LARUS

Bildquelle : wallsdesk.com

Innere Strukturen und Abläufe[1][2]

Dieses Handbuch ist noch eine Weile in den Geburtswehen.
Kommentare sind aber trotzdem willkommen.

1 Zur Nomenklatur : Handbuch Version h.m.n.x korrespondiert mit Software-Version h.m.n und ist die x.te Text-Iteration passend zu dieser Software-Version.

2 Die Ornithologen mögen mir verzeihen. Das ist keine Möwe, sondern ein Albatros. Aber es ist ein schönes Bild. Außerdem : Einige der Kontributoren zu diesem Projekt haben schon früher zusammen an einem Variometer gearbeitet, das hieß „Albatross“.

1 Einleitung

Der Autor der **LARUS** - FrontEnd-SW ist fast 75 Jahre alt und möchte in der Lage sein, den Code einem nachfolgenden „Hüter und Weiterentwickler“ in die Hand zu geben. Dazu ist es notwendig, all die Dinge niederzuschreiben und zu erklären, die sich nicht oder nur schwer aus dem Code selbst erschließen : Konzepte und hinterlistige Lösungen. Der Autor hat sich nach Kräften bemüht den Code lesbar zu gestalten und zu kommentieren, denn er weiß aus eigener Berufserfahrung, um wie viel leichter es einem Softwerker fällt, fremden Code zu verstehen, wenn ihm ein klein wenig Hilfestellung gewährt wird und ihm die Knackpunkte erklärt werden.

Dieses Papier ist für diese Person(en) geschrieben.

Es beschreibt Internas, Abläufe und Strukturen der SW im **LARUS** - FrontEnd-System und im **LARUS -** All-In-One-Modul[3].

Es ist erfahrungsgemäß[4] eine Herausforderung, ein Dokument wie das vorliegende zu schreiben, einmal, weil durch den Autor wegen seine Verbundenheit mit dem Programmcode Dinge, Themen, Zusammenhänge einfach als gegeben angenommen werden und hier vergessen werden, zum Zweiten, weil die Beschreibung schlichtweg nicht klar genug ist.

Deshalb meine Bitte an die Leser:

Legen sie Code und Dokument nebeneinander und lesen sie … und geben sie mir Feedback.

Dabei möchte ich ihre Aufmerksamkeit auf folgende Themen lenken :

- ***die Verarbeitung der Sensor-Daten im Modul CAN_Data_Digester***
- ***die Erzeugung der Menue-Repräsentation (die Module task_ImageBuilder.c, Renderer_Lib.c, ControlTables.c, Generic_Const.h und Generic_Types.h)***

Diese Module bilden den größten Teil der Komplexität der AD57-Software ab.

3 Da das **LARUS -** Utility-Modul fast funktionsgleich ist mit dem hier referenzierten **LARUS -** All-In-One-Modul , werden einige der Erklärungen auch auf diese SW zutreffen.

4 Der Autor hatte im Laufe seiner beruflichen Karriere diese Aufgabe schon mehrfach.

2 Hardware-Basis

Die FrontEnd-SW residiert auf einer HW-Plattform, dem AD57, von Air Avionics (kurz AA). AA nutzt diese Plattform für sein Produkt ATD (Air Traffic Display).

Für **LARUS** wurde das AD57 ihrer originären Firmware entkleidet und von Grund auf neu programmiert.

Da das AD57 intern keinen veritablen Audio-Verstärker und auch keinen Audio-Ausgang hat, war es notwendig, für das Audio eine separate Lösung zu suchen. Hier wurden in der Entwicklung bei festgezurrter CAN-Schnittstelle von Horst Rupp und dem restlichen Team des **LARUS -Projektes** unterschiedliche Lösungsansätze verfolgt. Das vorliegende Papier beschreibt die Lösung von Horst Rupp auf der Grundlage des STM32F4-Discovery-Boards, im Weiteren **LARUS -** Prototype-Utility-Board genannt.

Hier abgebildet ist die Front des Anzeige- und Steuergeräts für das **LARUS**-Vario-System mit seinen Eingabe- und Steuermöglichkeiten.

