More NQC

Using Rotation Sensors

- To make robot move in a straight line:
  - Can measure speed of rotation of each motor
  - Adjust if not the same
  - Example program:
    - Rotation_sensor

Subprograms

- Subroutine
  - Code that can be executed from many places in a program
  - Like procedures, but with restrictions
    - Up to 8 allowed
    - No parameters, no result returned
    - Cannot be nested
    - No recursive calls
    - Risky to call from different tasks
    - Code is only stored once, so efficient use of memory
  - Defining:

    sub sub_name() { };
  - Example: 6_subs
    - Main calls a subroutine that makes RCX turn 360 degrees several times

Inline Functions

- More like C functions
  - No return value (type void)
  - Can have value and reference parameters
  - Each time invoked a new copy of code is generated
    - Can use a lot of memory
  - No limit on number of inline functions
- Defining:

  void function_name(parameters) { };
  - Just like in C
- Example programs:
  - 6_inline2 (parameter is value of turn time)
  - 6_inline_by_ref)
    - Reference parameter increments n, which is used in caller for delays between outputting a sound

Macros

- Give small pieces of code a name
- Like inline functions in that each time invoked a new copy of the code is generated
- Can have arguments
  - Just placeholders for values to be used when invoked
- Defining:

  #define macro_name(argument_list) statements;
  - If more than one line is needed, must use '\' at end of line
- Example program: 6_macro
  - Power & time are arguments to forwards(s,t), backwards(s,t), turn_right(s,t), turn_left(s,t) macros

RCX Timers

- Four of them
  - Count from 0 to 32767 in 1/10 second increments
  - Then rollover to zero
  - Reading a timer:

    x = Timer(n)
  - Resetting a timer:

    ClearTimer(n) // Reset to zero

    SetTimer(n, value) // Reset to specified value
- Timers can also be read more precisely

  x = FastTimer(n) // 1/100 sec. (10 msec.) Intervals
- Example program: 12_timers
  - Go forward & turn randomly until timer times out

LCD Display

- RCX LCD has 8 display modes
  - DISPLAY_WATCH show system time, default
  - DISPLAY_SENSOR1 show value of sensor 1
  - DISPLAY_SENSOR2 show value of sensor 2
  - DISPLAY_SENSOR3 show value of sensor 3
  - DISPLAY_OUT_A show setting for output A
  - DISPLAY_OUT_B show setting for output B
  - DISPLAY_OUT_C show setting for output C
  - DISPLAY_USER show something else
- Set mode with SelectDisplay(mode)

LCD DISPLAY_USER Mode

- Continually read a source & update LCD display with value
  - Source can be a sensor, timer, global variable, etc.
  - Can display values with a decimal point
    SetUserDisplay (source, digits-after-dec-point)
- Example Programs:
  - timer_display, timer_display_ok

IR Communication

- RCX can send/receive messages using its IR port
- Message values: 0 to 255
- To retrieve most recently-sent message #:
  x = Message(); // 0 returned → no message received
- Sending a message:
  SendMessage(msg_number)
  - Receiving is disabled while sending
- Clearing the RCX's message buffer:
  ClearMessage();
- Example programs:
  - 11_Master, 11_Slave
    - Master RCX sends out messages to tell slave to go forward, backward, or stop
  - 11_leader
    - Robots decide who is master and who is slave

Proximity Sensor using IR

- Make robot react before bumping something
- Use IR communication port in conjunction with a light sensor
  - Light sensor emits/detects red and IR "light"
  - One task sends out IR message
  - Another task measures change in "light" (IR) intensity reflected back to light sensor
    - Detects it, detects it again and computes change
    - Large change → close; Small change → far
  - Example program: 9_proximity

Serial Transmission of Data Using IR Port

1. Set up serial communications Protocol
   SetSerialComm(SERIAL_COMM_DEFAULT);
   - 2400 baud, 50% duty cycle, 38 kHz carrier wave
     - Could be: SERIAL_COMM_4800
     - SERIAL_COMM_DUTY25, SERIAL_COMM_76KHZ
       - Boolean OR combinations
2. Set Up Packets (how to package data bytes)
   SetSerialPacket(Serial_PACKET_DEFAULT);
   - No packets, just data bytes
     - There are other possibilities, e.g.,
     - SERIAL_PACKET_RCX (RCX format with checksum)

3. Put bytes into serial transmit buffer (max=16)
   SetSerialData(index,value)
   - Index 0-15
   - Packets are built first
4. Send bytes in the buffer
   SendSerial(start_index, count);

- Reading a given byte from the buffer
  x = SerialData(i);

Arrays

- Maximum size = 32
- Declare just as in C
  - int my_array[4];
- No bounds checking is done

