

Robotics & Automation

ENGT4314

Tutorial 9

The Compass

Accessing the compass:

The compass in the robot is an I2C device which is connected to a I2C to serial converter. The compass is a slave device and hence unlike the GPS will only respond to queries we send to it. The device consists of a series of registers that can be read or written too.

Register	Function
0	Software Revision Number
1	Compass Bearing as a byte, i.e. 0-255 for a full circle
2,3	Compass Bearing as a word, i.e. 0-3599 for a full circle, representing 0-359.9 degrees.
4,5	Internal Test - Sensor1 difference signal - 16 bit signed word
6,7	Internal Test - Sensor2 difference signal - 16 bit signed word
8,9	Internal Test - Calibration value 1 - 16 bit signed word
10,11	Internal Test - Calibration value 2 - 16 bit signed word
12	Unused - Read as Zero
13	Unused - Read as Zero
14	Unused - Read as Undefined
15	Calibrate Command - Write 255 to perform calibration step. See text

We are primarily interested in registers 2 and 3. Which gives us a 16 bit number representing compass bearing.

So to communicate to the Compass module we need to use the following sequence of bytes.

0x55 - Read/Write to 1 byte addressable devices

This instructs the serial to I2C converter to make a connect to 1 byte addressable devices such as the compass.

0xC1- The read compass address.

The least significant bit of an I2C address indicates if the transaction is a read or a write operation. 1 indicates read while 0 indicates a write. So in this case we are only interested in read operations.

0x02 – This is the address of the register we wish to read from.

0x02 – The number of bytes we wish to read.

If the command is successful we should receive 2 bytes back. To calculate the compass bearing we simply shift the most significant byte to the left by 8 bits. Ie. Bearing = $\text{input}[0] \ll 8 + \text{input}[1]$.

Hence we can use the following code to read from the compass.

```
HANDLE compass;
int sent, rec;
int cstate;
unsigned char bearing[2];
unsigned char GPSData[500];
unsigned char data[4] = {0x55, // ReadWrite to 1 Byte addressable Device
                        0xC1, // Device Address
                        0x02, // Double Byte Bearing Address
                        0x02, // Read Two Bytes
};
cstate = openComPort(&compass, "\\.\COM2", 19200, NOPARITY, TWOSTOPBITS); // Compass Comm Port

sent = comSend(&compass, data, 4); // Send I2C request to compass
rec = comRead(&compass, bearing, 2); // Read Compass data as two bytes

printf("Compass Sent: %i, Rec %i\n", sent, rec);
printf("Bearing (x10): %i\n", ((int)bearing[0] << 8) + bearing[1]);

comClose(&compass);
```

GPS:

Unlike the compass the GPS doesn't require us to poll it for data. The GPS will send a string of characters every second.

Bellow is the standard output from the GPS receivers

```
$GPGSA,A,1,,,,,,,,,,,,,*1E
$GPGSV,3,1,12,25,73,340,22,13,59,211,36,23,52,150,16,16,14,115,37*73
$GPGSV,3,2,12,02,08,219,27,07,53,338,20,20,49,071,23,27,40,328,18*7C
$GPGSV,3,3,12,04,38,245,38,32,22,068,,08,15,331,,11,08,008,*7F
$GPRMC,071844.487,V,,,,,,,,,071008,,*24
$GPGGA,071845.487,,,,,0,00,,,M,0.0,M,,0000*52
$GPGLL,,,,,071845.487,V*1C
```

In particular we are interested in the line beginning with \$GPRMC which contains the coordinates and whether the stream is valid. The V in the stream above indicates that the stream is not valid, i.e we don't have a fix. Below is a stream when the gps does have a lock.

```
$GPRMC,072448.002,A,3158.7485,S,11548.9633,E,0.36,169.77,071008,,*13
```

You can use the same com methods as above, and adjust the max number of bytes to read as is appropriate. What is an appropriate value? Is it 1 or a large number?

Vision:

The image processing is handled externally to Aria so don't try to use any Aria functions to load an image. In the tutorial example for last week there is an example program that retrieves images from the webcam.