µSD-Slot

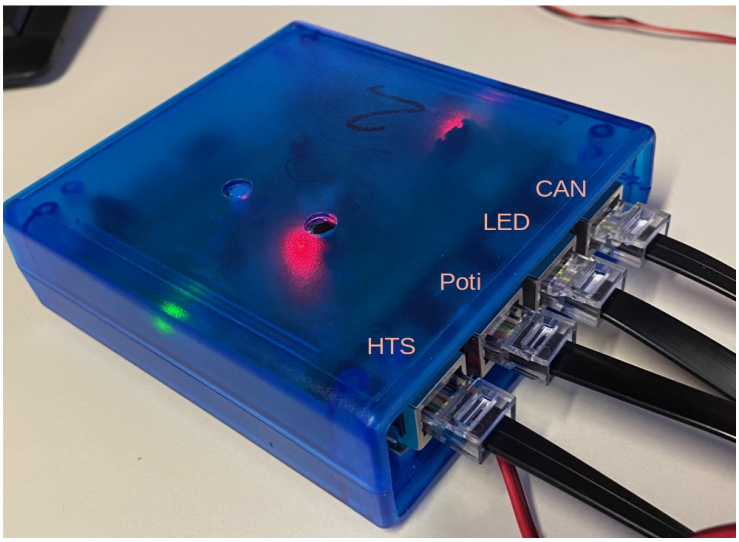
Notabene:

Der Soft-Ein-Aus-Schalter des AD57 funktioniert (noch) nicht. Diese Taste wirkt einfach nur als - Taste. Das Vario muss über einen externen Schalter in der Stromzuführung ein- und ausgeschaltet werden.

Zum Gesamt-System gehören darüber hinaus noch die folgenden Funktionseinheiten :

- die Main-Sensor-Einheit (misst oder errechnet Druck, Höhe, Fahrt, Lage im Raum, Position, Kurs, Wind, Beschleunigungen), unterscheidet automatisch die Modi „Geradeausflug“ und „Kreisen“.

- das Utility-Board (erzeugt die Ton-Signale, bedient den optionalen Sensor für die Wölbklappenstellung, treibt die optionale LED-Anzeige für die Darstellung der Soll-Ist-Wölbklappenstellung, liest die optionalen Mikro-Schalter für Fahrwerk, Bremsklappen und Wölbklappen aus, bedient den Sensor für Außentemperatur und Luftfeuchte und Umgebungslautstärke), überwacht (optional) Fahrwerk und Bremsklappen). Es gibt das Utility-Board in zwei Ausprägungen :
  - **das hier abgebildete „Prototype-Utility-Board“**[5] und alternativ
  - das Standard-Utility-Board (hier nicht abgebildet)

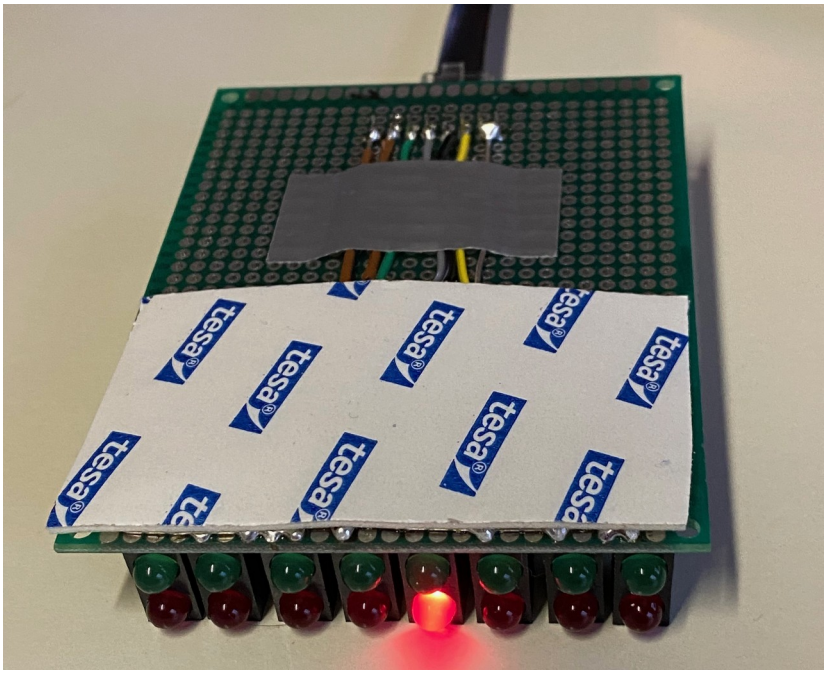


5 HTS steht für Humidity-Temperature-Sensor.
Poti steht für den Anschluss der Sensorik in der Seitenwand des Flugzeugs : drei Mikro-Schalter und ein Linear-Potentiometer.

