16/05/2014

CC3000 driver notes

CC3000 is a WiFi module by TI (see http://processors.wiki.ti.com/index.php/CC3000) providing a (supposedly friendly) socket-style interface for implementing Internet connections from small-footprint applications running in microcontrollers. From our point of view, the module is disappointing. It comes with its own software driver, the idea being that the driver should not be difficult to port to "typical" operating systems used in the micro world. Unfortunately, the driver is completely useless from our point of view, because we don't do things that way. As a matter of fact, the driver illustrates how things should not be done. The really unfortunate bit is that the module's low-level functionality seems to be rather strongly inspired by the driver.

Having said that, let me quickly add that there is a way to make the device work with PicOS. Considering that it is pointless to try to be power frugal while a WiFi module is active, we can afford to do a bit of polling in the driver without which nothing can be accomplished with the device. The present PicOS driver makes it possible to connect to a TCP or UDP socket and exchange packets as reliably as generally possible over a WiFi Internet connection (or at least it appears so based on my rather superficial tests so far).

The device does provide interrupts to signal some events. Those events (formal interrupts) are triggered in a messy way whereby some of them are responses to previously issued requests (these events are called "solicited") and some others can be raised spontaneously (asynchronously from the viewpoint of the driver) – those are called "unsolicited". Generally, when you issue a request (a command) to the module, the formal acceptance (sometimes it is in fact the completion) of the request will be marked by some specific (corresponding) synchronous event. The problem is that you have to know exactly what to expect and, what is truly disastrous, you cannot expect anything else until the command cycle runs to its prescribed completion. The design has the flavor of a student project where a sane and sound concept has been tweaked in a messy way to accomplish some momentary goal, which probably was to match the module's behavior to the expectations of the driver implementing a strictly sequential set of operations. The objective of the TI (reference) driver was to allow a sequential program to reference functions looking like standard BSD operations on sockets. The possible coexistence of that program with other processes or activities in the system was of no concern to the designers. This illustrates the kind of philosophy one would expect from the forthcoming generation of microprogramming experts educated on arduinos (where one cannot tell [or rather doesn't care about] the difference between an interrupt/event and a condition).

For illustration, to receive a packet, you have to issue a *receive* command. In a normal situation, the arrival of a packet would be signaled by an event (interrupt) to which the driver would respond by reading the data from the module into some buffer. While waiting for a packet to receive, the driver would be free to use the device for other tasks considered normal by most people, e.g., sending a packet out. In this case, once you have issued *receive*, the module becomes essentially blocked until a packet is in fact received! A reception event will be triggered all right when that happens (as one would expect), but you cannot do anything while waiting for that event! For example, if you decide that you have waited long enough without the reception having materialized, you may be tempted to ignore the outstanding *receive* command and issue another one. No way! When you do that, the module will get into a weird state from which the only way out is through a reset.

With this approach it would have been impossible to implement any kind of operation (except for some truly stupid, blocking, sequential scenarios too brain damaged even for arduino), if not for just one life-saving option: there is an operation mimicking BSD *select*. So the messy way to implement (safe) reception is this:

1. do *select* specifying a timeout
2. when the select says that that there is data available, issue *receive* (now you know that the operation will be responded to right away)
3. read the received packet
4. after a timeout do other things (like checking for an outgoing packet and transmitting it) and issue *select* again

Thus the driver must necessarily poll the module in a tight loop to be able to transmit and receive without rigidly ordering these operations in a blocking/stalling fashion. It looks like implementing an event driven driver having for the starting point a sequential program using a subset of BSD operations on sockets. Even worse, because the standard operations allow for non-blocking socket read. No such luck here. The select command is in fact one of the most complex operations for the device, so the polling cycle is far from economical.

Compilation options

The symbolic constant **`WIFI_OPTIONS`** is a collection of flags selecting the high-level modes of the driver. Two such flags are used at present:

`WIFI_OPTION_TCP` **(0x01)**

> Selects the TCP connection mode, rather than the (default) UDP mode. A connection functionally appears like a BSD socket. Two types of such sockets are available: UDP and TCP. We probably prefer UDP connections, because of their inherently packetized nature and much simpler maintenance rules.

