Pawel Gburzynski

DecaWave DW1000

driver

August 30, 2015

Preamble

DecaWave DW1000 is a UWB radio module with built in precise packet timing capabilities, intended for indoor positioning. The module is described in detail in these documents:

DW1000 User Manual
DW1000 Data Sheet, version 2.04
DWM1000 Data Sheet, version 1.01

available from DecaWave's web site at: http://www.decawave.com/

This document sketches the practical functionality (interface) of the present PicOS driver for the module. There is no need to refer to the above documents, unless you want to understand in depth what is going on inside the module (or in the driver). Needless to say, the present driver is preliminary and mostly intended for experiments (and this document is intended as a starting point for discussion).

The assumptions

At present, we focus solely on the location tracking capabilities of the module. While the device can be used for straightforward RF communication (say as a replacement for CC1100), and the present driver can be quite easily (almost trivially) extended in that direction, the option doesn't immediately appear attractive to me, because the device's power consumption is one order of magnitude (in fact almost exactly 10 times) higher than for CC1100 (say with the receiver on) while its RF range appears unimpressive, especially when LOS (line of sight) communication is difficult. On the other hand, DW1000 has a number of interesting features reducing the length of duty cycles in some communication scenarios (plus some lower power listening modes similar to WOR in CC1100). Those scenarios are related to IEEE 802.15.4 (ZigBee) and (sort of) assume (or at least facilitate) point-to-point communication, so it is not immediately clear how attractive the module can be for our applications (for general communication).

Consequently, the present driver **does not** make the device available as a PicOS PHY. Instead, it implements a sensor-actuator pair. To see how the two conceptually simple interfaces can represent the device for our purpose, we have to learn a bit about location tracking with DW1000.

The principles of location tracking with DW1000

The extra features of DW1000 (compared to other radios, e.g., CC1100) boil down to a configuration of precise timers that can be used to estimate a packet's Time of Flight (TOF), which can be translated into the distance between the sender and the receiver. Having three or more distances between a tracked Tag and a number of Pegs (whose locations are known) will let us estimate the location of the Tag.

Another option is to follow the TDOA (Time Difference of Arrival) approach. This one simplifies the operation of the Tag (i.e., the tracked device) and reduces its energy budget, while complicating the operation of the infrastructure of Pegs.

Regarding the first (TOF) option, suppose that some two nodes, call them A and B, want to determine the distance between themselves. The nodes are equipped with precise timers. The TOF option does not require that their timers be synchronized.

Thus, A sends a packet addressed to B. A marks the packet's transmission time as TSP, B marks its reception time as TRP. Having received the packet, B responds with its own packet transmitted at TSR > TRP which A receives at TRR > TSP.[1] Then TOF = (TRR – TSP) – (TSR – TRP) is twice the propagation distance between A and B. When A performs this kind of exchange with several (three or four) Pegs (whose locations are known), A's location can be estimated by triangulation.

The TDOA option assumes that the timers at Pegs (not at Tags)[2] are synchronized, which means that the entity running the location tracking algorithm, i.e., the OSS in our case, can meaningfully relate the indications of timers at different Pegs to a centralized (global) notion of time. Note that this does not mean that the timers at different Pegs actually show the same time, only that their indications are centrally correlated.

In this approach, the Tag sends (broadcasts) simple beacon-like packets. Different Pegs receiving those packets receive them at different times depending on the difference in their positions with respect to the Tag. Owing to the fact that their timers are globally coordinated, the OSS can calculate those differences. Then, by solving some equations, the OSS can estimate the Tag's location (assuming that the Peg's locations are known).

The TOF option is relatively easy to implement within an ad-hoc network of Pegs, because the communication among the Pegs can be sloppy. The complicated bit is the requisite exchange of packets (handshake) between the tracked Tag and some specific Pegs. The Tag must receive a packet back from a Peg, then it must communicate the result to the Peg again, so it can be forwarded to the OSS. Thus, the Tag gets involved in a series of non-trivial packet exchanges, which complicates the operation and strains the energy budget. A single location tracking event involves packet exchanges with several (at least three, preferably four or more) Pegs whose identity must in some way become known to the Tag at some point. This is not necessarily a fatal flaw of the scheme, especially in panic button applications (where the location tracking events are sparse), or even in personal wearable Tags (which can be recharged every once in a while), or in equipment tracking applications, where the tracking events can occur a few times per day, can be triggered by (not too frequent) motion, and so on.

