Translating a Behavior Requirement into Code
Terry L. Grant
NASA, Ames Associate, grant@email.arc.nasa.gov

Translating a Behavior Requirement into Code

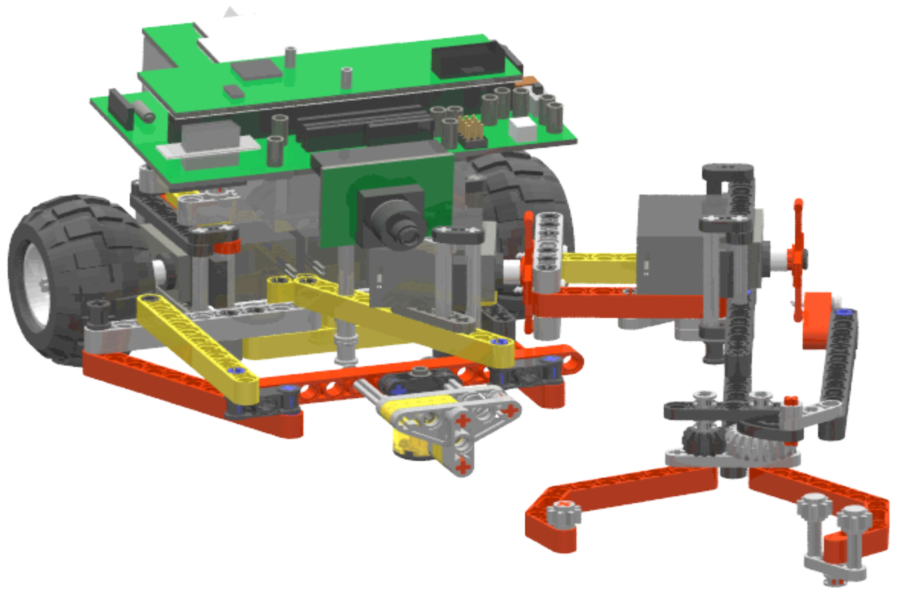
Introduction

This is both fundamental to robotics and also one of the least understood or discussed efforts. It requires complete understanding of the capability of your robot and full understanding of the microcontroller and its programming environment. We will first discuss the translation in the context of a couple examples as a stepwise conversion, then as a more realistic development process, and then as refinement into a code function that can be reused for similar behavior requirements. Defining behavioral requirements as written statements is not easy; usually it is much easier to show the behavior with a 'bot' model than to describe it. That is where the development cycle is needed. Often the testing of the first draft code translation results in refinement of both the code and the requirements. It can also lead to expanded behavioral capability or fully revised code design. Then, as the developer becomes familiar with the behavior, if time permits, the requirements can be generalized and the code parameterized into a 'component' or function that has broader utility, if fully documented.

Translation steps

The first issue is the need for a clear, fully defined behavior. This need is often overlooked or under emphasized by an instructor who is perhaps *too familiar* with the robotics controller and the robot structural design. We usually describe behavior in terms of actions and robots with which we are most familiar. Lets start with a simple example and its translation:

Requirement: Explicit - Move the robot forward (at full speed) for 3 seconds.
Implicit – Robot is similar to the starter bot with driver motors connected to each wheel and a skid in front. (see figure below)



Code design:

1.motors are turned on
–Wiring is adjusted so that + translates to ‘forward’ [be sure that motor ports are connected as desired]
–Power set to full or 100% for max speed
2.Set timer and wait for 3 seconds
3.Turn off drive motors

Actual code in Interactive C statements:

1. Call the library function to power motors:
   motor(1, power); motor(2, power);
   •Assume port 1 drives the right wheel; port 2 drives the left wheel
   •Set the ‘power’ to 100 (maximum)
2. Wait
   •Set a timer to zero, then start.
   •Wait for timer value (t) to equal 3 seconds, then continue.
   •We find that the **sleep(t);** library function does all this, where ‘t’ is 3.0, a floating point number.
3. Turn off
   •We find another library function does this:
   • **alloff();** or **ao();**

Development
Even for this simple example we don’t know if we have a good translation until we open the IC environment and write a program with these statements and download it to the XBC, then run the XBC on the robot several times to see what happens. As we run this test at least a few novices are likely to see their robot “turns-in-place” rather than move forward.

If this happens we can assume that one motor is connected backward, and, instead of guessing which one, we can use the XBC to diagnose the problem. The XBC O.S. actually computes the position in ‘ticks’ of each connected motor. If we push a shoulder button twice the motor positions are shown on the bottom two lines of the display, called the “status window”. By moving the robot forward a short distance, we can see that one motor position is decreasing or going negative. If we reverse the connection for that motor, we will now see the robot move forward. Now that the proper connections are identified we can mark each connector with the port number on one side and thus fully identify how to make the connections.