Mit dem Potentiometer wird die aktuelle Position des Klappengestänges gemessen. Daraus lässt sich auf die Klappenpostion zurückschließen. Die Mikro-Schalter sind so eingebaut, dass sie mit den Gestängen des Fahrwerks, der Bremsklappen und der Wölbklappen betätigt werden (siehe das speziell dafür geschrieben Installationshandbuch).

LED steht für den Anschluss der LED-Wölbklappen-Soll-Ist-Stellungsanzeige .

- den notwendigerweise an anderer Stelle untergebrachten Lautsprecher (Vermeidung magnetischer Störungen am Haupt-Sensor) (nicht abgebildet)
- die LED-Leiste für die Soll-Ist-Stellungsanzeige der Wölbklappen

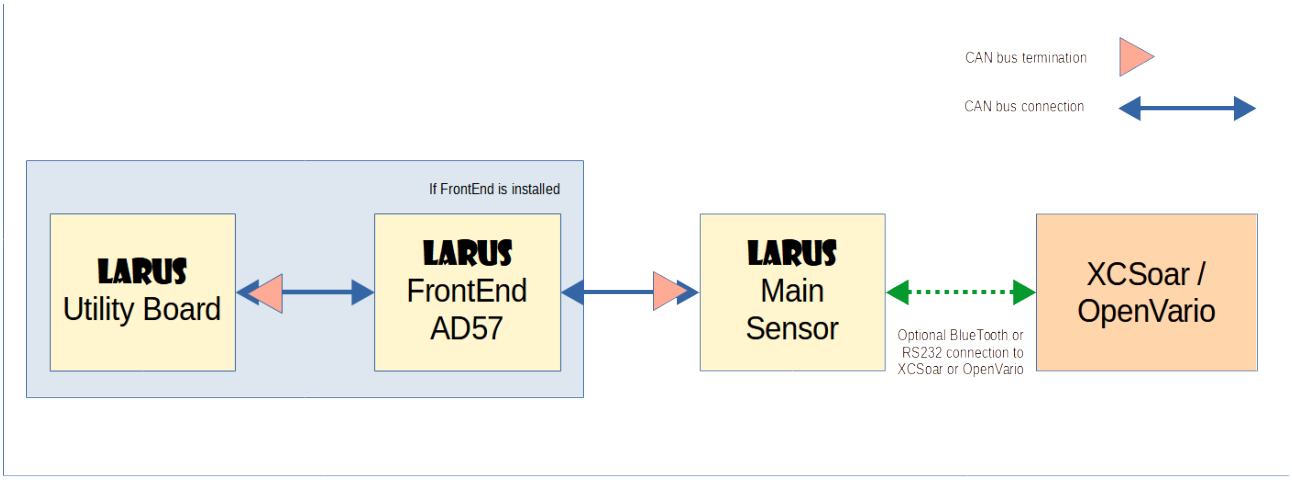


- einem optionalen Drei-Wege-Schalter (vorzugsweise im Knüppel eingebaut) zur Über-Steuerung des automatischen Steigen-Gleiten-Modus (nicht abgebildet)

3 Projektstruktur auf dem Repository

4 Anbindung der STMCUBEIDE an das Repository

5 Die Laufzeit- und Hardware-Struktur ganz grob



6 SW-Struktur

Die LARUS -SW ist ein C-, teilweise C++ basiertes Konstrukt. Sie wurde auf Windows und Unix unter Verwendung einer Entwicklungsumgebung von STM, der STMCubeIDE, gebaut.

STM als Herstellerfirma der verwendeten HW-Controller (STM32F407VG) stattet diese Entwicklungsumgebung (kurz EWU) mit einer umfangreichen Galerie von dedizierten Bibliotheken aus, so dass der Entwicklungsaufwand erheblich reduziert wird. Alle Treiber der Komponenten auf dem Controller-Chip sind schon vorgefertigt.

Außerdem folgen alle SW-Entwicklungen auf diesem Controller festen Richtlinien, was dazu führt, dass der hardware-nahe „Kern“ der LARUS -SW (`main.cpp`) mit den Mitteln, die STM bereitstellt, schon weitgehend generiert werden kann. Kenntnisse dieser Richtlinien und Vertrautheit mit dem Bau von STM-Applikation werden hier vorausgesetzt[6].