The image processing component of the lab is handled in the Video Class, which has been provided.

This constructor sets up the FLTK window for displaying the camera output in a Fl_Box

```
Video::Video(int x, int y, int width, int height) : Fl_Window(x, y, width
*2, height*2, "RA_LAB5 iVideoCap"){
    setUpDone = false;

    display = new Fl_Box(x,y,width,height); // Raw Image Display

    this->add(display);

    this->end();
    this->show();

    Fl::add_timeout(1.0, callback, this);
};
```

The Deconstructor frees up memory and stops gathering frames from the camera. The deconstructor is called when the FLTK window is closed or unloaded.

```
// Video Deconstructor
Video::~~Video(){
    VI.stopDevice(0);

    if (setUpDone) {delete [] pic;}
};
```

Set up camera simply sets the camera resolution, and allocates memory to store the RGB image. Pic is an array of unsigned chars, where each 3 consecutive bytes represents a pixel. The first byte is the Red components, and then green and blue. You can easily access and process the pic array.

```
// Setup Camera
void Video::setUpCam(){
    if(!setUpDone){
        VI.setupDevice(0,640,480);
        this->resize(this->x(),this->y(),VI.getWidth(),VI.getHeight());

        pic = new uchar[3*VI.getWidth()*VI.getHeight()];
        display->resize(0,0,VI.getWidth(),VI.getHeight());

        rgbImg = new Fl_RGB_Image(pic,VI.getWidth(),VI.getHeight());

        display->image(rgbImg);

        setUpDone = true;
    }
}
```

goahead grabs each frame, and redraws (refreshes) the Fl_Box hence displaying the current frame on the screen. If no camera is connected a error message will be printed in the box instead. This function is recalled via a timeout and would be where you call or run your processing code from.

The for loop, simply copies the new frame to the pic array and horizontal mirrors the image. You could replace it with a memcpy which may improve performance but wont mirror the image.

```
void Video::goahead() {
    int modo;
    int i;
    int j;
    int limit;

    if(camConnected()){

        this->setUpCam();

        if( VI.isFrameNew(0)){

            modo = VI.getWidth(0) *3 ;

            uchar *newPic = grabFrame();// +
            VI.getWidth(0)*VI.getHeight(0)*3 ;
            /*uchar *newPicc*/m_newPicc = const_cast<uchar* >(pic);

            for( i = VI.getWidth(0)*(VI.getHeight(0)-1)*3; i >= 0; i-
=modo){
                limit = i+ modo;
                for (j = i; j <limit;) {
                    m_newPicc[j++] = *(newPic++);
                    m_newPicc[j++] = *(newPic++);
                    m_newPicc[j++] = *(newPic++);

                } }

                // Place you image processing code here.

                rgbImg->uncache();

                display->redraw();

            }
        }
        else{
            display->copy_label("NO CAMERA CONNECTED");
            setUpDone = false;
        }

        Fl::repeat_timeout(0.003, callback, this);

    };
};
```

Driving

As mentioned in the previous lab, the driving code once initialized is simply a matter of calling a series of basic commands. Refer to the sheet from last week's tutorial for more information.

```
ArRobot robot;
ArSimpleConnector *connector;

int drive(int argc, char** argv) {
    connector = new ArSimpleConnector(&argc, argv);
    if (!connector->parseArgs() || argc > 1)
    {
        connector->logOptions();
        exit(1);
    }
    Aria::init();
    if (!connector->connectRobot(&robot))
    {
        printf("Could not connect to robot... exiting\n");
        Aria::shutdown();
        exit(1);
    }
    robot.enableMotors();
    robot.runAsync(true);
    robot.setDeltaHeading(140.0);
    while(!robot.isHeadingDone());
    robot.disableMotors();
    robot.disconnect();
    delete connector;
    return 0;
}
```

Installing the DVD.

