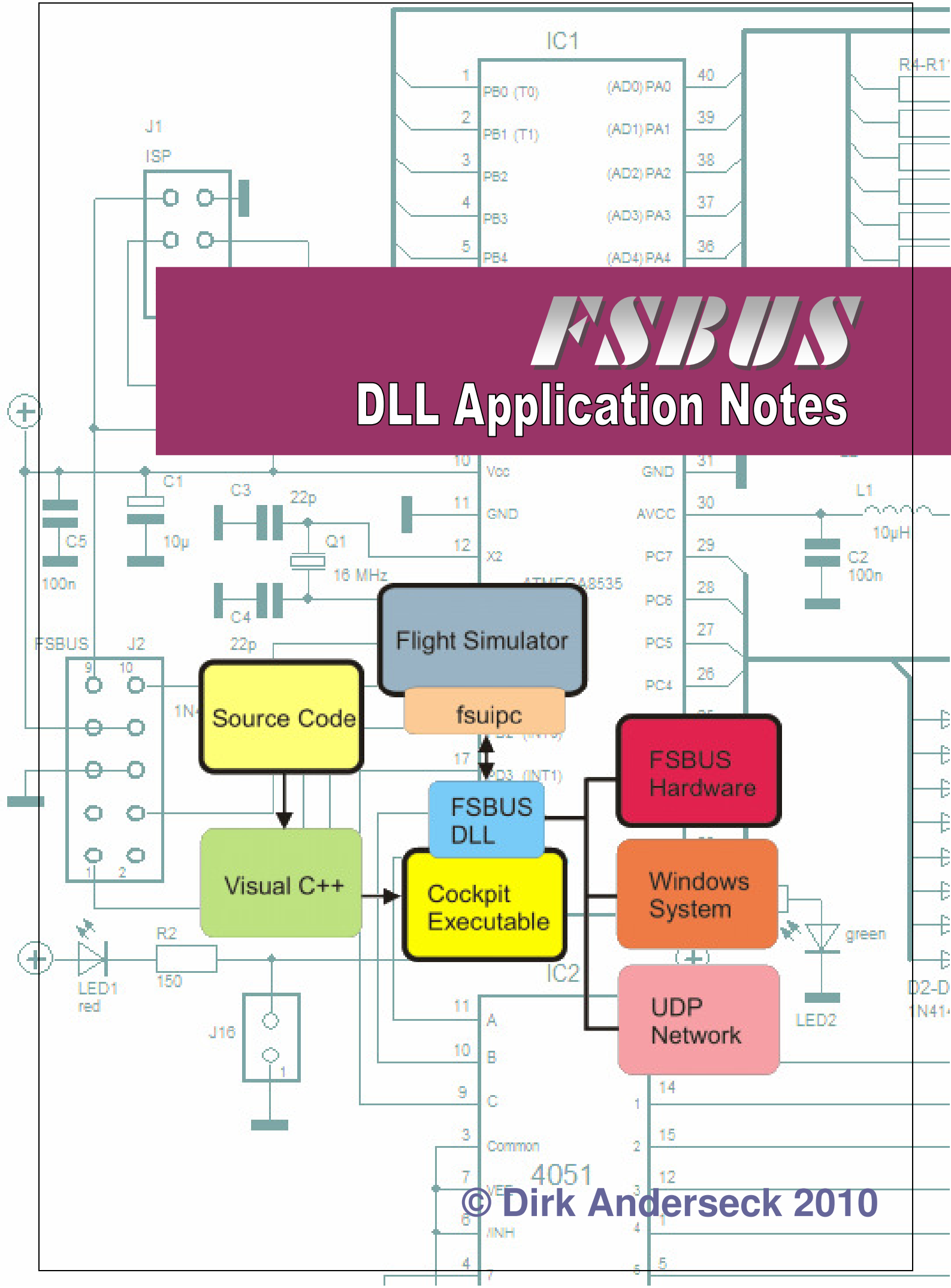
FSBUS

DLL Application Notes



Terms of use

The software and documentation included in the FSBUS DLL is copyrighted by Dirk Anderseck (dirk@anderseck-net).