The TDOA option basically requires interconnecting the Pegs with a wired network providing clock synchronization pulses. The network must be calibrated to account for the (fixed) propagation time of synchronizing signals in the cables. But then the Tags can simply emit trivial non-orchestrated broadcast beacons. They need not exchange any information with the Pegs (at least not over the energy-expensive DecaWave radios).

The high-level functionality of the present driver

In our present driver we only implement the TOF option. The other option requires a more serious commitment and has to wait until we are convinced that the device is going to be useful. One can say with some confidence that if the experiments with TOF tracking are successful, the TDOA option will be definitely worthy the additional investment.

The nodes carry out a handshake of sorts with the Tag (call it node A) going first. The Tag sends the first packet of the handshake which I call POLL. The packet needs no particular structure or content. The only important thing is that the other node, i.e., the Peg (call it B) should recognize it as a POLL packet and should also learn the identity of the initiating Tag. Any packet exchanged within the handshake is a *ranging* packet (it is marked in a special way), which means that:

[1] Note that we only compare timers belonging to the same node.
[2] In fact, Tags need no timers at all with this approach.

- the sending device precisely determines the time of its departure
- the receiving device precisely determines the time of its arrival

Technically, there is a special marker symbol within the packet's header whose transmission is used as the relatively sharp reference point for the timing (the device executes a rather complicated [microcoded] algorithm aimed at reducing the uncertainty in the timing of that symbol). As we said earlier, the timers are **local**, so the scheme does not assume that Pegs synchronize their clocks. As is clear from the formulas below, the TOF estimation is based on differences in the indications of local timers only.

When the initiating Tag A sends the POLL packet, it marks the time of its departure (using its local clock) as TSP. The Peg B receiving the POLL packet marks its arrival time (using its local clock) as TRP.

Having received the POLL packet, B responds with a RESP packet addressed to A. Again, the packet needs no special content[3] other than being identifiable as a RESP packet sent by B and addressed to A. B marks the departure time of the RESP packet as TSR, and A marks its reception time as TRR.

Having received the RESP packet, A sends the final packet of the handshake to B. I call this packet FIN. A inserts into the FIN packet (in contrast to the previous two packets, this one does have a payload) three timer values: TSP, TRR, and TSF, where TSF is the departure time of FIN. The role of FIN is:

1. to communicate the values of A's timers to B (the ones B doesn't know), so it can compute the time of flight (or rather pass the information to the OSS for the computation)
2. to provide for one more estimate of the time of flight (the three packets can be viewed as two exchanges)

The tricky bit is that A has to know the transmission time of FIN before sending the packet (so it can insert that time into the packet's payload). To make that possible, the device has a feature whereby it can be instructed to dispatch the next outgoing packet in such a way that the packet's marker symbol is transmitted precisely at a preset (advance) timer value. Thus, the driver calculates a safe (delayed) transmission time of FIN, based on current time + some processing interval, sets TSF to that time, inserts TSF into FIN (along with the other two timers) and tells the device to transmit the packet exactly at TSF. The actual action is a bit more complicated, but it pretty much amounts to this.

When the FIN packet gets to B, the Peg has enough information to calculate two estimates of the distance between the two nodes (note that it is the Peg that does the estimation). The node has these six time stamps:

TSP – departure time of POLL at A

TRP – arrival time of POLL at B

TSR – departure time of RESP at B

TRR – arrival time of RESP at A

TSF – departure time of FIN at A

TRF – arrival time of FIN at B

Now, the two estimates of TOF are calculated as:

[3] In fact, those packets, i.e. POLL and RESP, have no payload and consist exclusively of (some variants of) 802.15.4 headers.

TOF1 = ((TRR – TSP) – (TSR – TRP)) / 2

and:

TOF2 = ((TRF – TSR) – (TSF – TRR)) / 2

Then the node can take the average of the two values:

TOF = (TOF1 + TOF2) / 2

as the single estimate of the time of flight.

