user.h` when doing the driver porting as explained in [Section 2.4](#).

3.3.2 Enabling the Wi-Fi Subsystem

In `CC3000()`, `wlan_start()` was used to enable the CC3000 device and handle its initialization process. With the CC3100, `wlan_start()` are no longer used; instead, `sl_Start()` is used as an equivalent replacement for both calls. The `sl_Start()` function has three parameters that can be passed through to customize the start sequence:

- `plfHdl` - In case the interface must be opened outside the SimpleLink Driver, the user might give the opened handler to be used in any access the communication interface with the device (UART/SPI). The SimpleLink driver will open an interface port only if this parameter is null. For porting CC3000 code, this parameter should be "0".
- `pDevName` - The name of the device to open. Could be used when the `plfHdl` is null to transfer information to the open interface function. For porting CC3000 code, this parameter should be "0".
- `plnitCallback` - Is a pointer to function that can be called on completion of the initialization process. If this parameter is NULL the function is blocked until the device initialization is completed, otherwise the function returns immediately and the callback function pointed by `plnitCallback` will be called upon completion of the initialization process.

3.3.3 Working With the NVMEM

In CC3000, there were three main functions to work with the non-volatile memory:

signed long	nvmem_create_entry	(unsigned long file_id, unsigned long length)
long	nvmem_read	(unsigned long file_id, unsigned long length, unsigned long offset, unsigned char *buff)
long	nvmem_write	(unsigned long file_id, unsigned long length, unsigned long offset, unsigned char *buff)

The general concept of these API consists of files stored in electrically erasable programmable read-only memory (EEPROM), approaching data stored in these files is thru using file IDs and byte offsets from beginning of the file.

In CC3100, the main functions handling the non-volatile memory are:

_i32	sl_FsOpen	(_u8 *pFileName, _u32 AccessModeAndMaxSize, _u32 *pToken, _i32 *pFileHandle)
_i16	sl_FsClose	(_i32 FileHdl, _u8* pCertificateFileName, _u8* pSignature, _u32 SignatureLen)
_i32	sl_FsRead	(_i32 FileHdl, _u32 Offset, _u8* pData, _u32 Len)

_i32	sl_FsWrite	(_i32 FileHdl,_u32 Offset,_u8* pData,_u32 Len)
_i16	sl_FsDel	(_u8 *pFileName,_u32 Token)
_i16	sl_FsGetInfo	(_u8 *pFileName,_u32 Token,SIFsFileInfo_t* pFsFileInfo)

The general concept of these API consists of files stored in SFlash, approaching data stored in these files is thru using file names and byte offset from beginning of the file.

In addition, there is an optional layer of security, which is reflected by the use of security tokens and certificates.

3.3.4 Scanning

When referring to WLAN scanning process, there are three main functionalities that should be addressed.

- Scan parameters configuration

– CC3000

long **wlan_ioctl_set_scan_params** (unsigned long uiEnable, unsigned long uiMinDwellTime, unsigned long uiMaxDwellTime, unsigned long uiNumOfProbeRequests, unsigned long uiChannelMask, long iRSSIThreshold, unsigned long uiSNRThreshold, unsigned long uiDefaultTxPower, unsigned long *aiIntervalList)

– CC3100

int **sl_WlanPolicySet** (SL_POLICY_SCAN,SL_SCAN_ENABLE, (unsigned char *)&intervalInSeconds,sizeof(intervalInSeconds))

- Starting a scan operation

– CC3000

Scans are being enabled automatically by the device whenever required. In addition on every call to *wlan_ioctl_set_scan_params()*, scan operation is kicked started.

– CC3100

Scans are being enabled automatically by the device whenever required. In addition on every call **sl_WlanPolicySet**(SL_POLICY_SCAN,SL_SCAN_ENABLE, (_u8 *)&intervalInSeconds,sizeof(intervalInSeconds)), scan operation is kicked started.

- Retrieving scan results

– CC3000

long **wlan_ioctl_get_scan_results** (unsigned long ulScanTimeout, unsigned char *ucResults)

The scan results are returned one by one, so continue reads to this function are required till no results are retrieved (ucResults is NULL).

