Example logbook report for Apsc 1299:  

The Morse Decoder

This following pages contain an example logbook that a student might have written while performing an experiment, called the “Morse Decoder”, where they created code to translate Morse code digits into regular Arabic digits. This example logbook is particularly useful for seeing how to record your work while coding and while troubleshooting code that is not behaving correctly.

Note that you do not need to write your logbook exactly like this one in order to get full marks—there is quite a bit of flexibility in how an engineer can organize and record data in their logbook. However, this example should give you an idea of what level of detail you need to aim for, and what kinds of data are important to record.

An engineer’s logbook can be used as a legal document to, for example, establish who owns an idea or innovation in the case of a patent dispute. For this reason, your logbook must be written in ink, and have each page signed (or initialed) and dated. Graphs may be done in pencil. Computer code should be permanently pasted or stapled into the book.

Your logbook also needs to be a complete record of everything you did in the lab, so even your mistakes and unwanted results should be documented fully. Never remove pages, and never obscure information from the logbook. If you make a mistake, cross out the error with a single line (so it can still be read), and write the correction nearby.

When you join the work force, you’ll often be called upon to make an estimate on how long a job will take, or to bill your clients accurately for the time you spent on their project. For this reason, you’re required to record the current time in your logbook regularly. By getting into this habit now, you’ll develop a good “feel” for how long certain tasks take, and will always have a record of exactly how long you worked on a particular project.

The annotations in this logbook will point out things an instructor would think the student did well and things they would feel were lacking. Overall, however, this logbook displays good note-taking skills and would likely receive a very good grade.
Sign and date every page.

Start a new experiment's write-up by recording the experiment title and all the equipment and computer programs you'll use.

Record the time often.

Start to read manual 9:38 AM

Notes
- want to create code that will decode button pushes in Morse code into regular Arabic digit.
- will only do Morse code numbers because they always have five dots or dashes. The alphabet's codes can be longer or shorter so it's harder to code for.
- Morse code numbers are:
  - Morse code numbers are: 1 = 01110 6 = 10000
  - 2 = 00110 7 = 11000
  - 3 = 00011 8 = 11010
  - 4 = 00001 9 = 11100
  - 5 = 00000 +0 = 11111 9:43 AM

Tasks
- use morseDecoder.txt to create needed code
- will need to add a lot to make it work
- use timers to test whether a button push is long enough to be considered a dash.
- use 0.35 sec or longer as a dash
- will include 10 sec delay after button is pushed or released to prevent "button-bounce" errors
- code must accept button push of up to 2 sec with 9:47 AM no errors

If you don't record the time at least once per page, you'll lose marks.

The student takes notes as she reads through the manual's theory, recording all the important information in her logbook.

When she gets to the tasks, she jots them down in point form. This is a clear way to state her experimental objectives.
Record your plans, rough work, and thought processes as you prepare for a task.

Create new project in MPLAB IDE:
- Project > Project wizard > Next > PIC18F4525
- Next > Microchip C18 Toolset >
- MPLAB C18 & Compiler V3.35 > Next > Browse

Made new folder in Projects called: Morse Decoder
- Project name: Morse Decoder
- Next > Next > Finish

- Creating a file called morseDecoder.c from
- supplied file morseDecoder.txt

- Going to need to add code in switch1_risingedge_action() function to time how long button push is.
- I'll add a timer that runs for as long as button is pushed. Then ReadTimerc() to see how long the button was pushed for.

In lab 6, Timers, I found a prescaler of 256 allows me to time up to 256x7us x 2^16 = 2.047 seconds

- Minimum button push of 0.35 sec counts as a dash, not a dot. Code will have 0.1 sec delay for button bounce error prevention, so timer must run for 0.25 sec for a dash

0.25 sec = 4 x 256 x 125ns \Rightarrow 4 = 4812.5

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:48 AM</td>
<td>Button is already set up from Lab 6, Timers</td>
</tr>
<tr>
<td>9:48 AM</td>
<td>LCD still attached too</td>
</tr>
<tr>
<td>9:50 AM</td>
<td>Created new folder in Projects called: Morse Decoder</td>
</tr>
<tr>
<td>9:54 AM</td>
<td>Created a file called morseDecoder.c from supplied file morseDecoder.txt</td>
</tr>
<tr>
<td>9:59 AM</td>
<td>Going to need to add code in switch1_risingedge_action() function to time how long button push is</td>
</tr>
<tr>
<td>10:06 AM</td>
<td>In lab 6, Timers, I found a prescaler of 256</td>
</tr>
<tr>
<td>10:06 AM</td>
<td>Allows me to time up to 256 x 7us x 2^16 = 2.047 seconds</td>
</tr>
<tr>
<td>10:06 AM</td>
<td>Minimum button push of 0.35 sec counts as a dash, not a dot. Code will have 0.1 sec delay for button bounce error prevention, so timer must run for 0.25 sec for a dash</td>
</tr>
</tbody>
</table>
Think of the logbook as a real-time diary of your work. Record what you do, as you’re doing it.

