Tutorial 2: A Sweet Hardware Link Layer with Enhanced Shockburst

Written by Brennen Ball, © 2007

Introduction

Well, I now have two tutorials in the can and you're still with me, so that's definitely a good thing! In this tutorial, we're going to explore the 24L01's integrated link layer, Enhanced Shockburst.

Don't Get Ahead of Yourself

Before we get started, this tutorial builds off of Tutorial 1. In fact, there are only about 5 lines of code or less per main file that are different. For this reason, it's important that you were able to get the code working in Tutorial 1 before you start with this one if you're planning on trying to get the code running. If you just want some background and aren't testing out the code, then continue on regardless.

Pretty much the only difference in this tutorial and Tutorial 1 is that, in this tutorial, we will be using Enhanced Shockburst to take advantage of the nRF24L01's hardware link layer. This allows us to let the 24L01 handle packet acknowledgement and retransmission automatically. It takes no extra hardware and almost no extra code, and gives us an even more robust link to work with than before! The only trade off here is that the link will be slightly slower, but as long as you don't need every ounce of bandwidth the 24L01 has to offer, Enhanced Shockburst is definitely the way to go.

Enhanced Shockburst in a Nutshell

You might be wondering exactly what Enhanced Shockburst is. It is essentially a system that is implemented in the hardware of the 24L01 that will automatically handle packet acknowledgements and retransmissions if a packet is lost. It relieves the user of having to take care of these arduous tasks in software when it is necessary to have a more robust link where a very high percentage of packets make it through.

If you are at all familiar with internet protocols, using Enhanced Shockburst is a lot like using TCP, where regular Shockburst is like using UDP. While Enhanced Shockburst doesn't handle setting up and closing a connection like TCP does (AKA sockets), it does give you some sort of semi-guarantee that your packet will get through. Similar to TCP, Enhanced Shockburst will try so many times to send a packet that is not acknowledged (also handled by Enhanced Shockburst at the receiver) before it just gives up and flags an interrupt.

There are a few registers associated with controlling Enhanced Shockburst. The EN_AA register handles enabling (1) and disabling (0) of auto-acknowledgements on a per-pipe basis. The ARC field of the SETUP_RETR register holds the number of retransmits you want to allow when a packet doesn't get acknowledged (a value of 0000 will disable auto-retransmit). The ARD field of the SETUP_RETR register will set how long you want to wait between retransmits (the lower the value, the less time between). The CONFIG register's MAX_RT interrupt enable bit will allow you to reflect the status of the MAX_RT interrupt on the IRQ pin. Also, the MAX_RT bit in the STATUS register will allow you to tell when the MAX_RT interrupt is active.

The setup of Enhanced Shockburst is very simple to get done. First, you need to make sure that at the TX device, your pipe 0 RX address (RX_ADDR_P0 register) is the same as your TX address (TX_ADDR register). This is because auto-acknowledgements are received on pipe 0 of the TX device. Next, you would enable auto-acknowledgements on pipe 0 of the TX and whatever pipe(s) you are using at the RX. Finally, you would set up the maximum number of retries and the retry delay in the SETUP_RETR register (ARC and ARD fields, respectively) of the TX device.

Once you have Enhanced Shockburst setup, it is nearly transparent because everything is handled automatically. The only difference really is that you must watch for the MAX_RT interrupt when you're watching for the TX_DS interrupt, because you'll get one or the other, but not both. If the packet is sent and acknowledged, then TX_DS will go active. If the packet does not get acknowledged before the maximum number of retries is hit, then the MAX_RT interrupt will go active.

At this point, there is one very important difference to remember. If the packet is sent and TX_DS goes active, the packet is removed from the TX FIFO like it would be in normal Shockburst mode. However, if the packet is not acknowledged and MAX_RT goes active, the packet **is not removed from the TX FIFO**. With that said, if you want to send the packet again, you simply toggle CE and you can give it another try. However, if you want to drop the packet, you must clear the TX FIFO **before** you load any more packets for transmission. Otherwise, the 24L01 will try to send the unsent packet plus anymore packets you try to load up to be sent and it could mess up your transmission.

The Hardware I Used for this Tutorial

I used the exact same hardware in this tutorial as in Tutorial 1. Nothing added, nothing subtracted, so as long as you were able to get Tutorial 1 working, you should certainly be able to get this one going.

Tutorial Software Description/Requirements

Similar to the hardware required, the software for this tutorial requires the same support as in Tutorial 1. If you were able to get it going, then you're in the clear for this tutorial.

Local Main File: maintutorial2local.c

Assuming you read Tutorial 1, this file should look extremely familiar to you. It operates the same way as Tutorial 1's maintutorial1ocal.c file, with the exception of the addition of some support code for Enhanced Shockburst.

Everything's the same until we get into the main routine. The call to Initialize() is still there, but its body is slightly different to allow for Enhanced Shockburst. The difference is that here, when we call nrf24l01_intialize_debug(), we let the third argument be true so that we enable Enhanced Shockburst on pipe 0.