Permission is granted to anyone to use this software for any purpose, FSBUS DLL is free software for non-commercial purpose.

All rights in the Software (including any images or text incorporated into the Software) are owned by the author of this software, except otherwise stated.

The author of this software does not guarantee that this software is free from bugs or free from viruses.

The author may, from time to time, revise or update the Software. In so doing, the author incurs no obligation to furnish such revision or updates to you.

The software is provided "as is" and the author disclaims all other warrenties and conditions, fitness for a particular purpose, conformance with description, title and non-infringement of third party rights.

This **Application Notes** contains an introduction into the development process, followed by a Cookbook which shows demo code for the different hardware.

Development introduction

It was my intention, to make the development process as easy as possible while producing a fast, powerful and extendable result. That's why I have chosen C as the development platform.

The development environment, used in this example is the free Visual Studio express version. Since I abandoned any C, C++ or Windows speciality, this environment is absolute sufficient making large and powerful cockpit applications.

Requirements

The following list shows the software required to develop your own FSBUS cockpit control software.

Software	free	Website
Microsoft Flight Simulator FSX or 2004		www.microsoft.com/games/flightsimulatorx
fsuipc	(*)	www.schiratti.com/dowson.html This library is the interface into and out of flightsim.
Microsoft Visual Studio C++ Express (2010)	*	www.microsoft.com/germany/Express/download VC++ is one of many compilers and workbenches to develop own software.
FSBUS distribution (fsbus.dll, fsbus.lib, fsbus.h)	*	www.fsbus.de FSBUS distribution contains documents, libraries, programs, firmware and other things to connect FSBUS hardware to flightsimulator.

Hardware	mandatory	
FSBUS COM	*	this hardware connects the FSBUS buscable to a serial interface of a pc. If your PC no longer supports this kind of interface, you may use a USB to Serial converter. detailed info: **doc\com_en.pdf**

FSBUS IO	(*)	One controller supports <= 64 buttons <= 32 rotaries <= 16 digital outputs (LED) <= 8 analogue inputs (potentiometer) <= 8 analogue outputs (voltmeter) detailed info: **doc\ dio_en.pdf**
FSBUS DO64		Up to 64 LED's or other other loads are supported by this controller. detailed info: doc\ do64_en_1.pdf
FSBUS Servo		A maximum of 8 RC-servos are driven by one servo controller. detailed info: **doc\ servo.pdf**
FSBUS Display		This controller supports up to 6 seven-segment-displays. detailed info: **doc\ dsp_en.pdf**
FSBUS Stepper		Controller for 1 stepper motor. detailed info: **doc\ stepper.pdf**

Introduction

You begin a new project with VC++2010 express by selecting [file] [new][win32 console application]. Just give it a name (mycockpit), a directory (c:\develop) and the workbench will generate a few files for you.

Now you may copy fsbusdll\fsbus.dll, fsbusdll\fsbus.h, fsbusdll\fsbus.lib from the fsbus distribution into your new build directory (c:\develop\mycockpit\mycockpit).

Modify the automatic generated mycockpit.cpp. You may include the fsbus.h file and tell the linker to link the library file with the pragma command.

```
#include "stdafx.h"
#include "fsbus.h"
#pragma comment(lib, "fsbus.lib")

int main(int argc, char* argv[])
{
    printf ("FSBUS DLL version %d\r\n", GetFsbusDLLVersion ());
    Sleep (2000);
    return 0;
}
```

This is just a simple starter program which shows the DLL version for 2 seconds.

Organize files

We continue the development process by introducing the main strategy. I assume, our cockpit will be a big cockpit with hundreds of buttons and other stuff. This makes grouping of functions in different files very useful.

Let's make the code for a gear control. A gear control consists of a switch, which is down or up (digitally spoken 0 or 1), 3 LED's showing the state of the wheels and a red transition LED which is on during the wheels are in motion.

First, we add a new **gear.cpp** file to the sourcefiles and a **gear.h** file to our headerfiles section of the project.

