The Robot Control Program

MVRT Team 115

2006-2007

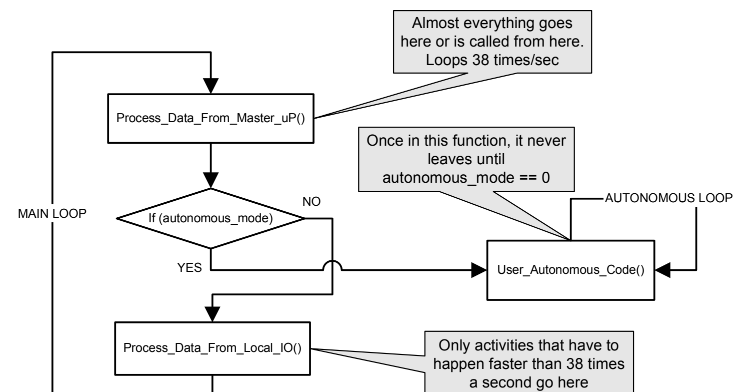
What the Control Program Does

- This lesson describes the control program.
- If you know the basic structure of the program, you know where to look for functions & you know where to put the code you write.
- You should also learn what NOT to do.

Overall Program Structure

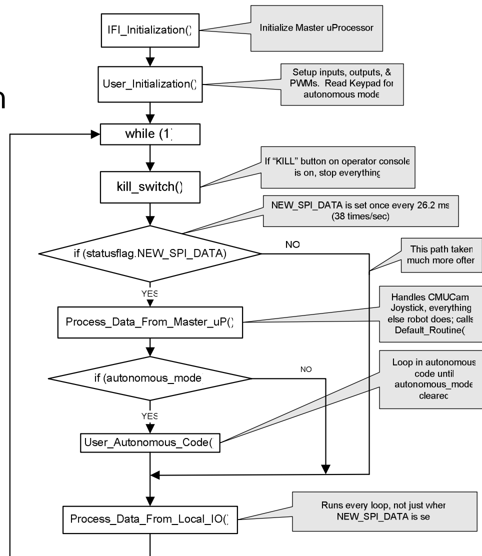
- The control program has 3 main sections:
  - Initialization
  - Main Loop
  - Autonomous Loop
- Initialization tells the controller what inputs & outputs the program will use. It reads the keypad data.
- After initialization, the main loop runs until you turn the controller off.
- If autonomous mode is on, the main loop transfers control to the autonomous loop, which runs until autonomous mode is turned off. Then the main loop runs again.

The Two Control Program Loops



A More Detailed Look at the Main Program Loop

in main.c



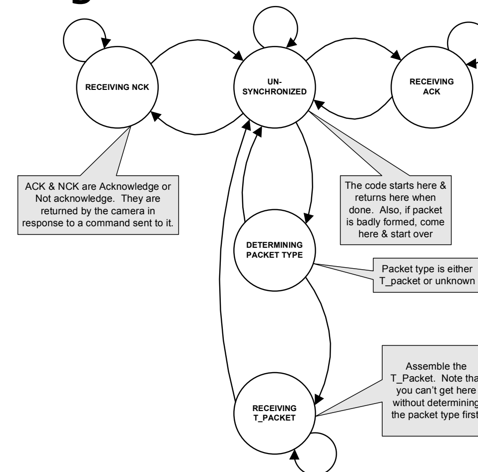
Main Loop

- The main loop is the fundamental structure of the control program
- Do NOT change this structure!
- Do NOT add
  - statements to main loop
  - while() statements elsewhere in the program
  - getdata(), putdata() elsewhere in the program
- It can be confusing to figure out how to add code without changing the fundamental structure, but there are techniques that help.
  - The first one to understand is the state machine

State Machines

- A state machine is an idea from computer science and electrical engineering. You make a model that breaks down a complex operation into a set of simple operations with clear rules about what to do next.
- Each simple operation is known as a state. The current state, plus inputs, determines what state the machine *transitions* to next.
- The state machine is always in one of the defined states. (The machine can't be in an undefined state.)
- To program a computer as a state machine, turn the code into a set of program pieces, each of which can be executed separately. Each program piece is considered a state.
- A common way to make a state machine in C is to use a *switch* statement. Inside each case, a variable is used to determine the next state. (Note that every possible case should be defined.)

