THE SPIN LANGUAGE & PropBot Programming

Chapter 3

Objectives

- Learn about the Propeller chip and the Spin programming language
- Understand how to program the PropBot
  - Using the sensors and servos
- Understand how to use the robot-to-PC communications software (RobotTracker v4.0) to coordinate code between the PC, the robot and the webcam-based tracking system.

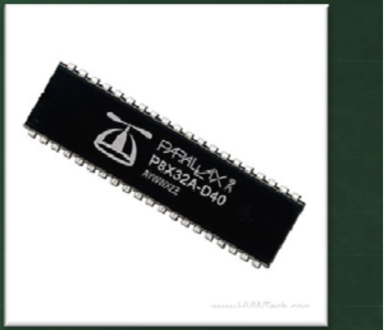
What's in Here ?

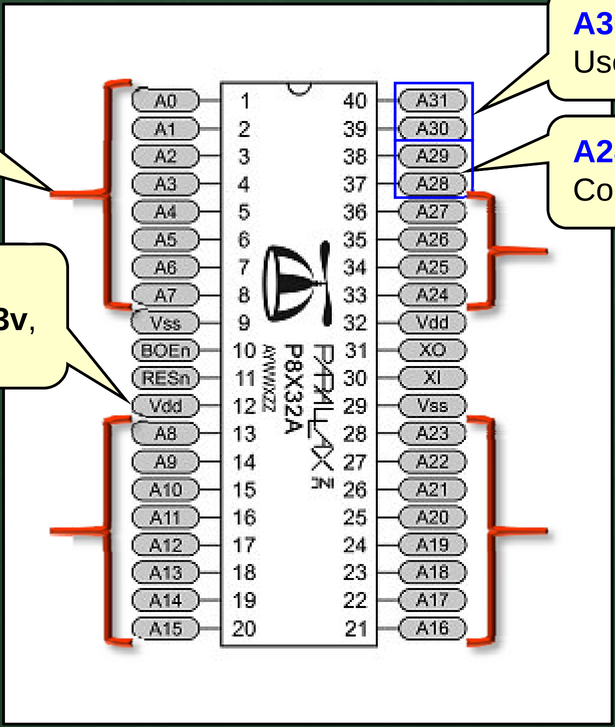
- The Propeller Chip & Spin Language
  - Propeller Chip, Propeller Tool IDE and Spin
  - Memory and Variables
  - Math Functions
  - Control Structures
  - Debugging
- Spin Programming Examples
  - Reading Sensors
  - Servo Control
- "Robot Tracker" Software
  - GUI and Settings
  - Tracing a Robot's Movements
  - Wireless Debugging
  - Trace Files
  - Mapping

The Propeller MicroProcessor

The Propeller

- The microprocessor that we will use is called the **Propeller**:





A0 – A31
32 general purpose I/O pins.

VDD
Runs on **3.3v**, not 5v.

A30,A31
Used for serial I/O (i.e., programming)

A28,A29
Connects to external EEPROM

#PROCESSORS = 8

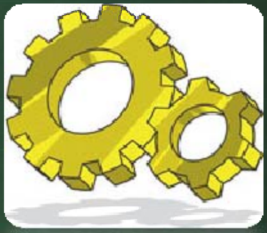
RAM = 32k, **ROM** = 32k

CLOCK SPEED = 12Mhz

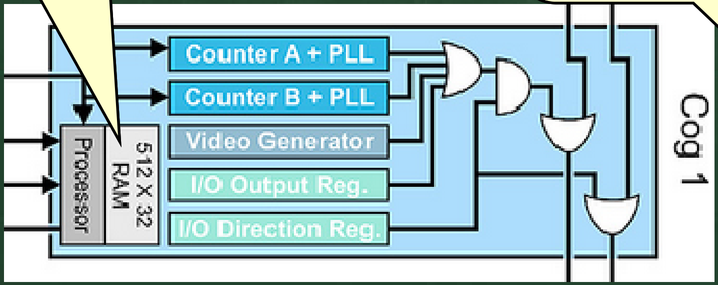
I/O PINS = 32

8 Processors

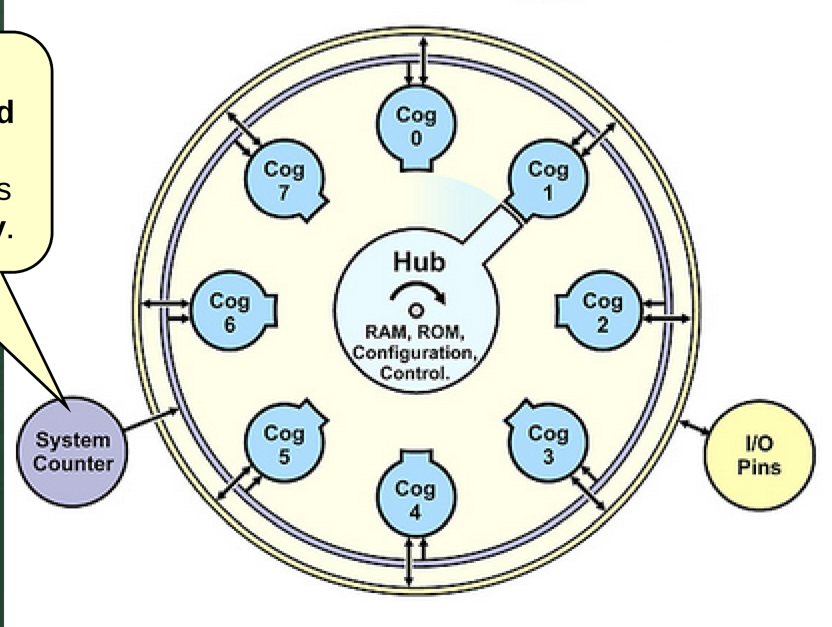
- The Propeller has 8 processors (called *cogs*):
  - "True" multitasking (parallel processing)
  - Shared memory (round robin fashion)
  - Shared I/O pins



2k bytes of **individual memory** per processor



Cogs all run at the **same speed** and execute their instructions **synchronously**.

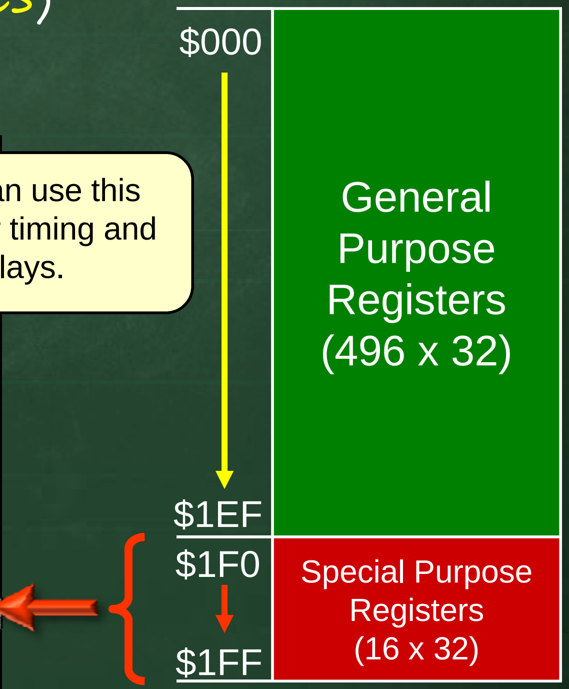


Cog memory

- Each Cog has a small amount of "local" memory:
  - 496 x 32bit words
  - faster than shared memory (i.e., access to shared memory can take anywhere from 7 to 22 clock cycles, whereas access to local memory takes at most 4 clock cycles)
  - use it for local variables & stack space

ADDRESS	NAME	TYPE	DESCRIPTION
$1F0	PAR	Read-Only	Boot Parameter
$1F1	**CNT**	**Read-Only**	**System Counter**
$1F2	INA	Read-Only	Input States for P31 - P0
$1F3	INB	Read-Only	Input States for P63- P322
$1F4	OUTA	Read/Write	Output States for P31 - P0
$1F5	OUTB	Read/Write	Output States for P63 – P322
$1F6	DIRA	Read/Write	Direction States for P31 - P0
$1F7	DIRB	Read/Write	Direction States for P63 - P322
$1F8	CTRA	Read/Write	Counter A Control
$1F9	CTRB	Read/Write	Counter B Control
$1FA	FRQA	Read/Write	Counter A Frequency
$1FB	FRQB	Read/Write	Counter B Frequency
$1FC	PHSA	Read/Write	Counter A Phase
$1FD	PHSB	Read/Write	Counter B Phase
$1FE	VCFG	Read/Write	Video Configuration
$1FF	VSCL	Read/Write	Video Scale

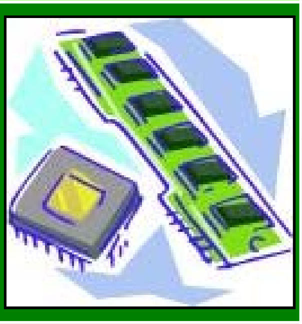
Can use this for timing and delays.


$000
General Purpose Registers (496 x 32)
$1EF
$1F0
Special Purpose Registers (16 x 32)
$1FF

Main (shared) Memory

- Available memory is 32k bytes:
  - All cog programs must fit in the 32k byte memory
  - its actually a lot of space (we used less than 2k altogether with the BS2 chip previously)
  - Tips:
    - don't allocate huge arrays
    - use registers when possible
    - re-use variables

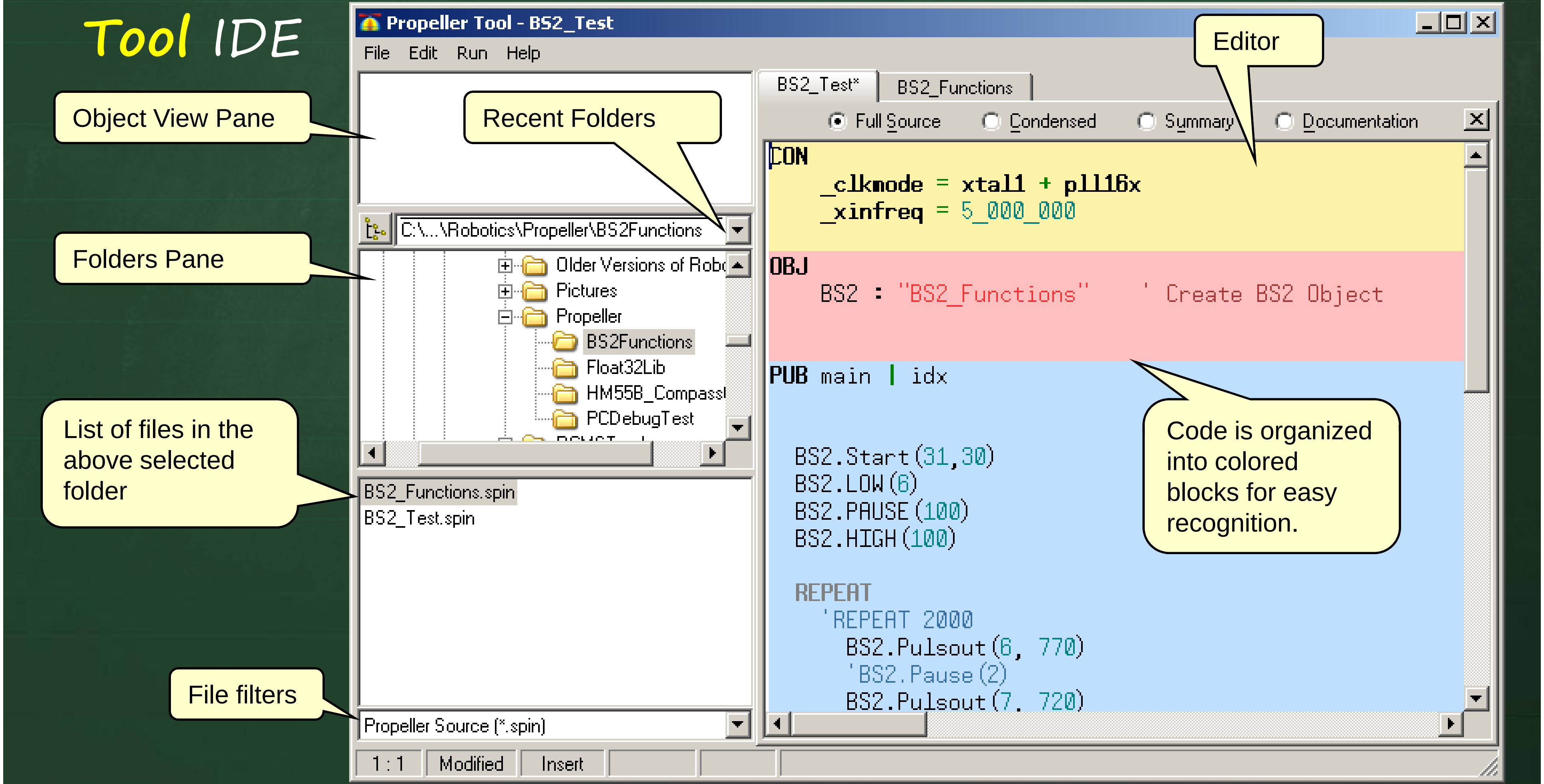
$0000	$000F	Reserved (16 bytes)	RAM (32k)
$0010	$7FFF	Your Code (32,752 bytes)	
$8000	$BFFF	Character Set (16,384 bytes)	ROM (32k)
$C000	$CFFF	Log Table (4,096 bytes)	
$D000	$DFFF	Anti-log Table (4,096 bytes)	
$E000	$F001	Sine Table (4,098 bytes)	
$F002	$FFFF	Boot Loader & Interpreter	



Programming Using SPIN

The Propeller Tool IDE

- Our robots will be programmed using the **Propeller Tool IDE**



```
CON
    _clkmode = xtal1 + pll16x
    _xinfreq = 5_000_000

OBJ
    BS2 : "BS2_Functions"     ' Create BS2 Object

PUB main | idx


     BS2.Start(31,30)
     BS2.LOW(6)
     BS2.PAUSE(100)
     BS2.HIGH(100)

     REPEAT
       'REPEAT 2000
         BS2.Pulsout(6, 770)
         'BS2.Pause(2)
         BS2.Pulsout(7, 720)
```

Object View Pane

Recent Folders

Editor

Folders Pane

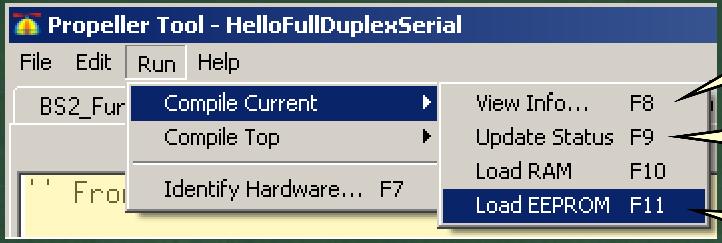
List of files in the above selected folder

Code is organized into colored blocks for easy recognition.

File filters

Downloading Your Code

- After you write your code …

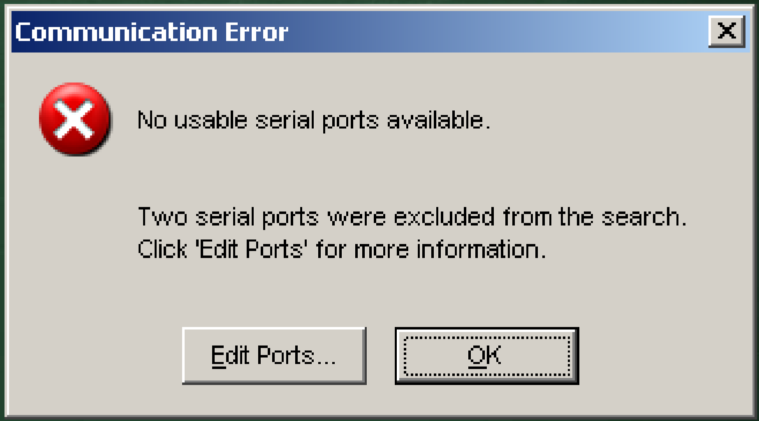


Select this or press **F8** to see how much space your program takes up.

Select this or press **F9** to compile without downloading.

Select this or press **F11** to download your code onto the robot.

- If this window appears, then you forgot to …
  - setup the COM port,
  - turn on & plug in the robot, or
  - disconnect from the Serial Terminal

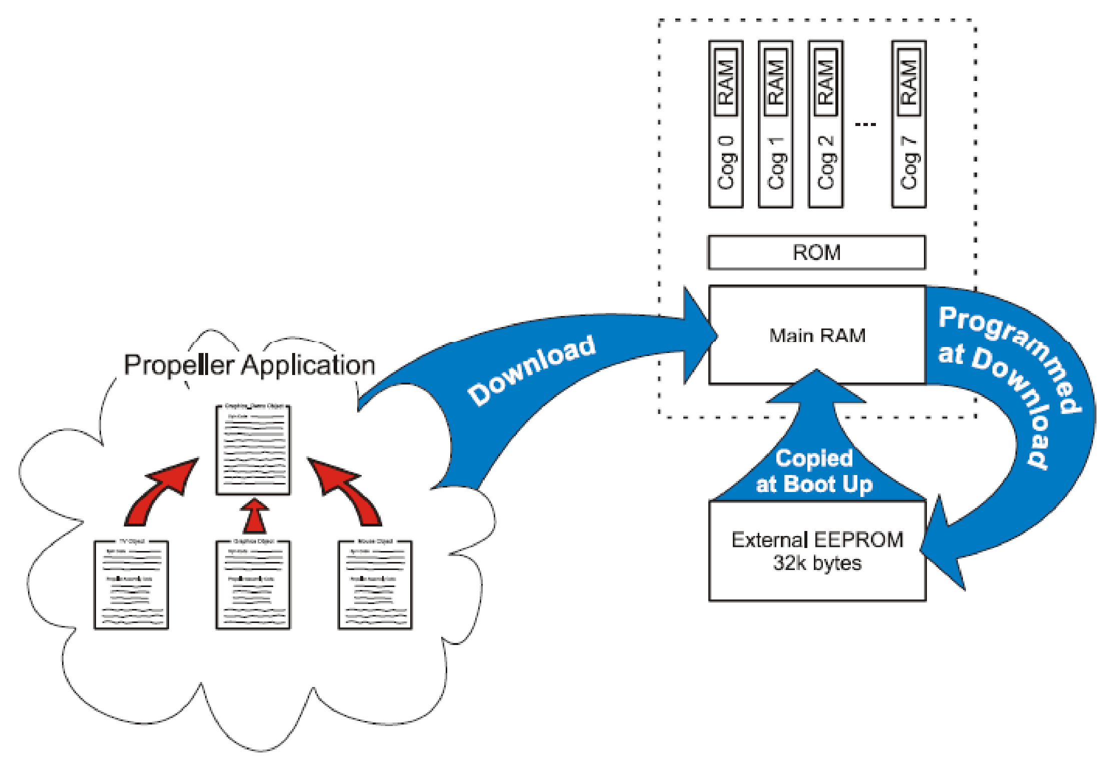


No usable serial ports available.

Two serial ports were excluded from the search.
Click 'Edit Ports' for more information.

Downloaded Code

- Downloaded code is stored onto EEPROM
- When the robot is turned on or reset, the EEPROM program is loaded onto RAM and then run.

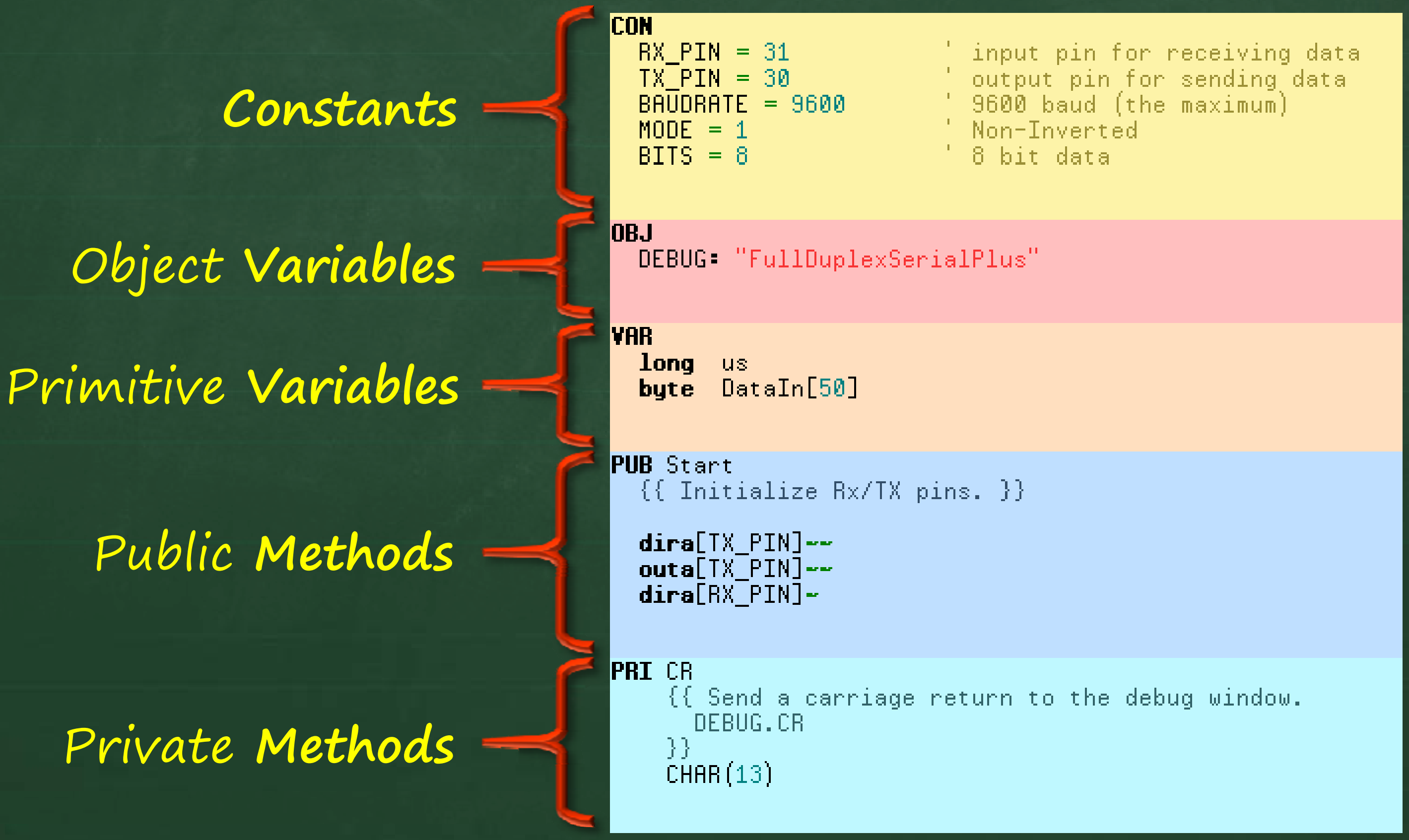


Spin Code

- Propeller code can be written in:
  - SPIN (i.e., an object-oriented interpreted language).
  - Assembly language (yech!)
- Spin code is organized into objects
  - Like JAVA, each .spin file defines an object
- Executable programs must have a public main method.
- Spin has no debugger or console screen:
  - We will use a pre-defined object that allows serial I/O through either a USB connection to the PC or through the wireless bluetooth device

SPIN Programs

- Spin code is automatically arranged into “colored” sections:



Constants

Object Variables

Primitive Variables

Public Methods

Private Methods

```
CON
  RX_PIN = 31                ' input pin for receiving data
  TX_PIN = 30                ' output pin for sending data
  BAUDRATE = 9600            ' 9600 baud (the maximum)
  MODE = 1                   ' Non-Inverted
  BITS = 8                   ' 8 bit data

OBJ
  DEBUG: "FullDuplexSerialPlus"

VAR
  long  us
  byte  DataIn[50]

PUB Start
  {{ Initialize Rx/TX pins. }}

  dira[TX_PIN]~~
  outa[TX_PIN]~~
  dira[RX_PIN]~

PRI CR
    {{ Send a carriage return to the debug window.
      DEBUG.CR
    }}
    CHAR(13)
```

Constants

- Constants may be:
  - Integers (booleans and characters are integers too)

    e.g.,
    ```
    outPin = 21
    ```
  - Floats

    e.g.,
    ```
    scale = 1.5
    ```
  - Expressions (must be constant algebraic)

    e.g.,
    ```
    number = 32.05 * 18.1
    ```
- They can be declared on the same line:

  e.g.,
  ```
  delay = 500, aChar = "A", baud = 9_600
  ```

_ character can be used to separate groups of 3 digits for integers to make them more readable.

Number Representation

- Numbers are actually represented as
  - decimal (**215**), hex (**$D7**) or binary (**%11010111**).
- Negative numbers stored using two's compliment:

  5 = 00000101_2 (can be represented as 5, $05 or %00000101 in SPIN)

  -5 = 11111010_2 in one's compliment notation (just flip bits)

  -5 = 11111011_2 in two's compliment notation (flip bits & add 1)
- The propeller performs ALL calculations using 32 bits (i.e., longs)
- Even floating point math calculations use longs

Float Constants

- Float constants can be declared directly if shown as real number values (i.e., with a decimal point).

```
CON
  Width = 83.651        ' Height set to float 83.651
```

- Can convert integers to floats and vice-versa:

```
CON
  Height = FLOAT(27)    ' Height set to float 27.0
  Size1 = ROUND(Width)  ' Size1 set to integer 84
  Size2 = TRUNC(Width)  ' Size2 set to integer 83
```

Predefined Constants

- There are some predefined constants:

TRUE	(value is -1 … or $FFFFFFFF)
FALSE	(value is 0 … or $00000000)
POSX	(value is 2,147,483,647 … or $7FFFFFFF)
NEGX	(value is -2,147,483,648 … or $80000000)
PI	(value is 3.141593 … or $40490FDB)

- There are some chip-specific constants as well:

```
_clkmode = xtal1 + pll16x
_xinfreq = 5_000_000
```

… many more …

We will not look into understanding these as they will be fixed for our purposes.

Variables

- SPIN has 3 main types of *integer* variables:

```
VAR
   byte        counter          ' 8-bits
   word        numReadings      ' 16-bits (2 bytes)
   long        timeLapse        ' 32-bits (4 bytes)

   byte        str[24]
   word        positions[100]
   long        averages[10]

   byte        a, b, c
   word        x1, y1, x2, y2
```

Can make arrays of these types.