The **stdafx.h** file is then extended with:

```
#pragma once

#include "fsbus.h"
#include "gear.h"

#define OID_GEAR (1 << OID_CONTROL_BITS)
```

We have to manage the object id's by ourselves. It's suggested to group id's into functional groups with up to **32** objects in each group. The maximum number of groups is **128**.

The Bit scheme is:

Bit 11	10	9	8	7	6	5	4	3	2	1	0
Group ID 0-127							Object ID 0-31				

- The OID's of all groups is defined in stdafx.h
- The object id's of the objects in each group is defined in the groups .h file

Our new **gear.h** file looks like this:

```
#ifndef __GEAR_H_
#define __GEAR_H_

enum e_gearoid {
	C_LED_LEFTGEAR = OID_GEAR,
	C_LED_RIGHTGEAR, C_LED_NOSEGEAR, C_SWITCH_GEAR, C_LED_GEARTRANS,
	FS_GEARCONTROL, FS_GEARPOSITIONNOSE, FS_GEARPOSITIONRIGHT,
	FS_GEARPOSITIONLEFT };
void BuildGearObjects ();
void cbGear(int oid, int val, double dval);

#endif
```

It contains the contants for each fsbus object (C_LED_LEFTGEAR …) and the prototypes for 2 new functions. The first is the function in which all objects are created and the second is the callback function. The callback function is called from the fsbus.dll, when a specific event occurs. In our example the cbGear will be called when the gear switch is moved and when flightsim reports any change of the gear position.

This brings us to the discussion of **gear.cpp** file:

```
#include "stdafx.h"

void BuildGearObjects ()
{
	MkFsbusObject (BTP_D_IN, C_SWITCH_GEAR,"SW_GEAR",cbGear, 2, 12,0);
	MkFsbusObject (BTP_D_OUT, C_LED_GEARTRANS,"LED_GEARINTRANS",NULL, 22, 24,0);
	MkFsbusObject (BTP_D_OUT, C_LED_LEFTGEAR,"LED_LEFTGEAR",NULL, 22, 27,0);
	MkFsbusObject (BTP_D_OUT, C_LED_RIGHTGEAR,"LED_RIGHTGEAR",NULL, 22, 25,0);
	MkFsbusObject (BTP_D_OUT, C_LED_NOSEGEAR,"LED_NOSEGEAR",NULL, 22, 26,0);
	FsbusPowerOffOptions (C_LED_GEARTRANS, PWRTYPE_MAINBUS, 0);
	FsbusPowerOffOptions (C_LED_LEFTGEAR, PWRTYPE_MAINBUS, 0);
	FsbusPowerOffOptions (C_LED_RIGHTGEAR, PWRTYPE_MAINBUS, 0);
	FsbusPowerOffOptions (C_LED_NOSEGEAR, PWRTYPE_MAINBUS, 0);
	MkFsObject(FS_GEARCONTROL,"",cbPlaneCtrl2,0x0BE8, 4,  TP_I32,FS_NONE,0);
	MkFsObject(FS_GEARPOSITIONNOSE,"",cbPlaneCtrl2, 0x0BEC, 4,TP_I32,FS_NORMAL,0);
	MkFsObject(FS_GEARPOSITIONRIGHT,"",cbPlaneCtrl2,0x0BF0, 4,TP_I32,FS_NORMAL,0);
	MkFsObject(FS_GEARPOSITIONLEFT,"",cbPlaneCtrl2, 0x0BF4, 4,TP_I32,FS_NORMAL,0);
}
```

```
void cbGear(int oid, int val, double dval)
{
	static bool gleft = false;
	static bool gright = false;
	static bool gnose = false;
	static int  gtrans = 0;

	switch (oid)
	{
	case C_SWITCH_GEAR: // Down = 0
		FsWriteInt (FS_GEARCONTROL, val ? 0 : 1);
		break;

	case FS_GEARPOSITIONNOSE:
		if (val > 100 && val < 16000)
			gtrans &= ~0x01;
		else
			gtrans |= 0x01;

		if (val > 16000 && gnose == false)
		{
			gnose = true;
			FsbusWrite (C_LED_NOSEGEAR, 1);
		}
		if (val < 100 && gnose == true)
		{
			gnose = false;
			FsbusWrite (C_LED_NOSEGEAR, 0);
		}
		break;
	case FS_GEARPOSITIONLEFT:
		if (val > 100 && val < 16000)
			gtrans &= ~0x02;
		else
			gtrans |= 0x02;

		if (val > 16000 && gleft == false)
		{
			gleft = true;
			FsbusWrite (C_LED_LEFTGEAR, 1);
		}
		if (val < 100 && gleft == true)
		{
			gleft = false;
			FsbusWrite (C_LED_LEFTGEAR, 0);
		}
		break;
	case FS_GEARPOSITIONRIGHT:
		if (val > 100 && val < 16000)
			gtrans &= ~0x04;
		else
			gtrans |= 0x04;

		if (val > 16000 && gright == false)
		{
			gright = true;
			FsbusWrite (C_LED_RIGHTGEAR, 1);
		}
		if (val < 100 && gright == true)
		{
			gright = false;
```

```
                    FsbusWrite (C_LED_RIGHTGEAR, 0);
                }
                break;
        }
        FsbusWrite (C_LED_GEARTRANS, gtrans  ? 1 : 0);
}
```

