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.

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.

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

### MORSE DECODER

<table>
<thead>
<tr>
<th>Equipment</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC18F4525 microcontroller</td>
<td>morsDecoder.txt, serialLCD.c, serialLCD.h</td>
</tr>
<tr>
<td>breadboard, wires</td>
<td>configuration_bits.h</td>
</tr>
<tr>
<td>Switch</td>
<td>osc.c, osc.h</td>
</tr>
<tr>
<td>LCD screen</td>
<td>configureUSART.c</td>
</tr>
<tr>
<td>10kΩ resistor</td>
<td>configureUSART.h</td>
</tr>
<tr>
<td>20kΩ resistor</td>
<td></td>
</tr>
</tbody>
</table>

Starting 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 it for 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 write code for

  - Morse code numbers are:
    - 1 = 01111
    - 2 = 00111
    - 3 = 00011
    - 4 = 00001
    - 5 = 00000
    - 6 = 10000
    - 7 = 11000
    - 8 = 11100
    - 9 = 11110
    - 0 = 11111

**Tasks:**
- use morsDecoder.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 s or longer as a dash
  - will include 0.1 s delay after button push or released to prevent "button bounce" errors
- code must accept button push of up to 2 s with no errors.
Record your plans, rough work, and thought processes as you prepare for a task.

- Button is already set up from Lab 6, Timers 9:48 AM
  - LCD still attached too

- Creating project using procedure on last page of logbook 9:50 AM
  - include files: serLCD.c, serLCD.h configuration.h

- Creating a file called morseDecoder.c from 9:54 AM
  - supplied file morseDecoder.txt

- Going to need to add code in switch1_rising_edge_action() function to time how long button push is
  - I’ll add a timer that runs for as long as button is pushed. Then, Read Timer#0() and see how long button was pushed

⇒ In Lab 6, Timers, I found a prescaler of 256 9:59 AM
  - allows me to time intervals up to 256×2⁻¹⁶ = 2.047 sec

- Want a total time of 0.35 sec for minimum button push length to count as a dash, not a dot.

- Code has a delay of 0.1 sec to prevent “button bounce” errors, so timer has to run for 0.25 s at least for a dash:

  \[
  0.25 \text{ sec} = 256 \times 125 \text{ ns}
  \]

⇒ \[
  y = \frac{0.25}{256 \times 125 \times 10^{-9}} = 7812.5 \approx 7813
  \]

⇒ So I’ll need to test whether Read Timer#0() got to 7813 or higher 10:06 AM

Consider putting the steps for often-performed tasks in an appendix of your logbook. That way, you can refer to it rather than re-writing the steps.
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.
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 arsene calculation of “morse”, as on previous page, to print_morse() function.
- Declare Morse as type char.

- Created get Arabic(morse) function.

  \[ \text{Using \( \frac{1}{110} / \frac{1}{10} \)} \]

  - \( \text{I have:} \)
    - \( 1 = 01111 = 15 > 7 + 2 + 1 \)
    - \( 2 = 00111 = 7 = 4 + 2 + 1 \)
    - \( 3 = 00011 = 3 = 2 + 1 \)
    - \( 4 = 00001 = 1 = 1 \)
    - \( 5 = 00000 = 0 = 0 \)
    - \( 6 = 10000 = 16 = 16 \)
    - \( 7 = 11000 = 24 = 16 + 8 \)
    - \( 8 = 11100 = 28 = 16 + 8 + 4 \)
    - \( 9 = 11110 = 30 = 16 + 8 + 4 + 2 \)
    - \( 0 = 11111 = 31 = 16 + 8 + 4 + 2 + 1 \)

- 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 test For Error” 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
  - Then program goes back to recording digits because it goes back into switch 1_rising_edge_action()

- Gibberish on screen is because I didn't configure USART! Argh, argh, argh...

- Added configure USART.c and configure USART.h to project
- Added #include "..\Functions\ConfigureUSART.h"
- Added lines: set_osc_32MHz();
  configureUSART(9600ul, 32);

Build failed: couldn't find definition of set_osc_32MHz

?? 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! Forgot to add it
    - adding osc.h, osc.c

Build succeeded, no errors

Program targeted device

Yah! Five taps of the button gives a correct-looking display. It gives "Morse: 111111"
Arabic: 
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</td>
<td>01111</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>00111</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>00011</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>00001</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>000000</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>10000</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>11000</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>11100</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>11110</td>
<td>1</td>
</tr>
<tr>
<td>0</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 case switch or my math.

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

  

- Changed definition 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!
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.
// 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
LCDCursorPosition(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
    {
        // an Arabic digit or an error code (20)
        case 15:
            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;
    }
}