Can use , to declare more than one on a line.

Variables are:

- Global to the object
- Not accessible outside the object (unless a pointer to its memory is used)

In almost all situations, all variable names must be unique globally, even local variable names!!

Byte Arrays

- Consider arrays of bytes:

```
VAR
   byte     bufferOne[100]
   byte     bufferTwo[100]
```

Use the @ sign to refer to the array's address (i.e., the first byte in the array)

- Can fill in an array of bytes with some value:

```
bytefill(@bufferOne, 0, 100)
bytefill(@bufferTwo+50, 1, 50)
```

Fills in all 100 values with 0

Fills in 2nd half of buffer with 1

- Can also copy bytes from one location to another:

```
bytemove(@bufferOne, @bufferTwo, 100)
```

- Similar commands exist for word and long types:

`wordfill, longfill, wordmove, longmove`

Strings

- Strings are just "arrays of bytes", terminated by 0.
- Can use STRING, STRSIZE, STRCOMP:

```
byte        size
long        myStringPtr
long        yourStringPtr
```

STRING declares a string constant and returns its address.

```
myStringPtr := STRING("Hello World")
yourStringPtr := STRING("Hello There")
size := STRSIZE(myStringPtr)
if STRCOMP(myStringPtr, yourStringPtr)
  '...strings are equal ...
else
  '...strings are not equal ...
```

STRSIZE returns the number of characters in the string number up to but not including the terminating zero byte at the end.

STRCOMP compares two strings for equality. Does not check < or > … just ==.

More Strings

- Strings can also be declared as byte arrays:

```
byte        size
byte        myString[10]
```

You can access each character individually.

```
myString[0]  := "H"
myString[1]  := "e"
myString[2]  := "l"
myString[3]  := "l"
myString[4]  := "o"
myString[5]  := 0
size := STRSIZE(@myString)
if STRCOMP(@myString, STRING("Hello"))
  '...strings are equal
else
  '...strings are not equal
```

Don’t forget to add a zero byte at the end!

Use @ to get the address of the string.

Other Useful Tools

- Lookupz is a useful tool for using fixed lists of data:

```
repeat i from 0 to 6
  temp := lookupz(i: 25, 300, 2510, 163, 17, 8000, 3)
  ' … now do something with temp …
```

temp gets assigned the number in this list at position **i** (starting index = 0)

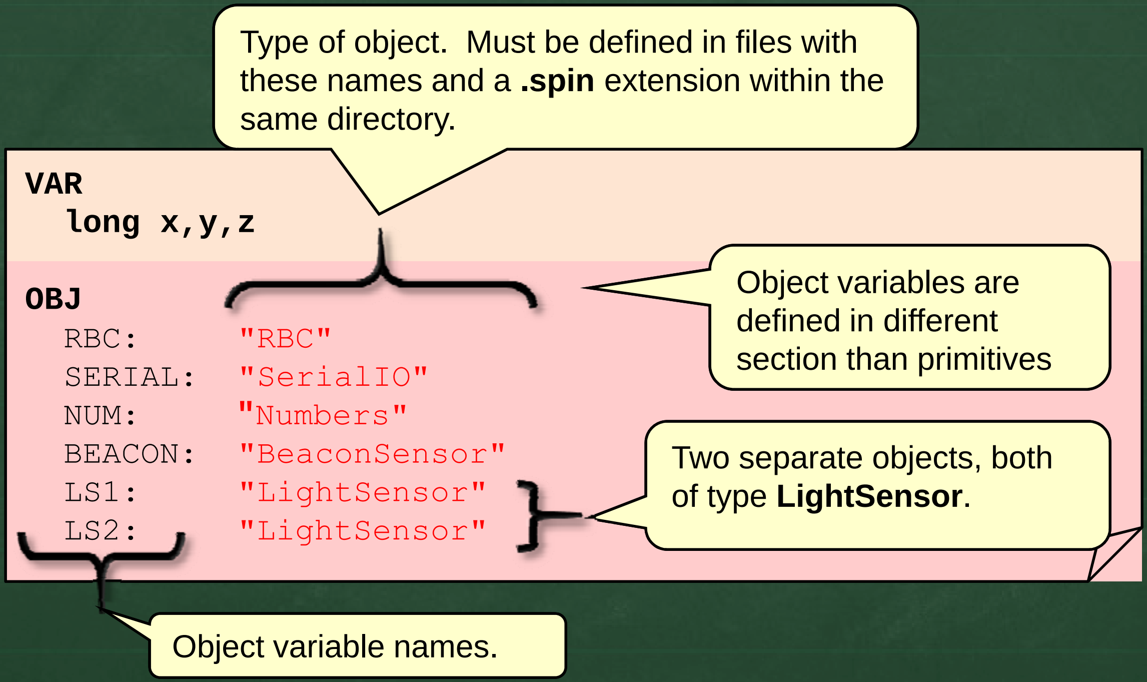
- Lookdownz returns the index of a list item's data:

```
val := 163
i := lookdownz(val: 25, 300, 2510, 163, 17, 8000, 3)
```

i gets assigned the index (starting from 0) of the first number in this list with value **val**. In this case it returns **3**.

Objects

- Spin allows object variables as well:



Type of object. Must be defined in files with these names and a **.spin** extension within the same directory.

```
VAR
  long x,y,z

OBJ
  RBC:      "RBC"
  SERIAL:   "SerialIO"
  NUM:      "Numbers"
  BEACON:   "BeaconSensor"
  LS1:      "LightSensor"
  LS2:      "LightSensor"
```

Object variables are defined in different section than primitives

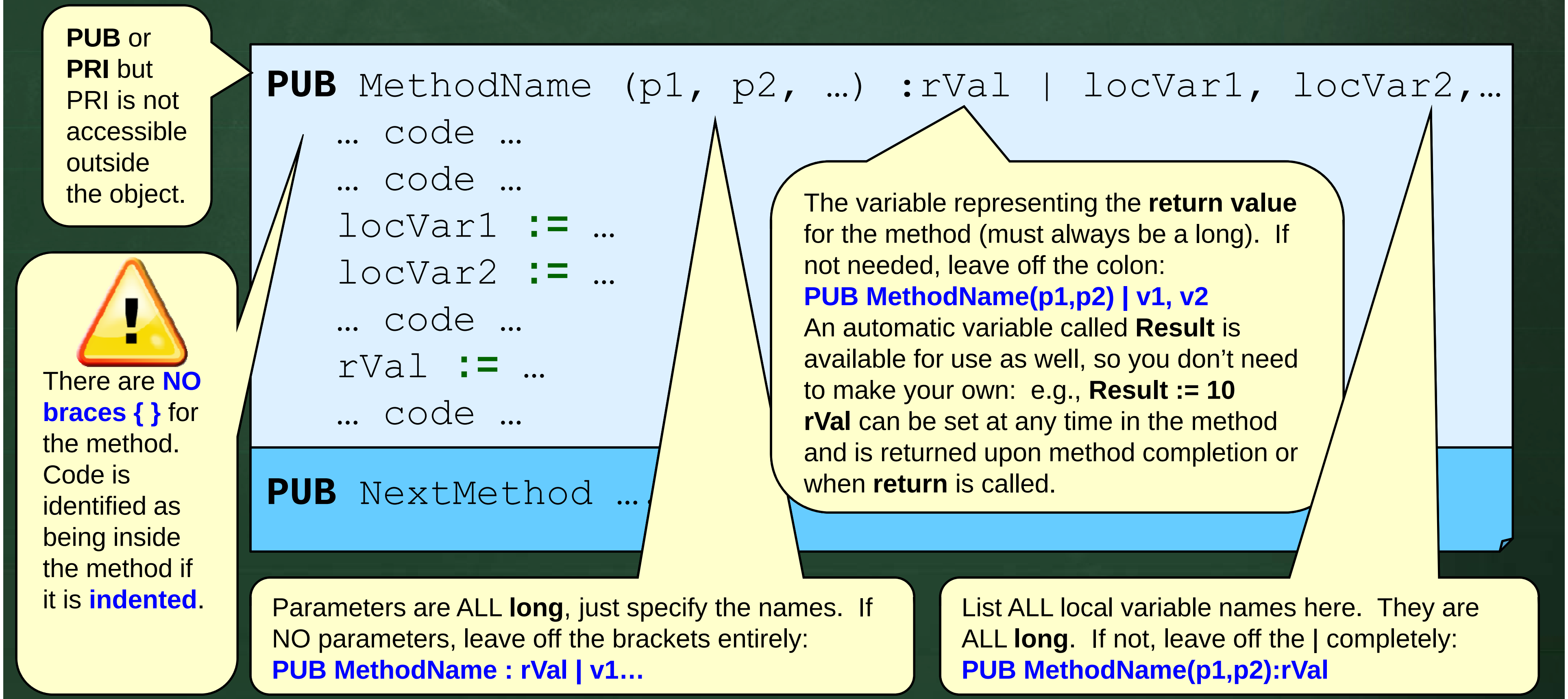
Two separate objects, both of type **LightSensor**.

Object variable names.

- Objects are automatically created upon startup.
  - Although, sometimes they have an **init** or **start** method that needs to be called.

Defining Methods

- Methods may be either PUBlic or PRIvate:
- Here is the general format for all methods:



```
PUB MethodName (p1, p2, …) :rVal | locVar1, locVar2,…
    … code …
    … code …
    locVar1 := …
    locVar2 := …
    … code …
    rVal := …
    … code …

PUB NextMethod …
```

PUB or **PRI** but PRI is not accessible outside the object.

There are **NO braces { }** for the method. Code is identified as being inside the method if it is **indented**.

Parameters are ALL **long**, just specify the names. If NO parameters, leave off the brackets entirely: **PUB MethodName : rVal | v1…**

The variable representing the **return value** for the method (must always be a long). If not needed, leave off the colon: **PUB MethodName(p1,p2) | v1, v2**
An automatic variable called **Result** is available for use as well, so you don't need to make your own: e.g., **Result := 10**
rVal can be set at any time in the method and is returned upon method completion or when **return** is called.

List ALL local variable names here. They are ALL **long**. If not, leave off the | completely: **PUB MethodName(p1,p2):rVal**

Method Calls

- To call a method defined in the spin file, just use its name:

```
PUB main
  SetUp(12)
  ...
PRI SetUp(range)
  ...
```

Calls the SetUp method below with parameter 12.

- To call an object's method, you merely use the object's name, followed by a dot and the method name:

```
PUB main
  RBC.Init
  d1 := LS1.GetReading
  d2 := LS2.GetReading
  CONV.ToStr(z, CONV#DEC)
  RBC.DebugStr(@myString)
```

If no parameters, don't use brackets.

Simple Math

- Here are some simple math operators:

```
x := x + 5    'if you don't understand this, go home…
x := y / 6    'simple divide
x := x // 10  'modulus (i.e., remainder after divide)
x := x * 4    'multiply and return low 32 bits of result
x := x ** y   'multiply and return high 32 bits of result
x := y #> 100 'highest of y or 100
x := y <# 100 'lowest of y or 100
```

- As in JAVA, you can also use

  +=, -=, *=, /=, //=, **=

  ++, -- (i.e., increment/decrement)

More Math

- Here are some more …

```
x := ^^ y              'square root
x := || y              'absolute value
x := ?x                'pseudo-random long value
x := check1 AND check2 'logical AND
x := check1 OR check2  'logical OR
x := NOT check1        'logical NOT
x == y                 'logical EQUALS
x <> y                 'logical NOT EQUALS
<, >, =<, =>           'logical comparisons
@string                'address / pointer
```

Can you believe that they reversed the order ? … a ridiculous decision !!

Bit Math

- *Here are some bit-related operators:*

```
z := ~x       'sign extend when x is byte and z is long
```

```
11111011                            assume (x = -5)
00000000_00000000_00000000_11111011 (z := x → 251)
11111111_11111111_11111111_11111011 (z := ~x → -5)
```

```
z := ~~y      'sign extend when y is word and z is long
x := y << 4   'shift left 4 bits
x := y >> 3   'shift right 3 bits
x := y ~> 2   'shift right 2 bits (keeps the sign)
x := !%00101100               'bitwise NOT
x := %00101100 & %00001111    'bitwise AND
x := %00101100 | %00001111    'bitwise OR
x := %00101100 ^ %00001111    'bitwise XOR
```

Floating Point Math

- SPIN operators DO NOT WORK on FLOATS!!!! ⚠
  - either convert everything to integers, or
  - use FloatMath.spin and FloatString.spin objects which are in the standard library:

```
VAR
  long x,y,z

OBJ
  F:    "FloatMath"

PUB main:
  x := 3.14159265
  y := F.FMUL(2.0,x)
  z := F.FDIV(y, FLOAT(2))
```

Floats are actually stored as longs.

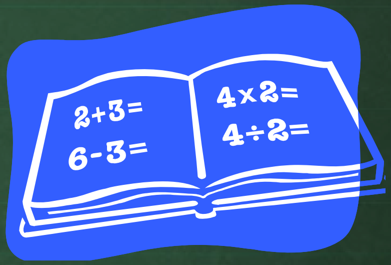
Too many digits to store as “single” float. This will be truncated to 7 significant digits.

⚠ Warning!! You MUST NOT use **2** here. It MUST be a float (i.e., **2.0**) or the results will be wrong!!

This works too…converts integer to float.

More Floating Point Math

- Here are all the **functions**/methods in **FloatMath**:

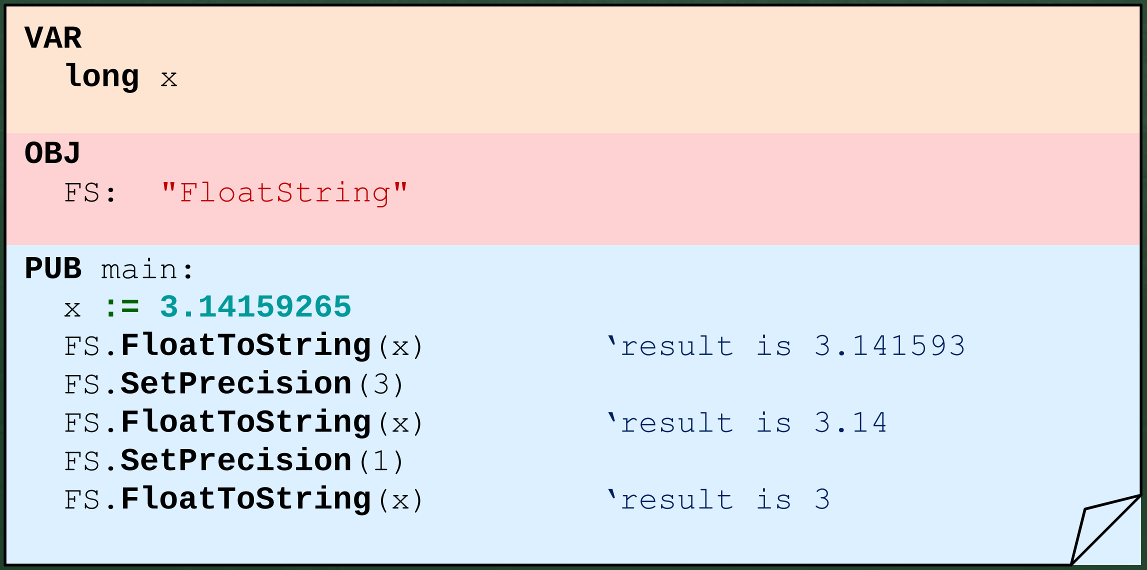


```
FAdd(single1, single2)     'add
FSub(single1, single2)     'subtract
FMul(single1, single2)     'multiply
FDiv(single1, single2)     'divide
FSqr(aSingle)              'square root
FNeg(aSingle)              'negate
FAbs(aSingle)              'absolute value
FTrunc(aSingle)            'truncate to integer
FRound(aSingle)            'round to integer
FFloat(anInteger)          'convert to float
```

Converting Floats To Strings

- The **FloatString.spin** library object converts floating point numbers into Strings:

```
FloatToString(aSingle)     ‘converts to a string
SetPrecision(anInteger)    ‘sets precision from 1 to 7
                            significant digits
```



```
VAR
  long x

OBJ
  FS:   "FloatString"

PUB main:
  x := 3.14159265
  FS.FloatToString(x)           'result is 3.141593
  FS.SetPrecision(3)
  FS.FloatToString(x)           'result is 3.14
  FS.SetPrecision(1)
  FS.FloatToString(x)           'result is 3
```

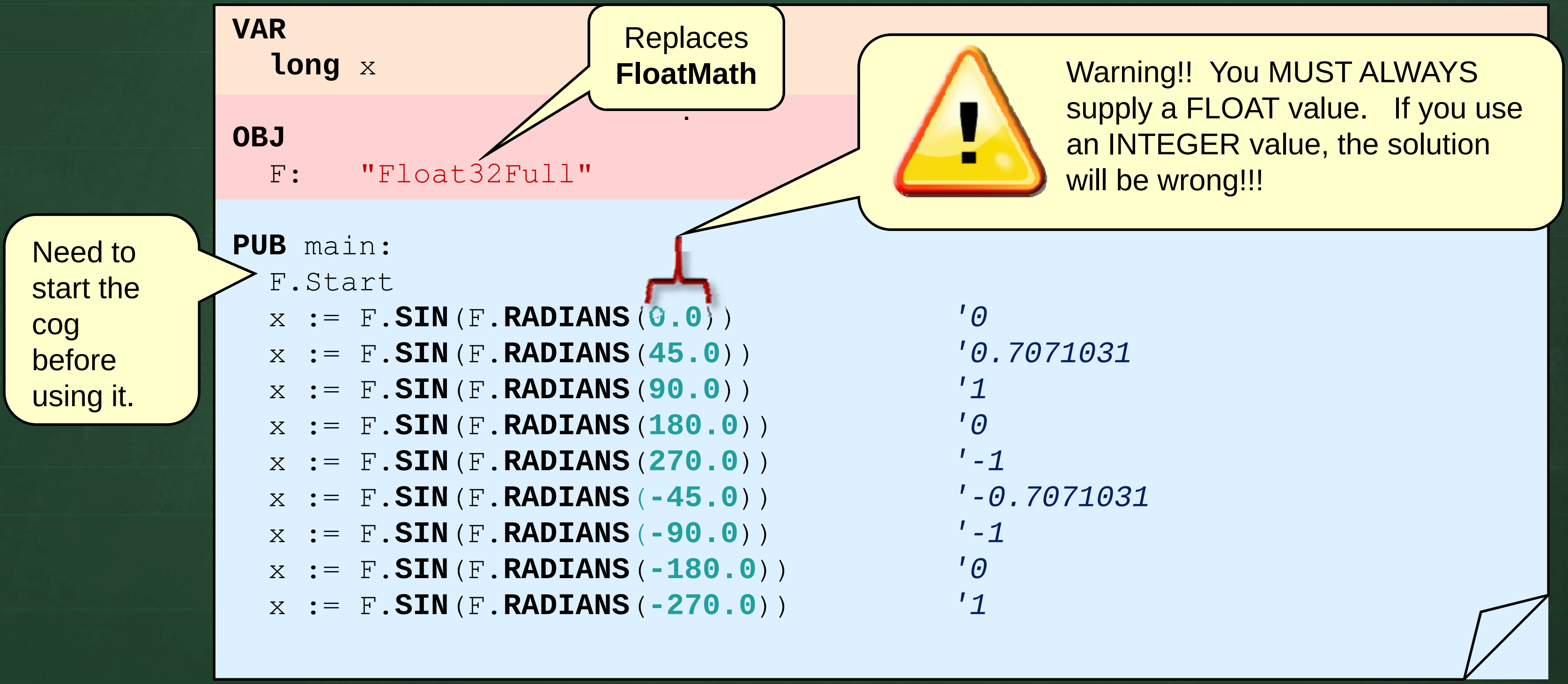
Trigonometry

- Propeller contains 2049-word **Sine table**:
  - sine values from 0° to 90° are “looked up”
  - sine values for all other quadrants can be calculated from simple transformations on this table.
  - can calculate COS/TAN from SIN

- Instead of “re-inventing the wheel”, we can use the library provided in **Float32Full.spin** and **Float32A.spin** which:
  - implements the usual float functions
  - all the useful trig functions (i.e., sin, cos, tan, asin, acos, atan, etc..)

The Extended Float32Full Object

- Use **Float32Full** object instead of **FloatMath**:



```
VAR
  long x

OBJ
  F:    "Float32Full"

PUB main:
  F.Start
  x := F.SIN(F.RADIANS(0.0))           '0
  x := F.SIN(F.RADIANS(45.0))          '0.7071031
  x := F.SIN(F.RADIANS(90.0))          '1
  x := F.SIN(F.RADIANS(180.0))         '0
  x := F.SIN(F.RADIANS(270.0))         '-1
  x := F.SIN(F.RADIANS(-45.0))         '-0.7071031
  x := F.SIN(F.RADIANS(-90.0))         '-1
  x := F.SIN(F.RADIANS(-180.0))        '0
  x := F.SIN(F.RADIANS(-270.0))        '1
```

Replaces **FloatMath**

Warning!! You MUST ALWAYS supply a FLOAT value. If you use an INTEGER value, the solution will be wrong!!!

Need to start the cog before using it.

- When using a **Float32Full** object, it will take up 2 additional COGs for itself !!

Float32Full's Functions

- Here are "most" of the functions in Float32Full:

```
FAdd, FSub, FMul, FDiv          'same as in FloatMath
FSqr, FNeg, Abs                 'same as in FloatMath
FTrunc, FRound, FFloat          'same as in FloatMath
Sin(r), Cos(r), Tan(r)          'Sin/Cos/Tan of radians val
ASin(r), ACos(r), ATan(r)       'ASin/ACos/ATan of rad val
Log(s), Log10(s)                'Log functions
Exp(s), Exp10(s)                'Exponent functions
Pow(s1, s2)                     's1 raised to power of s2
FMin(s1, s2), FMax(s1, s2)      'Min and Max of s1 & s2
Radians(deg), Degrees(rad)      'convert between rad/deg
```

Logical Control Structures

- The Spin *logical control structures* are as follows:

```
if (input < 100)
    … code …
    … more code …
```

No braces, all code beneath **indented** is within the **IF**'s body.

```
if (input < 100)
  … do something …
else
  … do something else …
```

```
if (input < 100)
  … do something …
elseif (input < 200)
  … do something else …
else
  … do yet something else …
```

You can use these binary operators:
==, **<**, **>**, **=<**, **=>**, or **<>**

or these logical operators:
NOT, **AND**, **OR**, or **XOR**

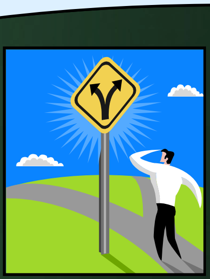
```
ifNot (input < 100)
  … do something …
elseIfNot (input < 200)
  … do something else …
else
  … do yet something else …
```

There are also **ifNot**, **elseIfNot** control structures as well.

No break statements. Only first matching case is evaluated.

```
case (X+Y)
  0:          … code to handle zero case
  1:          … code to handle one case
  10,15:      … code to handle 10 & 15 case
  A*2:        … code to compute and handle result case
  30..40:     … code to handle 3o to 40 range case
  OTHER:      … code for the default case

… this code is now outside the case statement
… more code …
```

```
ifNot (input < 100)
  … do something …
else
  … do something else …
```



Looping Control Structures

- Here are some of Spin's *looping control structures:*

```
repeat
  … some code …
  … more code …
  … yet more code …
```

An infinite loop with three lines of code which all must be indented.

```
repeat 10
  … some code …
  … more code …
```

Repeats 10 times.

```
repeat i from 0 to 10
  … some code …
  … more code …
```

Same as a FOR loop but **i** must either be a local or global variable.

```
repeat i from 10 to 0
  … some code …
  … more code …
```

Automatically counts backwards for you.

```
repeat i from 0 to 10 step 2
  … some code …
  … more code …
```

Can step by specified amount.

```
repeat
some code …
```

If you forget to indent your code, the repeat line by itself will hang forever because nothing is in the loop body.

```
x := 0
repeat while (x < 10)
  … do something …
  x++
```

A **WHILE** loop

```
x := 0
repeat until (x > 10)
  … do something …
  x++
```

A **REPEAT/UNTIL** loop

```
x := 0
repeat
  … do something …
  x++
while (x < 10)
```

Another **WHILE** loop

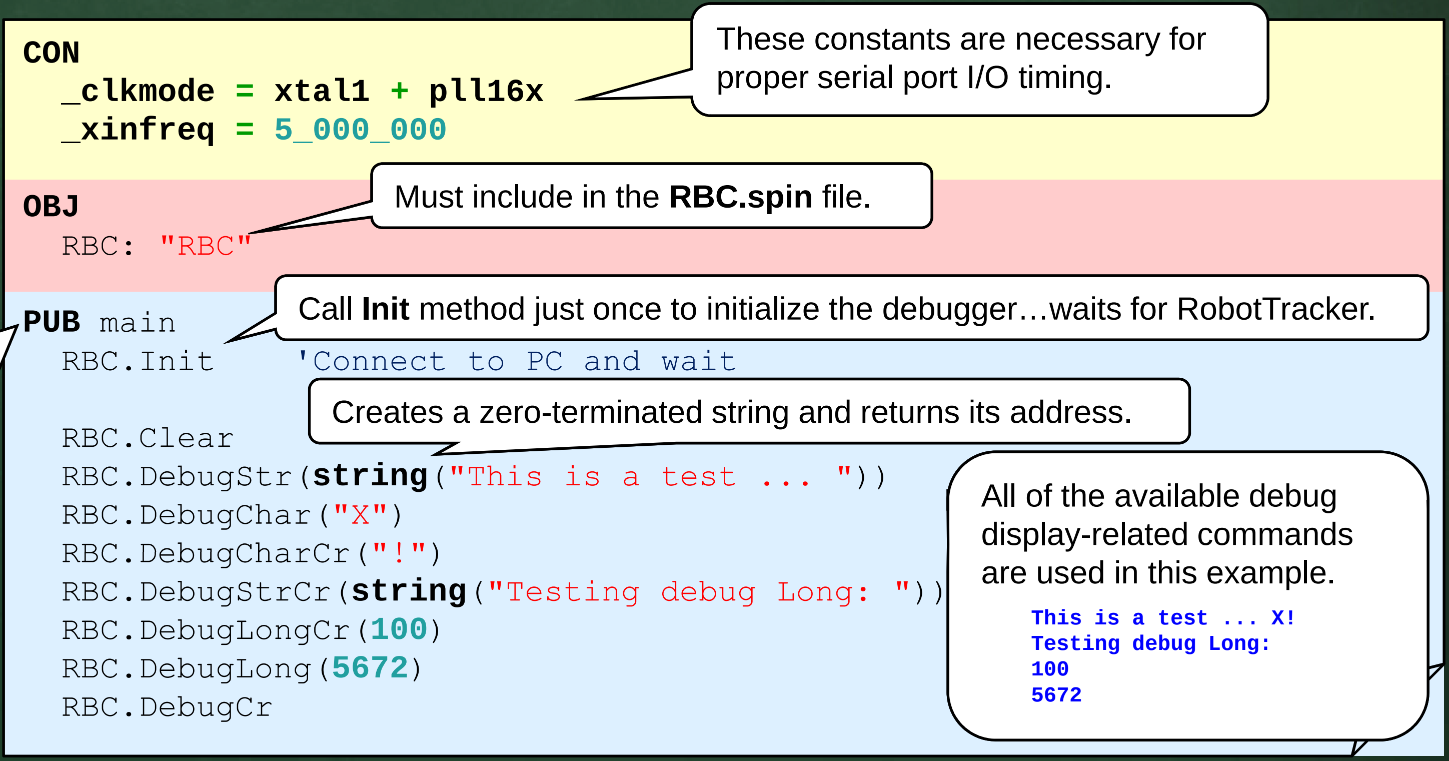
```
repeat i from 0 to 10
  if (somethingHappened)
    next
  elseif (somethingElseHappened)
    quit
  else
    … do something else …
```

Goes to next iteration of loop

Jumps out of loop

First Program: Hello World

- Displaying Hello World is not so simple:
  - There is no console screen on the robot
  - Can send data out wirelessly using RBC.spin



```
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

OBJ
  RBC: "RBC"

PUB main
  RBC.Init     'Connect to PC and wait

  RBC.Clear
  RBC.DebugStr(string("This is a test ... "))
  RBC.DebugChar("X")
  RBC.DebugCharCr("!")
  RBC.DebugStrCr(string("Testing debug Long: "))
  RBC.DebugLongCr(100)
  RBC.DebugLong(5672)
  RBC.DebugCr
```

These constants are necessary for proper serial port I/O timing.