The **BuildGearObjects** function creates each required object by assigning names and id's to it.
The gear switch is created with a call to **MkFsbusObject**. The type is BTP_D_IN which is a digital in, declared in fsbus.h. The object-id C_SWITCH_GEAR has been previously defined in **gear.h**. The parameter 2 is the so called controller id (CID). This is the assigned address of an FSBUS IO controller. Since this controller supports many switches, there is also a register-id (RID), which is 12 in this example. An important parameter is **cbGear**, which is the address of the callback function. FSBUS DLL know knows where to notify an event, if the gear switch is turned on or off.

The LED objects are created similar with one exception, an LED doesn't need a callback. There is another issue when controlling LED's. If battery level is low, no LED will shine. A very convenient way to achieve this is a call to **FsbusPowerOffOptions**.
It will then control the power off situation internally.

Now we have to create the flightsim objects. Flightsim must be signalled, when the gearswitch moves up or down. The movement of the gear is then notified by flightsim. The notification is done by sending a value from 0 to 16384 correspondig to the current wheel position. It's not only a binary down or up, it's like an analogue value.

All the flightsim objects have an offset, a type and a length parameter. The developer documentation of Peter Dowson's fsuipc will give you the details about each control.

Now, we will have a look at the callback function. As said before, the callback is performed by the FSBUS.DLL, when any FSBUS hardware event, or flightsim event occurs. The object-id is notified in the oid parameter and a value is passed either as an integer or a floating-point parameter depending on the valuetype.

The first event, we were notified, is the gear down switch, handled by the pilot. The action performed in the code is:

```
FsWriteInt (FS_GEARCONTROL, val ? 0 : 1);
```

In other words, this event is routed to flightsim, which then begins the gear down or up process. The gear cases will open, the wheels will come down and any change is reported to our callback function.

There is more than only setting things on or off. The threshold values to detect a down position is set to 16000 and the up position is assumed to be 100. This controls

the Gear-in-Trans position LED, which isn't supported by fsuipc. It's a good example to overcome missing capabilities of flightsim or fsuipc.

Finally, the things have to come together in our main function.

```
#include "stdafx.h"
#pragma comment (lib, "fsbus.lib")

int main(int argc, char* argv[])
{
    CheckIn();
    BuildGearObjects ();

    FsbusOpen("COM1");
    printf ("FSBUS connected ...\r\n");
    FsConnect();
    printf ("Flightsim connected ...\r\n");
    for (;;)
    {
       if (_kbhit())
       {
              switch (getch())
              {
              case 'x':
                     CheckOut();
                     return 0;
              }
       }
       FsbusMux(500);
    }
}
```

The main function is the start of any C-program. The CheckIn is the initialization function in FSBUS.DLL. Then the FSBUS objects are created which is done in our BuildGearObjects function.
The hardware is connected by FsbusOpen and the FsConnect makes the interface to flightsim available to this program. Don't forget to start flightsim before.

