

Nextest Applications Note

Using the GSI Lumonics M310 laser trimmer with Nextest Maverick and Lightning
By Dan Bullard, Mixed Signal Marketing Applications Engineer, Nextest Systems
7 October 2005

Laser trimming of silicon chip die has been around for many years, but you may have never been “exposed” to it before. If that’s your story, this applications note is for you. Here I will tell you how to connect, operate and drive a GSI Lumonics M310 laser trimmer from a Maverick or Lightning tester.



Figure 1: GSI Lumonics M310

Overview

The M310 works like any conventional wafer prober, indexing the wafer to the die to be tested, moving the die up to the probe tips for testing, then stepping to the next die as instructed by the tester. The M310 uses a Sun Sparc™ workstation (pictured above on an Anthro cart) running Solaris™ as the controller of all operations. While knowing a bit of Unix can be helpful when using the M310, it is not necessary as most operations are controlled by a GUI written by GSI. The Tester to Trimmer interface can be GPIB, RS232 or Ethernet (via TCP/IP). All communications is controlled by an interface GSI calls CTRIMS, which is a string based language. Additionally there is a “fast bus” that can be used for faster, more direct communication between the tester and trimmer. This bus is used for advanced operations and not covered in this apps note.

The M310 differs from a traditional prober by the addition of a laser and light directing and metering unit on top of the prober. This unit can generate, transport and direct a “low power” laser beam with very high accuracy and deliver it to any point on the die under test. Low power is a relative term, the laser on the M310 can put out up to 20uJ (micro-Joules). By comparison, the average laser pointer emits about 5mJ (milli-Joules), but the beam is much wider and therefore doesn’t go around slicing things. A 20uJ laser focused on a tiny area is powerful enough to cut through the sputtered aluminum or Nichrome coating on a wafer. In fact rarely do you need so much power, and the beam is modulated on and off at some rate (known as the Q rate) to reduce its power even more. The laser beam comes from an aperture directly over the probed die and cuts parts of the circuit for one of two purposes. The laser cuts either a link (similar to fuse blowing) or a trim (similar to a potentiometer). Cutting links is easy, the tester makes a measurement, decides on what links to cut based on a lookup table, then cuts the links. The lookup table is based on some empirical data that the device manufacturer has worked out in advance. In the case of a trim, “bites” are taken out of an area to change the resistance of a metal film resistor on the die. Either way, the tester doesn’t actually have to know the location of these features on the die, all that information is stored in a Trimmer SetUp file (.tsu) on the trimmer itself. All the tester has to do is tell the trimmer what feature to cut and how many cuts to make. The feature names are just strings passed to the trimmer via the CTRIMS interface, so mostly this is a matter of doing an sprintf() to send the feature name and number of cuts to the trimmer. It can get trickier, usually when you are doing a measure-trim-measure loop, but even that isn’t too difficult to understand. The tester makes a measurement (voltage, current, time, frequency, etc) then tells the trimmer to make a cut on a particular feature, then the tester measures again to see if another cut is needed. Laser power can be adjusted in such a loop, the laser power is set to some miniscule value, the tester forces a voltage on a Laser Target pin and current flows to the substrate. The tester repeatedly increments the laser power, requests a cut, then measures current, and so on until the current goes to zero. That’s the indication that the laser now has sufficient power to cut through the metal on this wafer (although some companies do this adjustment per die).

In addition of course you have to handle the other communication just like any other wafer prober, such as reading the XY location so you can display it in the datalog, but remember, the trimmer is in charge of all the indexing, so all the tester has to do is sit back and let the trimmer drive the probes across the wafer (actually it’s the other way around) and perform it’s testerly duties of deciding whether the part is good or not after trimming. The M310 takes care of the wafer mapping as well as generating the Ink List based on what bin the tester sends to the M310.