– CC3100

_i16 **sl_WlanGetNetworkList** (_u8 Index, _u8 Count, SI_WlanNetworkEntry_t *pEntries)

The scan results are returned in a list of up to 20 results, it is possible to retrieve more than 20 results by calling this API again.

3.3.5 Smart Config

The basic principles of smart configuration process remained similar in CC3100, however, the APIs are different due to improved functionality and new features introduced with CC3100.

NOTE: CC3100 Smart configuration algorithm has many improvements, both from features stand point (enhanced security, multi group support, multi field configurations) and from performance. However, CC3100 algorithm has full backward support for CC3000 smart configuration algorithm, but not the other way around.

The main functionalities used for smart configuration are:

- Start smart config
 - CC3000

long	wlan_smart_	(unsigned long algoEncryptedFlag)
	config_start	
 - CC3100

_i16	sl_WlanSmartConfigStart	(const _u32	groupIdBitmask,
		const _u8	cipher,
		const _u8	publicKeyLen,
		const _u8*	group1KeyLen,
		const _u8*	group2KeyLen,
		const _u8*	publicKey,
		const _u8*	group1Key,
			group1Key,

The CC3000 is able to run smartconfig in secure mode or open mode, which is supported in CC3100 by using the cipher parameter. However, CC3100 adds much more capabilities of configuring several group of devices in parallel each with its' own encryption. For more information, see the [CC31xx Programmer's Guide wiki](#). Upon completion of the smart config process, both devices will return asynchronous events signaling on the process completion.

- Stop smart config
 - CC3000

long	wlan_smart_	(void)
	config_stop	
 - CC3100

_i16	sl_WlanSmart	(void)
	ConfigStop	
- Handle smart config encryption
 - CC3000

long	wlan_smart_	(void)
	config_process	
 - CC3100

Not required. Unlike CC3000 in which the encryption is done on the host processor side. CC3100 has a HW encryption engine which is handling the smart config process in the chip itself.
- Set prefix
 - CC3000

long	wlan_smart_	wlan_smart_config_set_prefix
	config_set_prefix	
 - CC3100

Not required. This is an optional feature for CC3000, which is obsolete and not required for CC3100.

3.3.6 Mask Asynchronous Events

Masking asynchronous events on the CC31xx has been simplified dramatically versus the CC3000.

- CC3000
int **wlan_set_event_mask()**
- CC3100
_i16 **sl_EventMaskSet** (_u8 EventClass , _u32 Mask)

Asynchronous events are masked using defines within the user.h file. To handle this API removal in your application code, simply remove all calls of `wlan_set_event_mask()` and set your desired asynchronous event handlers in user.h. The different event groups that the user can register to are listed below; more information can be found in the user.h file.

- void `sl_GeneralEvtHdlr`(SIDeviceEvent_t *pSIDeviceEvent)
- void `sl_WlanEvtHdlr`(SIWlanEvent_t *pSIWlanEvent)
- void `sl_NetAppEvtHdlr`(SINetAppEvent_t *pSINetApp)
- void `sl_SockEvtHdlr`(SISockEvent_t *pSISockEvent)
- void `sl_HttpServerCallback`(SIHttpServerEvent_t *pSIHttpServerEvent, SIHttpServerResponse_t *pSIHttpServerResponse)

3.3.7 Inactivity Timeout

All of the socket related operations can be configured using `sl_SetSockOpt()` API. One example is setting the socket inactivity timeout which, in CC3000, was configured using proprietary API.

- CC3000
INT32 **netapp_timeout_values**(UINT32 *aucDHCP, UINT32 *aucARP, UINT32 *aucKeepalive, UINT32 *aucInactivity)
- CC3100
sl_SetSockOpt(SockID, SL_SOL_SOCKET, **SL_SO_RCVTIMEO**, (_u8 *)&timeVal, sizeof(timeVal));

3.3.8 Ping

- Starting a ping
 - CC3000
long **netapp_ping_send** (unsigned long *ip, unsigned long ulPingAttempts, unsigned long ulPingSize, unsigned long ulPingTimeout)
 - CC3100
_i16 **sl_NetAppPingStart**(SIPingStartCommand_t* pPingParams, _u8 family, SIPingReport_t *pReport, const P_SL_DEV_PING_CALLBACK pPingCallback)

NOTE: CC3100 ping API can be also called in “blocking mode” if the callback is set to NULL.