Must include in the **RBC.spin** file.

Call **Init** method just once to initialize the debugger…waits for RobotTracker.

Creates a zero-terminated string and returns its address.

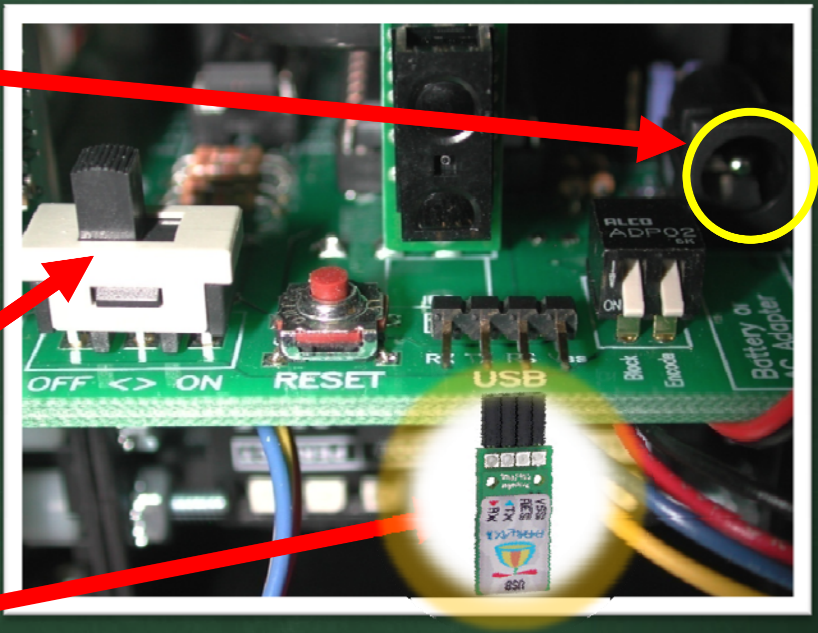
The 1st public method is where your program begins…it does not need to be called **main**.

All of the available debug display-related commands are used in this example.

```
This is a test ... X!
Testing debug Long:
100
5672
```

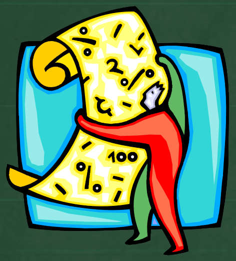
Double Check

- Before uploading a program to the robot:
  - robot must have power
    - from battery cable
  - robot must be turned on
  - robot MUST be connected to PC's USB port
    - port must be set up in the Propeller Tool IDE program...



Displaying Integers

- Can display numbers using **Numbers.spin** object:



```
VAR
  byte   x
  word   y
  long   z

OBJ
  RBC:   "RBC"
  CONV:  "Numbers"

PUB main
  RBC.Init
  CONV.Init

  x := 7
  y := -400
  z := 100 * ~x + ~~y

  RBC.DebugStr(CONV.ToStr(z, CONV#DEC))
  RBC.Cr
  RBC.DebugStr(CONV.ToStr(~~y, CONV#DEC))
  RBC.Ctr
  RBC.DebugStr(CONV.ToStr(~x, CONV#DEC))
  RBC.Cr
```

The **ToStr()** method converts a **long** value into a string formatted to look like a specified type (in this case a decimal as specified by the symbol CONV#DEC).

We need to initialize these.

Use # to access an object's constant.

To display a **word** we must use the **~~** to extend the sign by 16 bits. To display a **byte** we use **~** to extend it by 24 bits. We **MUST** do this in order to handle negative numbers properly.

Here is the output:
300
-400
7

Displaying Floats

- To display floats, you need to use the **FloatString.spin** object:

```
VAR
  long x

OBJ
  F:      "Float32Full"
  FS:     "FloatString"
  RBC:    "RBC"

PUB main:
  F.Start
  RBC.Init

  x := 3.14159265
  RBC.DebugStrCr(FS.FloatToString(x))
  x := F.SIN(F.RADIANS(45.0))
  RBC.DebugStrCr(FS.FloatToString(x))
```

Outputs **3.141593**

Outputs **0.707103**

Device I/O

- Each cog can communicate with various devices (e.g., sensors, bluetooth) through 32 shared pins
  - Each pin 0 – 31 is digital (can be high (1) or low (0))
  - An I/O pin should only be set by one cog at a time, but all cogs have free access to all 32 pins.

  - If two cogs try to set a pin at the same time, the result of the pin is an "OR"-ing of the requests.
  - Here are the rules:
    - pin outputs low **only** if all active cogs that set it to output also set it to low
    - pin outputs high **if any** active cog sets it to an output and also sets it high

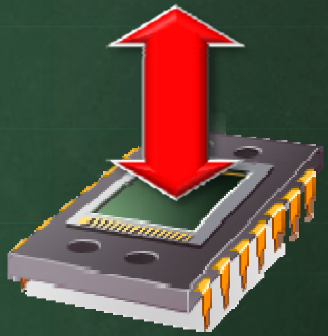
I/O Registers

- Device communication is done using 3 registers:

  DIRA – specifies the direction of all 32 I/O pins

  OUTA – sets the output state of the 32 pins

  INA – reads the input state of the 32 pins



- For example,
  - the 1st line of the following code specifies pins 26, 21, 20, 8, 7, 6, 5 and 4 to be ***output*** pins, defining the remaining pins as ***input***.
  - the 2nd line then sets pins 30, 26, 21, 20, 8, 7 and 4 to output ***high***, the rest being set to output ***low***.

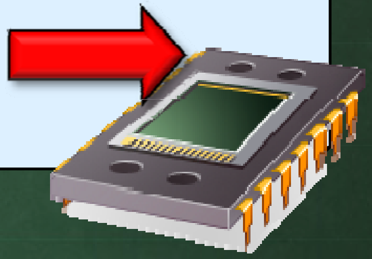
```
DIRA := %00000100_00110000_00000001_11110000
OUTA := %01000100_00110000_00000001_10010000
```

"Ignored" since **DIRA** at position **30** is set to *input*.

Setting/Reading PINs

- It is easier to set an individual PIN as follows:

```
DIRA[10]~~                    'Set P10 to output
OUTA[10]~                     'Make P10 low
OUTA[10]~~                    'Make P10 high
```

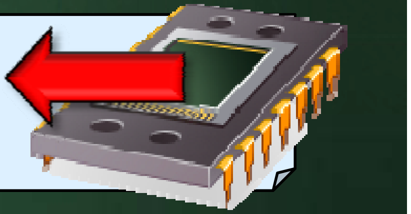


- Can also specify a range of pin settings:

```
DIRA[8..12]~~                 'Set pins P8-P12 to output
OUTA[8..12] := %11001         'Set pins P8 through P12 to 1,1,0,0,1, respectively
```

- Easy to read pins:

```
temp := INA                   'Get the state of ALL 32 I/O pins
temp := INA[10]               'Get the state of pin P10
temp := INA[8..12]            'Get the state of pins P8 through P12
```



Our Robot's I/O Connections

- For our robots, we have already defined various constants within the object files that correspond to the PIN numbers for the various robot components (shown here) →

- Below is a list of the object files that have been created:

Sensors:
```
  IR8SensorArray.spin
  BlockSensor.spin
  Encoders.spin
  PingSensor.spin
  DirrsSensor.spin
  CMUCam.spin
```

Control:
```
  ServoControl.spin
  Beeper.spin
  EasyBluetooth.spin
```

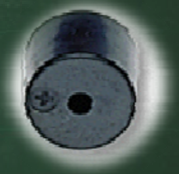
Optional:
```
  CompassHMC6352.spin
  AccelerometerLIS302DL.spin
  BeaconSensor.spin
```

```
CON
  PIN_RIGHT_ENCODER_A       = 0
  PIN_RIGHT_ENCODER_B       = 1
  PIN_RIGHT_GRIPPER_SERVO = 2
  PIN_RIGHT_SERVO           = 3
  PIN_BLUE_RX               = 4
  PIN_BLUE_TX               = 5
  PIN_BEACON_AHEAD          = 6
  PIN_BEACON_RIGHT          = 7
  PIN_BEACON_BEHIND         = 8
  PIN_BEACON_LEFT           = 9
  PIN_DIRRS                 = 11
  PIN_SONAR                 = 12
  PIN_CAMERA_RX             = 13
  PIN_CAMERA_TX             = 14
  PIN_BEEPER                = 15
  PIN_BLOCK_DETECT          = 16
  PIN_IR_SENSE_LOAD         = 17
  PIN_IR_SENSE_CLOCK        = 18
  PIN_IR_SENSE_DATA         = 19
  PIN_HEAD_YAW_SERVO        = 20
  PIN_HEAD_PITCH_SERVO      = 21
  PIN_COMPASS_SCL           = 22
  PIN_ACCEL_SCL             = 22
  PIN_COMPASS_SCA           = 23
  PIN_ACCEL_SCA             = 23
  PIN_LEFT_SERVO            = 24
  PIN_LEFT_GRIPPER_SERVO    = 25
  PIN_LEFT_ENCODER_B        = 26
  PIN_LEFT_ENCODER_A        = 27
```

PropBot Programming

Sensors and Servos

The Beeper



- The **Beeper.spin** object can be a very useful tool for debugging.
  - has various predefined beep routines:

```
Beeper.Startup    'Make a "Starting Up" sound
Beeper.Shutdown   'Make a "Shutting Down" sound
Beeper.Ok         'Make an "Ok" sound
Beeper.Error      'Make an "Error" sound
```

It's a good idea to ALWAYS **do this at the beginning of your code** to know when the robot starts or resets.

  - you can make your own kind of beep by specifying duration and frequency:

```
  Beeper.Beep(10, 4000)      'Make a 4000hz beep for 10ms
  Beeper.Beep(1000, 6000)    'Make a 6000hz beep for 1sec
```

You need to have this at the top of your code in order for any of this code to work:

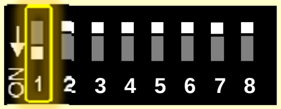
```
OBJ
  Beeper: "Beeper"
```

  - can even create musical tunes

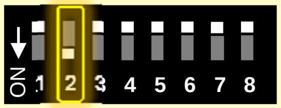
The IR Sensor Array

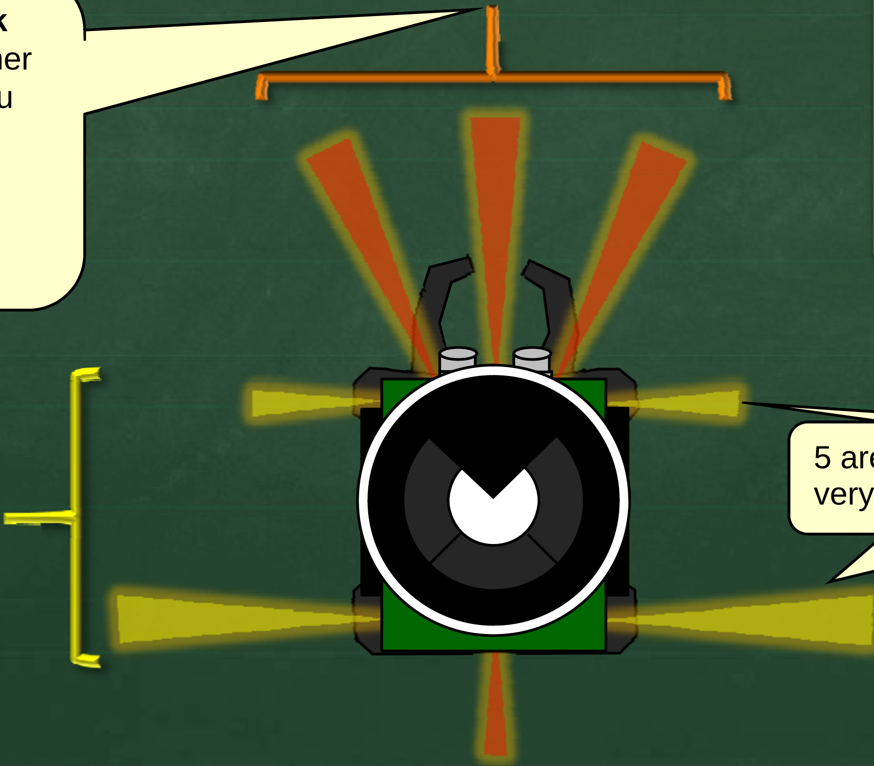
- There are 8 IR sensors surrounding the robot:
  - All 8 sensors are read in at one time

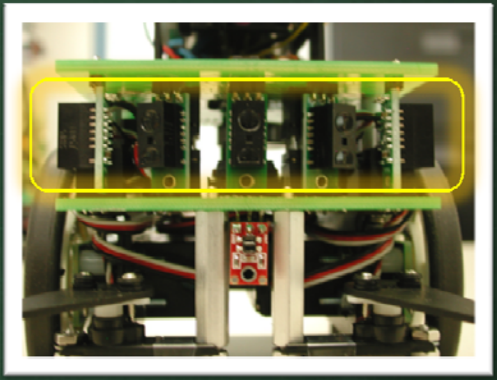
The 3 **front** and single **back** sensors are powered together on switch **1**. Turn it on if you want the sensors to work:



The 4 **side** sensors are powered together on switch **2.** Turn it on if you want the side sensors to work:



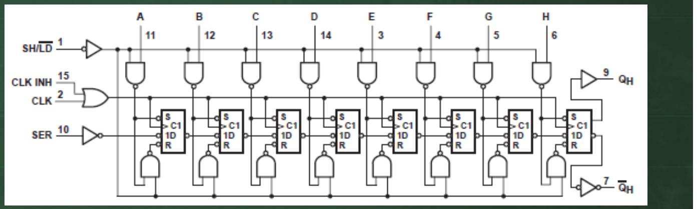




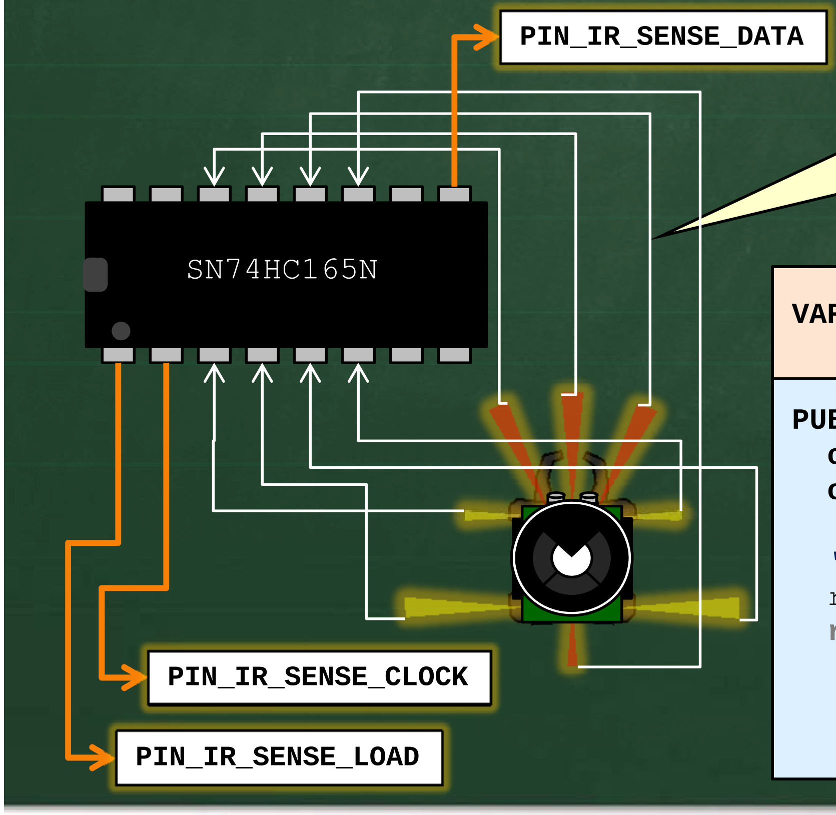
5 are short range (10cm) and 3 are very short range (5cm)

Reading the IR Sensor Array

- Sensors connected to an 8-bit Parallel Load Shift Register →



  - Allows 8 binary sensors to be read using only 3 I/O lines.

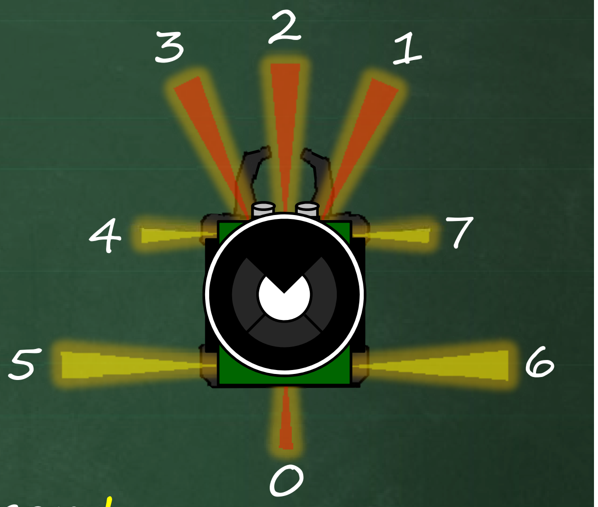


Connect 8 sensors to device, load the data with one line, then clock the data one at a time to get it through the output line.

```
VAR
   byte   readings[8]    ' Stores the latest readings

PUB MAIN
  outa[PIN_IR_SENSE_LOAD]~      ' Set pin low, then high to…
  outa[PIN_IR_SENSE_LOAD]~~     ' load all sensor readings

  ' Now shift the register to get each value in turn
  readings[7] := 1 - ina[PIN_IR_SENSE_DATA]
  repeat i from 1 to 7
     outa[PIN_IR_SENSE_CLOCK]~~   ' Set pin high then low to…
     outa[PIN_IR_SENSE_CLOCK]~    ' shift to next sensor
     readings[7-i] := 1 - ina[PIN_IR_SENSE_DATA]
```

Reading the IR Sensor Array

- Code defined in IR8SensorArray.spin
  - Reading the sensor is done by capturing the data and then just reading the appropriate sensor number →
  - Call capture to get the latest readings
  - Call Detect(i) to get the binary value of sensor I
    - (i.e., 1 = obstacle, 0 = no obstacle)

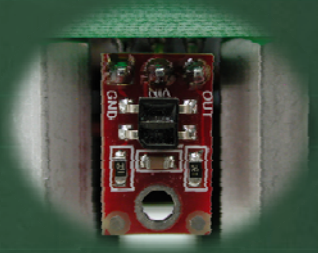


```
OBJ
  IRSensors:   "IR8SensorArray"

PUB main
  IRSensors.capture     'Do this to get latest sensor readings

  'Check for front collision
  if (IRSensors.Detect(1) OR IRSensors.Detect(2) OR IRSensors.Detect(3))
    'Front Collision … Avoid Obstacle
```

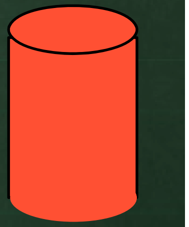
Example of how to use it:

The Block Sensor

- The *block sensor* is:
  - a Pololu QTR-1RC Reflectance Sensor
  - defined in the BlockSensor.spin file



- Detects objects within short range (around 3mm)
- It is used to detect presence of a cylindrical block



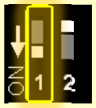
Device uses a “capacitor discharge circuit” that allows the digital I/O line to take an analog reading of reflected IR by measuring the discharge time of the capacitor. Shorter capacitor discharge time is an indication of greater reflection (i.e., closer object).

```
PUB Detect
  dira[PIN_BLOCK_DETECT]~~      'Set as output
  outa[PIN_BLOCK_DETECT] := 1   'Charge capacitor
  dira[PIN_BLOCK_DETECT]~       'Make pin input
  waitcnt(cnt + 100000)         'Wait a bit
  return 1 - ina[PIN_BlockDetect]) 'Read line
```

Example of how to use it:

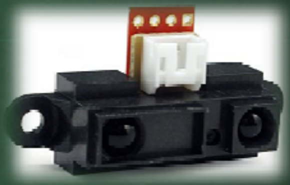
```
OBJ
  BlockSensor:   "BlockSensor"

PUB main
  if (BlockSensor.Detect)
    'Do something
```

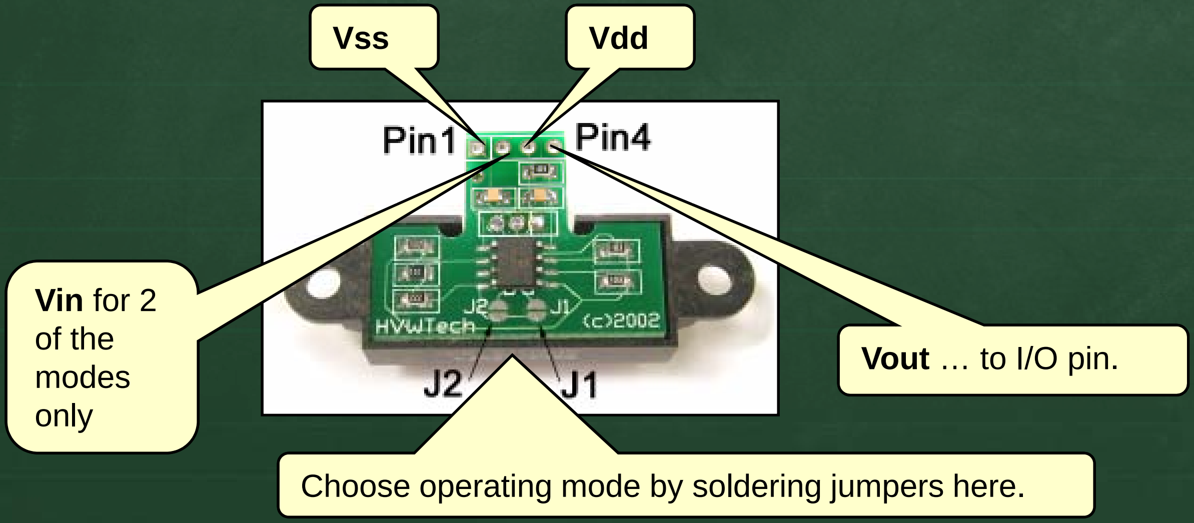
Switch **1** on the lower level dip switch must be ON in order for the **Block** sensor to work



DIRRS+ Range Sensor

- Our robots are equipped with a DIRRS+ sensor:
  - a Digital InfraRed Ranging System
  - gives a distance reading (in cm)
  - gives valid readings from 10cm to 80cm





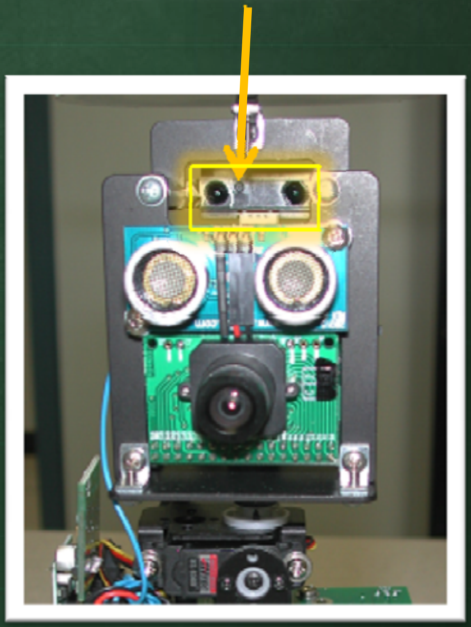
Vss

Vdd

Vin for 2 of the modes only

Vout … to I/O pin.

Choose operating mode by soldering jumpers here.



DIRRS+ Range Sensor

- Can operate in one of three modes:
  - Serial Hex:
    - device first sends byte 10101010 followed by 1 byte voltage val.
    - data is constantly sent on PIN 4 at 4800bps (8Bits/NoParity/1StopBit) This is roughly every 5_{ms}.
  - Synchronous Serial:
    - single 8-bit voltage value transmitted on pin 4 at rate corresponding to clock on pin 2.
    - must generate 8 pulses on PIN 2 and read PIN 4 in between.
  - Serial CM:
    - string of 3 ASCII characters transmitted serially on PIN 4
    - characters correspond to distances in $_{cm}$ (e.g., object at 10cm sends “1”, “0” and “0”)

DIRRS+ Range Sensor

- We will use the Serial CM mode. Why ?
  - + it is simple to use
  - + requires only one (precious) I/O line.
  - + it already calculates range in cm for us.