CTRIMS basics

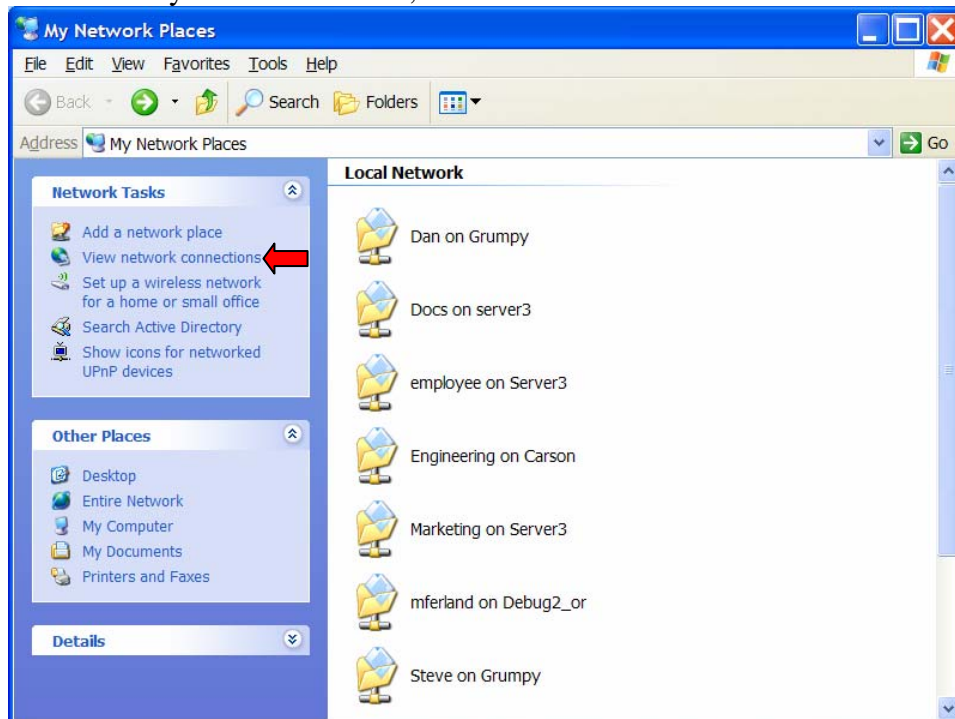
As mentioned above, the tester to trimmer communication is handled by a text based interface called CTRIMS. Basically you pass strings to the trimmer and wait for a response. When you get it, you can check to make sure everything was OK before proceeding. In one case, waiting for the response is what keeps the tester from proceeding while the trimmer indexes to the next die. That is the string "CURRENT_POS?". When the tester sends this string it can wait until the trimmer replies, which it will only do when it has set “down” on the next die. In Maverick lingo the flow goes something like this: in

the BEFORE_TESTING_BLOCK you send "CURRENT_POS?" to the trimmer, when it returns you can start testing, then in the AFTER_TESTING_BLOCK you send "NEXT <bin_number>". As the Maverick test program loops around from AFTER_TESTING_BLOCK to BEFORE_TESTING_BLOCK the trimmer will be indexing to the next die, and the tester will be waiting for that event by waiting for the response to "CURRENT_POS?". If you just wanted to use the M310 as a prober that would pretty much be the extent of your communications. How do you get to this point though? You need a communications channel to the trimmer and a function that can send and receive strings to and from the trimmer.