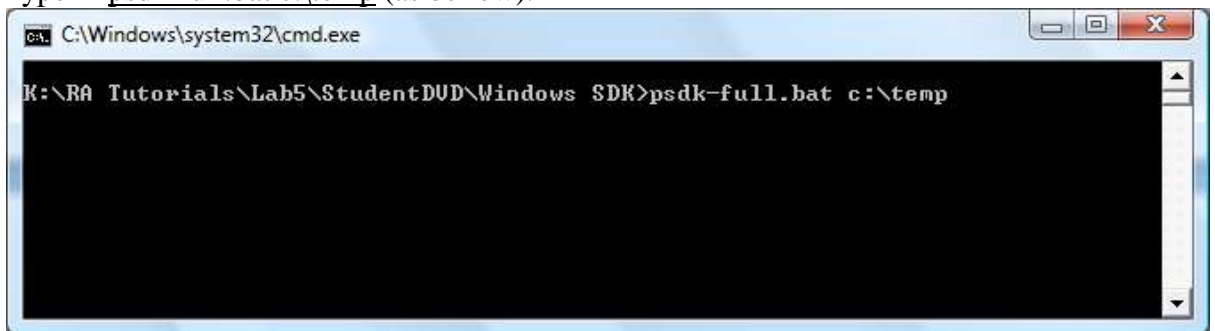
Some students have requested a step by step instruction on how to install the SDKs.

1. Before doing anything else make sure you have Visual Studio 2005 installed this is important as the SDKs add references to the linker. I do not recommend using 2008 as you won't be able to re-compile on the robot's notebooks.
2. Copy the student DVD folder from the disk.
3. From the copy run the DirectX sdk from the direct folder, and install to the default path. This is important to make sure that the linker settings remain the same on your PC and the robot.
4. Open a command prompt window, and navigate to the windows SDK folder.



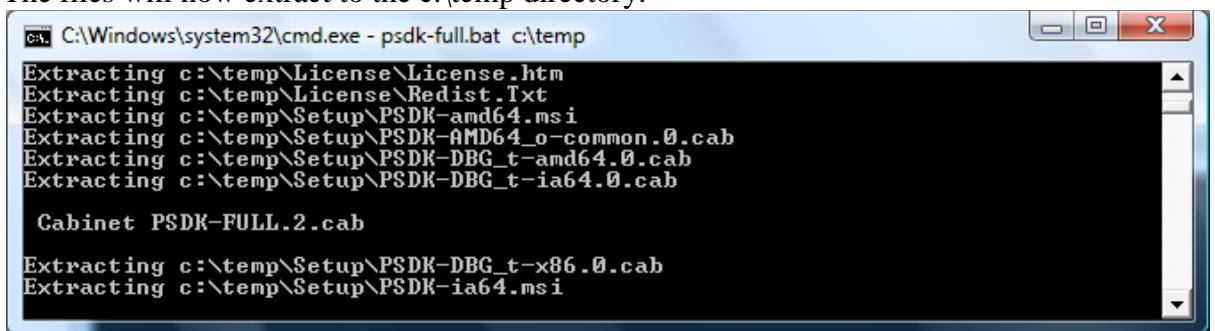
```
C:\Windows\system32\cmd.exe
K:\RA_Tutorials\Lab5\StudentDUD\Windows_SDK>
```

5. Type in `psdk-full.bat c:\temp` (as bellow).



```
C:\Windows\system32\cmd.exe
K:\RA_Tutorials\Lab5\StudentDUD\Windows_SDK>psdk-full.bat c:\temp
```

6. The files will now extract to the `c:\temp` directory.



```
C:\Windows\system32\cmd.exe - psdk-full.bat c:\temp
Extracting c:\temp\License\License.htm
Extracting c:\temp\License\Redist.Txt
Extracting c:\temp\Setup\PSDK-amd64.msi
Extracting c:\temp\Setup\PSDK-AMD64_o-common.0.cab
Extracting c:\temp\Setup\PSDK-DBG_t-amd64.0.cab
Extracting c:\temp\Setup\PSDK-DBG_t-ia64.0.cab

Cabinet PSDK-FULL.2.cab

Extracting c:\temp\Setup\PSDK-DBG_t-x86.0.cab
Extracting c:\temp\Setup\PSDK-ia64.msi
```

7. Once completed type `c:\temp\setup.exe` which should now open the windows sdk setup program. Once again you need to install it to the default directory.
8. Now copy the VS2005 Aria directory and unzip the Aria.zip inside (ignore the Aria folder). This contains a standard VS2005 project which is setup with all the required linker parameters. Only open the Aria.sln file. In VS you will find the Tutorial example which you can run.