Now, an endless loop begins. Normally this kind of endless loop seems to be a CPU consuming worthless task, but the FsbusMux with the timeout parameter set to 500 milliseconds is the keypoint. It waits without processorload until the time is elapsed or an event occurs and then executes the necessary actions which is the call of one of the callback functions.

Testing the keyboard state allows to exit the program by pressing the x-key.

The function FsbusMux() is the most valuable function in FSBUS.DLL. It looks very simple, but this function keeps all the balls (cockpit objects) playing.

Cookbook

Example stepper motor

Controlling a stepper motor in FSBUS is a little different from other FSBUS objects, because it can only be run by `FsbusWriteFmtVar()`. There is no need to declare an object for it. The following code shows a stepper application. The motor drives a needle for a compass.
For convenience all code is in one file.

```
#include "stdafx.h"
#include "fsbus.h"
#pragma comment (lib, "fsbus.lib")

void cb(int oid, int val, double dval);
#define FS_WHISKEYCOMPASS  1
#define C_COMPASS_STEPPER  2
```

```
int main(int argc, char* argv[])
{
    CheckIn();

    FsbusOpen("COM7");
    printf ("FSBUS connected ...\r\n");
    FsConnect();
    printf ("Flightsim connected ...\r\n");

       MkFsObject(FS_WHISKEYCOMPASS, "",  cb,  0x02CC, 8, TP_DBL,  FS_NORMAL, 0);
       MkFsbusObject (BTP_V_OUT, C_COMPASS_STEPPER, "", NULL, 23, 80, 0);

       for (;;)
    {
            if (_kbhit())
                  return 0;
            FsbusMux(500);
    }
}

void cb(int oid, int val, double dval)
{
       switch (oid)
       {
       case FS_WHISKEYCOMPASS:
              FsbusWrite (C_COMPASS_STEPPER, (int)(dval*1210.0/360.0));
              break;

       }
}
```

Magneto control

The starter switch(es) are 2 5-step rotary types marked Off,R,L,Both,Start. Each line is defined as an ordinary on-off switch. It controls a single offset in fsuipc with a specific value.

stdafx.h:

```
#define OID_MAGNETO_GROUP           (10 << OID_CONTROL_BITS)
#include "magneto.h"
```

magneto.h:

```
#ifndef __MAGNETO_H
#define __MAGNETO_H

enum e_magnetooid {
      C_LEFT_R_SWITCH = OID_MAGNETO_GROUP,
      C_LEFT_L_SWITCH,
      C_LEFT_BOTH_SWITCH,
      C_LEFT_START_SWITCH,

      C_RIGHT_R_SWITCH,
      C_RIGHT_L_SWITCH,
      C_RIGHT_BOTH_SWITCH,
      C_RIGHT_START_SWITCH,

      FS_ENGINE1STARTSWITCH,
      FS_ENGINE2STARTSWITCH
};

// declare the functions in the corresponding .cpp file
void BuildMagnetoObjects();
void cbMagneto (int oid, int val, double dval);
#endif
```