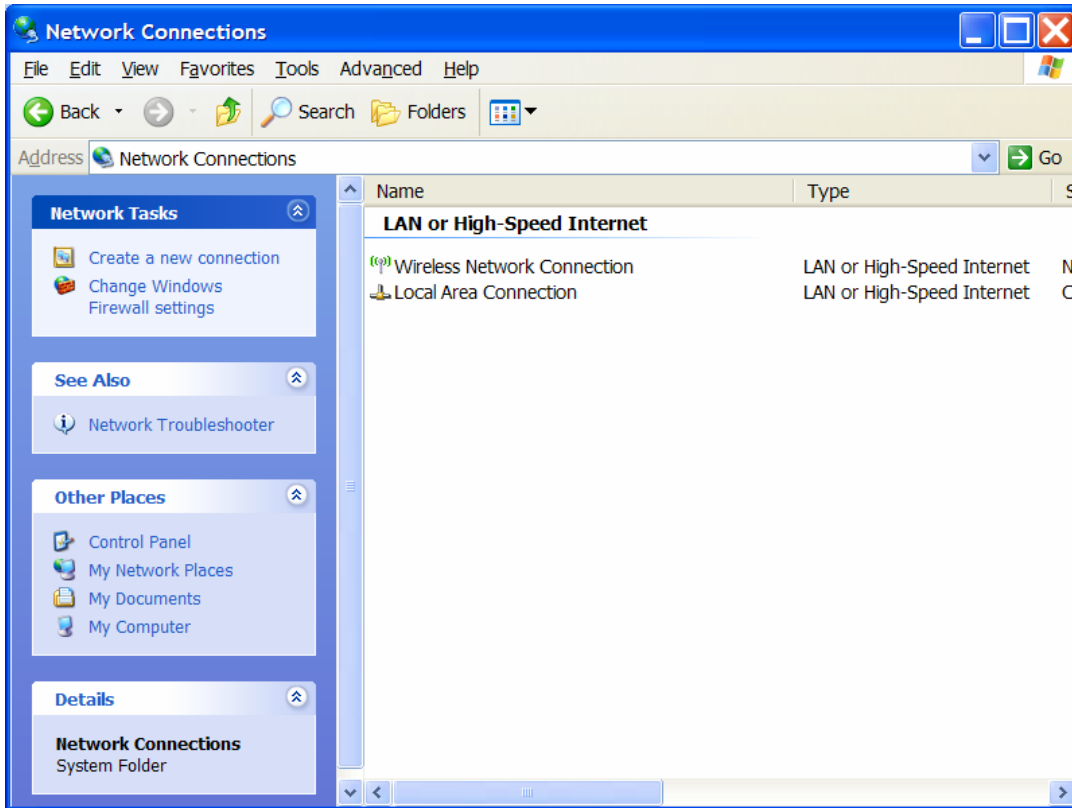
Communication

The most convenient interface solution is Ethernet. This is what I used and I recommend it for most applications. It's pretty fast and easy to configure. You could use the existing Ethernet port on the Host PC for this, however, I found it better to have a dedicated network between the Host and the M310. I bought a NetGear FA120 USB to Ethernet transceiver for my project. It's easy to install and has no more latency than the built-in Ethernet card. Here is how you set it up to talk to the M310.

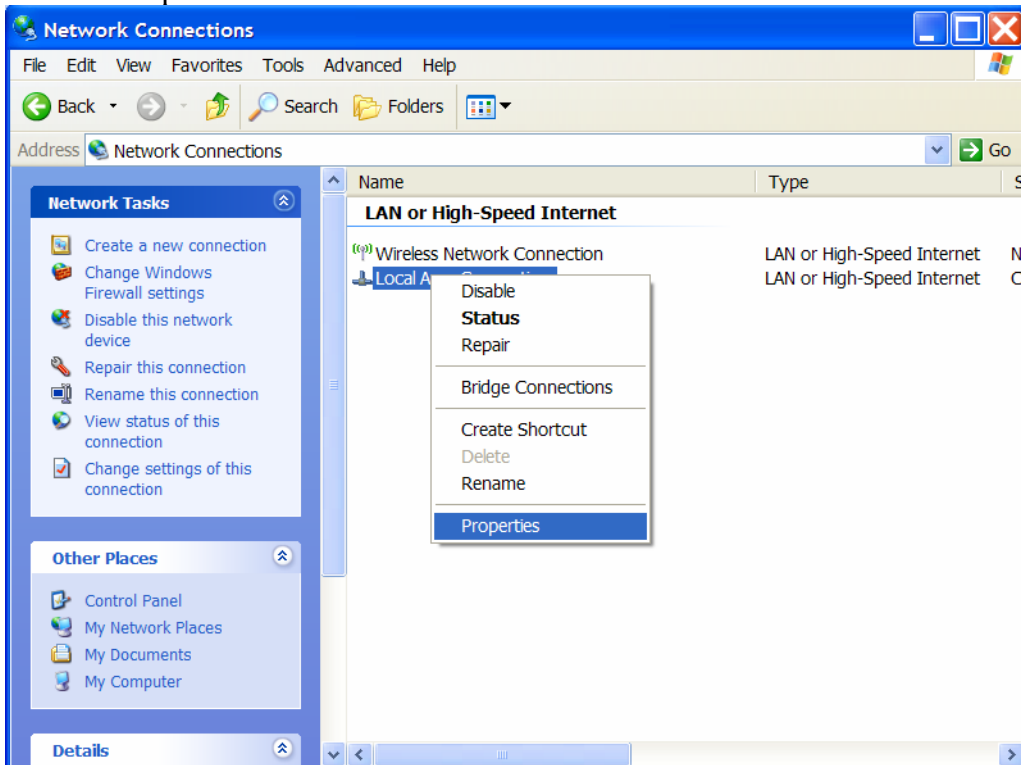
Click on "My Network Places", then click on "View Network Connections".