If the robot still turns slightly to the left or right, the motor on the inside of the turn is moving slower, probably because that wheel is rubbing or an axle has too much friction.
Try mechanical adjustments, and if it seems that one motor is just weak, swap it out. If all else fails, we can experiment with changing the motor() call to power the other motor at a slightly lower value.

Translation example #2
While the simple example illustrates the process, it is not a very useful in a broader context, such as action in a game. The second example is more generally applicable:

Requirement: Explicit - Move the robot forward 12" and then turn left 90 degrees.
Implicit - Robot is similar to the starter bot with driver motors connected to each wheel and a front skid. The distance between the centers of the wheels, r, is 4.5". The motors generate t (~1100) ticks/rev. of the wheel. The circumference (one rev. or 2 pi radians) of the wheel is c ~6.9").

Code design for example #2:

1.Use 'move_relative_position' (or mrp) with XBC to rotate drive motors equivalent to 12" of linear distance.
2.Wait for motors to finish rotation.
3.Assuming 'drive steering', use mrp to rotate the right motor equivalent to a 90 degree left turn.
4.Wait for the motor to finish its rotation.

Actual code in Interactive C statements:

1. Call mrp function for each drive motor:

mrp(1,speed,position); mrp(2,speed,position);

Assumes motor ports are still #1 for right, #2 for left wheel.
Make speed an adjustable constant, since it isn't explicitly required.
To convert linear distance to position 'ticks'
–Measure ticks/revolution of a wheel (This includes any gear ratio between the motor and the wheel.)
–Measure circumference of the wheel (or 2 *pi*radius) by rotating it once on a surface and measuring the forward distance (measure in the same units as the desired distance).
–If 'v' represents the number of position ticks (a long integer) in the mrp function, 'c' is the wheel circumference, 'd' is the desired distance (12"), and 't' is the ticks/rev, then:

position, v = t*d/c

and using the values in the requirements:
v= 1100*12/6.9 or v= 1913L
As a check see that the units match: $\frac{\text{ticks/rev*inches}}{\text{(inches/rev)}}$ = ticks

2. Call 'block_motor_done' (bmd) for each motor

Bmd(1); bmd(2);

3. Call mrp for the right motor

mrp(1, speed, position);

To calculate the distance (d) equivalent to a 90 deg (pi/2 rad) turn:
–Measure the distance between the wheels as the turn radius (r)
–The circumferential distance (d) for an angle of pi/2 is just d = r*pi/2
Using the formula from step 2, the position in ticks is then

v = t*d/c, or v = t*r*(pi/2)/c

and again using the values in the requirements:
v=1100*4.5*(3.14/2)/6.9 or v=1126L

4. Call 'block_motor_done' (bmd) for motor 1

bmd(1);

Development of example #2

As we test this code translation, we can assume that motor connection issues have been resolved. Once again we open the IC environment and write a program with these statements and download it to the XBC, then run the XBC on the robot several times to see what happens. To follow good coding practice, we should copy the requirements and some design computations into comments at the beginning as below:

```
/* Requirement: Explicit - Move the  robot forward 12" and then turn left 90
degrees.
Implicit - Robot is similar to the starter bot with driver motors  connected
directly to each wheel and a skid in front.
The distance between the center of the wheels, r, is 4.5"
The motors generate t (~1100) ticks/rev. of the wheel.
The circumference (one rev. or 2 pi radians) of the wheel is c ~6.9").
        - fwd_v = t*d/c = 1100*12/6.9
        - turn_v = t*r*a/c =1100*4.5*(3.14/2)/6.9, where a is the turn
         angle, in radians
*/
#define SPEED 500
#define FWD_V 1913L
#define TURN_V 1126L

void main(){
        mrp(1,SPEED,FWD_V); mrp(2,SPEED,FWD_V);
        //assumes motor ports are still #1 for right,   #2 for left wheel.
        bmd(1); bmd(2);
        mrp(1,SPEED, TURN_V);
        bmd(1);
}
```

After download to the XBC, run the robot several times to see if it behaves reliably as expected. If the forward distance or the turn-angle does not match, check to see if your wheels have the assumed circumference. If not adjust the FWD_V and TURN_V constants proportionately and retry. Then retry the test runs with maximum speed, and with half the nominal speed. Does the robot still behave as required? In computing the distances-to-tick conversion, we conveniently assumed that robot acceleration could be achieved without slippage, and that deceleration distance was negligible. At some point these assumptions will not be good, and the position values may require adjustment. Also, if the robot is nearly balanced on the wheels it will lift off the skid on startup, and may not move in a repeatable manor. In that case an additional set of statements may need to be added to slow the startup. The following calls cut the acceleration in half: **set_motion_acceleration(1, 2500); set_motion_acceleration(2, 2500);** Along with this added code, the requirements should state: "Cut acceleration in half to improve startup stability."