magneto.cpp:

```
#include "stdafx.h"

void BuildMagnetoObjects ()
{
  // declare the left magneto starter switch
  MkFsbusObject(BTP_D_IN, C_LEFT_R_SWITCH,"SW_MLR",cbMagneto,2,36);
  MkFsbusObject(BTP_D_IN, C_LEFT_L_SWITCH,"SW_MLL",cbMagneto,2,37);
  MkFsbusObject(BTP_D_IN, C_LEFT_BOTH_SWITCH,"SW_MLBOTH",cbMagneto,2,38);
  MkFsbusObject(BTP_D_IN,C_LEFT_START_SWITCH,"SW_MLSTART",cbMagneto,2,39);

  // declare the left magneto starter switch
  MkFsbusObject(BTP_D_IN, C_RIGHT_R_SWITCH,"SW_MAGRHTR",cbMagneto,2,32);
  MkFsbusObject(BTP_D_IN, C_RIGHT_L_SWITCH,"SW_MAGRIGHTL",cbMagneto,2,33);
  MkFsbusObject(BTP_D_IN, C_RIGHT_BOTH_SWITCH, "SW_MAGRIGHTBOTH",cbMagneto,
2, 34);
  MkFsbusObject(BTP_D_IN,C_RIGHT_START_SWITCH,"SW_MAGRST",cbMagneto,2,35);

  // declare the flightsim object h
  MkFsObject(FS_ENGINE1STARTSWITCH,"FSENG1START",cbMagneto,
             0x0892,2,TP_UI16,FS_NONE);
  MkFsObject(FS_ENGINE2STARTSWITCH,"FSENG2START",cbMagneto,
             0x092a,2,TP_UI16,FS_NONE);
```

```
}

// this is the callback function, where any event is routed to
// val reflects the state (1 or 0) of a switch
void cbMagneto (int oid, int val, double dval)
{
	int x;

	switch (oid)
	{
	case C_LEFT_R_SWITCH:
		// if this switch turns to 1, the position is off
		// a 0 determines the R position
		FsWrite (FS_ENGINE1STARTSWITCH, val ? 0 : 1);
		break;
	case C_LEFT_L_SWITCH:
		if (!val)
			FsWrite (FS_ENGINE1STARTSWITCH, 2);
		break;
	case C_LEFT_BOTH_SWITCH:
		if (!val)
			FsWrite (FS_ENGINE1STARTSWITCH, 3);
		break;
	case C_LEFT_START_SWITCH:
		if (!val)
			FsWrite (FS_ENGINE1STARTSWITCH, 4);
		break;

	case C_RIGHT_R_SWITCH:
		FsWrite (FS_ENGINE2STARTSWITCH, val ? 0 : 1);
		break;
	case C_RIGHT_L_SWITCH:
		if (!val)
			FsWrite (FS_ENGINE2STARTSWITCH, 2);
		break;
	case C_RIGHT_BOTH_SWITCH:
		if (!val)
			FsWrite (FS_ENGINE2STARTSWITCH, 3);
		break;
	case C_RIGHT_START_SWITCH:
		if (!val)
			FsWrite (FS_ENGINE2STARTSWITCH, 4);
		break;
	}
}
```

Twin Engine Throttle Control

This is an example for 6 potentiometer based throttle unit. The Calibrate function with individual setup for each pot allows you to make your own precise flight deck.

Stdafx.h:

```
#define OID_THROTTLE_GROUP          (5 << OID_CONTROL_BITS)
#include "throttle.h"
```

throttle.h:

```
#ifndef __THROTTLE_H
#define __THROTTLE_H
```

```
enum e_throttle_oid {
      C_THROTTLE_LEVER_LEFT = OID_THROTTLE_GROUP,
      C_THROTTLE_LEVER_RIGHT, C_PITCH_LEVER_LEFT, C_PITCH_LEVER_RIGHT,
      C_MIX_LEVER_LEFT, C_MIX_LEVER_RIGHT, C_COWL_LEVER_LEFT,
C_COWL_LEVER_RIGHT,
      FS_ENGINE1_THROTTLE_CONTROL, FS_ENGINE2_THROTTLE_CONTROL,
FS_ENGINE1_PROP_CONTROL,
      FS_ENGINE2_PROP_CONTROL, FS_ENGINE1_MIX_CONTROL,
FS_ENGINE2_MIX_CONTROL,
};

void BuildThrottleObjects();
void cbThrottle (int oid, int val, double dval);

#endif
```