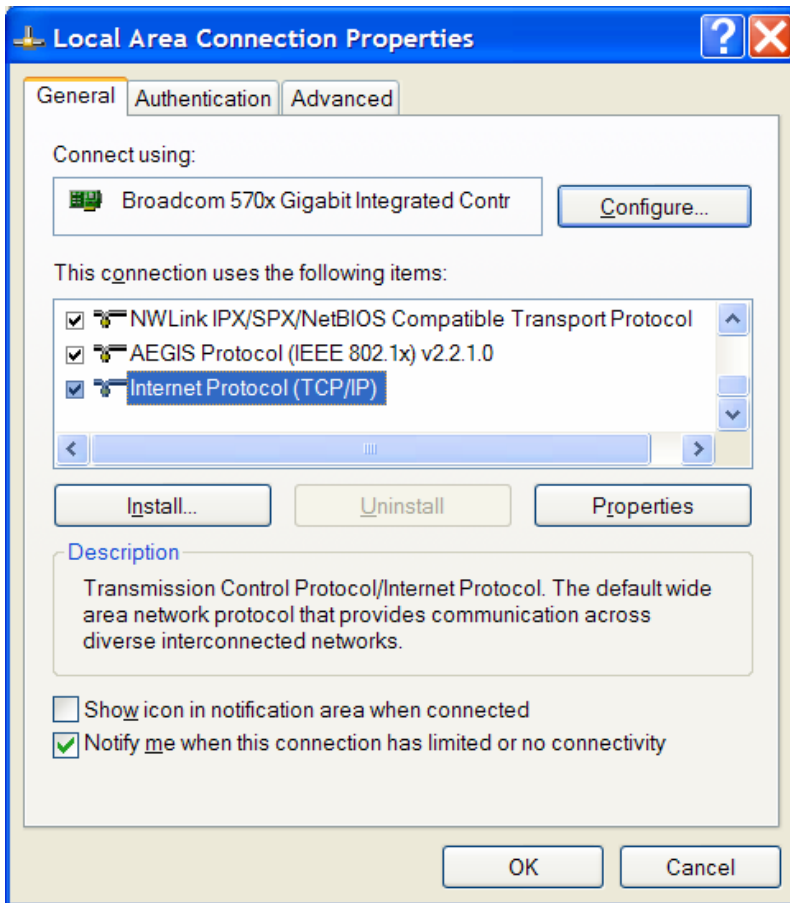
You should now see a window that looks like this:



