The device needs to accumulate bits to make up characters.

When it has received a character, it needs to add it to the characters already stored in memory, keeping a count of the number of unmatched parentheses, and it needs to output the message when it is complete.

Here’s a Structure Diagram showing the various steps involved in this process. This is not a requirement of the assignment, but it may help you to understand what’s involved. We’ll convert the structure diagram to a hardware design.

You could argue that there’s a slight problem with this arrangement. If two characters arrive without any delay between them, then the time taken to complete the output stored bytes phase will cause some bits of the second character to be missed. This is hardly likely to be a problem with the arriving radio message, since the bits are separated by days or weeks. And if the characters that have already been captured by the alert crew member were dumped into the circuit from a memory, then they could arrive without any delays, which would break the circuit.

However, I’ve assumed – and it is reasonable for you to assume – that under the circumstances of the assignment, the leading characters will be input by a human, slowly, so that that in machine terms, there will be significant delays between each bit and the next bit, and the circuit won’t lose any bits. At the end of this sample solution, there is a description of an alternative design that would handle this problem.
A shift register stores bits until a complete byte has accumulated. The byte is then transferred to byte-wide memory. A 12-bit counter keeps track of the number of characters that have arrived. And an 8-bit counter (the size of this counter is an arbitrary decision. To cope with the worst possible message, consisting only of parentheses, an 11-bit counter would suffice, but an 8-bit counter allows for a parenthesis:message ratio of 1:16, which seems reasonable) keeps track of the number of parentheses, increasing if the character is a left parenthesis, or decreasing if it is a right parenthesis. When the parenthesis count has decreased to 0, the message is deemed to be complete, and it is output. The memory uses the same bus for inputs and outputs, so the shift register has to have tristate outputs (the memory will automatically have them too). Figure 1 shows the main part of the architecture.

The device keeps track of the number of bits as they arrive using a 3-bit bit-counter (Figure 2), which outputs to an equal-to-zero detector (a 3-input, active-low inputs, AND gate, not shown). Because the bit count recycles to 0 when a complete byte has arrived, it does not need to have down-counting capabilities like the other counters in the architecture.

The parentheses in the message must match, so a parenthesis counter (Figure 3) is incremented when a left parenthesis arrives, and decremented when a right parenthesis arrives. The count will not decrease to 0 until all the left parentheses that have arrived have been matched – that is, when the complete message has arrived.
The parenthesis counter is clocked when either a left or a right parenthesis arrives. The type of character contained in the shift register is detected by a simple piece of combinational circuitry that detects the bit combinations 01111011 and 01111101 (ASCII 123 and 125, which represent “(“ and “)” respectively). The circuit in figure 4 includes a subcircuit for “other,” comprising an AND gate that detects two low inputs (i.e., not a left parenthesis AND not a right parenthesis).

Having designed the architecture of the device, we can now turn our thoughts to controlling it. There are three main components in the controller. First, it gets bits till the number of bits is 8 (signified by a 0 in the 3-bit bit-counter). Then it stores the character, keeping a count of the number of unmatched parentheses, and the number of characters. When the number of parentheses is 0, it outputs characters till the character count has decreased to 0. Figure 5 shows an ASM diagram for the controller, a transition table and a set of Boolean equations for the outputs. Note the Boolean expression for $B_n$. In state 2, there appear to be a complex condition governing whether $B_n$ should be T or F. However, a quick look at the diagram will show you that the next state number after two can only be 1 or 3, both of which have a 1 in their lower significance bit, so in state 2, $B_n$ is True.
Figure 5: The ASM for the device

Converting the ASM to a circuit diagram, using the multiplexor approach is trivial, though somewhat tedious.
ALTERNATIVE DESIGN

At the start of this sample solution, the problem of handling a byte sequence with no gaps was mentioned. The above design would lose data under these circumstances, because the store complete character phase of the ASM uses a clock cycle, during which the next bit of data would arrive and be lost.

In order to handle this, we add another register, (a buffer register), and separating the ASM into two.

The first ASM would be responsible for converting bits to bytes. It would load them into a shift register, just the same as the earlier design. Then, on the clock cycle when the last (eighth) bit was loaded into the shift register, the byte would be loaded into the buffer register. Thus the shift register would be free to load more bits, while the second ASM loaded the information in the buffer register into memory. The second ASM would also be responsible for outputting the completed message into the nav computer.