Wie fast alle eingebetteten Systeme ist LARUS ein zyklisches Dauerlaufsystem. Es wird einmal gestartet und läuft dann, bis die Spannungsversorgung ausgeschaltet wird.

Der Kern des Systems, `main.cpp,` ruft eine Laufzeit-Umgebung ins Leben, die auf Amazon-RTOS basiert. RTOS ist ein markt-gängiges präemptives Betriebssystem für Controller.

Die Neu-Entwicklung der Applikations-SW konnte mit dem applikationsnahen „Kern“, einen SW-Modul mit Namen `NV_Main.c` beginnen, der aus `main.cpp` heraus gestartet wird. Die Applikations-SW kann dann auf diese Laufzeitumgebung aufsetzen und die APIs dieser Umgebung nutzen.

`NV_Main` erzeugt zunächst verschiedene Queues und Semaphore (System-Ressourcen) und startet dann RTOS auf einem Konglomerat von interagierenden Tasks, die folgende Aufgaben haben (grob geschnitzt) :

`task_BackGround`	(setzt Default-Werte, koordiniert den Start des Systems und synchronisiert verschiedene Funktionen)
`task_FrontEnd`	(kontrolliert die Eingabe-Tasten und Encoder, setzt sie um in interne Kommandos)
`task_MMI`	(führt interne Kommandos aus und erzeugt Datenstrukturen, die vom `task_ImageBuilder` auf dem LCD ausgegeben werden können)
`task_ImageBuilder`	(verarbeitet die zuletzt genannte Datenstruktur und erzeugt das Pixel-Image auf dem LCD) (**Rendering)**
`task_CAN_Bus_Receiver`	(empfängt und decodiert DataGramme vom CAN Bus, die der Sensor oder ein anderer Mitspieler am CAN Bus liefert)
`task_CAN_Bus_Sender`	(encodiert und sendet DataGramme in den CAN Bus an den Sensor und an andere Mitspieler am CAN Bus.)

6 ***Anekdotisch** : Der Autor (Horst Rupp) hatte diese Kenntnisse a priori auch nicht. Er wurde von Klaus Schäfer eingewiesen. Das Ganze gestaltete sich als mehrjähriger (mühsam für einen alten Mann) Lernprozess, der die eigentliche Entwicklung der Vario-SW begleitete. Das Mantra von Klaus Schäfer war und ist : „Horst, du musst das Cortex-ARM-M3 Buch lesen“. Der Autor konnte das bis heute vermeiden, denn das ist ein Wälzer von 600 Seiten. Es war für ihn absehbar, dass der Aufwand das Ding zu lesen und im Detail zu verstehen in keinem Verhältnis zum Nutzen stand. Konsequenterweise hat es Klaus Schäfer dann auch unterlassen, die von Horst Rupp geschriebene Dokumentation für das Frontend zu lesen. Das sind weniger als 100 Seiten.*

`task_AudioController` (erzeugt die auditiven Ausgabe-Daten)

`task_I2C_Handler` (handhabt allen Datenverkehr auf I2C-verbundenen Devices)

`task_SD_Handler` (handhabt allen Datenverkehr mit der Mikro-SD)

`task_Buzzer` (initiiert haptische und auditive Signale zur Kommunikation mit dem Benutzer)

Hier folgend eine verbale Skizze der autonomen Schleife der AD57-SW : Data Acquisition and Rendering

Der Sensor versorgt die AD57-SW über den CAN-Bus alle 100 msec mit neuen Daten. Die Daten werden in `task_CAN_Bus_Receiver` entgegengenommen und in lokalen Datenstrukturen (*`g_SensorData`* und *`g_FrontEndData`*) abgelegt.

Mit jedem Empfang von neuen Daten, aus dem Sensor synchronisiert, spätestens jedoch, wenn die Synchronisationszeit um 2 Millisekunden überschritten wurde, übernimmt *`task_ImageBuilder`* diese Daten, verarbeitet sie (*`CAN_Data_Digester`*) und stellt sie dann dar (auf einer Text-Seite (einem Menue) oder auf einer graphischen Seite). Der *`task_ImageBuilder`* verzweigt für das Rendering in entsprechende Module (zB *`BuildHorizonPage`* *oder* *`BuildTextPage`*) . Alle Daten, die in dieser Schleife gewonnen oder erzeugt werden, stehen im Gesamtsystem global zur Verfügung.