Technically, the timers are 40-bit (5-byte) values and the arithmetic is a bit involved for MSP430, so it makes sense to pass the timers to the OSS for calculations (and this is what the present driver does). This makes even more sense considering that (rather unavoidably) some Tag/Peg-specific adjustments (calibrations) will have to be applied to them. While the chip incorporates internal adjustments, i.e., provides for storing pre-calibrated antenna delays (and a few other things) in the so-called one-time programmable (OTP) memory, it seems much easier (and more cost effective) to do the calibration in the OSS, which will simplify both the firmware as well as the calibration procedure (which can be carried out during production and using production firmware).

Note that the last leg of the handshake (i.e., the FIN packet) should arrive at a Peg, such that the relevant information can be directly passed to the OSS. Consequently, POLL packets should be issued by Tags. It can hardly be any other way, because keeping the receiver open in a Tag for any nontrivial amount of time is out of the question.

The timer resolution is formally about 1/63897600000 s which roughly translates into about 5 mm at the speed of light. This is the granularity, not the accuracy. There is a healthy bit of guessing involved in the timing of the marker symbols, and this seems to be the primary internal source of inaccuracies. Moreover, there are external sources: signals can get reflected, and so on. They suggest procedures and heuristics to detect and discard such signals (basically by looking into the very rich set of various quality indicators). There is no RSSI, but they say how it can be computed from those indicators in a rather messy sort of way. Notably, my preliminary experiments carried out in a somewhat idealistic environment, seem to confirm the manufacturer's claims that (with some calibration) the accuracy of distance estimation is within the 10 cm error margin.

The wrap around time of the 40-bit timer is about 17.2 seconds. Owing to the fact that a handshake is carried out without unnecessary (non-deterministic) delays (like LBT, or other collision avoidance schemes), its duration (if successful) is highly predictable and, regardless of the setting of RF parameters, always below 12 ms (counting from the transmission of POLL until the reception of FIN. When we strip the 40-bit timer of its most significant byte (to make it look like a simple and convenient 4-byte longword), its wrap around time becomes ca. 67 ms which, in confrontation with handshake duration, is still OK for ranging. I have included a compilation option in the driver to use 4-byte timers instead of the 5-byte ones. This option is probably preferred.

The interface

The (potentially relevant) compilation options of the driver are encoded in the symbolic constant DW1000_OPTIONS which is a collection of binary flags. The default value of that constant is 0 (no flags set). Here is the list:

0x01 DW1000_OPT_DEBUG

The option includes some internal tests and diag messages that may be useful for debugging. It should not be selected for production (or realistic, production grade, timing tests).

0x02 DW1000_OPT_SHORT_STAMPS

The option selects short (4-byte, as opposed to 5-byte) time stamps. Short stamps reduce the size of the FIN packet as well as the data structure representing the sensor value (see below), and also simplify some processing in the driver (not a lot).

0x04 DW1000_OPT_NO_ANT_DELAY

The option removes from the driver the code for internally adjusting the timer values by preset antenna delays. Such delays can be pre-calibrated and stored in the module's OTP (one-time programmable) memory. This option makes sense, if we prefer to store the calibration data in the OSS, which we most likely do.

0x08 DW1000_OPT_MAX_TX_POWER

The option ignores the recommended transmission power settings (which are different for different channels/data rates) and uses the maximum power for all RF options. Also, it turns off the so-called “smart power” feature which (sometimes) causes different portions of a single packet to be sent at different power levels in an attempt to comply with the regulations while delivering the maximum performance.

0x10 DW1000_OPT_DONT_SLEEP

The option causes that when the driver is playing a Tag's role (see below), it never puts the chip to sleep, thus draining much higher power, but also making the device responsive to commands, e.g., modifying its registers. This is useful for experimenting with different parameters (which can be reconfigured between the handshakes).

I am contemplating an option to replace the bunch of time stamps returned by the driver with a single-value TOF estimate. This is doable (and will simplify the interface), especially if:

- we use short (4-byte) time stamps
- we conclude that the (external) calibration procedure can work on the single TOF estimate and needs no individual time stamps; I am inclined to think that this is the case

Of course, the options can be eliminated when we get to a production version of the driver and preset them in the most suitable way.