Notice how the student jots down notes on everything she does, step by step.

Here, the student jots down some pseudo-code as part of her planning process for the project.

So I need to test whether `read_timer()` got to 7813 or higher.

- Copy/pasted the `open_timer` command from Lab 6 using timers.c.
- Changed prescaler to 256.
- Only need to open timer once, so I put `open_timer()` in `main()` outside the while(1) loop.

Now building code in switch1_risingedge_action().
- `add_write_timer()`;
- Add a while (portB.bit. RD1) loop to wait for button to be released.

Next will read timer and test whether it got past 7813. If yes, record 1, if no, record 0.

You have to record values in an array. They called it `dotdash_array[i]`.
- I is a global variable that acts as index for array.

So this is what I’ll do:

```c
if (Read_timer() > 7813)
    dotdash_array[i] = 1;
else
    dotdash_array[i] = 0;
```

- Resetting `i` to zero taken care of in the supplied code after 5 button pushes.
- Now adding this stuff to switch1_risingedge_action().

Function
Again, this is a step-by-step, real-time diary of what she’s doing, as she does it.

Someone could take the same code she started with and reproduce her work using only the notes in this logbook, with no lab manual. This is the level of detail you should be aiming for.
Added Morse calculation of "morse", as on previous page, to print-morse() function.
- declare morse as type char

Creating get Arabic(morse) function

Using XXXXXX, I have:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111</td>
<td>00111</td>
<td>00011</td>
<td>00001</td>
<td>00000</td>
<td>10000</td>
<td>11000</td>
<td>11100</td>
<td>11110</td>
</tr>
</tbody>
</table>

- Created a case switch in getArabic() to return correct digit.
  - What to return if none of these?
    - OKay, I made the return value of type unsigned int, so I can return a larger value than 0-9. I'll use 20 as the error code/default value for switch.
    - I'll also need to add some new variable in print-morse() to catch a "20" and display an error message.

- Added "unsigned int testForError" declaration to print-morse() function

- Added if/else loop to print "error" if 20 is returned, and print the value otherwise.
A common error is to not record enough notes when things start to go wrong.

Notice how this student handles coding problems: She notes every error message, including its code and warning.

Then, as she figures out what went wrong, she notes what the problem was and how to fix it.

It’s implied, here, that she made the fixes she noted. This is okay for simple problems. If a more complex solution was required, she would have needed to add more details about what she did.
Here she encounters a different kind of problem: Her project built successfully, but the circuit is not behaving as intended.

She jots down notes on everything she does to fix the problem, step by step. This is the same level of detail she used when she documented creating her code.
When you're testing or measuring something, record all your observations fully.

- Stepping through code...
  - Program goes into switch `1_rising_edge_action()` five times for five button presses, which is correct.
  - Then it goes to get Arabic(), which is correct.
  - Observation: my LCD screen is full of gibberish.
  - Then program goes back to recording digits because it goes back into `1_rising_edge_action()`.

The gibberish on screen is because I didn't configure USART! Argh, argh, argh... 11:34 AM

- Added `configure USART.c` and `configure USART.h` to project.
- Added `#include "..\Functions\Configure USART.h`.
- Added lines:  `set oscillate 32MHz();`  
  `configure USART(9600, 1, 32);`

Build failed: couldn't find definition of `set oscillate 32MHz` 11:39 AM

?? That's in `osc.h`, and the provided file has the `#include` statement for it. Don't know why this is happening...
  → Ack! It's not in the project! For got to add it
  - Adding `osc.h, osc.c`.

Build succeeded, no errors, 11:42 AM

Program target device

Yah! Five taps of the button gives a correct-looking display. It gives "Morse: 11111111 Arabic: φ" 11:43 AM

It's not enough to say something worked or didn't work. Record what you observed that led you to that conclusion.
Again, when testing or measuring something, record all your observations.