- Ranges returned as three bytes.
  - first two bytes contain whole cm portion
  - third byte contains fraction of cm as number of $1/10_{ths}$
  - if no obstacle detected, may return value of 0 or value in the high 70's due to voltage fluctuations (i.e., noise).

Reading the DIRRS+ Sensor

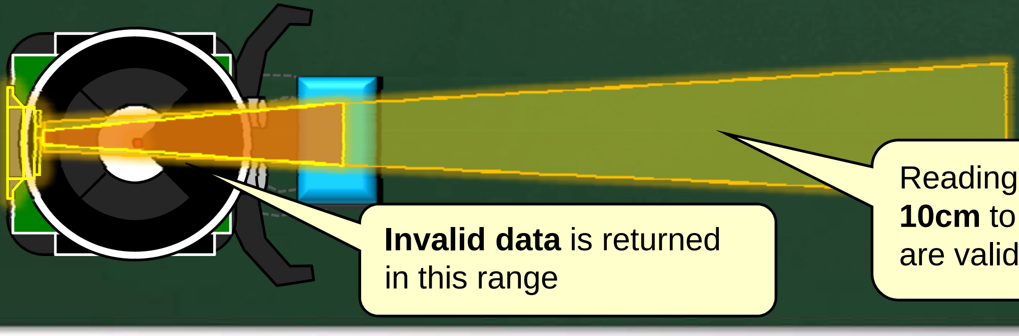
- All the “hard work” of serialization is already done for you in DirrsSensor.spin:
  - Just call the DistanceCM function to get an integer range reading as a long value:

    -1 is returned if no object is detected (e.g., > 80cm away)

    0 is returned if object is too close (i.e., within 10cm away)

Example of how to use it:

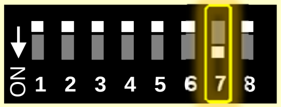
```
OBJ
  Dirrs:  "DirrsSensor"

PUB main | temp
  temp := Dirrs.DistanceCM
  if (temp =< 20)
    'too close
  else
    'add reading of temp cm to map
```



Invalid data is returned in this range

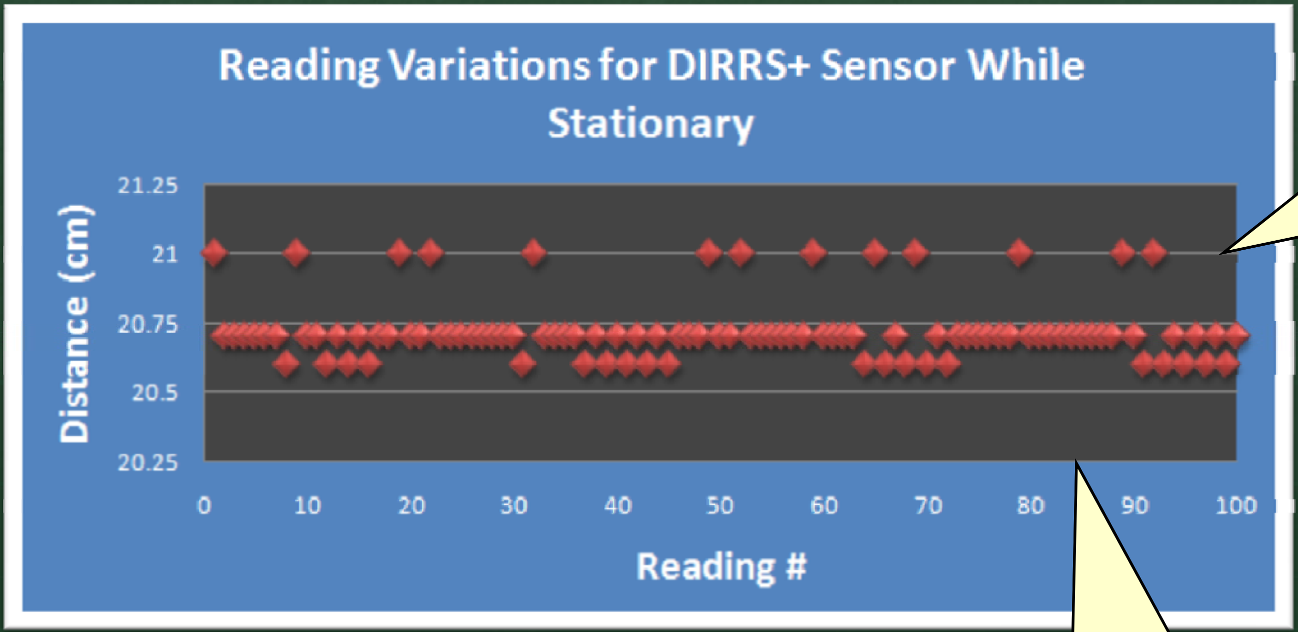
Readings from 10cm to 80cm are valid

Switch 7 must be ON in order for the DIRRS+ sensor to work:



Reading the DIRRS+ Sensor

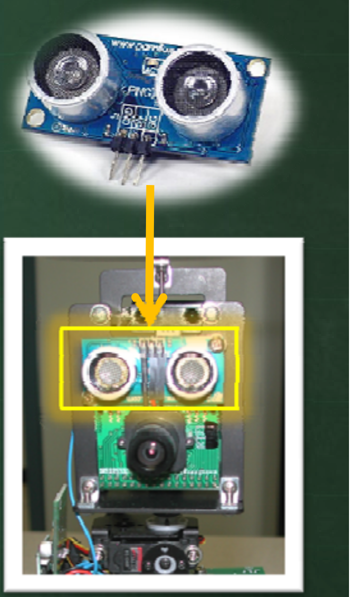
- Be aware that the readings will fluctuate by a **cm** or so between readings:



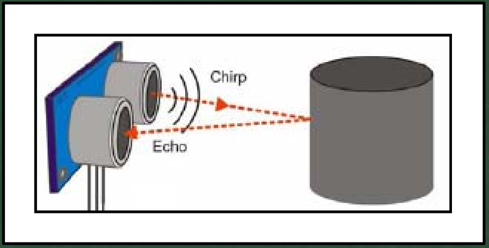
Our DIRRS code will round off to the nearest CM.

All these readings were taken with the sensor remaining stationary.

The Sonar Sensor



- Our robots are equipped with a Ping))) sonar sensor that emits ultrasonic sound
- Emits ultrasonic sound to measure distance to objects

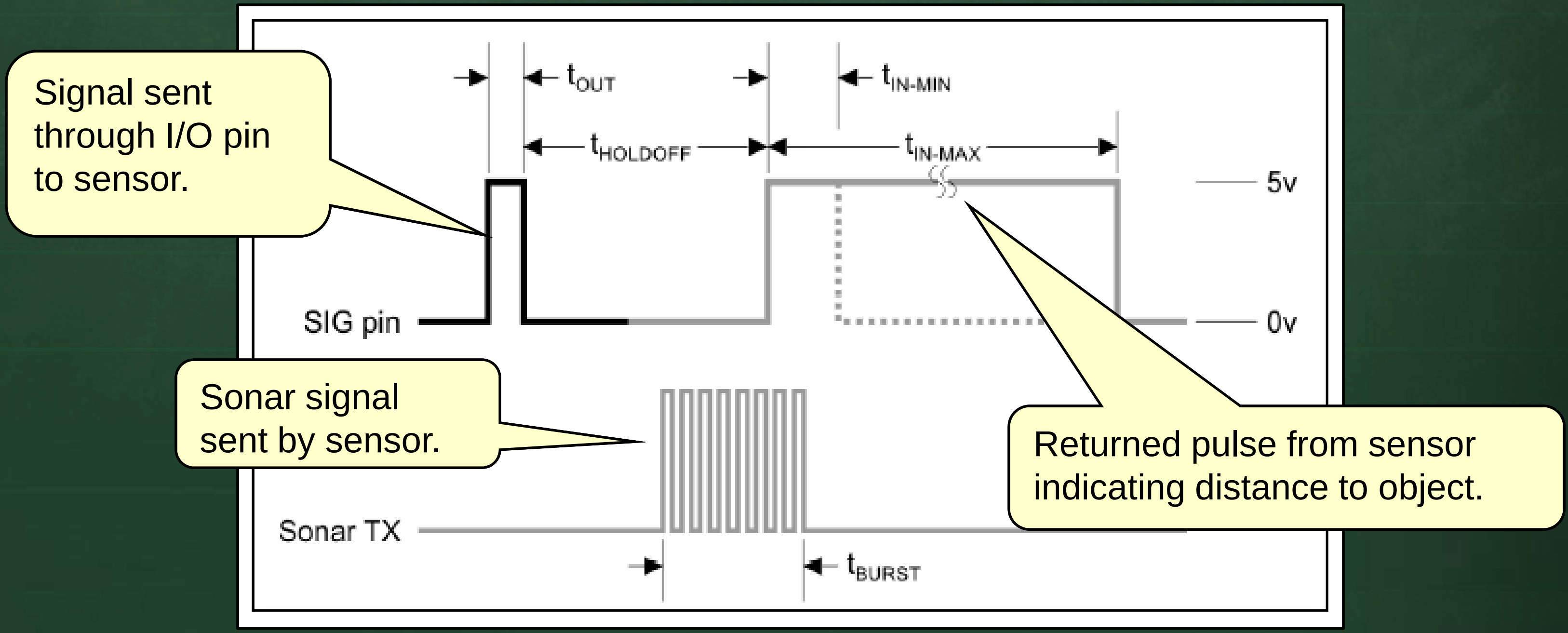


- Detection range from about 3_{cm} to 300_{cm}
- Data may be invalid if object is < 3cm away
- Connects to robot using one I/O pin

The Sonar Sensor

- Operated by:
  - first sending a 0-1-0 pulse to the sensor
  - then reading the pulse coming back from the sensor.

  width of the returned pulse reflects the distance to the object.



Signal sent through I/O pin to sensor.

Sonar signal sent by sensor.

Returned pulse from sensor indicating distance to object.

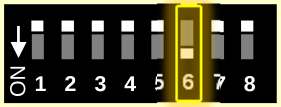
Reading the Sonar Sensor

- Hard work already done in PingSensor.spin file:
  - Use it the same way as the DIRRS sensor.

Example of how to use it:

```
OBJ
  Sonar:   "PingSensor"

PUB main | temp
  temp := Sonar.DistanceCM
  if (temp =< 20)
    'too close
  else
    'add reading of temp cm to map
```

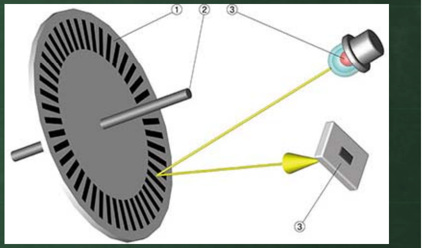
Switch **6** must be ON in order for the **Sonar** sensor to work:

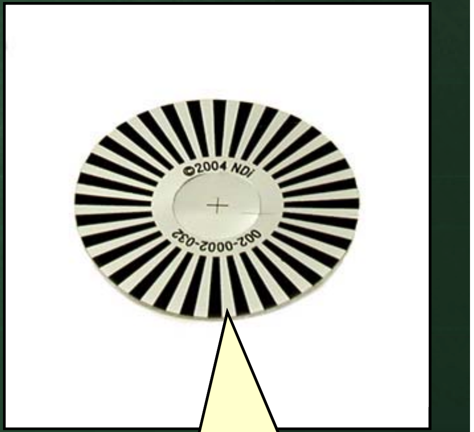


The Wheel Encoders

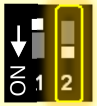
- Our robots have incremental optical encoders on each wheel
  - emits modulated IR light beam that is reflected back from wheel's sticker into a phototransister
  - easy to read status of encoder:

```
tickStatus := ina[PIN_LEFT_ENCODER_A]
```





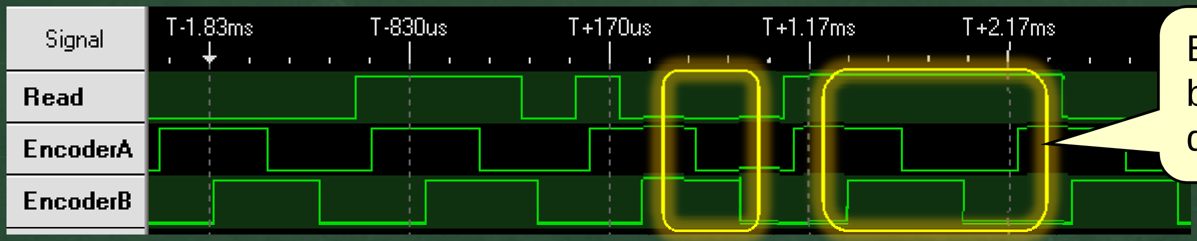
Switch **2** on the lower level dip switch must be ON in order for the **encoders** to work



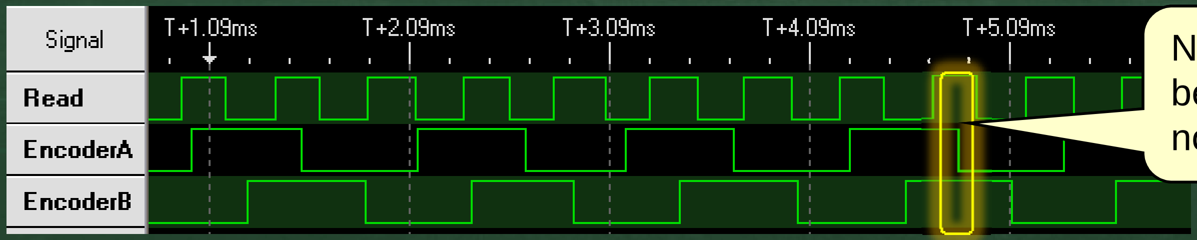
Our wheels contains 32 equally spaced stripes

Reading the Encoders

- Must read encoders fast enough so that no “pulse” is missed



Encoder ticks missed because reads not done fast enough.



No ticks missed because reads are now done quickly.

- Solution: use a dedicated **cog**

Wheel Encoders

- Encoders.spin file contains code to count ticks:

```
VAR
  byte leftA, leftB, rightA, rightB
  word leftCount, rightCount

PRI Run | newVal1, newVal2
  'get the binary readings from left/right encoders signals
  leftA := ina[PIN_LEFT_ENCODER_A]
  leftB := ina[PIN_LEFT_ENCODER_B]
  rightA := ina[PIN_RIGHT_ENCODER_A]
  rightB := ina[PIN_RIGHT_ENCODER_B]
  repeat
    newValA := ina[PIN_LEFT_ENCODER_A]
    newValB := ina[PIN_LEFT_ENCODER_B]
    ifnot ((newValA == leftA) AND (newValB == leftB))
      leftA := newValA
      leftB := newValB
      leftCount++
    newValA := ina[PIN_RIGHT_ENCODER_A]
    newValB := ina[PIN_RIGHT_ENCODER_B]
    ifnot ((newValA == rightA)  AND (newValB == rightB))
      rightA := newValA
      rightB := newValB
      rightCount++
```

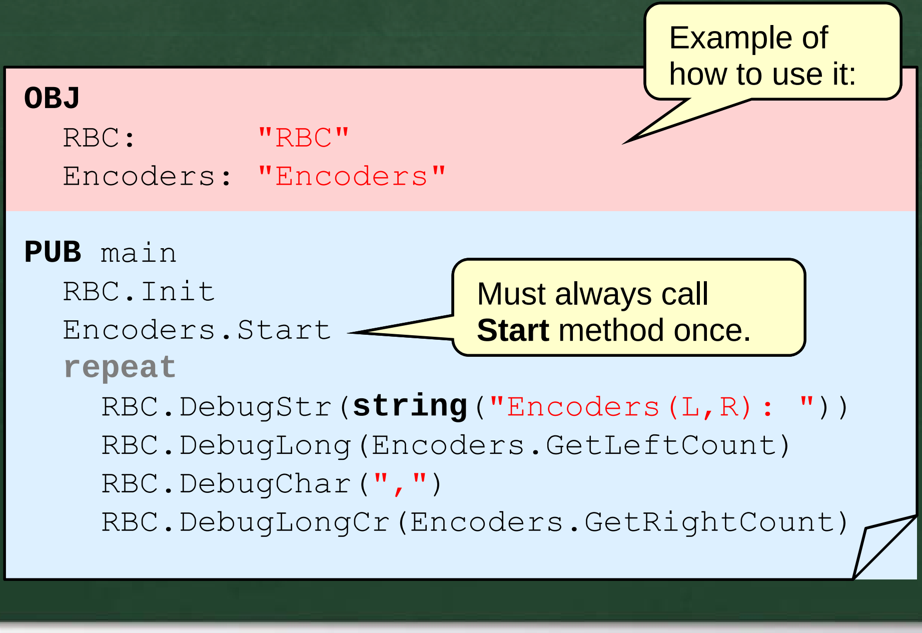
Maintains counters for left & right wheels individually. A **word** is used, which means a maximum count of **32,767** ticks which is **5,518cm** of traveling.

A **separate cog** is required in order not to miss any ticks. The **Start** method is called by the user which runs this **Run** method in an infinite loop.

Looks for changes in pulse from **0->1** or **1->0** on each channel.

Reading the Wheel Encoders

- Reading the encoders is done with 4 methods:
  - Start – call once to start the process that counts the ticks
  - GetLeftCount /GetRightCount – returns the number of ticks that the left/right wheel made since the last counter reset.
  - ResetCounters - reset both counters to zero.



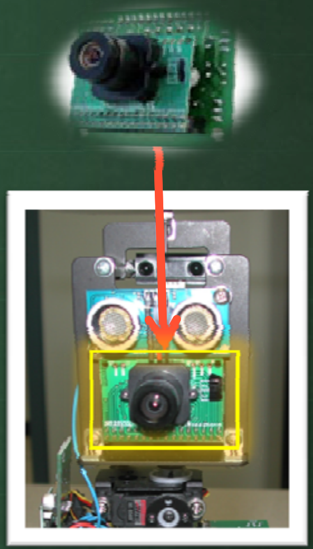
Example of how to use it:

```
OBJ
  RBC:      "RBC"
  Encoders: "Encoders"

PUB main
  RBC.Init
  Encoders.Start
  repeat
    RBC.DebugStr(string("Encoders(L,R): "))
    RBC.DebugLong(Encoders.GetLeftCount)
    RBC.DebugChar(",")
    RBC.DebugLongCr(Encoders.GetRightCount)
```

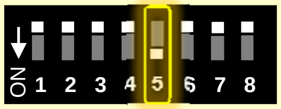
Must always call **Start** method once.

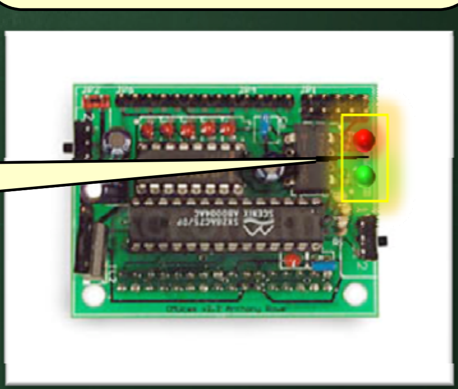
The CMUCam

- Robot has a CMUcam1 Vision System
  - can be used to track (or identify):
    - blocks, other robots, walls of environment
  - can track a color "blob" at 17 frames/sec
    - tracking color can be changed "on the fly"
  - has resolution of 80 x 143 pixels
  - can get statistics (e.g., centroid of blob, mean color and variance data)
  - can extract a frame dump of image
  - Communicates serially with propeller at 115.2k baud



Switch **5** must be ON in order for the **CMUCam** to work:





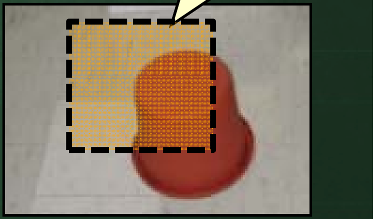
Red light on when camera power is on. **Green** light on when "blob" is identified and **being tracked**.

The CMUCam

- The CMUcam.spin file contains predefined code:
  - makes use of FullDuplexSerial.spin and Numbers.spin
  - requires extra cog (for serial I/O)
  - here are some configuration-related commands:

```
Start
   Initialize the CMUcam (need to call this once)

SetFullWindow
   Set the camera's image window to full size (i.e., 80x143)

SetConstrainedWindow(Xleft, Ytop, Xright, Ybottom)
   Set the portion of the camera's image that you
   want to process (up to 80x143)

ReadColor
   Read the mean color value in terms of red, green and blue components.

GetRed, GetGreen, GetBlue
   Get the red, green or blue value from the last call to ReadColor.
```

The **window** is the portion of the image that is used for tracking.



The CMUCam

- Here are tracking-related commands:

`SetTrackColor(r, g, b, sensitivity)`

Set the color to be tracked currently. The sensitivity is the allowable +- range for each color component during tracking. Call this before calling **TrackColor**.

`TrackColor`

Track the color previously specified by the call to **SetTrackColor**.

- The following should ONLY be called AFTER calling TrackColor:

`GetCenterX`, `GetCenterY`

Return the x/y component of the blob's center of mass.

`GetTopLeftX`, `GetTopLeftY`, `GetBottomRightX`, `GetBottomRightY`

Return the x/y component of the blob bounding box's topLeft/bottomRight corner.

`GetPixels`

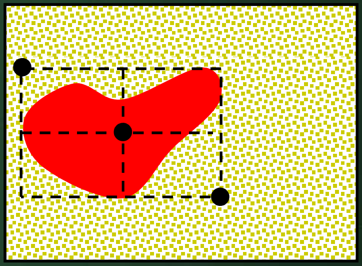
Actual value should be (pixels+4)/8

Return the number of pixels in the tracked blob.

`GetConfidence`

A value of 8 is poor & 50 is very good.

Return the confidence level of the track (i.e., 0 to 255).



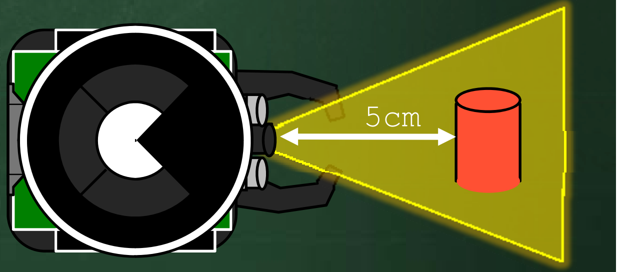
Calibrating the CMUCam

- To determine a color to track, run *CameraColorSampler.spin* and follow these steps:

    1. plug robot into USB and start Parallax Serial Terminal
    2. place object to track (e.g., block) 5cm in front of camera
    3. turn on the robot to start the program and then wait
    4. write down the RGB values from the output window:

```
Place Object about 5cm in front of the camera …
Getting color from 10 samples …
Color value of object: (R,G,B) = 189, 21, 16
```

Be aware that the **lighting conditions** in the room (including the **shadow** from your hand) can greatly affect the readings. As you let the program run, you will notice the **variation**. Try to take an **average** value.



Reading the CMUCam

- To track a color now …

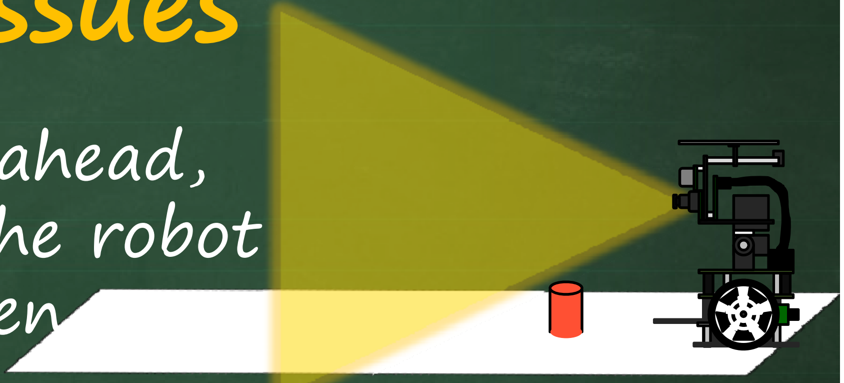
```
CON
  RED_TRACK   = 189
  GREEN_TRACK = 21
  BLUE_TRACK  = 16
  SENSITIVITY = 30
```

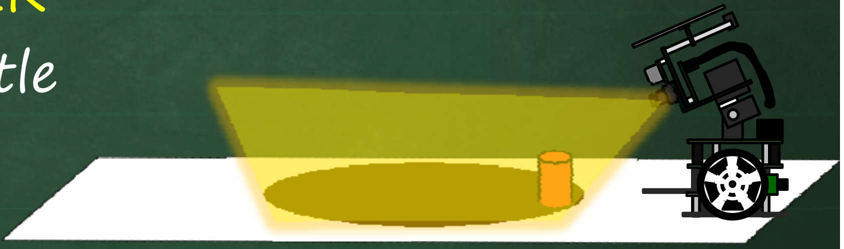
Choose these values yourself, based on **CameraColorSampler.spin**

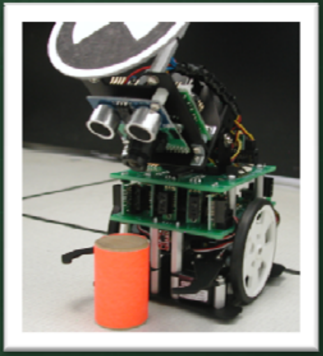
```
OBJ
  Camera:   "CMUCam"

PUB main
  Camera.Start
  Camera.SetTrackColor(RED_TRACK, GREEN_TRACK, BLUE_TRACK, SENSITIVITY)
  repeat
    Camera.TrackColor
    if (Camera.GetCenterX == 0) AND (Camera.GetConfidence == 0)
      'Object has not been found
    elseif (Camera.GetCenterX > 55) AND (Camera.GetConfidence > 5)
      'Object is on the left side
    elseif (Camera.GetCenterX < 15) AND (Camera.GetConfidence > 5)
      'Object is on the right side
    else
      'Object is straight ahead
```

Camera Tracking Issues

- When robot's head is straight ahead, there is a blind spot close to the robot where the blocks cannot be seen
  - You may want to adjust the head (USING YOUR PROGRAM … NEVER MANUALLY!!) to tip forward a little so that the robot could see the blocks when they are close. Trial and error will inform you of the "best" tilt amount to use.