Note that when DW1000_OPT_NO_ANT_DELAY is not selected, the driver assumes some default value for the antenna delay recommended by DecaWave. That value seems to work almost fine (with only a slight shift) for three of the four modules I have, but produces a large discrepancy for the fourth one.[4] That fourth device is generally problematic (I have had some problems with it, but it seems to work now [except for the discrepancy]), so I am not sure what to think. With DW1000_OPT_NO_ANT_DELAY set, the antenna delay is assumed to be zero, so the estimates need to be offset.

The device is started (initialized) with the following operation:

void dw1000_start (byte mode, byte role, word pan);

where:

mode is one of 8 RF configurations (frequency, preamble, channel, etc.) which I have copied from the reference driver by DecaWave

[4] The manual states that the antenna delay should be very stable for a given design, and the calibration generally should be done once per design, although individual calibration may still give somewhat better results. Perhaps one of our modules is some older version?

role — determines the node's role, i.e., which end of the handshake the node wants to play; 0 means Tag (node A, according to the procedure described above), 1 means Peg (node B)

pan — is basically the network Id (the so-called PAN in 802.15.4 parlance); nodes belonging to different networks will not talk to (handshake with) each other (similar to our Network Id)

Note: theoretically, I could use our Network Id for the PAN, but the driver cannot assume that another radio (like CC1100, where the Network Id is prominent) is configured into the system. On the other hand, I am using our standard Node Id (the lower word of host_id, which is always known for every node) as the node's (short) address.

To switch the device off execute this:

```
void dw1000_stop ();
```

When the device is started with role 1 (Peg), the receiver is immediately switched on (note that it drains about 160mA) and the device begins to listen for POLL packets. Completed handshakes are communicated to the praxis as (compound) values arriving on a sensor (see BOARDS/OLIMEX_CCRF_DW1000 for the way that sensor is configured). A single value is described by this data structure (defined in dw1000.h):

```
typedef struct {
        word tag;                               // Source
        byte tst [6*DW1000_TSTAMP_LEN];         // Six time stamps
        byte seq;                               // Sequence number
} dw1000_locdata_t;
```

where:

tag — is the Tag Id of the handshake (the node sending the initial POLL packet)

tst — is a byte array storing the six time stamps produced by the handshake and consisting of 6*5 = 30 bytes, if DW1000_USE_SHORT_STAMPS is 0 (DW1000_TSTAMP_LEN is then 5), or 24 bytes, if short stamps are selected at compilation. The byte offsets to the respective timers are represented by these constants (defined in dw1000.h):

```
#define DW1000_TSOFF_TSP (DW1000_TSTAMP_LEN * 0)
#define DW1000_TSOFF_TRR (DW1000_TSTAMP_LEN * 1)
#define DW1000_TSOFF_TSF (DW1000_TSTAMP_LEN * 2)
#define DW1000_TSOFF_TRP (DW1000_TSTAMP_LEN * 3)
#define DW1000_TSOFF_TSR (DW1000_TSTAMP_LEN * 4)
#define DW1000_TSOFF_TRF (DW1000_TSTAMP_LEN * 5)
```

seq — is the handshake's sequence number (a single-byte value) arriving in the initial POLL packet (see below)

To retrieve the handshakes, the praxis should run an FSM with a state organized like this:

```
...
dw1000_locdata_t        location;
…
```


```
fsm … {

        ...

        state EVENT:

                read_sensor (EVENT, SENSOR_RANGE, (address)&location);
                if (location.tag == 0) {
                        wait_sensor (SENSOR_RANGE, EVENT);
                        release;
                }

                ...
```

If the tag attribute of the sensor value is zero, it means that there is no new handshake value to see. The sensor generates an event whenever a new handshake is received. Then, the tag attribute of the returned structure is set to the originating Tag Id.

When the node's role is 0 (Tag), the device is put to sleep after startup (the current drain is of order nanoamps). To start a handshake, the praxis has to store a value in an actuator (see BOARDS/OLIMEX_CCRF_DW1000 for the way that actuator is configured). The value consists of a single byte specifying the sequence number of the POLL packet, e.g.,

```
…

byte seq;

        ...

        state POLL:

                write_actuator (POLL, ACTUATOR_RANGE, (address)&seq);

                ...
```

If the driver is busy doing the previous handshake, the operation will block and auto-restart when ready.