Threading

I hope so far that you have divided up the tasks in your group between each person, eg Driving, Image Processing, etc. You may find it beneficial to run each members code at the same time, that way you don't need to worry about waiting for the driving code to finish before you can process the image for example. To achieve this we can use threads. For the simple threading requirements for this project we can simply implement them with `_beginthreadex` and `_endthreadex`.

You will need to include `<process.h>` and `<window.h>`.

```
uintptr_t _beginthreadex(  
    void *security,  
    unsigned stack_size,  
    unsigned ( *start_address )( void * ),  
    void *arglist,  
    unsigned initflag,  
    unsigned *thrdaddr  
);
```

Security: Affects the way threads and be accessed, for our purposes this can be just set to 0.

Stack_size: Set this to 0 to use the default stack size.

Start_address: The function that will be executed in the thread. This function should not return until the thread is ready to exit.

Arglist: A pointer to a parameter that will be passed to the thread's function. This can be null if not required.

Thradr: An unsigned integer that will be written to with the an ID for the thread.

The function returns a pointer to the thread which we can cast as a Handle.

`WaitForSingleObject` allows us to wait for a thread to terminate so that we can clean up after it.

```
DWORD WaitForSingleObject(  
    HANDLE hHandle,  
    DWORD dwMilliseconds  
);
```

hHandle: The handle to the resource in this case a thread from `_beginthreadex`.

dwMilliseconds: The time to wait for object in this case we use `INFINITE`.

Once a thread has exited we close the handle with `CloseHandle(Handle h)`;

Each thread function should be typed like the following

```
unsigned __stdcall threadA(void * argument) {  
}
```

And must return a value (typically 0).

Example:

```
#include "stdafx.h"
#include <process.h>
#include <windows.h>
#include <conio.h>

unsigned __stdcall threadA(void * argument) {

    while (*((char*)argument) != 0) {
        printf("A");
        Sleep(300);
    }
    printf("\nThread A exiting....\n");
    Sleep(1000);
    return 0;
}

unsigned __stdcall threadB(void * argument) {
    while (*((char*)argument) != 0) {
        printf("B");
        Sleep(300);
    }
    printf("\nThread B exiting....\n");
    Sleep(1000);
    return 0;
}

unsigned __stdcall threadC(void * argument) {
    while (*((char*)argument) != 0) {
        printf("C");
        Sleep(300);
    }
    printf("\nThread C exiting....\n");
    Sleep(1000);
    return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
    char stopA = 1, stopB = 1, stopC = 1;
    unsigned int threadAID, threadBID, threadCID;
    HANDLE A, B, C;

    A = (HANDLE)_beginthreadex(NULL,0,&threadA,(void*)&stopA,0,&threadAID);
    B = (HANDLE)_beginthreadex(NULL,0,&threadB,(void*)&stopB,0,&threadBID);
    C = (HANDLE)_beginthreadex(NULL,0,&threadC,(void*)&stopC,0,&threadCID);

    while(!_getwch());
    stopA = 0;
    WaitForSingleObject(A, INFINITE );
    printf("\nThread A has been terminated\n");
    CloseHandle(A);

    while(!_getwch());
    stopB = 0;
    WaitForSingleObject(B, INFINITE );
    printf("\nThread B has been terminated\n");
    CloseHandle(B);

    while(!_getwch());
    stopC = 0;
    WaitForSingleObject(C, INFINITE );
    printf("\nThread C has been terminated\n");
    CloseHandle(C);
    Sleep(1000);

    return 0;}
}
```