`WIFI_OPTION_NETID` **(0x02)**

> When set, indicates that the Network Id field of packets (the first two bytes) will be interpreted in the same way as for packets transmitted by our standard RF modules. That is, a received packet whose Network Id doesn't match the Network Id assigned to the PHY will be ignored. For an outgoing packet, the Network Id will be automatically inserted by the driver into the first two bytes. The standard interpretations of the special Id's 0x0000 and 0xFFFF will also apply.

By default, there is no Network Id, i.e., all bytes of a packet are treated as payload bytes. This makes sense for point-to-point communication, e.g., providing a wireless OSS link. Effectively the same behaviour is selected with the NETID option compiled in by setting the Network Id to 0xFFFF.

The length of the polling interval (for receiving packets) is determined by two symbolic constants, **`CC3000_POLLINT_MIN`** and **`CC3000_POLLINT_MAX`** whose default values are 1 and 15, respectively. They are translated into seconds by multiplying them by 0.065536 (the unit of polling interval is 64K microseconds). The default minimum interval is one unit, i.e., 64K microseconds, corresponding to about 15.28 polls per second, and the maximum is 15 units yielding about one poll per second. Any activity (transmission/reception) resets the interval to the minimum; then after every idle cycle, the delay increases linearly by one unit (64K microseconds) until it reaches the maximum (where it stays put).

Note that the polling interval also determines the amount of waiting time before a transmission. Recall that once the driver starts the polling cycle, it can do nothing until the select timer expires. Thus, the

average delay for transmitting a spontaneous packet is about 0.5s. But two (or more) packets in a row will be transmitted back to back, if the second one is queued before the first one is transmitted.

By default, the driver issues keep-alive packets at regular intervals. If there is no activity (transmission/reception) the connection will die (somewhere within the module) in about two minutes. This happens without a notice, but with a stubborn regularity suggesting its being some kind of feature. Keep-alive packets seem to solve the problem. They may be generally a good idea, especially for UDP communication.

The constant **WIFI_KAL_INTERVAL** determines the frequency of keep-alive packets in the units of polling interval increments (i.e., 0.065536s). In fact, the actual frequency will be slightly lower (the interval between keep-alive packets will be slightly higher), because the time for issuing the polling commands counts extra. The default value of the constant is 240 amounting to about 16 seconds.

If the NETID option (see above) is off, a keep-alive packet consists of four zero bytes. With the option compiled in, the packet consists of the Network Id followed by the Node Id (i.e., the lower word of the 32-bit Host Id). Its length is four bytes in both cases.

Note that the time until the next keep-alive packet is reset after every packet transmission. In particular, a continuous supply of outgoing packets showing up faster than the keep-alive interval will result in no keep-alive packets being sent at all.

If **WIFI_KAL_INTERVAL** is defined as zero, no keep-alive packets are sent automatically. This is OK, e.g., is the user program makes sure to send enough regular packets to keep the connection up.

Controlling the driver

The module is visible to the system as a standard PHY. Here is a sample initialization sequence (copied form App/TEST/CC3000/app.cc):

```
#include "sysio.h"
#include "plug_null.h"
#include "phys_cc3000.h"
...
...
cc3000_wlan_params_t WP = {
      CC3000_POLICY_DONT,
      CC3000_WLAN_SEC_WPA,
      7,
      "OLSONETbacabacaba"
};
cc3000_server_params_t SP = { { 192, 168, 1, 16 }, 2234 };
      ...
      ...
      phys_cc3000 (0, maxplen, &SP, &WP);
      tcv_plug (0, &plug_null);
      SFD = tcv_open (NONE, 0, 0);
      ...
      ...
```

The first two arguments of phys_cc3000 are standard; in particular, the second argument specifies the maximum length of a packet payload. There is no CRC (and no room should be reserved in the packet for any extra attributes). If the NETID option is compiled in, the Network Id formally counts to the payload (as for other RF modules).

The driver reserves (malloc's) a buffer for receiving packets. The length of that buffer equals maxplen + 30 bytes.