Hier die verbale Skizze einer zweiten – autonomen – Schleife (asynchron zur vorher genannten Schleife) : Audio and Utility Driving

Der `task_AudioController` bezieht sich (völlig asynchron zur erstgenannten Schleife) auf die dort erzeugten Daten und generiert seinerseits Ausgabe-Daten, übergibt sie an den `task_CAN_Bus_Sender`, der sie bei Bedarf per CAN-Bus an das Utility-Board mit dem Tongeber sendet. Der `task_AudioController` läuft in einer Loop alle 20 msec.

Als Nebenprodukt macht der *`CAN_Data_Digester`* den Soll/Ist-Vergleich für die Wölbklappen-Stellung von Wölbklappenflugzeugen. Auch diese Daten werden bei Bedarf an das Utility-Board weitergegeben.

Die restlichen Tasks bilden einen Kontroll-Rahmen und das Nutzer-Interface : UI and Control

Ein Wort zur Nomenklatur :

In der gesamten LARUS-Dokumentation werden die Begriffe „Programm“ und „Menue“ alternativ, fast synonym verwendet. Das ist zugegebenermaßen ein Unschönheit, aber sprachlich sinnvoll.

Intern werden die wählbaren Seiten in der AD57-SW durchnummeriert von -8 bis + 20. Die negativen Zahlen bezeichnen graphische Ausgabe-Programme (Vario, Horizont, etc), die nicht-negativen Zahlen bezeichnen Text-Seiten (Menues). Die Wahl der Seite (des Programms / des Menues) erfolgt stets mit dem gleichen Eingabe-Werkzeug Seite : dem großen Encoder-Knopf.

7 Data Acquisition and Rendering

Der grobe Ablauf im **task_ImageBuilder** :

```
 while ( 1 )
  {
    //
    //  Wait for new data to arrive,
    //  but loop anyway if data is late or receiver not active
    //
    //  Synchronize with semaphore controlled by task-CAN_Bus_Receiver
    //
    (void) xSemaphoreTake( CAN_2_UI_synchronizer, c_WakeImageBuilder + 2);
                                                                    //Synchronisation
    //
    //  Digest latest data having arrived from sensor.
    //  This is a thick routine. Fasten seatbelts and dive into it.
    //
    CAN_Data_Digester ();                                           // Datenübernahme
    .
    .
    //
    //  clear canvas
    //
    ba_graphicsClrScreen ( .. );
    //
    //  paint pages to canvas
    //
    switch ( l_CurProg )
    {
      //
      //  graphic pages - non menues
      //
      case
      .
      .
      case c_Vario_1 :
        Adjust_Scale ();
        BuildWindPage ();           // Rendering beispielhaft für graphische Seiten
        break;
      .
      .
      //
      //  menue pages
      //
      default :
        BuildTextPage ();           // Rendering beispielhaft für alle Text-Seiten
        break;
    }

    .
    .
    //
    //  Output all data to canvas
    //
    ba_displayRefresh();
  }
}
```