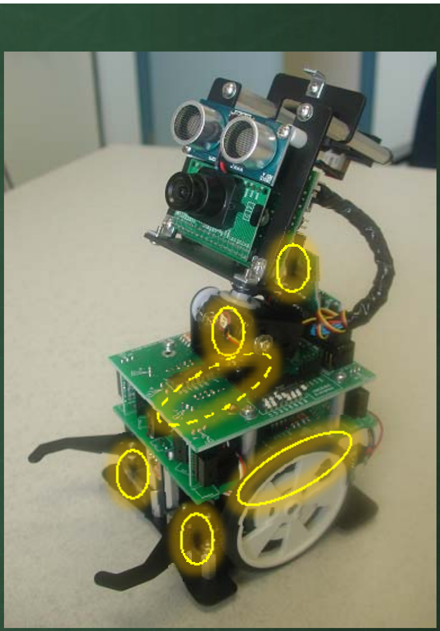






Servos

- Our robots are equipped with 6 ***servos***:
  - geared down motors with electronic circuitry that receive electronic pulses telling it the position, speed or direction that the motor should have.
- There are two types of servo motors:
  - Standard Servos – receives signals indicating the position that the servo should hold (good for controlling grippers, pan/tilt mechanisms, steering mechanisms etc…)
  - Continuous Rotation Servos - receives signals indicating the speed and direction that the servo should have (good for wheels and pulleys)



Switch under bottom board provides power to all 6 servos. Turn it off only when the robot does not need to move (e.g., when on desk).

Wheel Servos

- The wheel servos are parallax continuous rotation servos connected to pins 3 & 24
  - To turn on a servo, we must send it a pulse.
  - Servos have been adjusted so that:

    pulse = 1.5ms keeps servo still

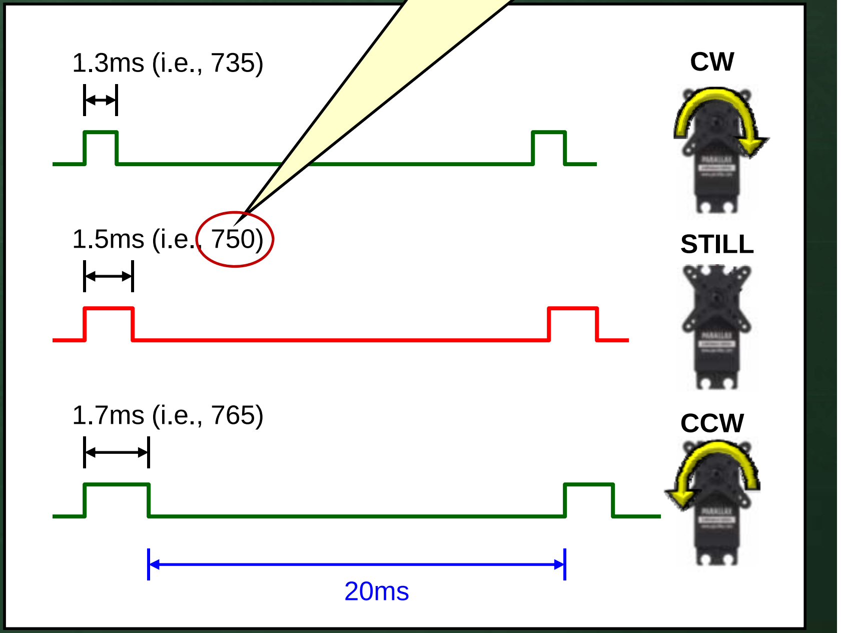
    pulse < 1.5ms rotates servo CW

    pulse > 1.5ms rotates servo CCW

- Over time, however, servos may need centering adjustments.

  (e.g., 748 on one servo …
  751 on the other)

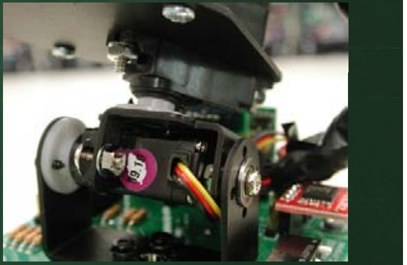
A "stopped" motor value of **750** was chosen to correspond with that of the original BoeBot, but is somewhat arbitrary.



Gripper and Head Servos

- The grippers are controlled by GWS Pico servos
  - These are VERY delicate. NEVER, EVER, EVER try to move the grippers manually … always use a program to set their position.

- The head's pan and tilt (i.e., yaw and pitch) are controlled by HS-85BB Micro Servos
  - These are also quite delicate. NEVER move the head manually … always use a program to set its position. In some cases the head may appear "stuck"… do not try to force it into a position. Please be very gentle.



- Operate on pulses like continuous rotation servos. The pulse values indicate a position (0° to ~180°), NOT a speed.

Servo Control – Wheels

- ServoControl.spin has been created to control servos:

```
VAR
 long  stoppedLeftValue     ' stopped state value of left wheel servo
 long  stoppedRightValue    ' stopped state value of right wheel servo
 long  leftSpeed            ' current speed of left servo
 long  rightSpeed           ' current speed of right servo

PUB Start(leftServoStoppedValue, rightServoStoppedValue, useWheels, useGrippers, usePitch, useYaw)
  ... Code omitted here ...
  result := (cog := cognew(Run, @stack) + 1) > 0

PRI Run | i
  repeat
   MoveWheels
   MovePitch
   MoveYaw
   MoveGrippers
   waitcnt(1600000+cnt)

PRI MoveWheels | clkCycles
  clkCycles := ((stoppedLeftValue+leftSpeed)*160-1250)#>400   ' duration*160 (=2µs) clock cycles
  !outa[PIN_LEFT_SERVO]      ' set to opposite state
  waitcnt(clkCycles + cnt)  ' wait until clk gets there
  !outa[PIN_LEFT_SERVO]      ' return to original state

 clkCycles := ((stoppedRightValue+rightSpeed)*160-1250)#>400   ' duration*160 (=2µs) clock cycles
  !outa[PIN_RIGHT_SERVO]      ' set to opposite state
  waitcnt(clkCycles + cnt)  ' wait until clk gets there
  !outa[PIN_RIGHT_SERVO]      ' return to original state
```

Stopped values determined from calibration (more on this later).

Maintains **separate speed values** for each wheel servo.

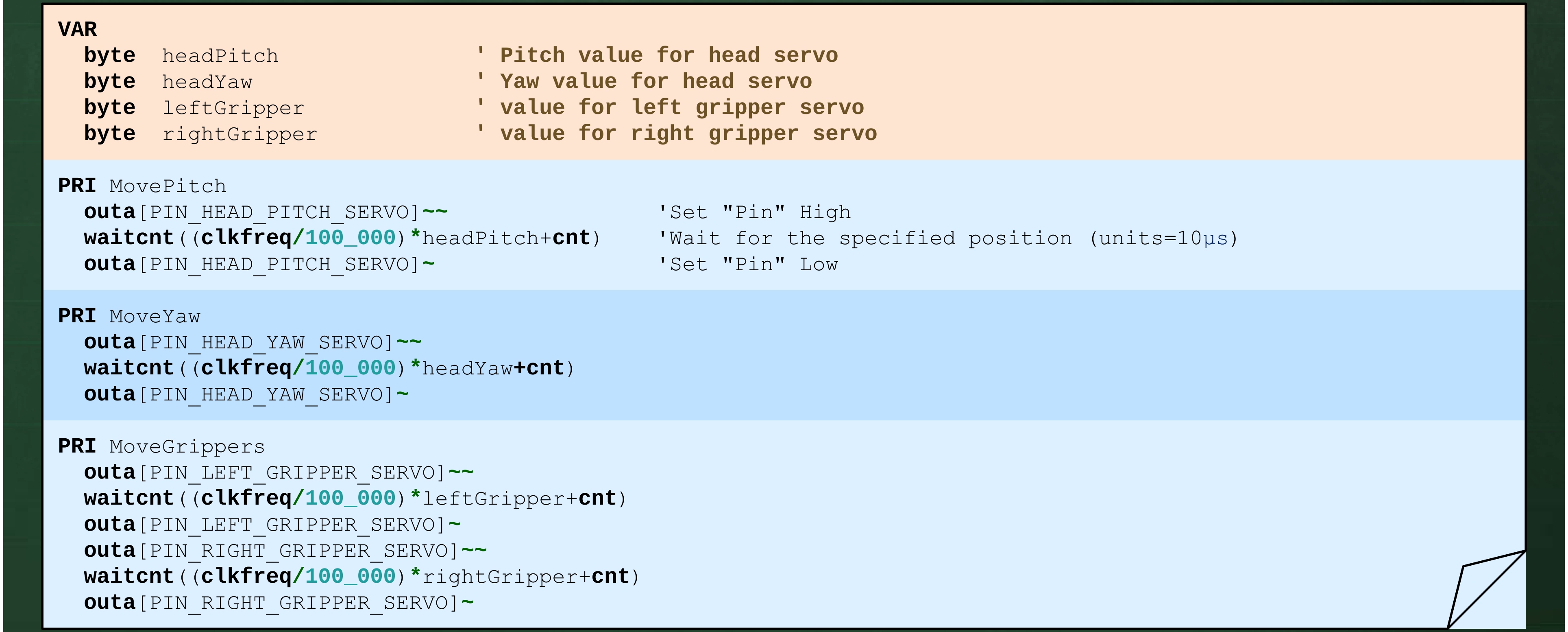
Booleans to enable movement of servos

Infinite loop moves wheels, grippers and head.

A **separate cog** is required so that consistent/smooth speed is obtained. (Must keep sending pulses to servos in order for them to keep moving). The **Start** method is called by the user which begins the **Run** method.

Servo Control – Head & Grippers

- The code for controlling the head and grippers is quite similar:

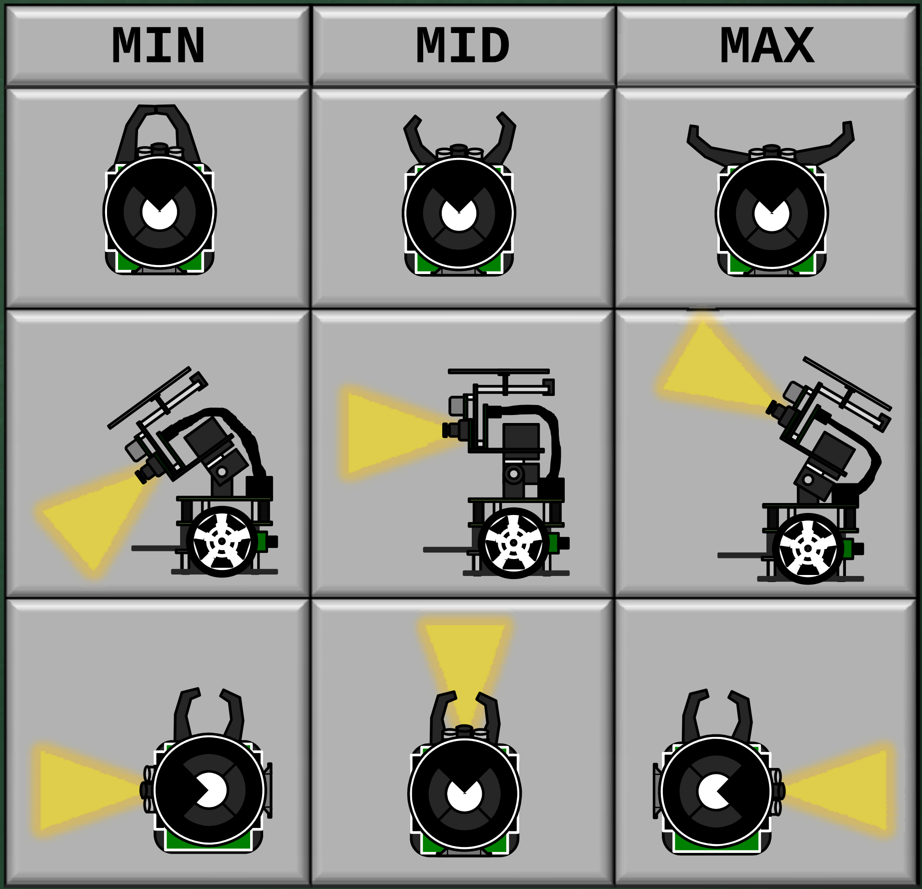


```
VAR
  byte  headPitch               ' Pitch value for head servo
  byte  headYaw                 ' Yaw value for head servo
  byte  leftGripper             ' value for left gripper servo
  byte  rightGripper            ' value for right gripper servo

PRI MovePitch
  outa[PIN_HEAD_PITCH_SERVO]~~                'Set "Pin" High
  waitcnt((clkfreq/100_000)*headPitch+cnt)    'Wait for the specified position (units=10µs)
  outa[PIN_HEAD_PITCH_SERVO]~                 'Set "Pin" Low

PRI MoveYaw
  outa[PIN_HEAD_YAW_SERVO]~~
  waitcnt((clkfreq/100_000)*headYaw+cnt)
  outa[PIN_HEAD_YAW_SERVO]~

PRI MoveGrippers
  outa[PIN_LEFT_GRIPPER_SERVO]~~
  waitcnt((clkfreq/100_000)*leftGripper+cnt)
  outa[PIN_LEFT_GRIPPER_SERVO]~
  outa[PIN_RIGHT_GRIPPER_SERVO]~~
  waitcnt((clkfreq/100_000)*rightGripper+cnt)
  outa[PIN_RIGHT_GRIPPER_SERVO]~
```

Servo Control – Head & Grippers

- Some constants are defined as "fixed positions" for the servos (although values vary slightly from robot to robot)

```
CON
   LEFT_GRIPPER_MIN = 215
   LEFT_GRIPPER_MID = 170
   LEFT_GRIPPER_MAX = 140

   RIGHT_GRIPPER_MIN = 104
   RIGHT_GRIPPER_MID = 150
   RIGHT_GRIPPER_MAX = 181

   PITCH_MIN = 95
   PITCH_MID = 137
   PITCH_MAX = 170

   YAW_MIN = 61
   YAW_MID = 146
   YAW_MAX = 225
```



Servo Control

- *Here are the available commands:*

```
Start(leftServoStoppedValue,
      rightServoStoppedValue,
      useWheels, useGrippers,
      usePitch, useYaw)
```

Starts the cog to control servos. Call this once at the beginning of your code. The 1st two parameters indicate the values that must be sent to the servos to stop them. These are obtained by running the program: **ServoCalibration.spin**.

The remaining 4 parameters are booleans indicating whether or not those particular servos are going to be used by this program. For example, if the grippers will not be used by your code, set **useGrippers** to **false**. Also, you may need to set **usePitch** to **true** in order to keep power on the head servo so that it does not tip forward on its own. ⚠

```
SetLeftSpeed(s)
SetRightSpeed(s)
SetSpeeds(sL, sR)
```

Sets the speed of the servos (usually ranging from -40 to +40). 0 means stop the servo, + is forwards and – is backwards. Higher values means faster.

```
SetHeadPitch(value)
SetHeadYaw(value)
```

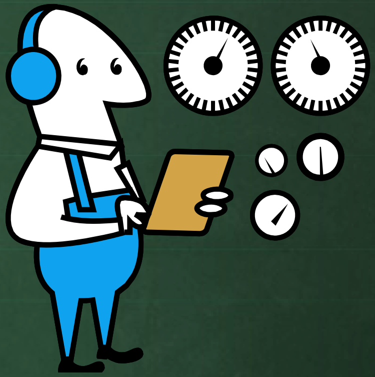
Sets the position of the head servos (usually ranging from **61** to **225**). Be careful that the head pitch does not cause the head to rub against the top board and/ or bluetooth device.

```
SetLeftGripper(value)
SetRightGripper(value)
```

Sets the position of the gripper servos (usually ranging from **61** to **225**). Be careful that the grippers do not press against each other when closed and that they do not rub against the wheels when fully open.

Better Servo Control

- You can add your **own methods** /constants to allow:
  - spinning, turning in arcs, stopping
  - moving backwards
  - moving at various speeds
  - etc..



- You can actually move all servos at once

⚠ Be careful! Some head pitch and yaw positions DO NOT work well on the robot. For example, putting the head down all the way and then panning it (i.e., rotating along the yaw direction) can cause physical damage to the sensors and top board of the robot as well as pull cables loose.

Ramping the Wheel Servos

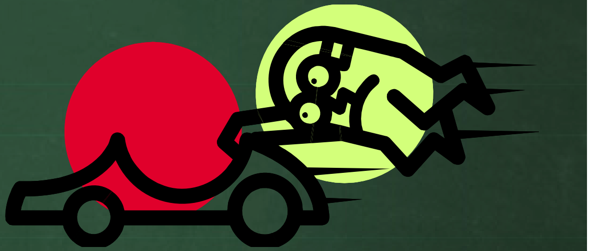
- Servos experience wear & tear more quickly when abrupt changes in speed and/or direction are made (e.g., stopped to very fast).

- To reduce wear & tear, ***ramping*** should be used:
  - gradually accelerate and/or decelerate the servos over time to the desired speed.

- We must be careful to realize that it takes time to decelerate, and so collision avoidance and other maneuvering behaviors must compensate
  - E.g., the front IR sensors will not seem to respond quickly when deceleration is too slow.

Ramping the Wheel Servos

- To do this, just keep track of:
  - currentLeftSpeed, currentRightSpeed
  - desiredLeftSpeed, desiredRightSpeed



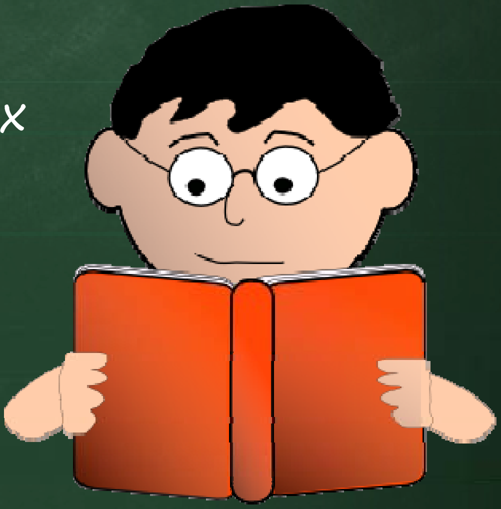
- If you want to turn or change speed:

1. set the desiredLeftSpeed and desiredRightSpeed in the SetLeftSpeed, SetRightSpeed and SetSpeeds methods.
2. modify the run method in the ServoControl.spin code to automatically increase/decrease the **current** speed values a little each time … until they match the **desired** speed values.
   - The amount of increase/decrease each time through the run loop represents the rate of acceleration/deceleration.

For More Information …

- There are many more functions and procedures defined in the Spin language
- Each sensor also has its own documentation.
- For more information on the Propeller:
  - The Propeller's Documentation website:

    http://www.parallax.com/tabid/442/Default.aspx

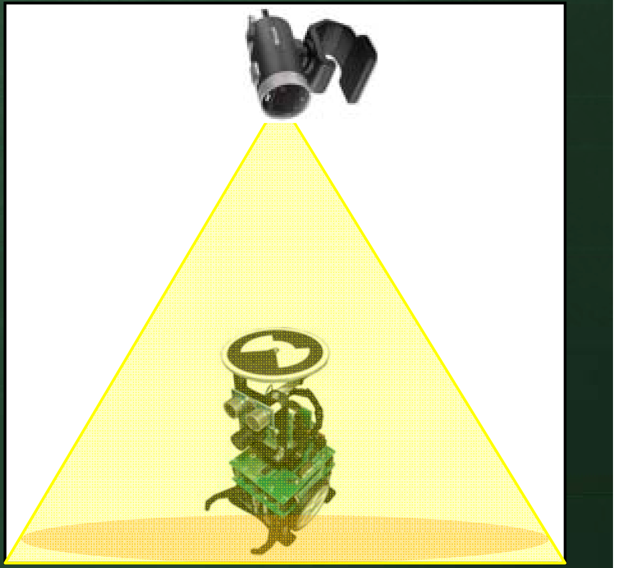


Robot Tracking System

Robot Tracker v4.0

Robot Tracker 4.0

- In the labs, we have a kind of local GPS tracking system called **Robot Tracker**.
- This system employs a **webcam** on the ceiling to track black and white tags placed on the robot.
- It will be used to provide absolute (x, y) **positions** for our robots as well as their **angle**.



- Can send this data to the robot or process it offline.

Robot Tracker – PC Communications

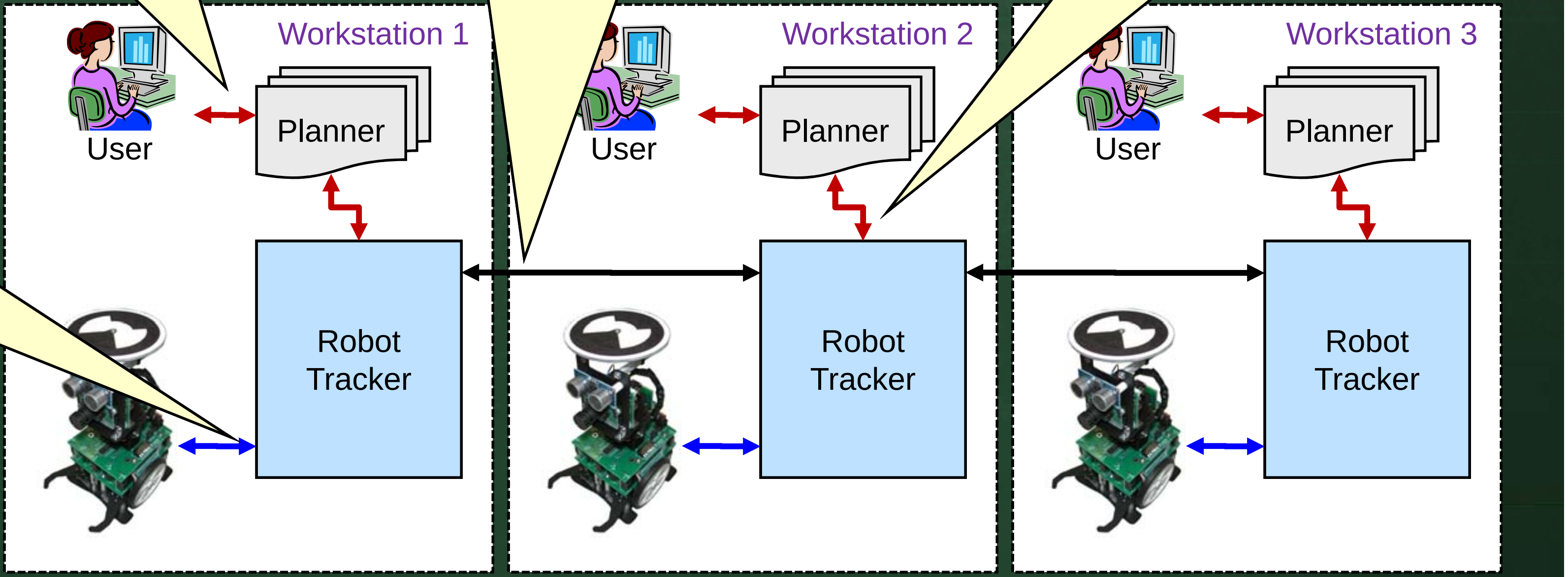
- All communications occur through the RobotTracker
  - Inter-robot communication
  - Planners also communicate

Write **your own JAVA code** to plan the robot's motion.

Robot Trackers communicate amongst themselves to exchange pose information as well as user data.

Your compiled java code **automatically** connects to (and is started by) the RobotTracker.

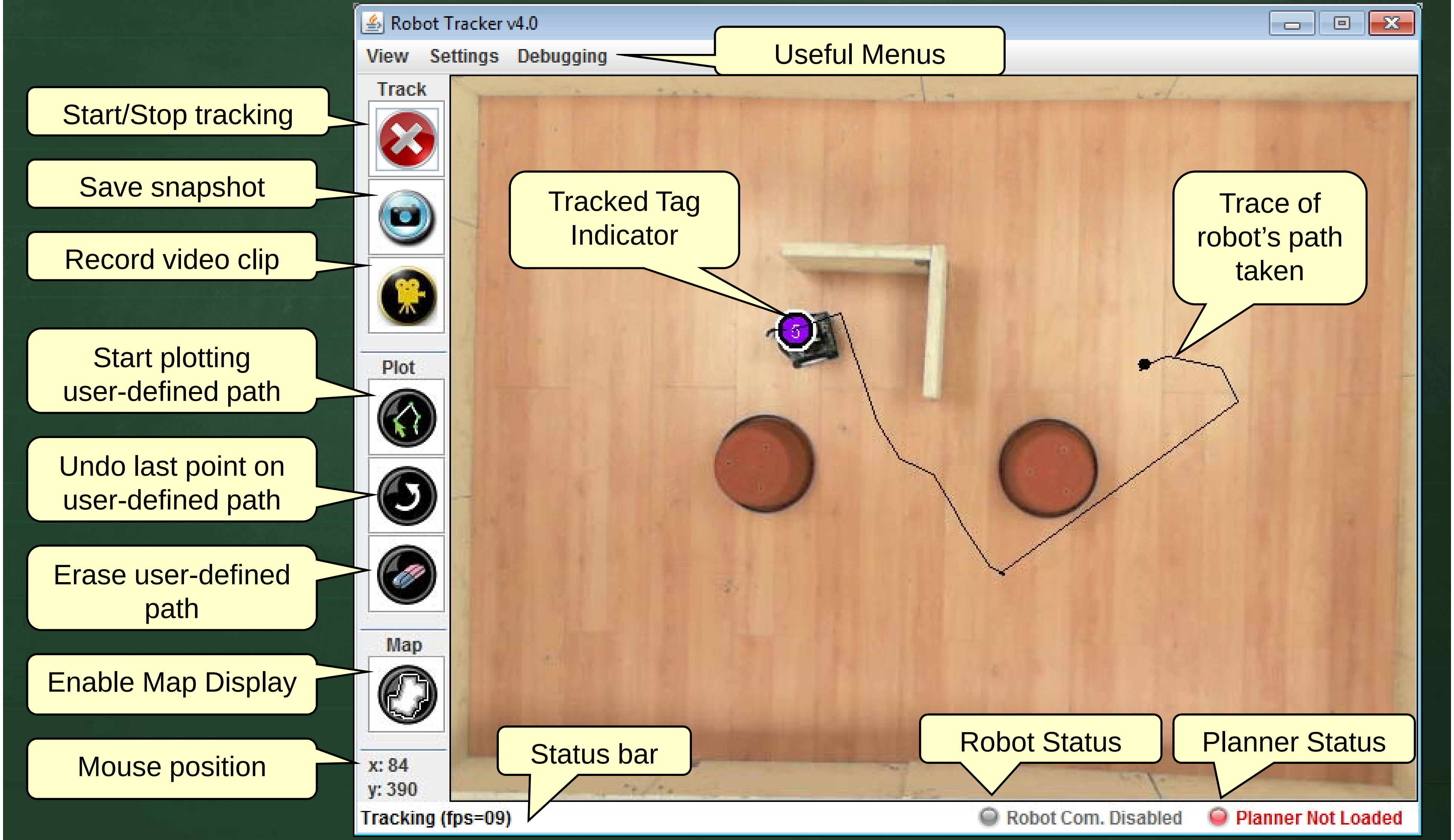
Each **Robot** communicates with one **RobotTracker** via bluetooth.