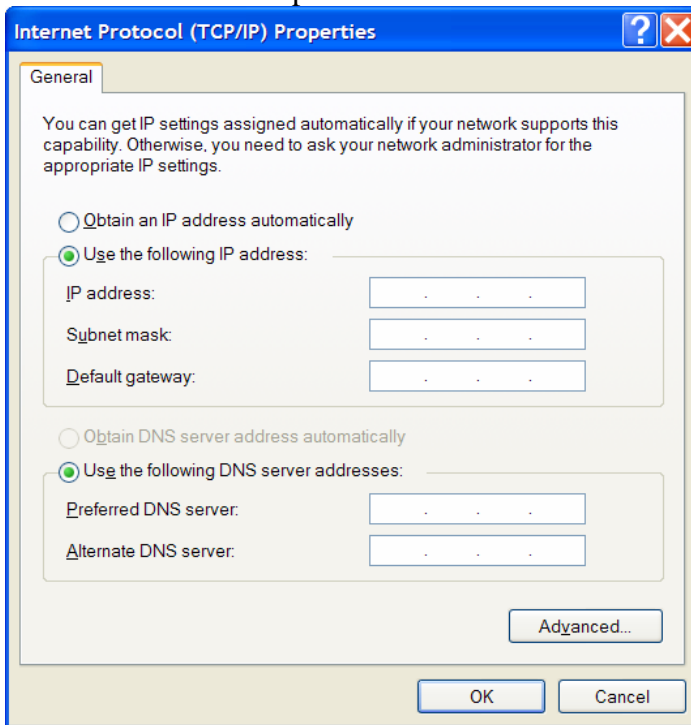
Right click on the network connection that you are going to use with the M310 and choose “Properties”.



Under Properties you’ll find a load of options, scroll to the bottom of the list and click on “Internet Protocol (TCP/IP)”.

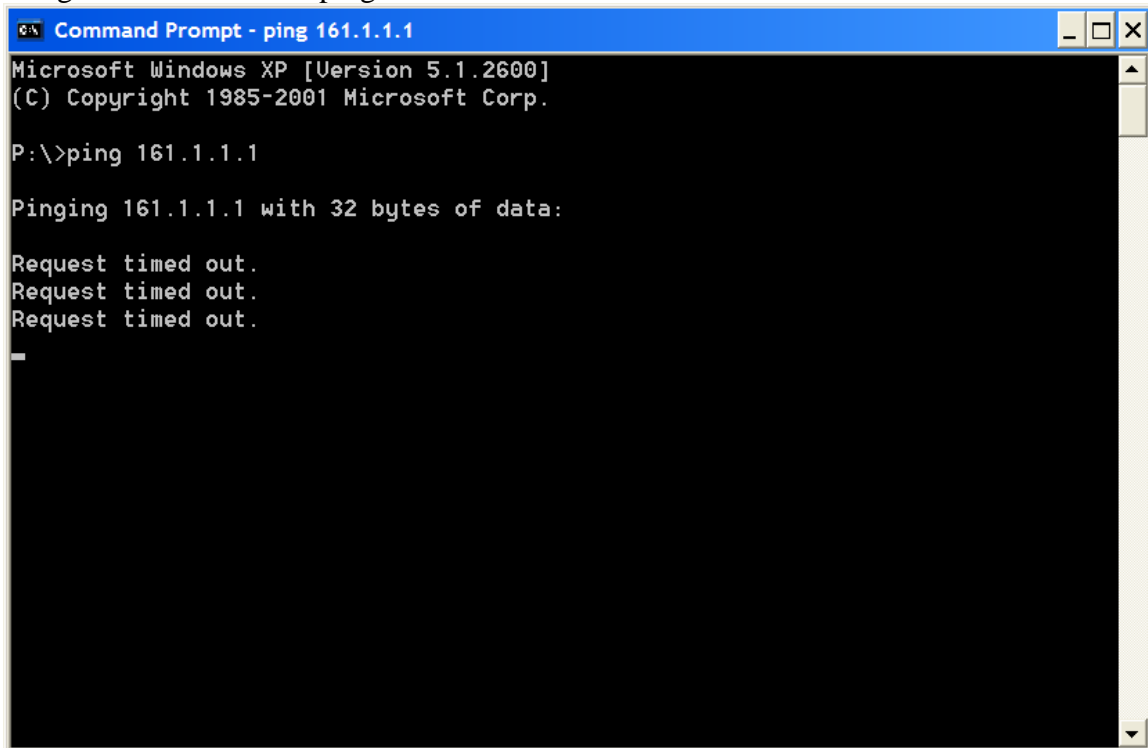


Now click on the “Properties” button in this window.



Here select “Use the following IP address” and fill in the IP address that the trimmer expects to hear from. Also fill in the “Subnet Mask” with “255.255.0.0. Then click “OK” and back out of the windows. Note that there is no need to specify a DNS Server.