While running the development tests, the question might arise as to how the robot should behave if it bumps into something while moving forward. Maybe the added requirement should be (2a) "If front sensor makes contact, stop and wait for A-button."

Code Design for Example #2a

1. Call move_at_velocity, mav() functions for the drive motors, and clear_motor_position_counter() then loop while the get_motor_position_counter() for one motor is less than v, and test the front bumper switch. If front bumper is contacted, stop and wait for A-button push.
2. Continue to wait while the motor position counter is less than v, then stop both motors.

Actual code in Interactive C statements for #2a:

1. would change to:

```
clear_motor_position_counter(1);
mav(1, 500); mav(2, 500);
while(get_motor_position_counter(1) < FWD_V){
        if(digital(15) == 1){ //test for front contact, on port #15
                ao();
                while(a_button()==0); //wait for A-button
                mav(1,500); mav(2, 500); //resume
                break;
                }
        }
```

2. would change to:

```
while(get_motor_position_counter(1) < FWD_V);
ao();
```

Development test for #2a

The code should be edited into the previous code, along with the new requirement, and saved with a different file title. The new code is then tested as above for reliability.
Code testing also serves to validate the code design logic. In this case, the original code design did not have a clear motor position call, but assumed that mav would start the motor position at zero (it doesn't). After the test failed, the clear_motor_position_counter() function was added to the design and the actual code statements.

Creating Components or Functions

After developing a behavior, we often can see a general use for it if we just make it a component or function with inputs and/or outputs. The two parts of the above behavior lend themselves to creating reusable functions: 'Go_fwd', and 'Turn', each with and single input and a common variable for speed, or with a pair of inputs for speed and distance, and speed and turn angle. Very little is needed to change the requirements: Just make the distance or turn angle a variable. The code design just adds a statement for the math to convert inches-to-ticks or angle-to-ticks respectively. Here are the functions:

```
/* Go_Fwd Requirement:
Explicit - Move the  robot forward a distance, d, in inches at a speed
(0-1000) in ticks/sec.
Implicit - Robot is similar to the starter bot with driver motors  (#1 & 2)
connected directly to each wheel and a skid in front.
```

```
The motors generate t (~1100) ticks/rev. of the wheel.
The circumference (one rev. or 2 pi radians) of the wheel is c ~6.9").
        The position in ticks is then v = t*d/c = 1100*d/6.9 = d*160
*/
void Go_Fwd(int speed, int x){
        long v =160L*(long)x; //see assumptions above
        mrp(1,speed,v); mrp(2,speed,v);
        //assumes motor ports are still #1 for right,   #2 for left wheel.
        bmd(1); bmd(2);
}
```

```
/* Turn Requirement:
Explicit - Turn left at an angle, in degrees, at a speed (0-1000) in ticks/sec.
Implicit - Robot is similar to the starter bot with driver motors  connected
directly to each wheel and a skid in front.
The distance between the center of the wheels, r, is 4.5"
The motors generate t (~1100) ticks/rev. of the wheel.
The circumference (one rev. or 2 pi radians) of the wheel is c ~6.9").
        The turn position, in ticks is then v = t*r*a/c, where a is the turn
        angle, in radians, or if a is in degrees then
        v =1100*4.5*a(3.14/180)/6.9 = a*12.5
*/
void Turn(int speed, int a){
        long at = (long)((float)a*12.5);
        mrp(1,speed, at);
        bmd(1);
}
```

Here is a sample main program that tests the functions to drive the robot:

```
 //forward, fancy turn and return, values in inches & degrees
void main(){
   Go_Fwd(SPEED, 12); Turn(SPEED, -45);
   Go_Fwd(SPEED,3); Turn(SPEED, 270); Go_Fwd(SPEED,3);
   Turn(SPEED, -45); Go_Fwd(SPEED,12);
}
```

Conclusion

We have explored the process of translating behavior requirements in code for use with the XBC Controller and a Botball starter robot. The examples show a process of writing clear requirements, then a code design with the IC functions for the XBC in mind. Then actual code is written in the Interactive C environment, checked for syntax, downloaded and tested on the target robot. The initial tests seldom go exactly as planned, and that leads to changes to accommodate physical properties, or errors in logic. It also can lead to expanded requirements and new developments. Finally, when the behavior and coding is well understood, the behavior can sometimes be encapsulated in a generalized function to be reused for more complex programmed behavior.