7.1 Rendering der graphischen Seiten

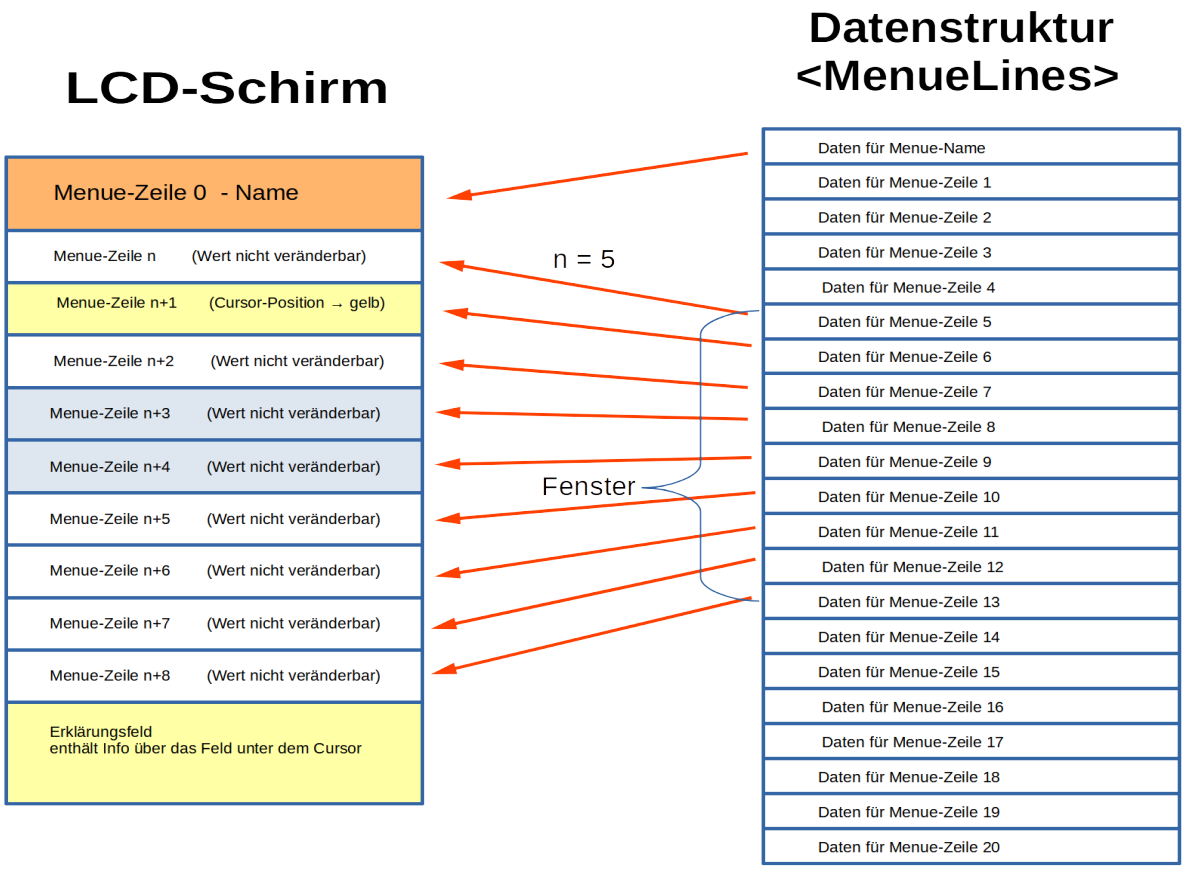
Für den Fall, dass die Darstellung dieser Daten in graphischer Form geschieht (Variometer-Bild, Horizont-Bild, Wendezeiger-Bild, Wind-Anzeige), rendern dedizierte Routinen (`BuildVarioPage.c`, `BuildHorizon.Page.c, ..`) im `task_ImageBuilder` das graphische Bild auf und lassen es auf dem LCD sichtbar werden.

Die graphischen Seiten (Vario, WindPfeile, Wendezeiger , Horizont, G-Meter…) sind „durchprogrammiert“, das heißt : Der jeweilige Code, der diese Seiten aufblendet, kann nur genau diese Seite aufbauen. Alle Information ist direkt in dem entsprechenden Code-Stück vorhanden. Diese Seiten erfordern in ihrer inneren Struktur keine Erklärung an dieser Stelle. Der Code sollte direkt lesbar und weitestgehend selbsterklärend.

7.2 Rendering der Menue-Seiten

Hier eine beispielhafte Menue-Seite :

Menue-Name
Menue-Zeile 1
Menue-Zeile 2
Menue-Zeile 3
Menue-Zeile 4
Menue-Zeile 5
Menue-Zeile 6
Menue-Zeile 7
Menue-Zeile 8
Menue-Zeile 9
Erklärungsfeld



7.2.1 Rendering im Detail

Das Rendering der Text-Seiten erfolgt in der Routine ***BuildTextPage*** in **task_ImageBuilder**. Hier ist die Struktur dieser Routine in grober Darstellung :

```
void BuildTextPage ( void  )
{
  .
  .
  //
  //  Prepare output data structure
  //
  BuildTheMenueContent ();                                  // Aufbau von MenueLines

  l_CurProg = g_ProgramStatus.CurrentProgram;
  .
  .
  .
  .
  //
  //  iterate through the current program's definition of fields and show those fields.
  //  Output takes care of scrolling of lines in case the current program requires more
  //  lines than c_max_no_menue_lines.
  //
  for ( uint8_t j = 0; j < c_max_no_menue_lines; j++ )      // Ausgabe von MenueLines
    OutputAMenueLine2Screen ( j );

  if ( g_ProgramStatus.Focus >= c_Focussed )                // Ausgabe des Erklärungstextes
  {
    At.x = c_explanation_x_offset;
    At.y = c_explanation_y_offset;
    DisplayTheExplanationAt ( At );
  }
}
```

Der Aufbau des sichtbaren Bildes erfolgt tabellengesteuert im Modul ***Renderer_Lib.c***. Die Tabelle für die Steuerung steckt in Modul ***Control_Tables.c***.