- Stopping an ongoing ping process
 - CC3000
void **netapp_ping_stop** ()
 - CC3100
Same as starting a ping, but with the destination IP address set to 0.

- Retrieving the ping report
 - CC3000
The CC3000 API, *netapp_ping_report()* is called to request the ping status. With the CC3000, the *netapp_ping_report()* function triggers the CC3000 to send the asynchronous event HCI_EVNT_WLAN_ASYNC_PING_REPORT. This event carries the report within the struct *netapp_pingreport_args_t*.
 - CC3100
In the CC3100 equivalent, *sl_PingReport*, the ping report is no longer sent with an asynchronous event. Instead, the function passes the report output structure pointer as a parameter. The new structure is redefined in *netapp.h*, and the format and typecasts have changed from the CC3000.

CC3000	CC31xx
<pre>typedef struct _netapp_pingreport_args{ unsigned long packets_sent; unsigned long packets_received; unsigned long min_round_time; unsigned long max_round_time; unsigned long avg_round_time; } netapp_pingreport_args_t;</pre>	<pre>CC3000 CC31xx typedef struct { _u32 PacketsSent; _u32 PacketsReceived; _u16 MinRoundTime; _u16 MaxRoundTime; _u16 AvgRoundTime; _u32 TestTime; }slPingReport_t</pre>

3.4 Obsolete CC3000 APIs and Features

- Open connection policy
 - CC3000 supported a connection policy in which the device connected automatically to any non-secured AP. This mode was found to be not useful without automatic validation of internet access.
 - CC3100 is not supporting this mode
- Arp flush - long *netapp_arp_flush ()*
 - CC3100 doesn't support this API. The networking sub-system handles the internal ARP table automatically
- WLAN status - long *wlan_ioctl_statusget()*
 - CC3000 is able to report on the internal status of the WLAN connection: WLAN_STATUS_DISCONNECTED, WLAN_STATUS_SCANNING, STATUS_CONNECTING or WLAN_STATUS_CONNECTED
 - CC3100 doesn't support this exact status report.
It has a different device status API - *sl_DevGet()*, which reports if the Wi-Fi subsystem is connected or disconnected, in smartconfig mode and on any dropped asynchronous event.
- Timeout values
 - CC3000 supports an API, *netapp_timeout_values ()*, which enables the option to configure the following internal timeouts in the networking sub-system:
 - DHCP
 - ARP
 - KeepAlive ⁽¹⁾
 - CC3100 does not support this API, since it has strong effects on the system stability and power consumption. The CC3100 network sub-system handles these timeouts automatically.

⁽¹⁾ KeepAlive timeout is a roadmap item that will be available in CC3100.

4 References

- CC3100 Programmers Guide: http://processors.wiki.ti.com/index.php/CC31xx_Programmers_Guide
- CC3100 Host Interface User's Guide: http://processors.wiki.ti.com/index.php/CC31xx_Host_Interface
- SimpleLink Wi-Fi Programmers User's Guide:
http://processors.wiki.ti.com/index.php/CC31xx_Programmers_Guide
- CC31xx Programmer's Guide: http://processors.wiki.ti.com/index.php/CC31xx_Programmers_Guide

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products

Audio	www.ti.com/audio
Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
OMAP Applications Processors	www.ti.com/omap
Wireless Connectivity	www.ti.com/wirelessconnectivity

Applications

Automotive and Transportation	www.ti.com/automotive
Communications and Telecom	www.ti.com/communications
Computers and Peripherals	www.ti.com/computers
Consumer Electronics	www.ti.com/consumer-apps
Energy and Lighting	www.ti.com/energy
Industrial	www.ti.com/industrial
Medical	www.ti.com/medical
Security	www.ti.com/security
Space, Avionics and Defense	www.ti.com/space-avionics-defense
Video and Imaging	www.ti.com/video

TI E2E Community

e2e.ti.com