Robot Tracker - Advantages

- The **Robot Tracker** is quite useful. It allows you to …
  - send data to the robot wirelessly (e.g., tracked position)
  - receive data from the robot wirelessly (e.g., map data)
  - perform wireless debugging
  - do inter-robot communications
  - use JAVA code (called a Planner) to plan your robot's movements
  - send data to the PC for display (e.g., estimated path)
  - display mapping data with full Gaussian distributions
- It handles all bluetooth communication between the robot and the PC.

Robot Tracker – GUI



x: 84
y: 390

Tracking (fps=09)

Robot Com. Disabled

Planner Not Loaded

Useful Menus

Start/Stop tracking

Save snapshot

Record video clip

Start plotting user-defined path

Undo last point on user-defined path

Erase user-defined path

Enable Map Display

Mouse position

Tracked Tag Indicator

Trace of robot’s path taken

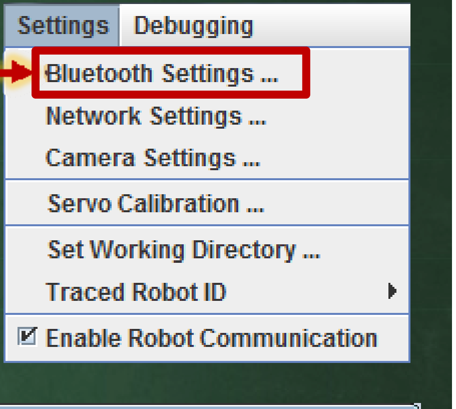
Status bar

Robot Status

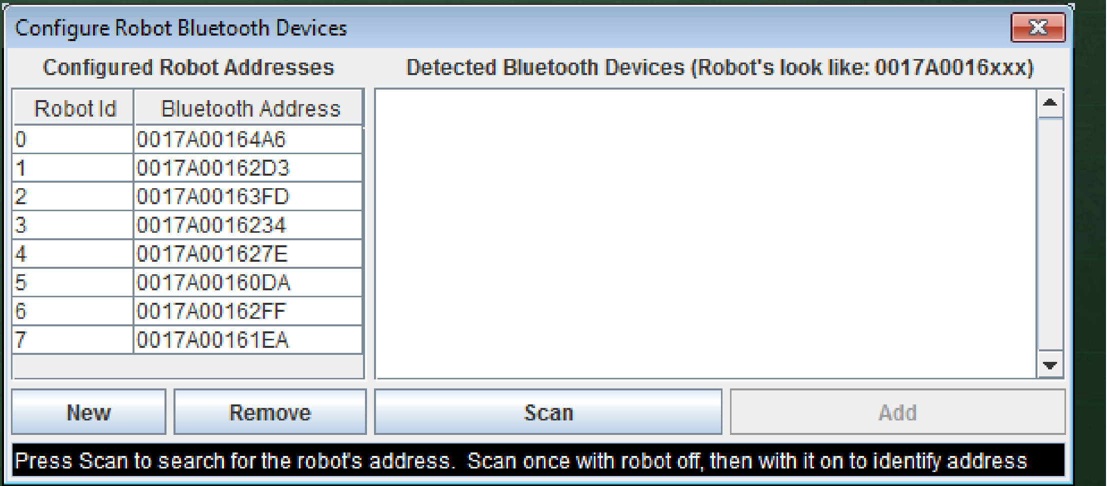
Planner Status

Bluetooth Setup

- Bluetooth devices must be configured one time in Settings menu



- All devices addresses need to be registered (you should not need to make any changes here)



Configure Robot Bluetooth Devices

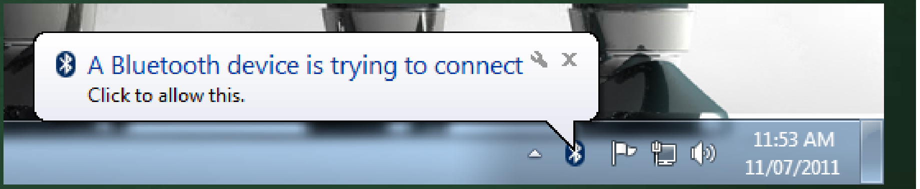
Configured Robot Addresses

Robot Id	Bluetooth Address
0	0017A00164A6
1	0017A00162D3
2	0017A00163FD
3	0017A0016234
4	0017A001627E
5	0017A00160DA
6	0017A00162FF
7	0017A00161EA

Detected Bluetooth Devices (Robot's look like: 0017A0016xxx)

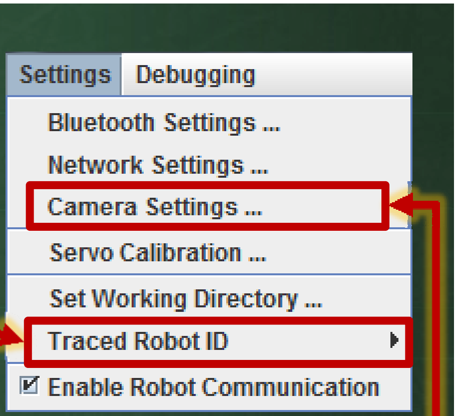
Press Scan to search for the robot's address. Scan once with robot off, then with it on to identify address

- First time connections will require a pairing code … which is 0000



A Bluetooth device is trying to connect
Click to allow this.

Robot Tracker – Setup



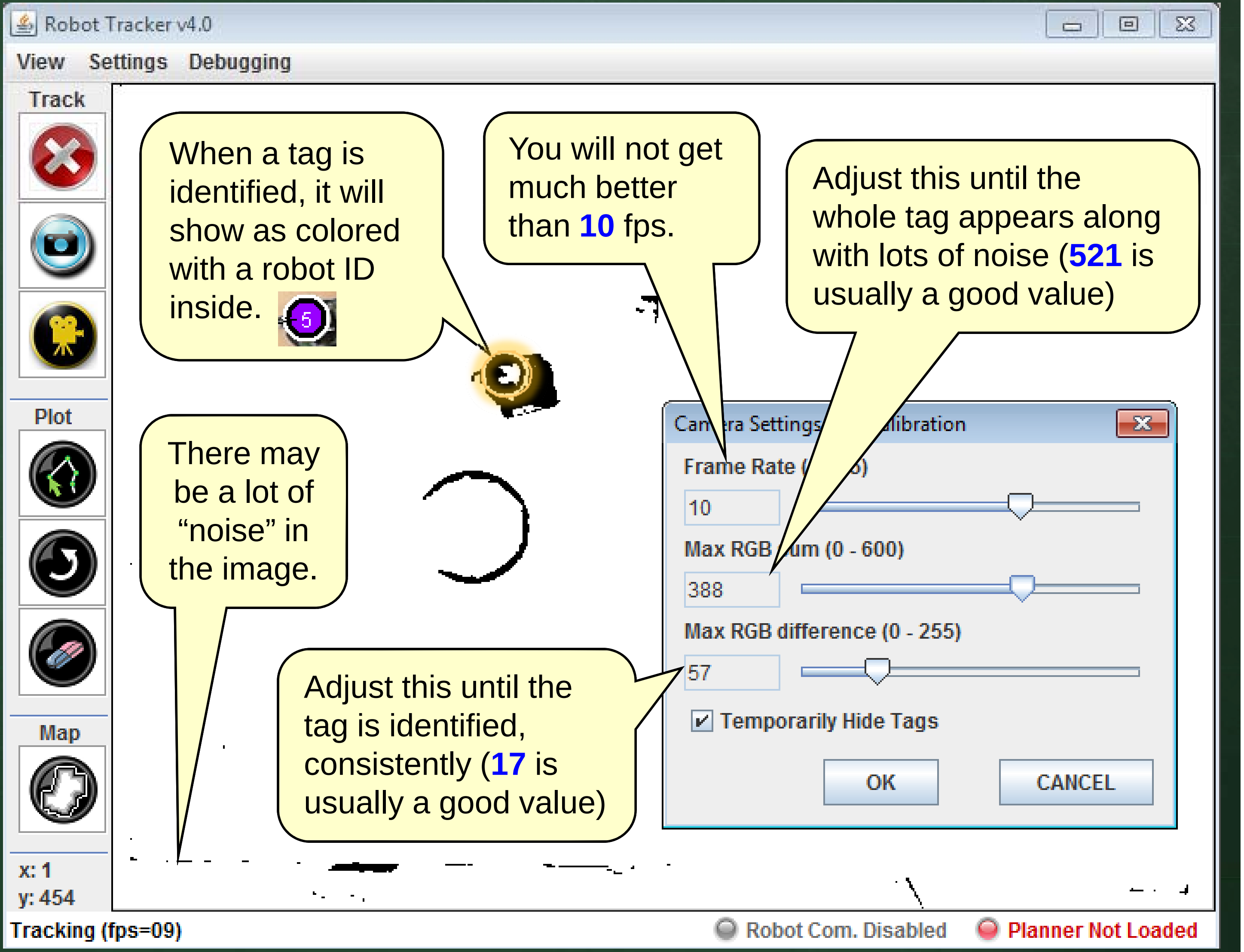
- Before each lab session, start the tracker and select the Traced Robot ID from the Settings menu to make sure that the ID correctly matches the robot that you are using.

- You will likely need to calibrate the camera under the (Camera Settings option) since it sometimes has a hard time identifying the tags … see next slide for details. Also, each time the computer has been restarted, you may need to re-configure the properties of the Microsoft Lifecam … here are good values

Resolution = 800x448
Settings:
 truecolor = off
 Properties ... Camera Control tab:
  Focus = 0 Auto = off
  Zoom = 32 Pan = 0 Tilt = 0
 Video Settings tab:
  truecolor = off
  Brightness = 161
  White Balance = 4250 Auto = off
  Saturation = 80
  Exposure = Auto = on
  Contrast = 5
  Powerline Frequence (Anti Flicker) = 60Hz

Robot Tracker – Calibration

- Calibrating the camera settings can be tricky.
  - lighting conditions plays a huge role



When a tag is identified, it will show as colored with a robot ID inside.

You will not get much better than **10** fps.

Adjust this until the whole tag appears along with lots of noise (**521** is usually a good value)

There may be a lot of “noise” in the image.

Adjust this until the tag is identified, consistently (**17** is usually a good value)

Can era Settings libration

Frame Rate (s): 10

Max RGB um (0 - 600): 388

Max RGB difference (0 - 255): 57

☑ Temporarily Hide Tags

x: 1
y: 454

Tracking (fps=09)

● Robot Com. Disabled ● Planner Not Loaded

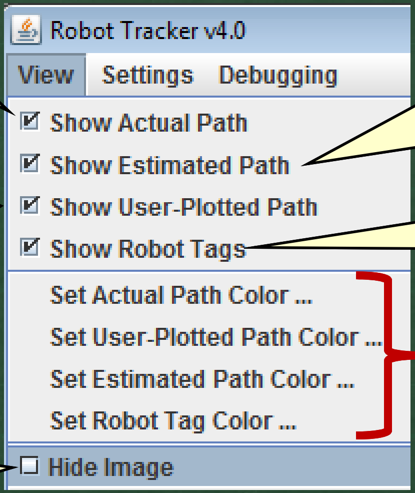
Robot Tracker – View Menu

- The *View* menu allows you to display various things:

The **Actual Path** that the robot traveled since it started.

The **User-Plotted Path** is a path provided by clicking at various locations on the screen. This path can be passed into your program.

The latest webcam image can be hidden or shown at any time.



The **Estimated Path** is a path provided by your code indicating the position that the robot "thought" that it was in as it moved … more later …

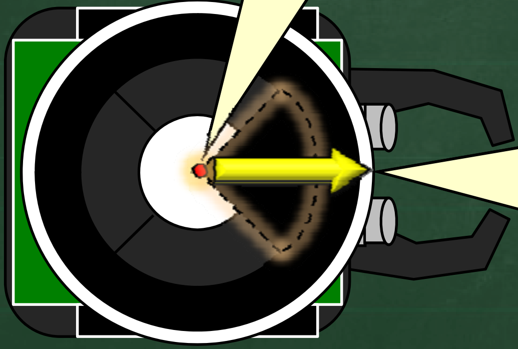
The **Robot Tag** markers can be shown or hidden at any time.

The colors of all paths (and tags) can be adjusted.

Robot Tracker -Pose Information

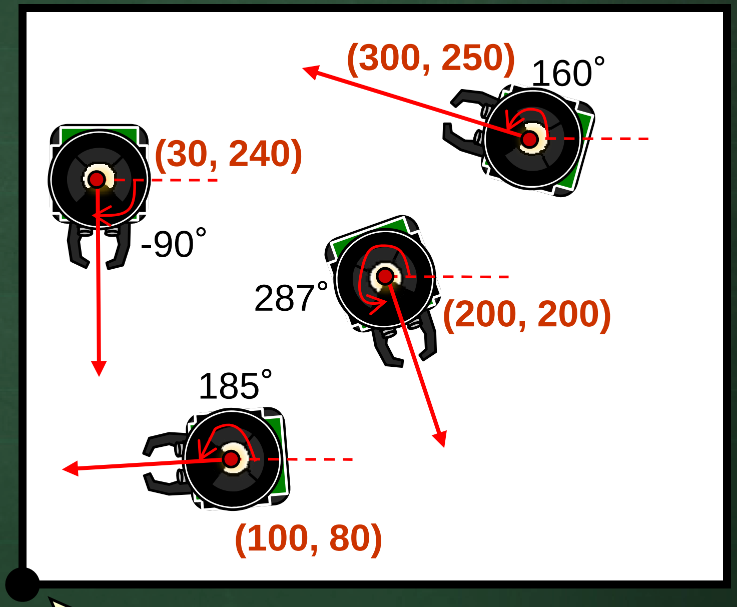
- The angle is computed with respect to the horizontal (positive x-axis) of camera's image.

Robot's tracked position
(x, y)



Tracked **angle.**

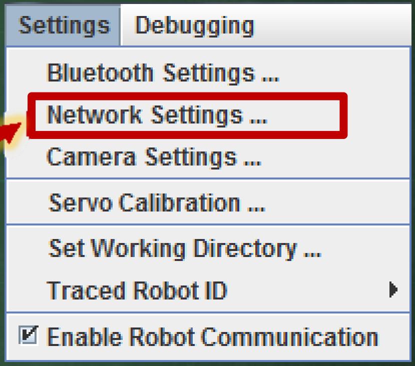
Make sure that the black wedge is facing forward (sometimes the screw holding the tag becomes loose).



The origin **(0,0)** is always at the bottom left of the screen.

Robot Tracker – Networked Tracking

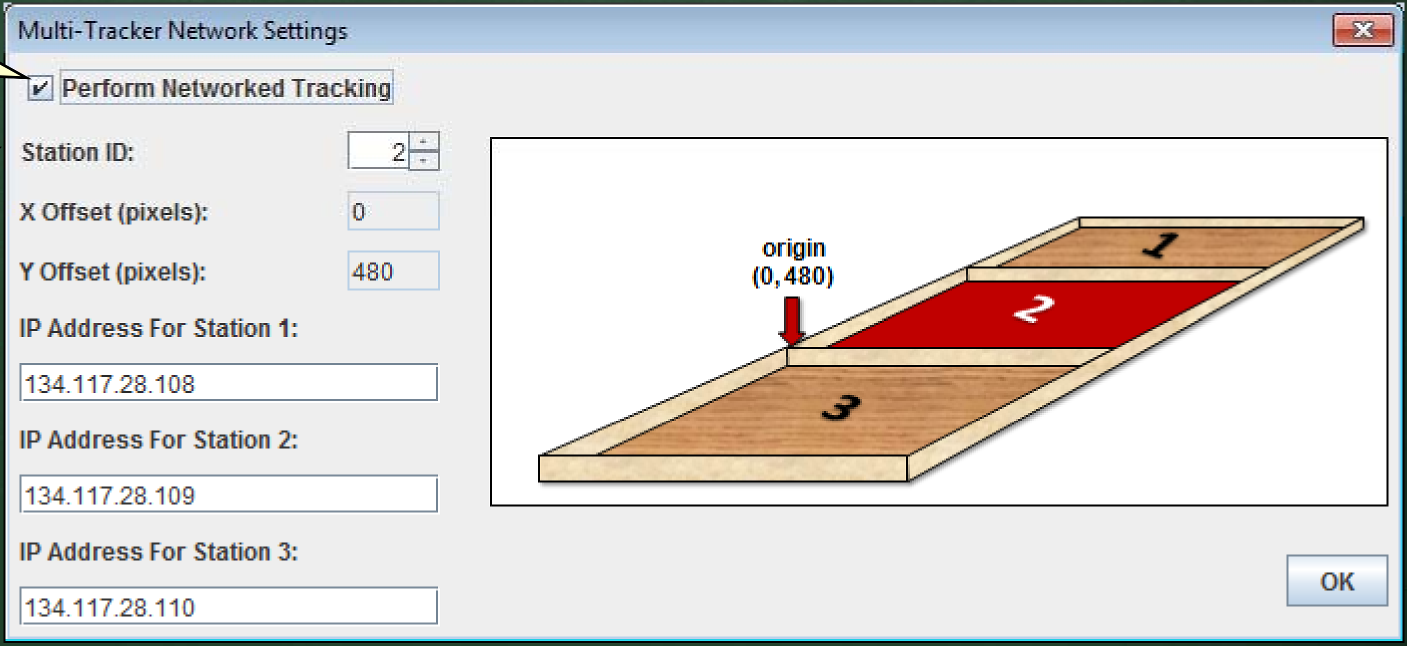
- The Robot Tracker allows you to track your robot in all 3 zones
  - choose Network Settings from Settings menu
  - Tracker must be running on ALL 3 machines.



Disable this when using a single tracker.

Each computer has unique station ID (see picture).

Each computer has unique IP address which should not be changed.



Multi-Tracker Network Settings

☑ Perform Networked Tracking

Station ID: 2

X Offset (pixels): 0

Y Offset (pixels): 480

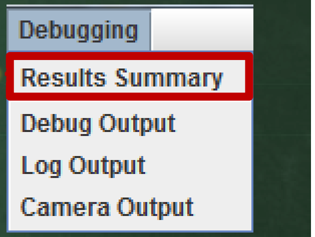
IP Address For Station 1: 134.117.28.108

IP Address For Station 2: 134.117.28.109

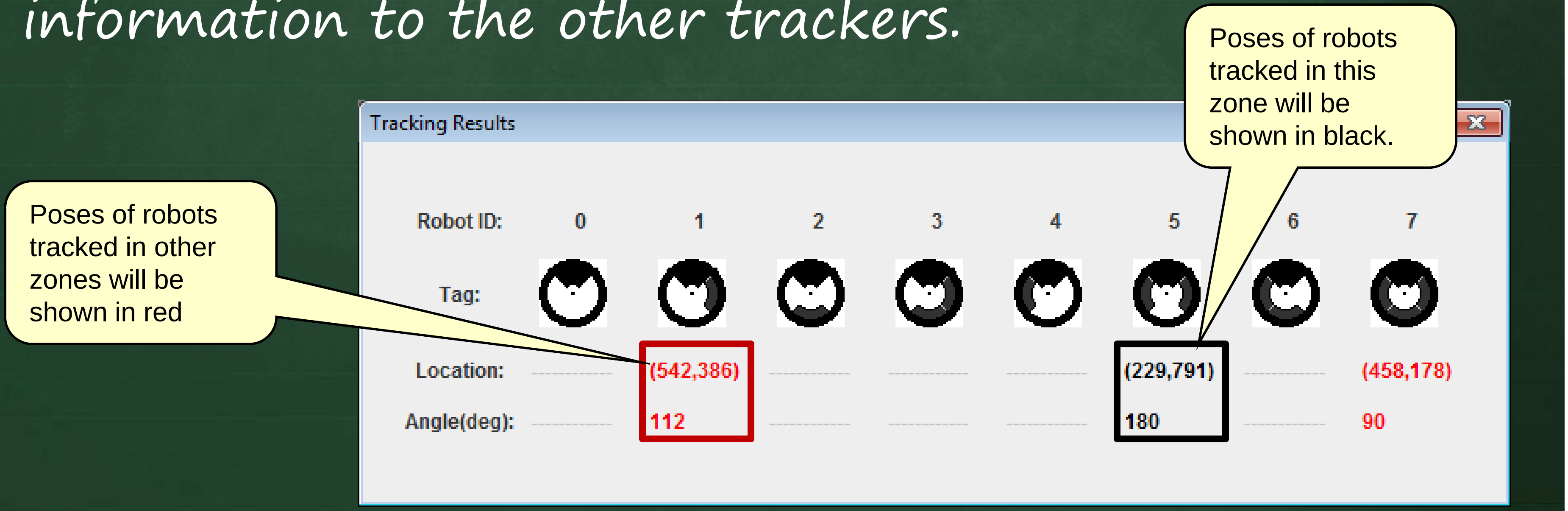
IP Address For Station 3: 134.117.28.110

Robot Tracker – Tracked Results

- The pose of each tracked robot may be monitored in real time by selecting Results Summary from the **Debugging** menu.



- Each tracker constantly sends updated pose information to the other trackers.



Tracking Results

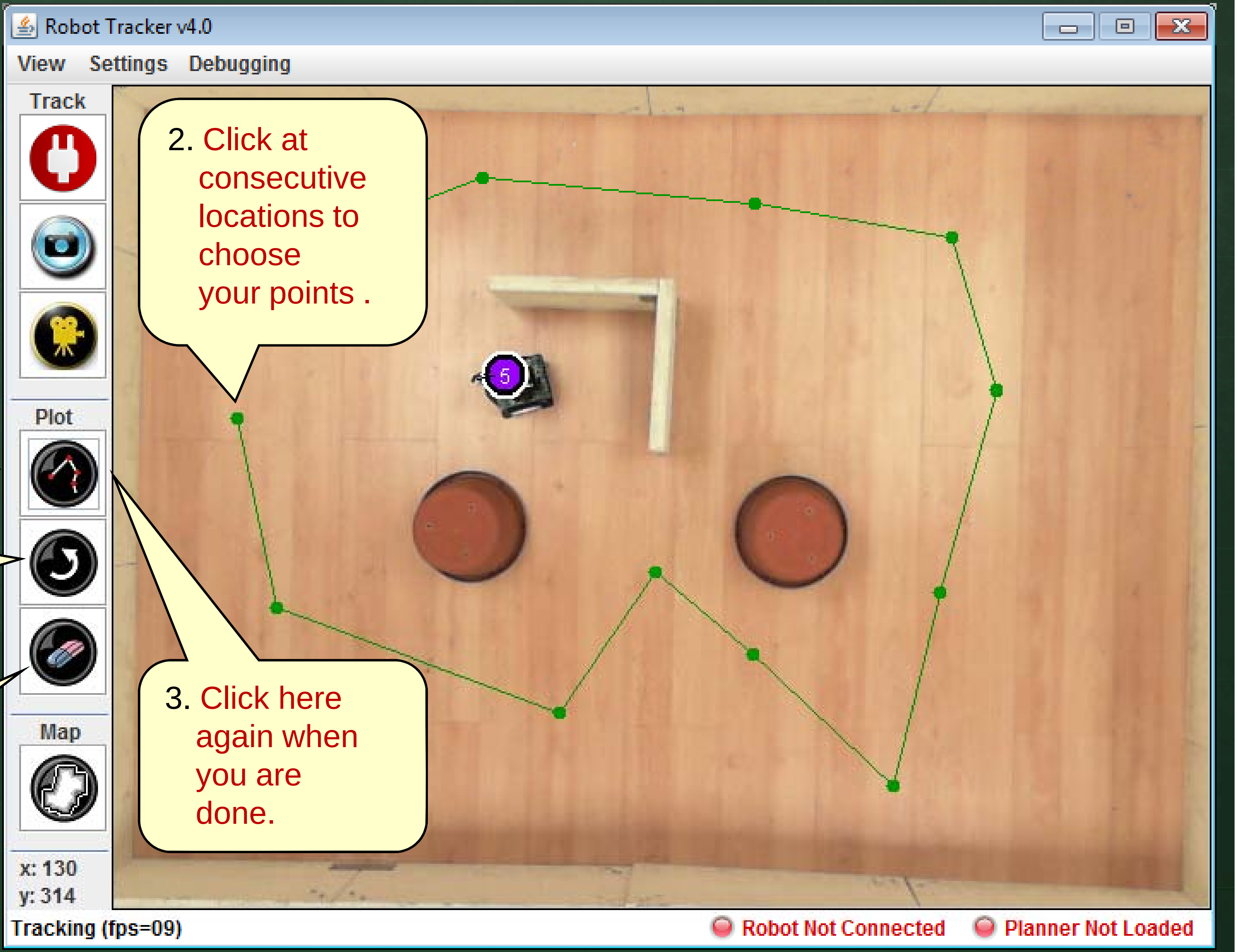
Robot ID:	0	1	2	3	4	5	6	7
Tag:								
Location:	------------	(542,386)	------------	------------	------------	(229,791)	------------	(458,178)
Angle(deg):	------------	112	------------	------------	------------	180	------------	90

Poses of robots tracked in other zones will be shown in red

Poses of robots tracked in this zone will be shown in black.

Robot Tracker – User Paths

- You can create a path as a sequence of points:



x: 130
y: 314

Tracking (fps=09)

Robot Not Connected

Planner Not Loaded

1. Click here to begin plotting.

2. Click at consecutive locations to choose your points .

3. Click here again when you are done.

Click to remove last point added to the path.

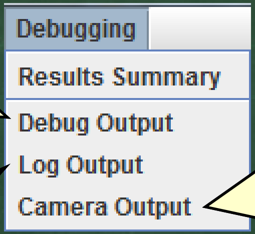
Click here to erase the whole path.