Data Logging

- RCX can store data in a "datalog"
  - From sensors, timers, variables, etc.
- Can be uploaded to a host computer
  - CreateDatalog(const size); // to create it
    - Uses same 6K RAM as programs
    - Each point logged uses 3 bytes
    - This instruction erases previous data
  - AddToDatalog(x); // to add data to it
    - x can be a variable, sensor value, timer value, etc.
  - UploadDataLog(start_index, count);
    - Not very useful since host computer usually initiates the upload of data
- Example program: datalog
  - Use BricxCC Datalog tool to look at data retrieved

Interference Between Tasks

- Program: 10_wrong
  - Task move_square() makes robot move in square
    - While turning enters into a Wait()
  - Task check_sensors() checks for bumper hit and backs up and turns away
    - While backing up enters into a Wait()
  - Everything is OK unless bump occurs while turning
    - Instead of turning away, it moves forward & bumps obstacle again
- While check_sensors is sleeping, move_square() is still running; so when check_sensors wakes up move_square() drives it forward into obstacle again
- Both tasks are driving motors at cross purposes
- One solution: make sure only one task is driving the motors at any time
  - Program: 10_stopping

- But there's still a problem
  - When move_square() restarts, it starts at the beginning
  - OK for small tasks, but we really should stop and resume at the same place in the task
  - One way to assure that happens: use a semaphore
- Semaphore – a global variable accessed by both tasks
  - Semaphore = 0 → no task is driving motors
  - Semaphore = 1 → a task is driving motors
- When a task wants to use the motors, execute following code:

```
until (semaphore == 0);
semaphore = 1;
// Use the motors
semaphore = 0;
```

- Program: 10_semaphore

NQC Access Control

- Setting task priorities for accessing resources
- Automates and generalizes the idea of semaphores
- Allows a task to request ownership of a resource
  - Motor, speaker, or a user-defined resource
- Code in a task:

```
acquire(list of resources)
   { body } // If resource is not owned by a higher-priority task
            // the task gets the resource & the body executes
catch
   {  };    // If resource is owned or taken away by a higher-
            // priority task, this task doesn't get the resource
            // body doesn't execute, & catch block executes
```

Access Control Resources

- Motors: ACQUIRE_OUT_A
  - Same for B and C
- Speaker:
  - ACQUIRE_SOUND
- User-defined resources
  - ACQUIRE_USER_1
    - Same for 2, 3, 4
  - Each is like a token
    - The task that has it runs
- Difference:
  - When ownership of motor is lost, default action is to stop motor
  - When ownership of speaker is lost, sound is turned off
  - No default action for user-defined resource

Setting Task Priorities in Access Control

SetPriority(priroity_level);

- 0 to 255
- lower values higher priorities
- Use at the top of a task

- Example program:
  - 10_acquire_usr

Event Monitoring

- Like using interrupts instead of polling sensors
- 16 types of events can be monitored and responses programmed (See NQC documentation for types)
  1. Set up event numbers
     - i.e., associate event #'s with event sources & types, e.g.,

     ```
     SetEvent(1, SENSOR_1, EVENT_TYPE_PRESSED);
     SetEvent(2, SENSOR_1, EVENT_TYPE_RELEASED);
     ```

  2. Monitor those events

     ```
     monitor (EVENT_MASK(1) + EVENT_MASK(2))
        {Normal code when events have not occurred}
     catch (EVENT_MASK(1))
       {event 1 handler code}
     catch (EVENT_MASK(2))
       {event 2 handler code}
     ```

- Example Pgm: events_two_touch_sensor

Range Event Types & Hysteresis

- Some sensors & event sources need to work with a range of values
  - Want to detect two threshold levels
  - E.g., light sensor trying to follow edge of a black zone
    - Take black = 40, white = 60
    - If sensor is between, go forward
    - If > 60 turn back toward black area (one way)
    - If < 40, turn away from black area (other way)
- Range events
  - Specify upper & lower limit for event with:
    - SetUpperLimit(event #, value)
    - SetLowerLimit(event #, value)
      - EVENT_TYPE_HIGH: when source enters high range
      - EVENT_TYPE_LOW: when source enters low range

Hysteresis

- But could have problems with sharp thresholds
- Sensors don't react instantaneously and there can be small errors in the readings (jitter)
- So use different cutoffs for entering and leaving the normal range
- Difference between two thresholds called hysteresis
  - Example w/o hysteresis: Upper Limit 60 (spurious value 58->61)
    - 52 55 58 60 (Event triggers a desired response) 59 58 61 57 (spurious 61 reading triggers another undesired response)
  - Same example with a hysteresis of 5:
    - 52 55 58 60 (Event triggers desired response) 58 61 (difference < hysteresis so no reaction) 59 58 55 52 49 53 57 62 (Event triggers desired response again)
- Set_Hysteresis(Event #, value)
- Event_hysteresis example program