throttle.cpp:

```
#include "stdafx.h"

void BuildThrottleObjects ()
{
  MkFsObject(FS_ENGINE1_THROTTLE_CONTROL,"",cbThrottle, 0x089A, 2,TP_I16,
      FS_NONE);
  MkFsObject(FS_ENGINE2_THROTTLE_CONTROL,"",cbThrottle, 0x0932, 2,TP_I16,
      FS_NONE);

      // -4096 to 16384
  MkFsObject(FS_ENGINE1_PROP_CONTROL,"",cbThrottle, 0x088e, 2,TP_I16,
      FS_NONE);
  MkFsObject(FS_ENGINE2_PROP_CONTROL,"",cbThrottle, 0x0926, 2,TP_I16,
      FS_NONE);

      // 0 to 16384
  MkFsObject(FS_ENGINE1_MIX_CONTROL,"",cbThrottle, 0x0890, 2,TP_I16,
      FS_NONE);
  MkFsObject(FS_ENGINE2_MIX_CONTROL,"",cbThrottle, 0x0928, 2,TP_I16,
      FS_NONE);

  MkFsbusObject (BTP_A_IN, C_THROTTLE_LEVER_LEFT,"",cbThrottle, 2, 72);
  MkFsbusObject (BTP_A_IN, C_THROTTLE_LEVER_RIGHT,"",cbThrottle, 2, 73);
  MkFsbusObject (BTP_A_IN, C_PITCH_LEVER_LEFT,"",cbThrottle, 2, 74);
  MkFsbusObject (BTP_A_IN, C_PITCH_LEVER_RIGHT,"",cbThrottle, 2, 75);
  MkFsbusObject (BTP_A_IN, C_MIX_LEVER_LEFT,"",cbThrottle, 2, 76);
  MkFsbusObject (BTP_A_IN, C_MIX_LEVER_RIGHT,"",cbThrottle, 2, 77);
  MkFsbusObject (BTP_A_IN, C_COWL_LEVER_LEFT,"",cbThrottle, 2, 78);
  MkFsbusObject (BTP_A_IN, C_COWL_LEVER_RIGHT,"",cbThrottle, 2, 79);
}

void cbThrottle (int oid, int val, double dval)
{
      switch (oid)
      {
            case C_THROTTLE_LEVER_LEFT:
                  {

// the next table describes 4 points on the pot axis. The values were
// taken with the fsadmin program. You may interprete it as follows:
// as long as the pot value is 3 or below, the result of Calibrate is 0
```

```
// a value between 3 and 28 is interpolated between 0 and 128.
// a value between 28 and 58 is interpolated between 128 and 200.
// a value greater 92 results in 255

                  static CALTAB LeftThrottleLeverCal[] = {
                              {3,0},{28,128},{58,200}, {92, 255}
                  };
                  val = Calibrate (val, LeftThrottleLeverCal,4);
                  FsWrite (FS_ENGINE1_THROTTLE_CONTROL, val * 64);
                  }
                  break;
            case C_THROTTLE_LEVER_RIGHT:
                  {
                  static CALTAB RightThrottleLeverCal[] = {
                              {3,0},{28,128},{58,200}, {92, 255}
                        };
                  val = Calibrate (val, RightThrottleLeverCal,4);
                  FsWrite (FS_ENGINE2_THROTTLE_CONTROL, val * 64);
                  }
                  break;
            case C_PITCH_LEVER_LEFT:
                  {
                        static CALTAB LeftPitchLeverCal[]    = {
                              {3,0},{28,128},{58,200}, {92, 255}
                        };
                  val = Calibrate (val, LeftPitchLeverCal,4);
                  FsWrite (FS_ENGINE1_PROP_CONTROL, val * 80 - 4096);
                  }
                  break;
            case C_PITCH_LEVER_RIGHT:
                  {
                        static CALTAB RightPitchLeverCal[]   = {
                              {3,0},{28,128},{58,200}, {92, 255}
                        };
                  val = Calibrate (val, RightPitchLeverCal,4);
                  FsWrite (FS_ENGINE2_PROP_CONTROL, val * 80 - 4096);
                  }
                  break;

            case C_MIX_LEVER_LEFT:
                  {
                        static CALTAB LeftMixLeverCal[]      = {
                              {3,0},{28,128},{58,200}, {92, 255}
                        };
                  val = Calibrate (val, LeftMixLeverCal,4);
                  FsWrite (FS_ENGINE1_MIX_CONTROL, val * 64);
                  }
                  break;

            case C_MIX_LEVER_RIGHT:
                  {
                        static CALTAB RightMixLeverCal[]     = {
                              {3,0},{28,128},{58,200}, {92, 255}
                        };
                  val = Calibrate (val, RightMixLeverCal,4);
                  FsWrite (FS_ENGINE2_MIX_CONTROL, val * 64);
                  }
                  break;

            case C_COWL_LEVER_LEFT:
                  break;
```

```
            case C_COWL_LEVER_RIGHT:
                  break;
      }
}
```