Robot Tracker – Debug Dialogs

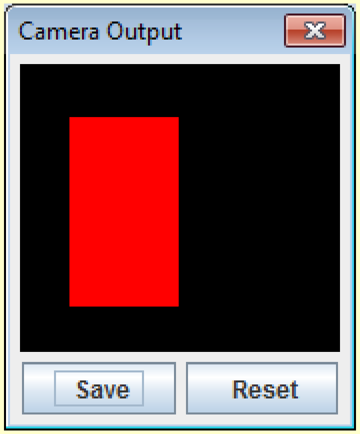
- There are other dialog boxes (available from the *Debugging* menu) that are useful:

The **Debug Output** dialog can be used to display debug data coming from the robot. This data is sent wirelessly.

The **Log Output** dialog can be used to display data being sent to the log file.

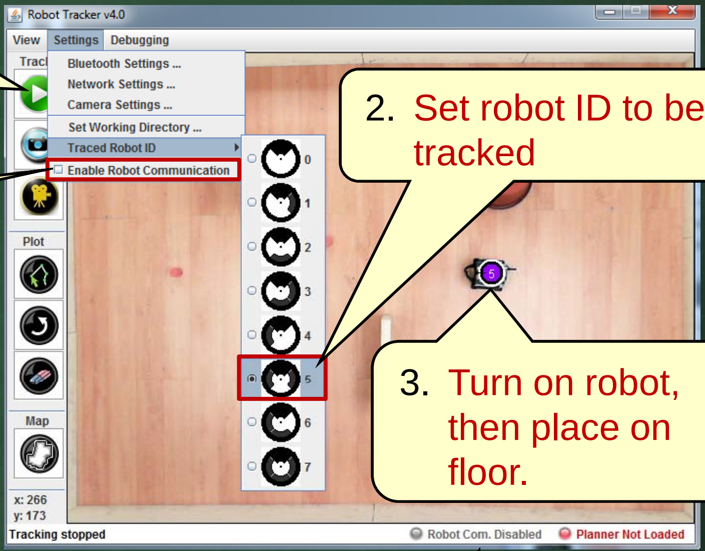


The **Camera Output** can be used to display tracked blobs from the robot's CMU camera (but it does not display a screen capture of the image).



Simple Robot Tracking

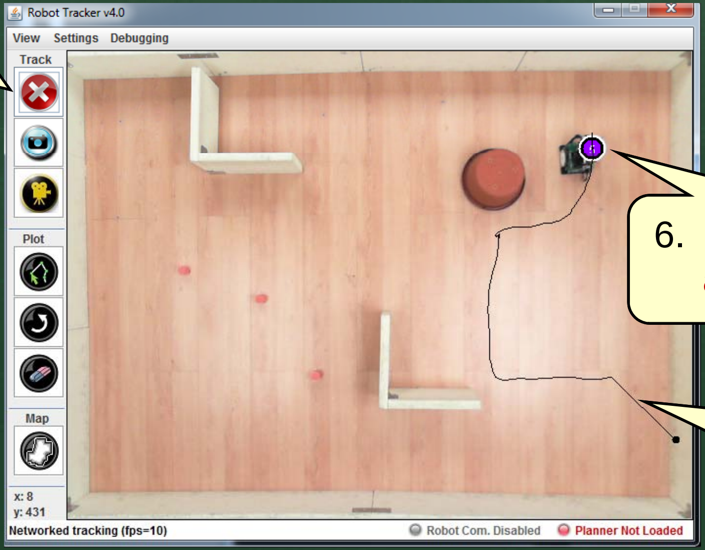
- To track a robot without wireless debugging follow these steps in order:



Tracking stopped

1. Disable robot communications
2. Set robot ID to be tracked
3. Turn on robot, then place on floor.
4. Press **Play** button

Indicates that robot is disabled



Networked tracking (fps=10)

5. Press **Stop** button when you are done.
6. Pick up robot and turn it off.

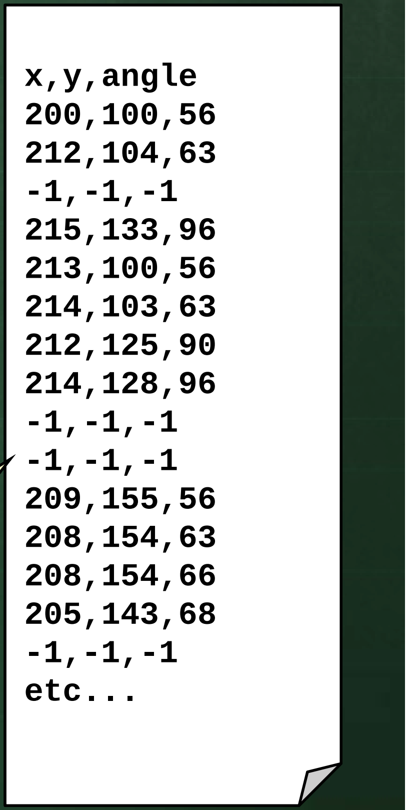
Robot's path will be displayed as long as **Show Actual Path** is selected in the **View** menu.

Simple Robot Tracking

- As the RobotTracker is running, it constantly writes the robot's pose data (i.e., x, y, angle) to a trace file:
- This file will be written in the Working Directory which is set from the Settings menu.
- Each time the RobotTracker is stopped and restarted, a new trace file is created.
- The files are automatically numbered and named in sequence as follows:

```
trace1.trc
trace2.trc
trace3.trc
```

etc..



```
x,y,angle
200,100,56
212,104,63
-1,-1,-1
215,133,96
213,100,56
214,103,63
212,125,90
214,128,96
-1,-1,-1
-1,-1,-1
209,155,56
208,154,63
208,154,66
205,143,68
-1,-1,-1
etc...
```

-1,-1,-1 indicates that robot's tag was not identified during that frame … usually due to lighting conditions, or obstructions.

Robot Tracking – With Debugging

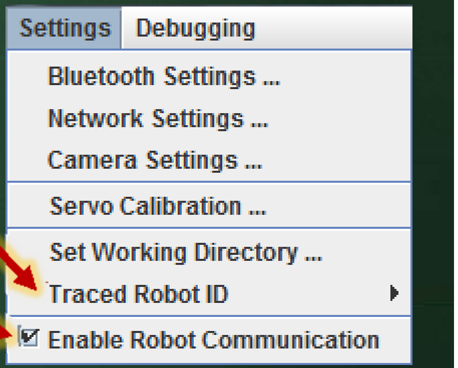
1. Your SPIN code must always connect via bluetooth to the PC. You must use RBC.spin to do this.

```
OBJ
  RBC:      "RBC"       'Required to communicate with PC
  Beeper    "Beeper"    'Required to use the beeper

PUB main
  Beeper.Startup       'Make a "Starting Up" sound (good idea to do this)

  RBC.Init             'Connect to PC and wait until Play button is pressed

  ...
```

Your code will stop/block here until the **Play** button ▶ is pressed on the RobotTracker.

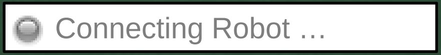
2. Ensure that the correct Traced Robot ID has been selected.

3. Ensure that ***Enable Robot Communication*** checkbox is selected on ***Settings*** menu

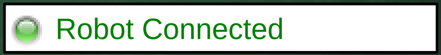


Robot Tracking – With Debugging

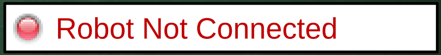
4. Turn on the robot.
5. Press the connection button: 🔌
   - Connection button will turn to hourglass ⌛
   - Robot status bar will say


● Connecting Robot …

6. Wait until robot connects:
   - If connection worked, status bar will say


● Robot Connected

   - If connection timed-out, status bar will say

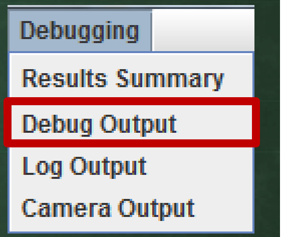

● Robot Not Connected

   - Ensure robot turned on and that Traced Robot ID is proper.

7. Press the **Play** ▶ button to start the robot.
8. Use the **Stop** ✖ button when done.
9. Pick up robot and turn it off.

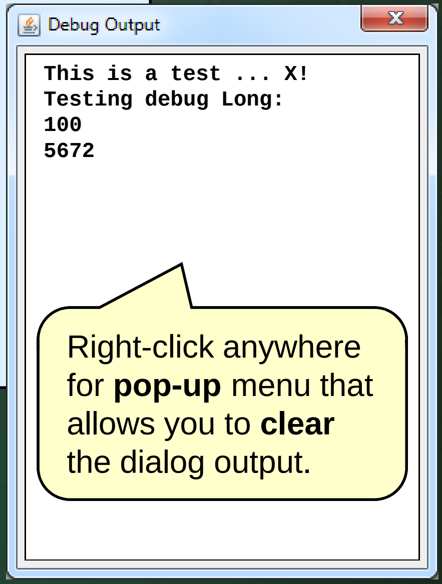
Robot Tracking – With Debugging

- All debug output will appear in **Debug Output** dialog box, selected from **Debugging** menu



```
OBJ
  RBC:    "RBC"        'Required to communicate with PC

PUB main
  RBC.Init    'Connect to PC and wait

  RBC.DebugClear
  RBC.DebugStr(string("This is a test ... "))
  RBC.DebugChar("X")
  RBC.DebugCharCr("!")
  RBC.DebugStrCr(string("Testing debug Long: "))
  RBC.DebugLongCr(100)
  RBC.DebugLong(5672)
  RBC.DebugCr
```

All of the available debug display-related commands are used in this example.

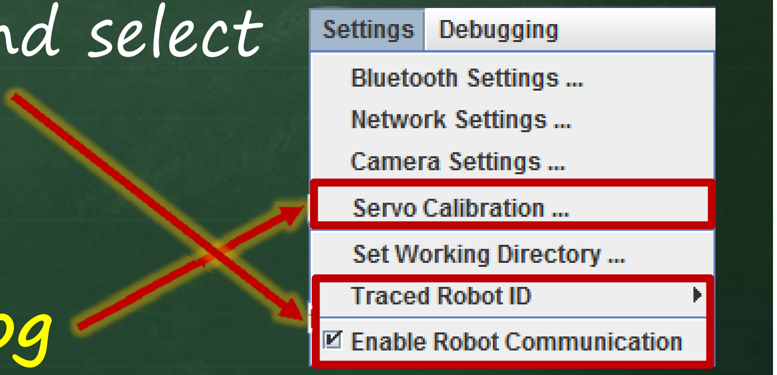


```
This is a test ... X!
Testing debug Long:
100
5672
```

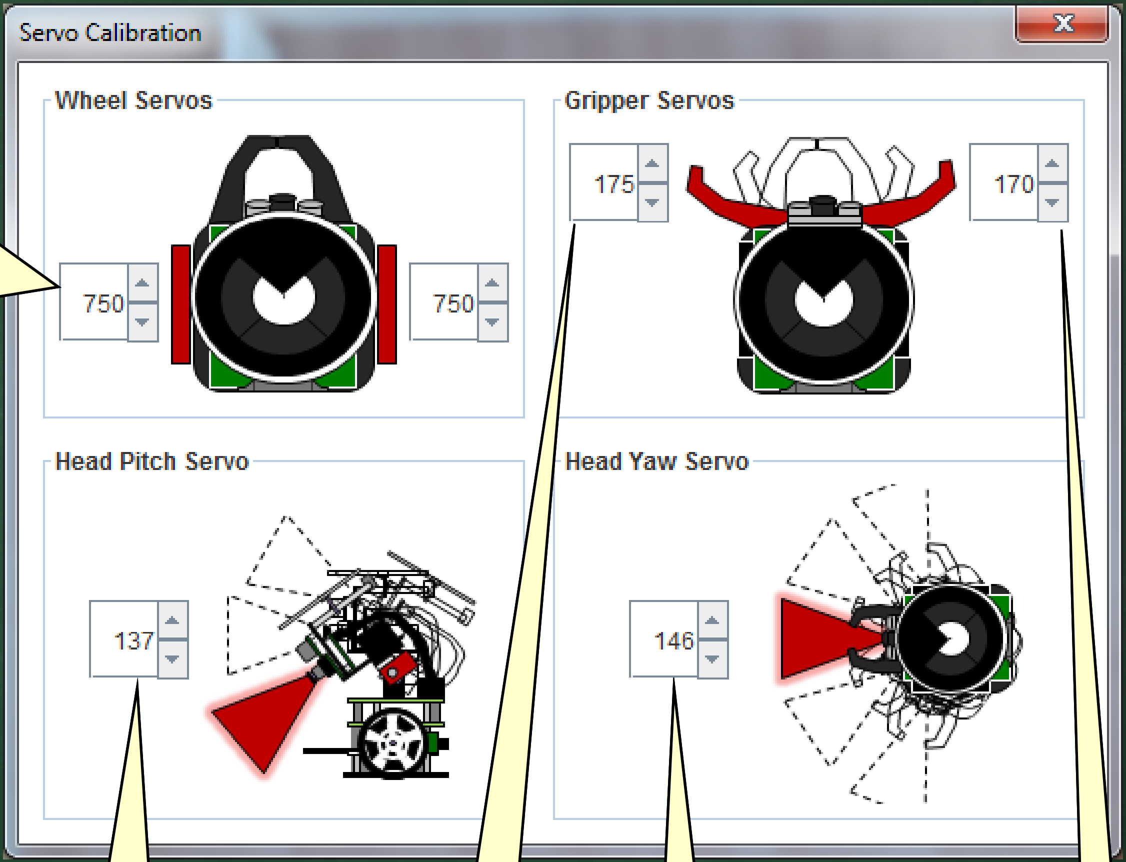
Right-click anywhere for **pop-up** menu that allows you to **clear** the dialog output.

Robot Tracker – Servo Calibration

- Each robot has slightly different servos which are off a little with respect to the values that stop them from moving.
- You can determine these "stopped values" by using the ServoCalibration.spin code (available on the course website).
- Enable robot communications and select the correct robot id as before.
- Connect 🔌 the robot as before
- Open the **Servo Calibration Dialog**
- Press the Play ▶ button.



Robot Tracker – Servo Calibration



Use the up/down sliders for each **wheel servo** (one at a time) to determine which number causes the wheel to stop moving. There will be a range of values that all cause the servo to remain still. Choose the middle value of this range as the stopped value for that servo **(see slide 3-76).** You should check these numbers each time you change robots.

Use the other up/down sliders to make fine-tuned adjustments for the various pre-defined constants for the **gripper** and **head** servos **(see slide 3-75).** You may want to make such fine adjustments each time you change robots.

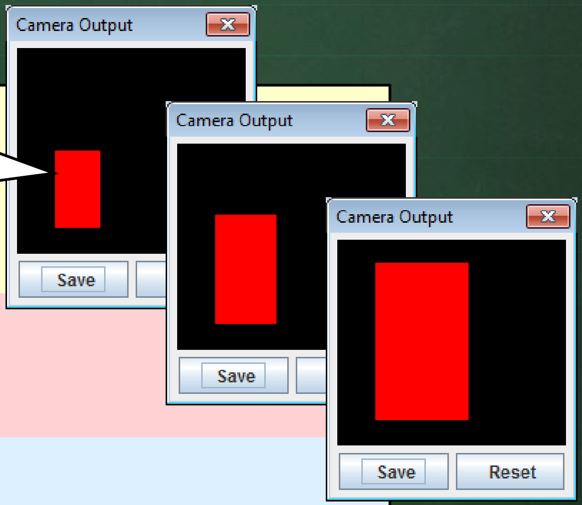
CMUCam Monitoring

- You can also use the debugger to get feedback from the CMUcam:

```
CON
  RED = 189
  GREEN = 19
  BLUE = 16
  SENSITIVITY = 30

OBJ
  RBC:    "RBC"
  CAM:    "CMUCam"

PUB main
  RBC.Init    'Connect to PC and wait for "Start Robot"

  CAM.Start
  CAM.SetTrackColor(RED,GREEN,BLUE,SENSITIVITY)  'Set color to track

  RBC.SendTrackedColorToPc(RED,GREEN,BLUE)  'Send color to RBC
  repeat
    CAM.TrackColor   'Track the color on the camera
    RBC.SendTrackedDataToPc(CAM.GetTopLeftX, CAM.GetTopLeftY,
                            CAM.GetBottomRightX, CAM.GetBottomRightY)
```

See next slide

Bounding box will get bigger as robot approaches the block.



Sends tracking color to RBC for display purposes only.

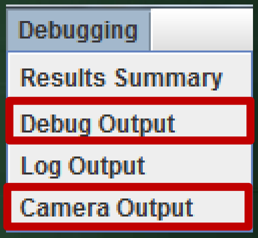
Sends the bounding box of the tracked blob for display purposes.

CMUCam Monitoring

- You can also use the **CameraColorSampler.spin** code (available on the course website) to determine the color values that you want to track.

```
CON
  RED = 189
  GREEN = 19
  BLUE = 16
  SENSITIVITY = 30

PUB main
  ...
  CAM.SetTrackColor(RED,GREEN,BLUE,SENSITIVITY)  'Set color to track
```

- Hold the object that you want to track about **5cm** from the robot's camera and look at the **Debug Output** and **Camera Output** window to see the values that are being read in from the camera.



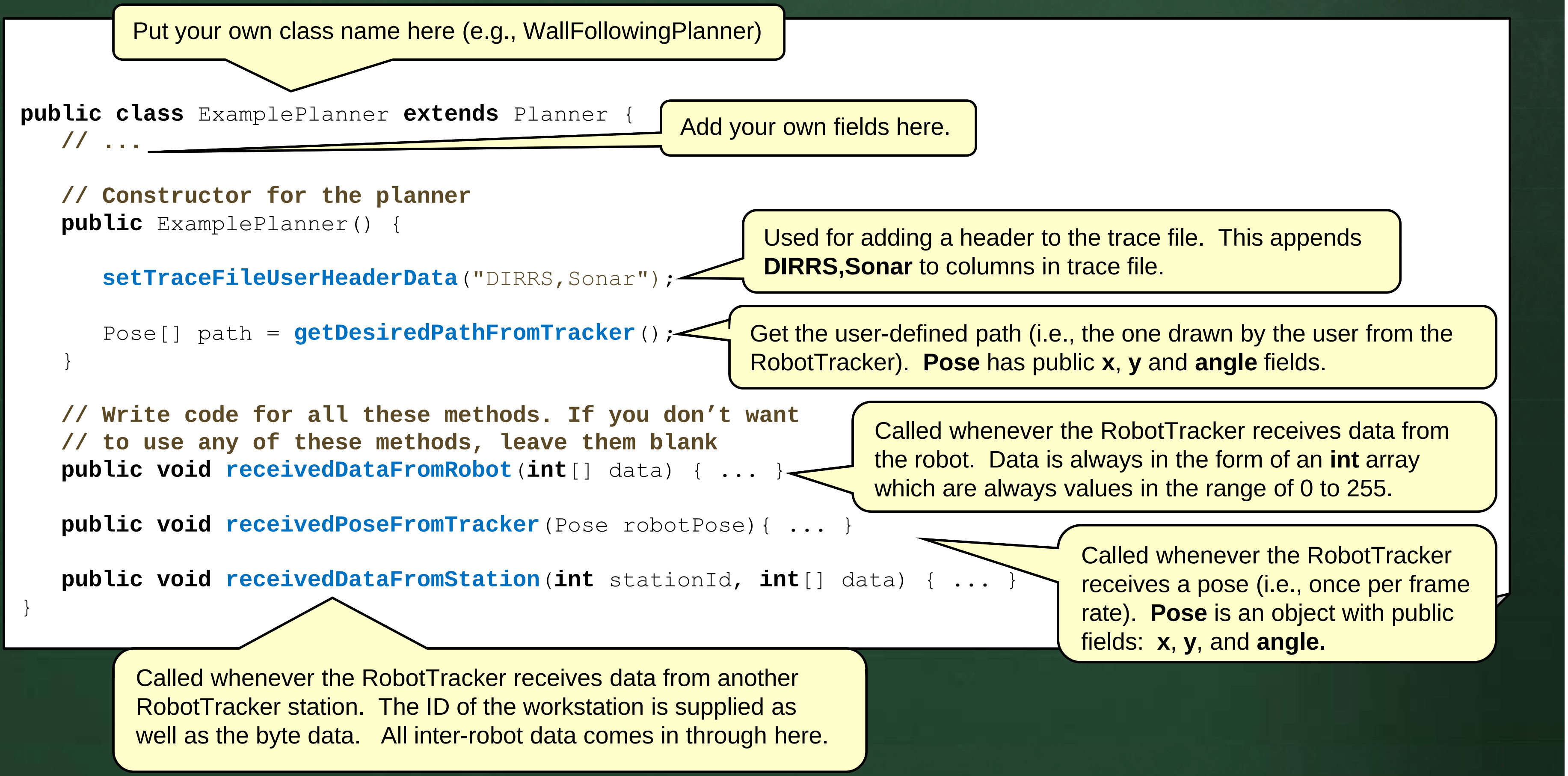
Robot Data Transfer

- In addition to debugging data, you can also send/receive data to/from the PC arbitrarily
  - store sensor readings to a file
  - get and use location from RobotTracker
  - send computed data back to PC (e.g., estimated position)

- Communicating with the PC in this way requires you to write a Planner
  - a Planner is JAVA code that communicates with the robot through the RBC.

Using a Planner

- Here is a template for writing a planner:



Put your own class name here (e.g., WallFollowingPlanner)

```
public class ExamplePlanner extends Planner {
  // ...

  // Constructor for the planner
  public ExamplePlanner() {

    setTraceFileUserHeaderData("DIRRS,Sonar");

    Pose[] path = getDesiredPathFromTracker();
  }

  // Write code for all these methods. If you don't want
  // to use any of these methods, leave them blank
  public void receivedDataFromRobot(int[] data) { ... }

  public void receivedPoseFromTracker(Pose robotPose){ ... }

  public void receivedDataFromStation(int stationId, int[] data) { ... }
}
```

Add your own fields here.

Used for adding a header to the trace file. This appends **DIRRS,Sonar** to columns in trace file.

Get the user-defined path (i.e., the one drawn by the user from the RobotTracker). **Pose** has public **x**, **y** and **angle** fields.

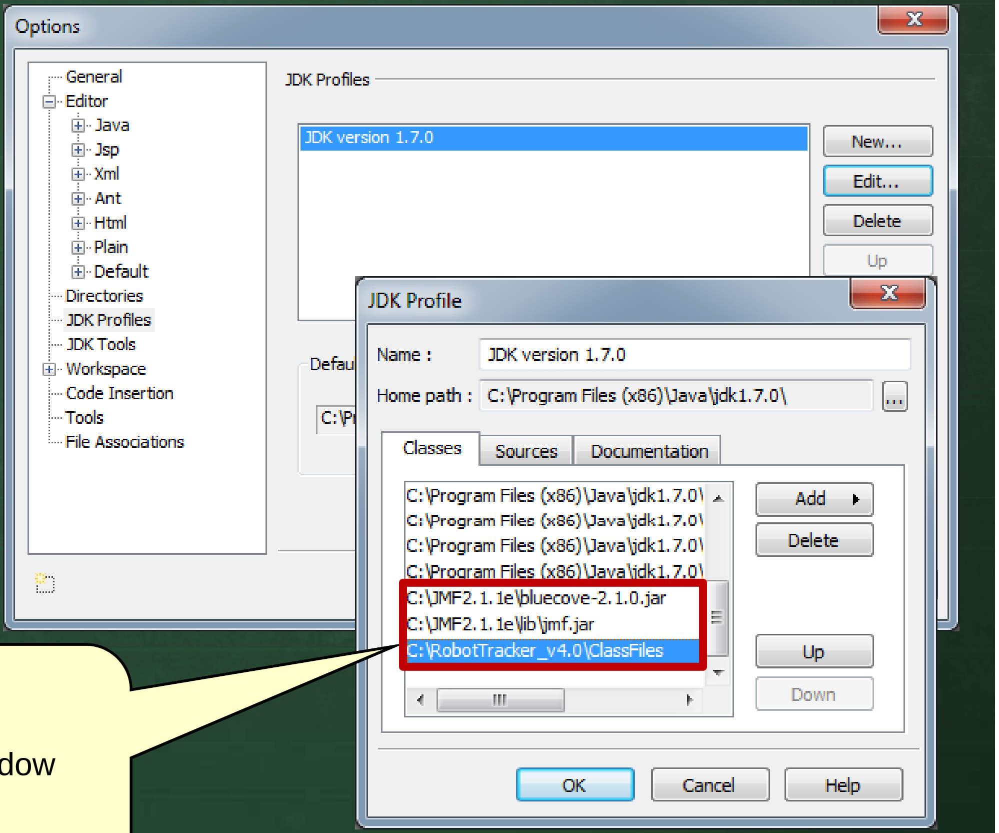
Called whenever the RobotTracker receives data from the robot. Data is always in the form of an **int** array which are always values in the range of 0 to 255.

Called whenever the RobotTracker receives a pose (i.e., once per frame rate). **Pose** is an object with public fields: **x**, **y**, and **angle.**

Called whenever the RobotTracker receives data from another RobotTracker station. The ID of the workstation is supplied as well as the byte data. All inter-robot data comes in through here.

Compiling Your Planner Code

- Your planner code must be compiled on its own.
- You will need to include the necessary .jar files and also include in the class path the folder that contains the compiled RobotTracker classes:



JDK version 1.7.0

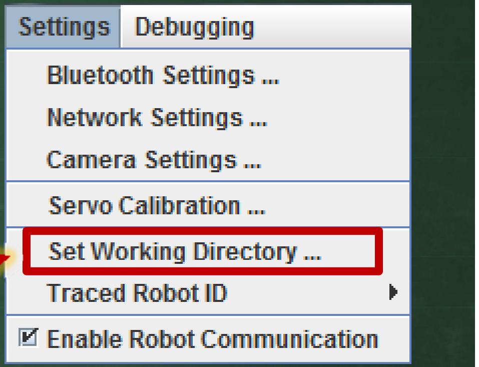
Name : JDK version 1.7.0

Home path : C:\Program Files (x86)\Java\jdk1.7.0\

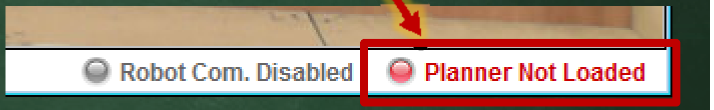
```
C:\Program Files (x86)\Java\jdk1.7.0\
C:\Program Files (x86)\Java\jdk1.7.0\
C:\Program Files (x86)\Java\jdk1.7.0\
C:\Program Files (x86)\Java\jdk1.7.0\
C:\JMF2.1.1e\bluecove-2.1.0.jar
C:\JMF2.1.1e\lib\jmf.jar
C:\RobotTracker_v4.0\ClassFiles
```

In JCreator, add the archive files **jmf.jar** and **bluecove-2.1.0.jar** files and the path to **C:\RobotTracker_v4.0\ClassFiles**. The window here was obtained by selecting **Configure/ Options…/ JDK Profiles** and select the JDK installed and then pressing the **Edit…** button.