You can test your connection to the trimmer by using the “ping” command; Below I am using a DOS window to ping a non-existent IP address.



```
Command Prompt - ping 161.1.1.1
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

P:\>ping 161.1.1.1

Pinging 161.1.1.1 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.

-
```

Obviously you should try pinging the M310’s IP address with a **CROSSOVER** Ethernet cable connected. The M310’s IP address should be in the /etc/hosts file. Use the Unix command “more” to read the hosts file, i.e.

```
>more /etc/hosts
```

Once you have that set up we can start with the code to talk to the M310.

Establishing the connection to the M310

My first project involved a PT, so I didn’t worry about having to pass strings between the site controller and the host CPU to talk to the M310. Because of that I tried to make things as simple as possible and do all communications from the Site process. Obviously you can’t do that if you use an ST->GT, but since the M310 can’t do multi-die trim, there is no sense in even discussing multi-site.

With that disclaimer out of the way, here is your first piece of code.

In the site_begin block I use a function that was provided by GSI to make the connection to the client. It returns a pointer to an object that I declare in the Site_begin.cpp file.

The objects and functions shown here are defined in two files provided by GSI, channel.h and channel.cpp, both are available from me at <mailto:dbullard@nextest.com>.

```

gsCommChannel * TheChannel;

TheChannel = new gsEtherClient ("122.167.0.1");
if(TheChannel->openChannel ())
{
    if (CommErrorOccur ())
        output ("Communication error occured: %s",
CommErrorDescriptionBuffer);
    else
        output ("Can't connect to GSI M310");
}

```

Note that I just put the M310s IP address in a string and the function `gsEtherClient` returns the channel object to `TheChannel`. Once that is done, I can freely communicate with M310 with the function `sendAMessage`.

For example:

```

char channel_com[40];

sprintf(channel_com, "LOAD?");
TheChannel->sendAMessage (channel_com);

```

Now, for every action there is a reaction, and for every message sent, there should be a reply. To get the reply (so you can check for errors, responses to queries, etc) use the code:

```

char the_response[120];

GetTheResponse(the_response);

```

The function `GetTheResponse()` is something you have to put in your code as a global function, that is, any function that is going to talk to the M310 is going to need to get access to this function. It simply uses the function `readAMessage` to get the response to a message that was sent to the M310. Below is the code for this function. One of the nice features of this function is that we can set the Boolean user variable “`echo_response`” True and then watch the responses from the M310 come back as they are sent. This aids in debugging your program when communicating to the M310.

```

void GetTheResponse(char *response)
{
    int HowManyCharsAreRead;
    char TheInput[1024];
    Boolean AMessageWasRead = False;
    while (! AMessageWasRead)
    {
        if(TheChannel->readAMessage
(&HowManyCharsAreRead, TheInput))
        {
            output("Read Message Error\n");
            break;
        }

        if(HowManyCharsAreRead > 0)
        {
            AMessageWasRead = True;
            if(echo_response) output ("%s\n", TheInput);
            strcpy(response, TheInput);
        }
    }
}

```

Examples

Now that you have seen the pieces let's throw it together into a couple of unrelated examples.

Example 1: Inquiring the identity of the loaded setup file (.tsu file).

```

sprintf(channel_com, "LOAD?");
TheChannel->sendAMessage (channel_com);
GetTheResponse(the_response);
output("the loaded TSU file is %s", the_response);

```

Example 2: Setting the laser power (in micro-Joules)

```

sprintf(channel_com, "TDLT %1.2f", 20.0); // set to
max, let's burn some stuff
TheChannel->sendAMessage (channel_com);
GetTheResponse(the_response);