Testing to make sure all numbers translate correctly:

<table>
<thead>
<tr>
<th>Tap pattern</th>
<th>Morse:</th>
<th>Arabic:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = 01111</td>
<td>01111</td>
<td>9 ✓</td>
</tr>
<tr>
<td>2 = 00111</td>
<td>00111</td>
<td>8 ✗</td>
</tr>
<tr>
<td>3 = 00011</td>
<td>00011</td>
<td>7 ✓</td>
</tr>
<tr>
<td>4 = 00001</td>
<td>00001</td>
<td>6 ✓</td>
</tr>
<tr>
<td>5 = 00000</td>
<td>00000</td>
<td>5 ✓</td>
</tr>
<tr>
<td>6 = 10000</td>
<td>10000</td>
<td>4 ✓</td>
</tr>
<tr>
<td>7 = 11000</td>
<td>11000</td>
<td>3 ✓</td>
</tr>
<tr>
<td>8 = 11100</td>
<td>11100</td>
<td>2 ✓</td>
</tr>
<tr>
<td>9 = 11110</td>
<td>11110</td>
<td>1 ✓</td>
</tr>
<tr>
<td>0 = 11111</td>
<td>11111</td>
<td>0 ✓</td>
</tr>
<tr>
<td>10101</td>
<td>10101</td>
<td>error!</td>
</tr>
<tr>
<td>01000</td>
<td>01000</td>
<td>error!</td>
</tr>
<tr>
<td>00010</td>
<td>00010</td>
<td>error!</td>
</tr>
</tbody>
</table>

- I have to be careful with fast taps. It's very touchy and I still get "button bounce" errors a lot.

- So Morse numbers are fine. Only Arabic numbers aren't being translated correctly. The problem is probably in my code switch or my math.

- Okay, yeah. I display Morse code as [0][1][2][3][4] so I need to change how I multiply those elements to convert to a number. Correct is:
  
  \[0\times 16 + [1]\times 8 + [2]\times 4 + [3]\times 2 + [4]\times 1\]

- Changed default assignment to Morse variable to do this.
Again, record the observations that led you to believe the circuit or code is working (or not working) as intended.

Always attach your code to your logbook. The level of detail you should aim for is that someone could reproduce the experiment based only on your notes. They would need a copy of your code to do that.

Before you print out, however, double-check whether you included enough commenting. Comments are one of the easiest things to forget!

<table>
<thead>
<tr>
<th>Tap pattern</th>
<th>Morse</th>
<th>Arabic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01111</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>00111</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>00001</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>00000</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>10000</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>11000</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>11100</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>11110</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: When it triggers correctly, it’s fine, but I get a lot of errors due to button. A long press at the start seems more likely to generate an error.

Otherwise, works as intended!
- Displays: “Morse: Correct Morse code
  Arabic: Correct digit or ‘error!’”
  in response to Morse code numbers being tapped in with button presses.

Code on attached on next page

→ Wait! Adding more comments and re-printing 12:07 PM
Code should have enough commenting that someone could understand how the program works based only on the comments.

In your opinion, does this code have enough comments?
These were the comments in the original, supplied code.

The student probably should have changed them, but she included her own comments after her added code, so all the necessary information is here.
dotdash_array[3],
dotdash_array[4]);

// Insert code to display the corresponding Arabic digit on the
// second line of the LCD. Put the heading "Arabic: " before the digit.
morse = dotdash_array[0]*16+    // convert Morse code into a number
dotdash_array[1]*8+           
dotdash_array[2]*4+           
dotdash_array[3]*2+           
dotdash_array[4]*1;

testForError = getArabic(morse);  // converts Morse to Arabic

LCD SetPosition(2,1);  // set to 2nd line, 1st space of LCD
printf("Arabic: ");

if(testForError==20)  
{
    printf("error!");  // print error message if not Morse code
}
else
{
    printf("%s",testForError); // print Arabic digit
}

unsigned int getArabic(char morse)
{
    // function translates Morse code number into Arabic number
    // and returns it

    switch(morse)  // Translates the Morse code and returns
    {
        case 15:  // an Arabic digit or an error code (20)
            return 1;
            break;
        case 7:
            return 2;
            break;
        case 3:
            return 3;
            break;
        case 1:
            return 4;
            break;
        case 0:
            return 5;
            break;
        case 16:
            return 6;
            break;
        case 24:
            return 7;
            break;
        case 28:
            return 8;
            break;
        case 30:
            return 9;
            break;
        case 31:
            return 0;
            break;
        default:
            return 20;
            break;
    }
}