Now we enter the main loop of the program. Here we wait for data from the UART, and then when we receive it, we transmit the character just like we did in Tutorial 1. Here's where things get different, though. When you are using Enhanced Shockburst, you have to watch for the MAX_RT interrupt, which is added to the while loop that waits to see if the packet was transmitted. Remember that when you transmit a packet in Enhanced Shockburst, TX_DS **will not** go active if the packet didn't go through. Instead, MAX_RT will become active. This is the reason you have to watch for both.

At this point is another addition. Here, we check to see if the MAX_RT interrupt was active. If it was, then we process the data exactly like we did in Tutorial 1. However, if MAX_RT is active, then we know the data didn't make it to the receiver and print a new debug character, "*", to reflect this information. That way we can tell the difference in HyperTerminal whether the packet made it to the remote unit or not when it was transmitted from the local unit if it didn't get transmitted back to the local unit. Also remember that, as I discussed above, when the MAX_RT interrupt is asserted, the data is left in the TX FIFO. For this reason, we call the nrf24l01_flush_tx() function to remove the packet. Finally we clear the interrupts and the loop starts again to continue to look for new characters at the UART.

Remote Main File: maintutorial2remote.c

Just as with the local file main file, maintutorial2remote.c is almost identical to the one used in Tutorial 1. It also simply adds in the concessions required to get Enhanced Shockburst up and running.

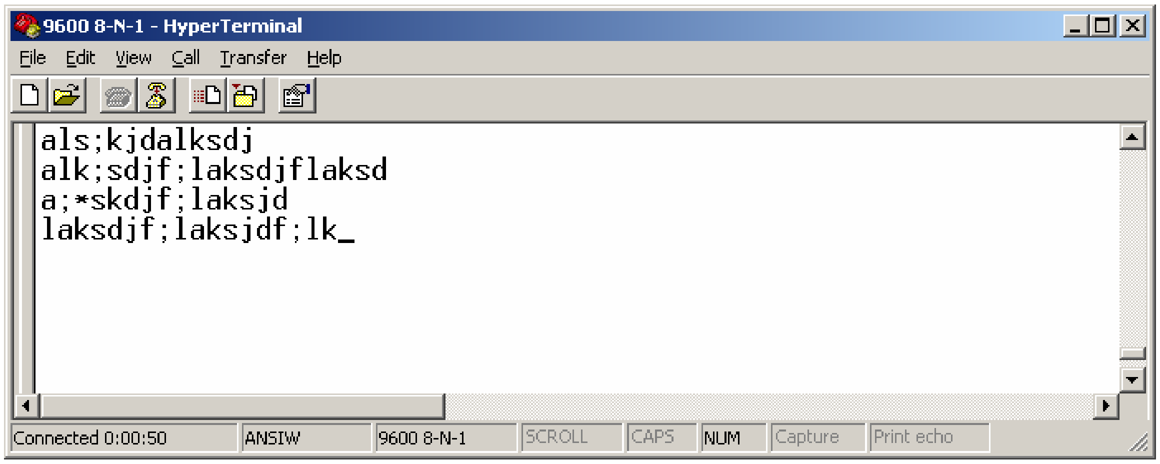
As with the local file, all of the declarations and defines before main are the same as those in Tutorial 1's remote main file. When we get into the main routine, once again, the call to Initialize() is there, but its body is different to reflect using Enhanced Shockburst. In Initialize(), the call to nrf24l01_initialize_debug() has the third parameter changed to "true" to enable Enhanced Shockburst, as in the local main file.

After Initialize() is called, we move into the main program loop. Here, we wait for a packet to arrive and once one is received, we simply send it back over the RF link. Similar to the local file, once we send the packet out we have to watch for both the TX_DS and MAX_RT interrupts to ensure that we don't get locked up. Whether the packet was sent correctly or not is ignored (TX_DS or MAX_RT interrupt active, respectively), and we clear interrupts and go back to the beginning of the loop.

Actual Program Output

In Figure 1, you can see the window in HyperTerminal that shows the program working. The 24L01s are in very close proximity to one another (around 6" apart), so there is an extremely high percentage of successful packets (the same as in Tutorial 1). Only one of the packets actually did not make it through. This packet can be seen on the

3rd line and the 3rd character as an asterisk. The fact that it is an asterisk indicates that the MAX_RT interrupt was asserted.



```
als;kjdalksdj
alk;sdjf;laksdjflaksd
a;*skdjf;laksjd
laksdjf;laksjdf;lk_
```

```
Connected 0:00:50   ANSIW   9600 8-N-1   SCROLL   CAPS   NUM   Capture   Print echo
```

Figure 1. Output window from HyperTerminal

Concluding Remarks

At this point, you should now have a pretty firm understanding of Enhanced Shockburst. If you have any questions, feel free to email me at brennen@diyembedded.com.