Diagram of Camera States



```
switch(state)
 {
  case UNSYNCHRONIZED:
   if(byte == 255) // start of a new data packet?
   {
    state = DETERMINING_PACKET_TYPE;
   }
   else if(byte == 'A') // start of an ACK?
   {
    packet_char_count = 2;
    state = RECEIVING_ACK;
   }
   else if(byte == 'N') // start of a NCK?
   {
    packet_char_count = 2;
    state = RECEIVING_NCK;
   }
   break;
  case DETERMINING_PACKET_TYPE:
   if(byte == 'T') // are we receiving a "t packet"?
   {
    packet_buffer_index = 0;
    state = RECEIVING_T_PACKET;
   }
   // unknown packet type;
   //go back to the unsynchronized state
   else
   {
    state = UNSYNCHRONIZED;
   }
   break;
  case RECEIVING_T_PACKET:
   // still building the packet?
   if(packet_buffer_index < sizeof(T_Packet_Data_Type))
   {
    // move packet character to our buffer
    packet_buffer[packet_buffer_index] = byte;
    packet_buffer_index++;
   }
   // complete packet?
   if(packet_buffer_index == sizeof(T_Packet_Data_Type))
   {
    T_Packet_Data.mx = packet_buffer[0];
    T_Packet_Data.my = packet_buffer[1];
    T_Packet_Data.x1 = packet_buffer[2];
    T_Packet_Data.y1 = packet_buffer[3];
    T_Packet_Data.x2 = packet_buffer[4];
    T_Packet_Data.y2 = packet_buffer[5];
    T_Packet_Data.pixels = packet_buffer[6];
    T_Packet_Data.confidence = packet_buffer[7];
    camera_t_packets++;
    // we're done; go back to the unsynchronized state
    state = UNSYNCHRONIZED;
   }
   break;
  case RECEIVING_ACK:
   // second character a C?
   if(packet_char_count == 2 && byte == 'C')     {
    packet_char_count++;
   }
   // third character a K?
   else if(packet_char_count == 3 && byte == 'K')
   {
    packet_char_count++;
   }
   // fourth character a return?
   else if(packet_char_count == 4 && byte == '\r')
   {
    camera_acks++;
    state = UNSYNCHRONIZED;
   }
   else
   {
    state = UNSYNCHRONIZED;
   }
   break;
  case RECEIVING_NCK:
   // second character a C?
   if(packet_char_count == 2 && byte == 'C')     {
    packet_char_count++;
   }
   // third character a K?
   else if(packet_char_count == 3 && byte == 'K')
   {
    packet_char_count++;
   }
   // fourth character a return?
   else if(packet_char_count == 4 && byte == '\r')
   {
    camera_ncks++;
    state = UNSYNCHRONIZED;
   }
   else
   {
    state = UNSYNCHRONIZED;
   }
   break;
 }
```

How to Start

- Probably the best way to design a state machine is to start with a pencil & paper diagram.
  1) What is the starting state? Draw a circle (or box) & label it.
  2) If everything works perfectly, what state do I want to go to next? What inputs or conditions tell me what to do next? Draw & label the 2nd state and draw a line with an arrow (telling direction) what condition causes you to go from the initial state to the 2nd state.
  3) If the input is bad, what state do I go to? Do I stay where I am? Draw error state & arrow.
  4) Repeat steps 2 & 3 for all the states.
  5) What is the final state? Is it just the initial state again?
- Note: you can go to the same state or a different state.
- If you are in state "A," the same input (or conditions) should always get you to state "B."
- Once you're sure you've identified all the states & all the transitions, you can create "if" or "switch" statements to implement the model in code.