```

Example 3: Requesting the wafer parameters from the M310

```
sprintf(channel_com, "WAFE");
TheChannel->sendAMessage (channel_com);
GetTheResponse(the_response);
output("the response to WAFE is %s", the_response);
token=strtok(the_response, " ");
if(strcmp(token, "WAFER") == 0)
{
    strcpy(Columnstr, strtok(NULL, " "));
    Columns = atoi(Columnstr);
    strcpy(Rowstr, strtok(NULL, " "));
    Rows = atoi(Rowstr);
    strcpy(CuOriginCol, strtok(NULL, " "));
    strcpy(CuOriginRow, strtok(NULL, " "));
    strcpy(RefCol, strtok(NULL, " "));
    strcpy(RefRow, strtok(NULL, " "));
    strcpy(DirectXstr, strtok(NULL, " "));
    DirectX = atoi(DirectXstr);
    strcpy(DirectYstr, strtok(NULL, " "));
    DirectY = atoi(DirectYstr);
    strcpy(DieSizeXstr, strtok(NULL, " "));
    DieSizeX = atof(DieSizeXstr);
    strcpy(DieSizeYstr, strtok(NULL, " "));
    DieSizeY = atof(DieSizeYstr);
    strcpy(WaferDiamstr, strtok(NULL, " "));
    WaferDiam = atof(WaferDiamstr);
    strcpy(WaferOrient, strtok(NULL, " "));
    strcpy(WaferType, strtok(NULL, " "));
    token = strtok(NULL, " ");
    if( token == NULL)
    {
        output("Wafer ID not Set!, Defaulting Wafer
ID to wafer1");
        strcpy(WaferID, "wafer1");
    } else
        strcpy(WaferID, token);
    if(strcmp(WaferType, "1") == 0)
        output("The wafer %s is %f microns across,
and has a notch", WaferID, WaferDiam);
    else
        output("The wafer %s is %f microns across,
and does not have a notch", WaferID, WaferDiam);
```

Example 4: Commanding the M310 to align the laser

```
strcpy(channel_com, "AL");
TheChannel->sendAMessage (channel_com);
GetTheResponse(the_response);
if(strcmp(the_response, "AL 1 A") != 0)
    output("laser failed to align");
```

Example 5: Commanding the M310 to cut some links based on a measured value

```
if(gain_err <= -0.5 && gain_err > -1.3)        t_cuts = 0x1;
if(gain_err <= -1.3 && gain_err > -2.2)        t_cuts = 0x2;
if(gain_err <= -2.2 && gain_err > -3.1)        t_cuts = 0x4;
if(gain_err <= -3.1 && gain_err > -3.6)        t_cuts = 0x8;

CutLinks("GAIN_CH1", t_cuts);
```

Note in the last example I included the code that calculates what links to cut. In this application, gain_err was measured by the tester (with the PMU) and based on that value we can deduce which links need to be cut with the command CutLinks. Below is the code for our CutLinks function.

```
int CutLinks(CString TrimName, unsigned int LinkCuts)
{
    unsigned response = 0;
    int MaxLinks = 9;
    CString r_msg;
    CString temp = "CLRT "+ TrimName;
    r_msg = SendReceiveM310(temp);
    for(int c=0;c<MaxLinks;c++)
    {
        if((1 << c ) & LinkCuts)
        {
            char cut_val[20];
            char buffer[120];
            char *w;
            itoa(c+1, cut_val, 10);
            CString cutstring = "CUT "+ TrimName + " " +
                (CString)(cut_val);
            r_msg = SendReceiveM310(cutstring);
            strcpy(buffer, r_msg); // expect CS 9
            w=strtok(buffer," \t\n");
            if (w != NULL) w=strtok(NULL, " \t\n");
            if (w != NULL) response = atoi(w); else
                response = -1;
            if (response != 3)
```

```

        {
            CString skipcutstring = "SKPC " +
TrimName;
            r_msg = SendReceiveM310(cutstring);
        }
    }
    return(PASS);
}

```

And finally, this function uses another function called SendRecieveM310, this is just a less transparent, albeit simpler function that sends messages to and gets responses from the M310.

```

CString SendReceiveM310(CString msg)
{
    CString resp;
    char recvMsg[80];
    char sendMsg[80];
    sprintf(sendMsg, msg);
    TheChannel->sendAMessage (sendMsg);
    Delay(10 US);
    GetTheResponse(recvMsg);
    resp = recvMsg;
    return (resp);
}

```

As I mentioned before, all we have to do is send the name of the cut and the links to cut (using the CutLinks function). The name of the cut corresponds to an entry in the TSU file (Trim Setup file) on the M310. CutLinks Clears the Trim, the cuts the links one at a time in a “for” loop according the bit position of the LinkCuts parameter. It’s pretty simple looking really.