The last two arguments of phys_cc3000 are special and provide for identifying the network and the server. Here is how the respective structures are defined (in phys_cc3000.h):

```
typedef struct {
      byte  policy;
      byte  security_mode;
      byte  ssid_length;
      char  ssid_n_key [33];
} cc3000_wlan_params_t;

typedef struct {
      byte ip [4];
      word port;
} cc3000_server_params_t;
```

The first structure describes the network. Note that the actual length of the last attribute can be less or more: it is supposed to be a string terminated by a NULL byte.

The first attribute, policy, stands for the so-called connection policy and can be one of these:

CC3000_POLICY_DIRECT (0)

Connect to the specified network (and make it the standard policy).

CC3000_POLICY_PROFILE (1)

Connect to the specified network, store the network in a profile, and make it the standard policy to connect to one of the networks stored in the profile (whichever is available). Profiles can be also created and stored in the module using a smartphone configuration tool. I haven't tried it yet. If needed, the driver can be (relatively easily) extended by a function to put the module into a state whereby it will accept profiles over RF from a smartphone.

CC3000_POLICY_LAST (2)

Connect to the specified network and make it the standard policy to connect to the last network connected to.

CC3000_POLICY_ANY (3)

Connect to the specified network, and make it the standard policy to connect to any network you can connect to.

CC3000_POLICY_OLD (4)

Connect to the specified network and do not change the standard connection policy.

CC3000_POLICY_DONT (5)

Connect automatically according to the current (last set) connection policy. The remaining parameters are ignored.

Note that I haven't tested everything yet (and there are probably things I don't understand). All I have tested is connecting to a specific network (my home network), setting the policy to LAST, and then using DONT to connect to the same network without having to specify its parameters. Note that in the last case (e.g., if you have preset the module to always connect to the same network, or one from the networks stored in the profile), the data structure may consist of a single byte (the remaining attributes are never looked up).

If the network parameters are needed, then the second attribute, security_mode, must be one of these:

CC3000_WLAN_SEC_UNSEC (0)

```
CC3000_WLAN_SEC_WEP          (1)
CC3000_WLAN_SEC_WPA          (2)
CC3000_WLAN_SEC_WPA2         (3)
```

to select the security mode of the network (the first value stands for an unsecured network).

The network's SSID is combined into a single string with the password (if a password is needed). Attribute ssid_length tells the length of the SSID part in the string. Thus, in the above example, the SSID is “OLSONET” and the password is “bacabacaba”.

The data structure describing the server consists of the IP address and the port number in the most straightforward way. Note that no connection occurs immediately. The PHY starts in the OFF state with the module powered down. To turn it on, execute:

```
tcv_control (SFD, PHYSOPT_ON, NULL);
```

In contrast to a typical RF module, the operation of switching CC3000 on and off is rather complicated and may take time. Also, it may quite legitimately fail, e.g., if the network is unavailable of (for the TCP mode) the server doesn't respond. When you switch the PHY on, the driver will try to connect to the network (according to the policy). Then, if that stage succeeds, it will create a socket and, for the TCP mode, try to connect to the server. The connection is considered established, if the connection to the server succeeds. For the UDP mode, there is no server connection: individual datagrams are (blindly) sent to the specified IP address and port. In that case, the startup succeeds as soon as the socket has been created.

The individual stages of the connection procedure are guarded by timeouts and retried periodically until the procedure formally succeeds. Once the PHY is turned on, the driver will keep trying until it succeeds, or until the PHY is turned off. When you turn the PHY off (and it has been on before), the driver will try to gracefully close the socket and go through some prescribed drill. Owing to the complexity and the somewhat special nature of WiFi communication, the device is not turned on (powered up) automatically when a packet is put into the output queue (by tcv_wnp), nor is the output queue emptied before formally turning off the module. Note that when the module gets disassociated from the network or from the server (which may happen asynchronously for a completely external reason), there is no way to sensibly expedite the packets waiting in the output queue. The only way to make sense out of the situation is to let the user program detect it and decide what to do (empty the packet queue manually via explicit operations of VNETI, wait with those packets queued until reconnection, and so on).