A handshake is retried up to 4 times, if the RESP packet fails to arrive within 10 milliseconds after POLL. The Tag doesn't verify whether its FIN packet makes it to the Peg. Its end of the handshake completes with the departure of the FIN packet.

It is possible to modify and/or read device registers using these operations (note that you have to know what you are doing):

```
void dw1000_register_write (byte reg, word index, word length, byte *stuff);
void dw1000_register_read (byte reg, word index, word length, byte *stuff);
```

where reg is the register number, index is the offset (some registers are arrays of sorts), length is the number of bytes, and stuff is a pointer to data. Note that the device must not be sleeping for these operations to succeed, i.e., it must be on. In its Tag role, the device is sleeping permanently, except for the very brief (a few millisecond) handshakes so you have to compile the driver with DW1000_OPT_DONT_SLEEP, if you want to take advantage of the above operations in that role.

Comments on the energy budget

In the full listen mode, the device drains ca. 160 mA. This (plus whatever comparatively negligible extras are needed for the microcontroller) is the current drain for a Peg constantly listening for POLL packets from Tags. With CC1100 on (in its full listen

mode), the total drain is going to be about 180 mA (i.e., a lot compared to our typical devices).

From the viewpoint of a Tag, about this much current is needed for the entire duration of a (proper) handshake which consists of a short TX burst (at 140 mA), followed by listen (at 160 mA), followed by another short TX burst. The duration of TX bursts is practically negligible (besides, the current drain is almost the same, only slightly less for TX than RX).

Before starting a handshake, the Tag wakes up the chip. Fortunately, while sleeping, the device needs a negligible amount of current (about 200 nA). We can assume that the handshake cycle (as viewed by a Tag) consists of:

1. some initial delay at the current drain of 13-14 mA (the device getting ready)
2. 9 ms of the proper handshake at the current drain of 160 mA

This is when the payload data rate is 100 kbps (modes 0, 2, 4, 6). For the other data rate option, 6.8 Mbps (modes 1, 3, 5, 7), the handshake duration reduces to about 4 ms. For a Tag, this is the amount of time measured from the beginning of POLL transmission until the complete transmission of FIN. Optimistically, the first delay is just 1 ms extra, so it (probably) can be ignored.

Comments on accuracy, calibration, range ...

My experiments so far have been very superficial. Given a pair of nodes, one acting as Peg, the other as Tag, I have compared estimates to actual distances (measured with a tape). Here are a few numbers.

Node 1 is Peg, Node 2 is Tag. The nodes are located 160 cm apart (measured from the centers of the antennas). No obstacles between the nodes. The estimate comes up as 155.69 (the average of 15 tests, the differences being surprisingly low, below 5 cm). I have tried some other distances under the same circumstances. Here is the table:

distance (m)	estimate (m)	offset (m)
1.60	155.69	154.09
2.46	156.52	154.06
3.00	157.12	154.12

Note that the estimate is evidently offset by the antenna delay (which I don't calibrate out internally); however, once we account for it, the readings are well within the advertised 10 cm accuracy margin (even considering the rather poor quality of my distance measurement). When I put myself in front of the Tag's antenna, the estimate for the 3.00 m distance became 157.57, i.e., 45 cm more, which may suggest that obstacles do play a role.

When I replaced the Tag node with another specimen (Node Id 3), the estimate for 3.00 m distance was 157.11 (very close to that for Node 2). But for Node Id 4 (yet another specimen), it was 118.85, i.e., way off (I suspect it is a slightly different design). Accounting for the different offset, node 4 exhibited stable and correct indications.

The maximum range with the recommended power setting is about 10-12 m (line of sight in my home), i.e., rather low compared to CC1100. When I set the device at its maximum (apparently illegal) power, the range increased to (at least) 150 m in open field

(I tested it in front of my home, and this is how far I can go before hitting the woods).[5] With the recommended power setting, it is about 25-30 m. The signal passes through (soft) walls, but it doesn't pass through the ceiling in my home (unlike the ISM signal from CC1101), even at a short distance (directly above) and at the maximum power setting.

[5] Trees seem to block the signal.