Debugging Tools

There are two places you can see what is going on between the tester and the M310. On the M310 is a feature that lets you see all the IO on the communications channel, and on the tester I showed you how to implement a way to see the responses to CTRIMS directives (using the “echo_response” user variable). Another nifty trick is being able to send CTRIMS commands directly to the M310 from UI. I used a user variable for that too. Here is how you set it up.

```
CSTRING_VARIABLE(send_command, "", "Send Command to M310")
{
    strcpy(channel_com, send_command);
    TheChannel->sendAMessage (channel_com);
    GetTheResponse(the_response);
    output("The response to the command %s is %s",
send_command, the_response);
}
```

Now if you want to send any CTRIMS command to the M310, just pop open the User Variables window and enter a string in the send_command user variable widget and click Set. You’ll get your response in the Site output window.

Wafer Prober Handshaking

The M310 is a wafer prober and a trimmer, so in order to step from die to die on the wafer you’ll need to tell the prober when you are done testing, what the bin of the tested die is, and then you’ll have to wait until M310 has positioned the next die under the probes before beginning the next execution of the sequencer.

In my program I used the BEFORE_TESTING and AFTER_TESTING blocks to do this handshaking. Again, I only run this program on a PT, so communicating directly from the Site process is OK.

Once I have determined the bin of the die I send that to M310 with the NEXT command inside the AFTER_TESTING block:

```
    sprintf(channel_com, "NEXT %2d", dut_bin_num);
    TheChannel->sendAMessage (channel_com);
    GetTheResponse(the_response);
    token=strtok(the_response, " ");
    token=strtok(NULL, " ");
    if(strcmp(token, "END_OF_WAFER") == 0)
        end_of_wafer();
```

The variable dut_bin_num is an int representing the bin number. In my case it’s never more than 99, so this code works fine. Note that one possible response to the NEXT command is that this is the end of the wafer. In that case I call a function that handles that properly. When the M310 gets the NEXT command it writes the bin into the ink table

then steps to the next die. In the meantime I finish out the AFTER_TESTING block and then the Nextest main calls the BEFORE_TESTING block. In that block I have this code:

```
strcpy(channel_com, "CURRENT_POS?");
TheChannel->sendAMessage (channel_com);
GetTheResponse(the_response);
token=strtok(NULL, " ");
strcpy(x_pos, token);
token=strtok(NULL, " ");
token=strtok(NULL, " ");
strcpy(y_pos, token);
```

The CURRENT_POS? query asks the M310 what die it is currently over. It will not respond until it is aligned over the next die and has finished the Z up operation and has the probes in contact with the die. So, the tester will hang here until the M310 replies with the X and Y position of the current die. In my case the customer wanted the X and Y position for several reasons. One is that the datalog needs to log the die location, so that becomes part of the datalog record. Also the customer wanted us to do wafer mapping (even though the M310 is also mapping on it's Sparc workstation screen). For that I convert the X and Y positions into strings and pass them to the wafer mapper in the AFTER_TESTING block like this:

```
BOOL ok = wmap_die_set( wmap_die_bin, atoi(x_pos),
atoi(y_pos), binID );
```

In this case binID is a string that names the bin in the wafer map configuration file (see the Maverick Programmers manual for more information on using the wafer map tool). The binID is simply a string that relates to the integer bin number that was set when the device failed or is still "1" if we passed everything.

Conclusion

I hope that I calmed some of your trepidations about learning to use the GSI Lumonics M310 Laser Trimmer. There is plenty more for me to learn, but I hope that this applications note gives you an idea where to start if you have to develop a test program that will talk to an M310. I have several additional references that will help you, so please feel free to ask for them by emailing me at dbullard@nextest.com. In addition as I mentioned before there is additional code you will need (channel.h, channel.cpp) and more examples, plus a step by step instruction sheet on loading and aligning a wafer on the M310.

Happy Trimming!

Dan Bullard