To know the actual (and possibly momentary) state of the PHY the user program can issue a status request, like this:

```
stat = tcv_control (SFD, PHYSOPT_STATUS, (address)&dstat);
```

where the argument points to this data structure:

```
typedef struct {
      byte  dstate;
      byte  freebuffers;
      word  mkalcnt;
      word  dkalcnt;
} cc3000_phy_status_t;
```

whose contents are filled by the function. Attribute dstate shows the current state of the driver which can be one of these:

```
#define     CC3000_STATE_DEAD 0          // Disabled, switched off
#define     CC3000_STATE_INIT 1          // Initializing
#define     CC3000_STATE_WBIN 2          // Waiting for buffer info
```

```
#define     CC3000_STATE_POLI 3          // Setting connection policy
#define     CC3000_STATE_RCON 4          // Connecting to AP
#define     CC3000_STATE_APCN 5          // Connected to AP
#define     CC3000_STATE_SOKA 6          // Socket available
#define     CC3000_STATE_CSNG 7          // Closing server connection
#define     CC3000_STATE_ESTB 8          // Connection established
#define     CC3000_STATE_RECV 9          // Receive poll
#define     CC3000_STATE_READ 10         // Read received packet
#define     CC3000_STATE_SENT 11         // Packet sent
```

The connection is healthy when dstate >= **CC3000_STATE_ESTB** (it makes sense to try to send packets out). Any other value means that the connection is being established or the PHY is off. In the latter case, dstate will be **CC3000_STATE_DEAD**. Note, however, that that state will not be assumed immediately when you execute **tcv_control (SFD, PHYSOPT_OFF, NULL)**, although it is guaranteed to settle there after some finite time. On the other hand, when you turn the module on, it will reach one of the last four connected states only if the network is available and (for the TCP mode) the server is willing to receive TCP connections.

Attribute freebuffers tells the number of free buffers of the device available for sending packets out. The maximum is 6 (for the present device version). A transmission occupies one buffer until the packet is considered done and can be discarded from the device. Buffer management is handled automatically and you need not worry about it.

Attribute mkalcnt tells the number of keep-alive events generated by the device (don't confuse them with the driver's keep-alive packets). The device periodically (about every 15 seconds) triggers an (unsolicited) event to tell the driver that it is operational.

Attribute dkalcnt tells the number of elapsed units of polling time. Each time a polling cycle completes, this value is incremented by the last setting of the select timer. So this value increments rather fast and wraps around quite often. This attribute is only set if **WIFI_KAL_INTERVAL** is nonzero.

The function value returned by the status query is the standard simple on/off value, i.e., 2 for off and 3 for on, reflecting the formal on/off status of the PHY. This value is updated immediately after formally turning the PHY on or off. Let me repeat once again that the simple binary status of the device does not indicate whether the device is in fact operational, only its desired target state.

In addition to the control calls mentioned above, the driver implements these control calls:

PHYSOPT_GETMAXPL

> returning the maximum packet length that was specified as the second argument for phys_cc3000

PHYSOPT_SETSID and **PHYSOPT_GETID**

> to set and read the Network Id; these calls are only available if the NETID option has been compiled in.

We may add more calls later, e.g., if a more precise control of the device's state is needed by the application.

Final notes

The device offers more possibilities than we take advantage of in the present version of the driver. It is not much more from the practical point of view, but, for example, it is possible to:

- have multiple sessions (multiple sockets) open simultaneously
- set up a TCP service

- use static parameters for the IP address, gateway, mask, etc., instead of getting them via DHCP

I don't think these options appear extremely attractive at this point, but ... one never knows.

One thing that I may have to do (if we decide to use the device for production) is patching, i.e., firmware upgrades, which are a bit complicated. My assumption was that we would need (expect from the device) the most basic functionality which (normally) should be fine since version 1.0. Maintaining firmware patches will certainly add burden to configuration and distribution of modules to target nodes, so we may want to avoid that operation, if the patches prove practically useless. The present patch level of the (not upgraded) device is 1.10 while the maximum available from TI is 1.12. Looking at the release notes, I see no interesting changes between the versions, certainly no changes that would help the most fundamental problems that the driver has to cope with.