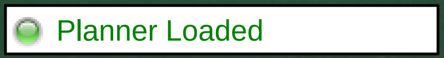
Using the Planner



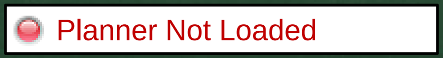
1. Load up the planner
   - Choose the Working Directory from the Settings menu. This should always point to the folder that contains your assignment work.
   - Double-click on **Planner Not Loaded** (at the bottom right corner of the RobotTracker window) and choose the **compiled** class file for your planner.



     - If successful, status bar will say



     - In failed, status bar will say



       - Examine RobotTracker's dos prompt window for indication of error. Possibly, you forgot to include one of the necessary planner methods, or you may have spelled one incorrectly, or may have wrong parameters.

2. Turn on the robot, establish the connection and press Play ▶ button as before.

Changes to Your Planner Code

- As you test your code, you will often re-compile your planner code.
- Each time you make changes and re-compile, you MUST re-load the planner by double clicking on the Planner Status bar at the bottom right of the RobotTracker window, even though it may already indicate Planner Loaded (i.e., it is the old version that is currently loaded and you need the new version).
- Ensure that the Stop button ✖ has been pressed before you re-load and that the robot has been reset before trying to re-establish the connection.

Planner Example 1

- Example that repeatedly receives RobotTracker poses and prints that pose information on the PC using the wireless debugger.

```
public class PlannerEx1 extends Planner {

   public PlannerEx1() {...}

   ...

   public void receivedPoseFromTracker(Pose p) {
      byte[] outData = new byte[6];
      outData[0] = (byte)(p.x / 256);
      outData[1] = (byte)(p.x % 256);
      outData[2] = (byte)(p.y / 256);
      outData[3] = (byte)(p.y % 256);
      outData[4] = (byte)(p.angle / 256);
      outData[5] = (byte)(p.angle % 256);
      sendDataToRobot(outData);
   }
}
```

This method is called repeatedly at the frame rate set in the RobotTracker settings.

This code assumes that the **x**, **y** and **angle** values are all positive.

Planner method that sends a **byte**[] to the robot.

```
OBJ
  RBC:    "RBC"

VAR
  byte dataIn[7]

PUB main | size
  RBC.Init

  repeat
    RBC.ReceiveData(@dataIn)

    size := dataIn[0]
    RBC.DebugStr("(")
    RBC.DebugLong(dataIn[1]*256 + dataIn[2])

    RBC.DebugStr(",")
    RBC.DebugLong(dataIn[3]*256 + dataIn[4])

    RBC.DebugStr(string(") angle: "))
    RBC.DebugLong(dataIn[5]*256 + dataIn[6])

    RBC.DebugStr(string(" degrees."))
    RBC.DebugCr
```

Wait for the pose.

First byte received is # of bytes sent from the planner.

Rebuild 2 bytes into a word and display. This code assumes positive **x**, **y** and **angle**.

Planner Example 2

- Example of displaying an estimated pose on the PC:

```
public class PlannerEx2 extends Planner {
   boolean firstPose;

   public PlannerEx2() {
      firstPose = true;
   }
   ...
   public void receivedPoseFromTracker(Pose p) {
     if (firstPose) {
        byte[] outData = new byte[6];
        outData[0] = (byte)(p.x / 256);
        outData[1] = (byte)(p.x % 256);
        outData[2] = (byte)(p.y / 256);
        outData[3] = (byte)(p.y % 256);
        outData[4] = (byte)(p.angle / 256);
        outData[5] = (byte)(p.angle % 256);
        sendDataToRobot(outData);
        firstPose = false;
     }
   }
   public void receivedDataFromRobot(int[] data) {
      int x = data[0]*256 + data[1];
      int y = data[2]*256 + data[3];
      int a = data[4]*256 + data[5];
      sendEstimatedPoseToTracker(x, y, a);
   }
}
```

Used to send the very first pose to the robot so that it knows its initial position.

Assumes that robot sends back estimated pose repeatedly.

Planner method that adds a pose to the estimated path. This path will appear on the RobotTracker assuming that **Show Estimated Path** is selected from the **View** menu.

```
OBJ
  RBC:    "RBC"

VAR
 byte dataIn[7]
 byte dataOut[6]
 long x, y, a

PUB main
  RBC.Init
  RBC.ReceiveData(@dataIn)
  x := dataIn[1]*256 + dataIn[2]
  y := dataIn[3]*256 + dataIn[4]
  a := dataIn[5]*256 + dataIn[6]
  repeat
    CalculatePose

    dataOut[0] := x / 256
    dataOut[1] := x // 256
    dataOut[2] := y / 256
    dataOut[3] := y // 256
    dataOut[4] := a / 256
    dataOut[5] := a // 256

    RBC.SendDataToPc(@dataOut, 6,
                          RBC#OUTPUT_TO_NONE)
```

Wait for 1st pose ... comes back as 7 bytes ... ignore 1st as it is the size.

CalculatePose is a private method that computes **x**, **y** and angle **a**. You need to write this …

Output options are:

```
OUTPUT_TO_LOG
OUTPUT_TO_FILE
OUTPUT_TO_LOG_AND_FILE
OUTPUT_TO_NONE
```

Planner Example 3

- Example of sending data to a trace file:

```
public class PlannerEx3 extends Planner {

   public PlannerEx3(RBCPlannerHandler handler) {
      ...
      setTraceFileUserHeaderData("Sonar,EncLeft,EncRight");
   }

   ...


   public void receivedDataFromRobot(int[] data) {
      int sonar = data[0];
      int el = data[1]*256 + data[2];
      int er = data[3]*256 + data[4];

      String traceData = "" + sonar + "," + el + "," + er;

      sendDataToTraceFile(traceData);
   }
}
```

This example assumes that the incoming robot data contains a 1 byte sonar reading followed by two 2-byte-words for the left and right encoder counters.

Planner method that writes data into the trace file. You should ensure to match the header format that you specified in the constructor.

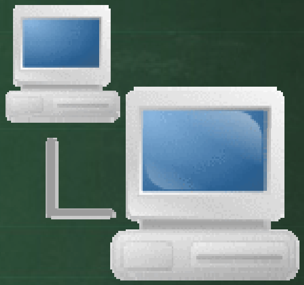
Text will match what is defined in the planner.

```
x,y,angle,Sonar,EncLeft,EncRight
200,100,56,73,17,28
212,104,63,72,29,32,71,36,38
-1,-1,-1,73,37,42
215,133,96,68,41,48,65,45,53,66,49,57
213,100,56,67,53,67,68,59,70
214,103,63,65,65,79,64,78,80,66,80,82
212,125,90,60,83,90
214,128,96,57,89,101
-1,-1,-1,54,99,112, 50,103,123,55,112,130
-1,-1,-1,48,116,140
209,155,56,65,119,152
208,154,63,72,130,157
208,154,66,89,139,166,90,150,181
205,143,68,101,167,190,102,184,204
-1,-1,-1,122,199,220
etc...
```

Sometimes the RobotTracker will miss certain poses of the robot (due to lighting changes, obstructions, noise etc…). In this case, the x, y and angle will all be -1, yet data sent to trace file will still appear.

Since robot data usually arrives quicker than robot poses, multiple readings will appear for the same pose line in the file.

Planner Example 4



- Example of sending and receiving data between workstations:
  - shows how one robot's data can be sent to the RobotTracker on a different workstation.

You need not send poses, but you can in fact send any data … perhaps various commands to coordinate other robots.

THIS CODE RUNS ON STATION 1:

In this example, any time a pose is received from the tracker on station 1, it is sent to station 2 and 3.

```java
public void receivedPoseFromTracker(Pose p) {
   String data = "(" + p.x + "," + p.y + "," + p.angle + ")";
   sendDataToStation(2, data);
   sendDataToStation(3, data);
}
```

THIS CODE RUNS ON STATIONS 2 AND 3:

In this example, any time data is received from station 1, it is simply printed out.

```java
public void receivedDataFromStation(int stationId, String data) {
   System.out.println("Received from Station " + stationId + ": " + data);
}
```

Planner Example 5

- Sending data to the PC for mapping purposes:

```
public class PlannerEx5 extends Planner {
   public PlannerEx5() {
     setTraceFileUserHeaderData("Dirrs+|0|-3|0|0.05|3,Sonar|0|4|0|0.10|19,IR6|90|-5|-5.5|0.85|5.5");
   }
   ...

   public void receivedDataFromRobot(int[] data) {
      int d = data[0]*256 + data[1];
      int s = data[2]*256 + data[3];
      int i = data[4]*256 + data[5];
      sendDataToTraceFile("" + d + "," + s + "," + i);
   }
}
```

RobotTracker expects a very particular format for the trace file header. This example assumes that the robot sends back data from three range sensors … the DIRRS+, the sonar and IR sensor #6 (i.e., right side at back). Each sensor specification is separated by a comma.

Each group of sensor data is made up of 6 pieces of information separated by the vertical bar | as follows:

```
name|angleOffset|xOffset|yOffset|distanceError|angularResolution
```

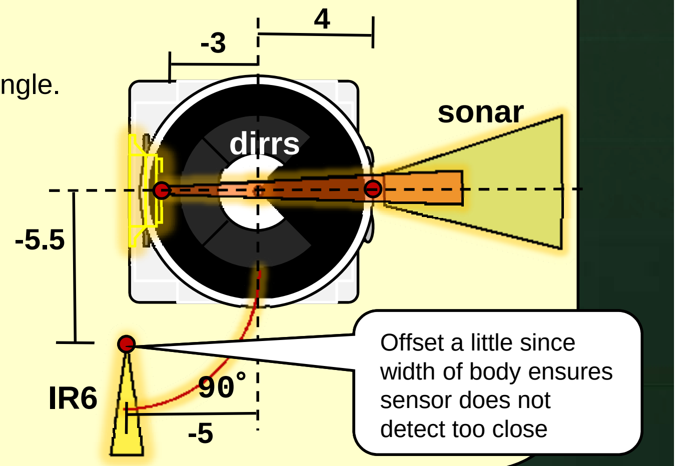
name ………………… Label given to this data set.

angleOffset ………… Angle (in degrees) that sensor is w.r.t. robot's forward facing angle. Notice that **sonar** and **dirrs** face in same direction as robot (i.e., 0° offset), while right side facing **IR6** is 90° from robot's forward direction ... should have been -90 ... oh well).

xOffset & **yOffset** ….. Distance (in cm) that sensor is w.r.t. robot's center. Notice that **sonar** is offset (4,0), **dirrs** is offset (-3,0) and **IR6** is offset (-5,-5.5).

distanceError ……… Error associated with measurement from sensor (**dirrs** is ±5%, **sonar** is ±10%, **IR6** is ±85%).

angularResolution …. Beam width (in degrees) of sensor (**dirrs** is 3° , **sonar** is 19° , **IR6** is 5.5°).

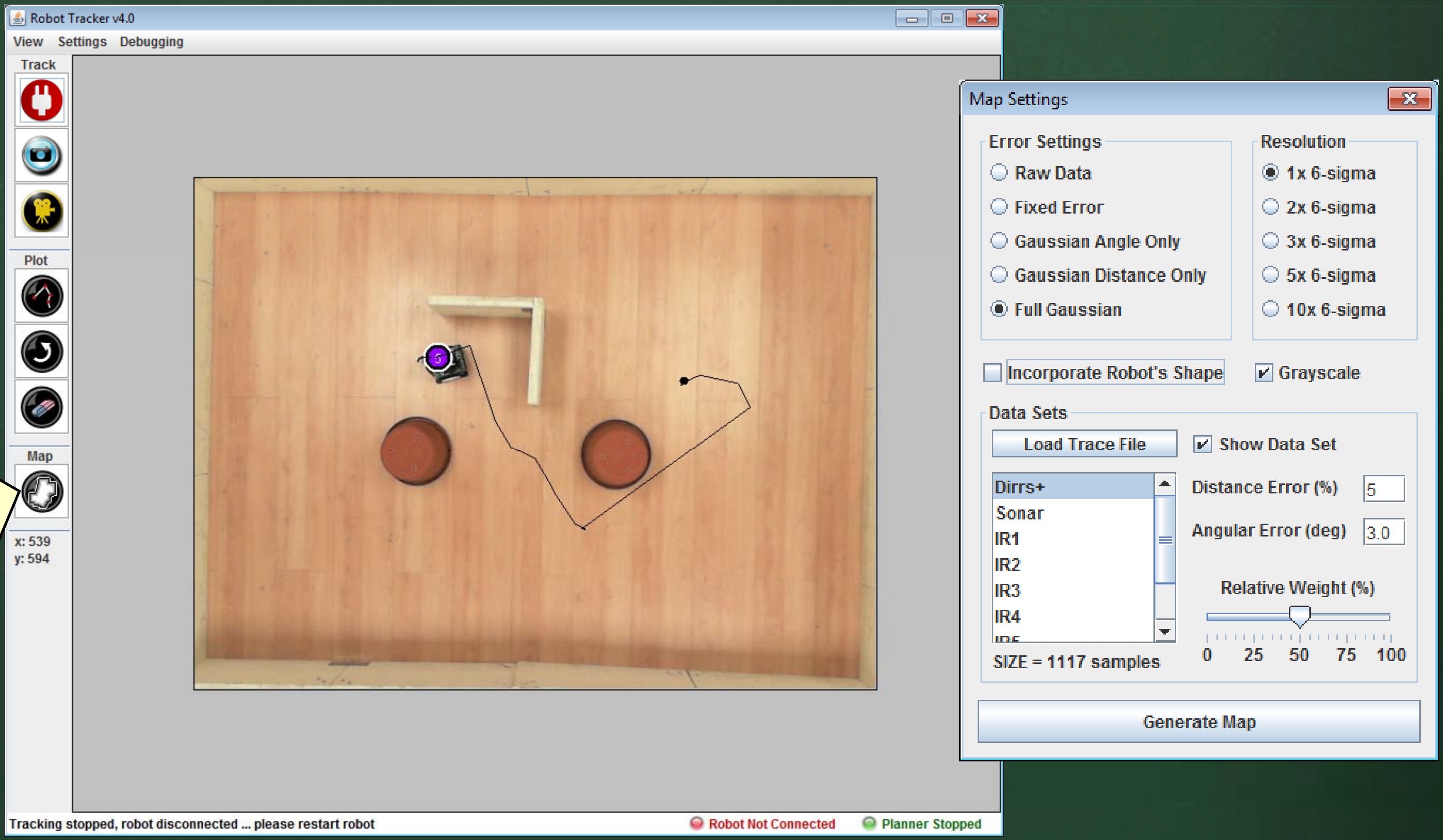


Offset a little since width of body ensures sensor does not detect too close

Planner Example 5

- Once trace file has been created, you can display a map:

Click here to enable mapping. RobotTracker window will become larger and a **Mapping Dialog** box will appear.



x: 539
y: 594

Tracking stopped, robot disconnected ... please restart robot

Robot Not Connected

Planner Stopped

Map Settings

Error Settings
○ Raw Data
○ Fixed Error
○ Gaussian Angle Only
○ Gaussian Distance Only
● Full Gaussian

Resolution
● 1x 6-sigma
○ 2x 6-sigma
○ 3x 6-sigma
○ 5x 6-sigma
○ 10x 6-sigma

□ Incorporate Robot's Shape
☑ Grayscale

Data Sets

Load Trace File
☑ Show Data Set

Dirrs+
Sonar
IR1
IR2
IR3
IR4
IR5

Distance Error (%) 5
Angular Error (deg) 3.0

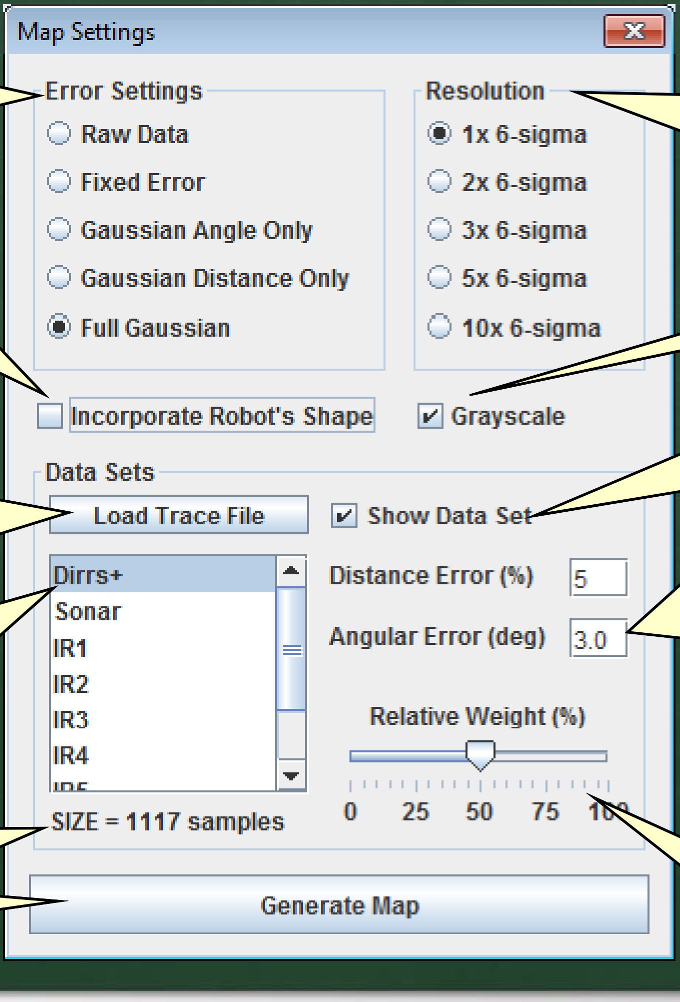
Relative Weight (%)
0 25 50 75 100

SIZE = 1117 samples

Generate Map

Planner Example 5

- Map Settings dialog allows setting of various mapping parameters:



Error Settings

○ Raw Data
○ Fixed Error
○ Gaussian Angle Only
○ Gaussian Distance Only
● Full Gaussian

Resolution

● 1x 6-sigma
○ 2x 6-sigma
○ 3x 6-sigma
○ 5x 6-sigma
○ 10x 6-sigma

□ Incorporate Robot's Shape ☑ Grayscale

Data Sets

☑ Show Data Set

Dirrs+
Sonar
IR1
IR2
IR3
IR4

Distance Error (%) 5

Angular Error (deg) 3.0

Relative Weight (%)

SIZE = 1117 samples

Specifies sensor error model for fusing all range data into the map.

When checked, **fills in area underneath robot** (for each pose in trace file) as an "open" area in the map.

Load a trace file. Does not display map right away. You can load the trace file as the robot is building it to see if your map is coming along nicely.

This list shows all data sets from the trace file (according to the header that was defined). Click on each item to adjust it's settings on the right.

The number of trace file readings for the data set selected in the list.

Click here to generate and re-draw map.

Specifies amount of **6-sigma resolution** to use on the Full Gaussian setting. 1x is normal, 10x is smoother.

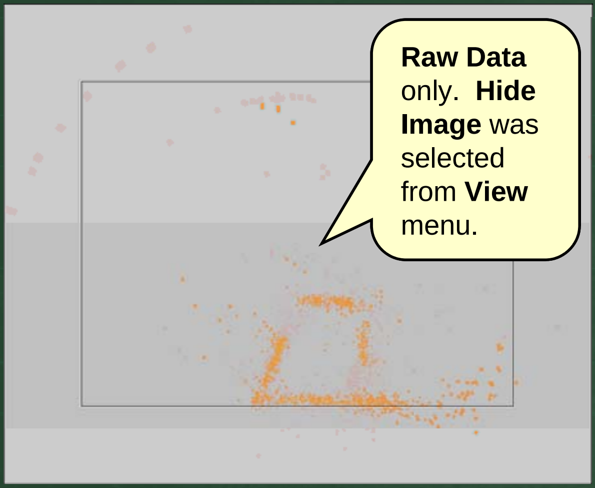
Switch to/from grayscale map

Select to **enable/disable** the selected dataset in the map. You will need to click **Generate Map** for this to take effect.

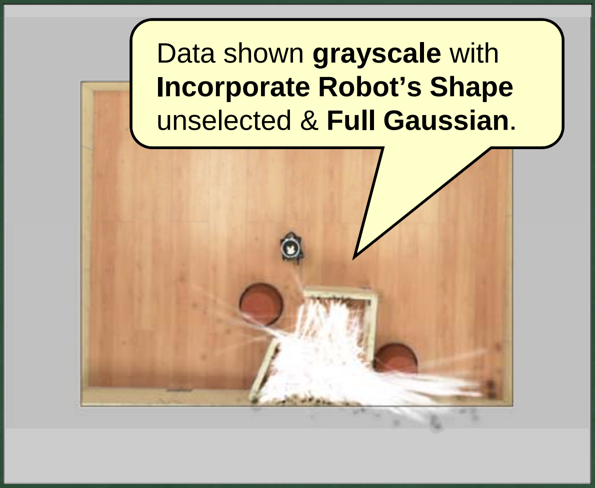
These are the sensor model **parameters** defined in the header of the trace file for the selected dataset. You can modify them, but upon reloading the trace file, they will be reset to the trace file header defaults.

Adjust this to specify the weight of the selected dataset in the overall fusion process.

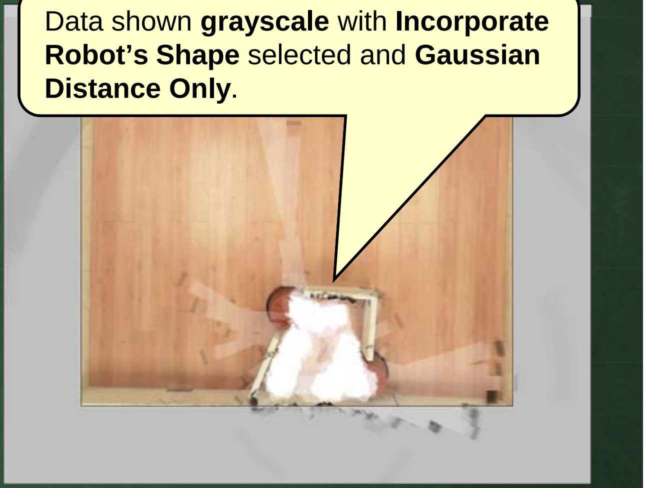
Planner Example 5



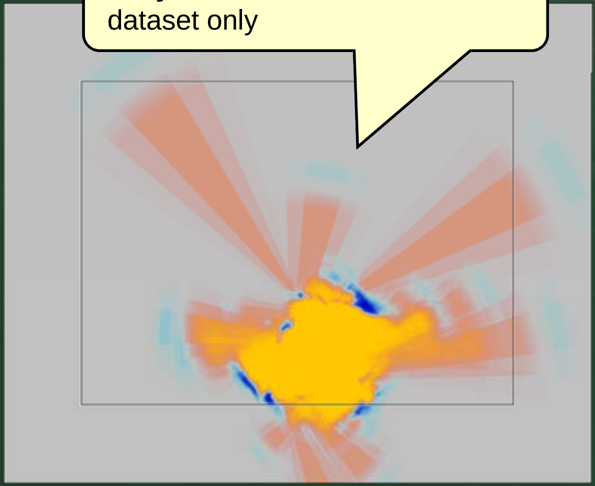
Raw Data only. **Hide Image** was selected from **View** menu.



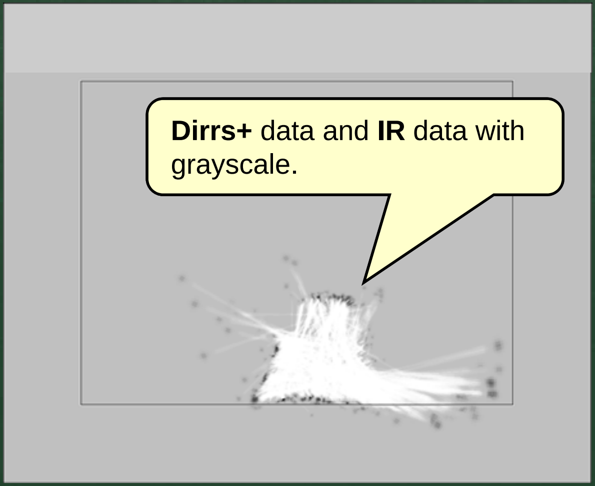
Data shown **grayscale** with **Incorporate Robot's Shape** unselected & **Full Gaussian**.



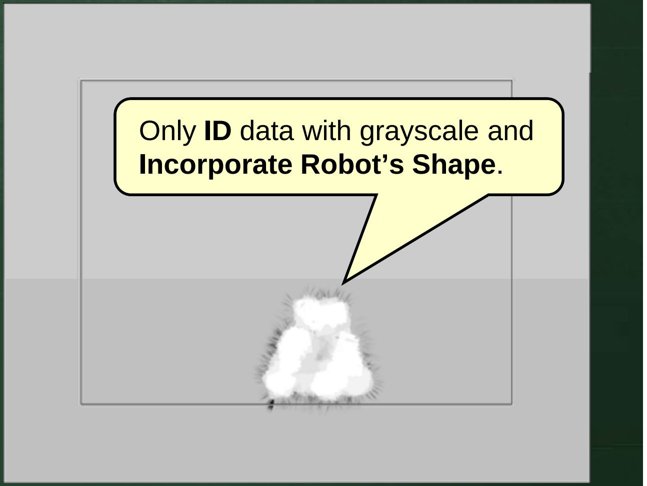
Data shown **grayscale** with **Incorporate Robot's Shape** selected and **Gaussian Distance Only**.



Grayscale unselected and sonar dataset only



Dirrs+ data and **IR** data with grayscale.



Only **ID** data with grayscale and **Incorporate Robot's Shape**.

Summary

- You should now understand how to:
  - write/compile and run Spin programs for the Propeller microprocessor
  - operate the robot servos and read the sensors
  - coordinate your PC code with your robot code using the RobotTracker software