Die Darstellung des Renderings ist leichter verständlich, wenn man den Ablauf zeitlich „von hinten nach vorne“, von der Physik des Schirms hin zur Steuerung und zur Datenaufbereitung betrachtet.

Schauen wir also zunächst auf die Ausgabe auf den Schirm : die Routine ***OutputAMenueLine2Screen*** im Modul ***Renderer_Lib.c*** erledigt diese Aufgabe.

Der LCD-Schirm kann 228 * 286 Pixel darstellen. Darauf wird folgende Datenstruktur dargestellt :

MenueLine_t MenueLines[*c_max_no_of_fields_in_program*]; // zZ 20

```
  typedef struct
  {
    uint8_t   TextLeft[c_size_TextWholeLine];
    uint8_t   TextValue[c_size_TextValue];
    uint8_t   TextUnits[c_size_TextUnits];
    uint8_t   Selectable;
    uint8_t   Focus;
    uint8_t   CurrFldIdx;
    union
    {
      float   FloatValue;
      int32_t IntValue;
    };
  } MenueLines_t
```

Diese Datenstruktur wird durch die Routine ***OutputAMenueLine2Screen (j)*** auf den Schirm übertragen .

Die Zahl der Zeilen auf dem Schirm ist kleiner als die Zahl der möglichen Zeilen in der Datenstruktur.

Die Datenstruktur muss nicht notwendigerweise vollständig gefüllt sein, kann weniger Zeilen enthalten als der Schirm, kann aber auch mehr Zeilen enthalten.

Wenn die Füll-Größe der Datenstruktur kleiner oder gleich der Zahl der Schirm-Zeilen ist, ist die Abbildung trivial.

Wenn die Füll-Größe der Datenstruktur größer ist als die Zahl der Schirm-Zeilen, wird ein „Fenster“ von Zeilen der Datenstruktur auf den Schirm abgebildet. Das Fenster kann auf der Datenstruktur bewegt werden. Das entspricht dem „Scrollen“ auf dem Schirm.

Nun der vorherige Schritt im Ablauf des Renderings :

In der Routine ***BuildTheMenueContent*** wird die Datenstruktur **<MenueLines>** gefüllt.

Darin wird die Erzeugung der unterschiedlichen Menues gesteuert durch Interpretation einer großen Datenstruktur (***ControlTables.c***). Das zentrale ausführende Organ der Tabellensteuerung ist dabei die Routine ***Renderer_Lib.c***.. Sie füllt bei jedem Durchlauf die Datenstruktur (**TheMenueLines[c_max_no_of_fields_in_program]**) neu, die dann ohne weitere Informationszufuhr durch ***OutputAMenueLine2Screen*** ausgegeben werden kann und dann genau ein Menue-Muster ergibt.

Mit diesem Mechanismus werden im derzeitigen Ausbauzustand des Varios ***20*** verschiedene Menue-Seiten erzeugt, in denen so unterschiedliche Datentypen wie Koordinaten, Komma-Zahlen, Kursangaben, Tageszeiten, … vorkommen. Die Diversität der Datentypen und ihre Darstellbarkeit werden gewährleistet durch ***Renderer_Lib.c*** als ausführendes Organ und ***ControlTables.c.*** als steuernde Datenstruktur (Tabellensteuerung).

Der Renderer interpretiert dabei für jede Menue-Zeile genau einen Eintrag in der folgenden Datenstruktur

ROM LongFieldDescriptorItem_t FieldDescriptors[c_Max_Fields]

Hier die Typ-Definition der Beschreibung einer Menue-Zeile :

```
typedef  struct
  {
    uint16_t  Idx_Field_Descr;                «««« eindeutiger Index des Feldes
    uint8_t   ConfigRelevant;                 «««« Konfigurationsrelevant ja/nein
    uint8_t   TextLeft[c_size_TextLeft];      «««« textuelle Beschreibungen
    uint8_t   TextShort[c_size_TextShort];    «««« des Feldes
    uint8_t   Explanation1[c_size_Explanation];«««« Text der im Erklärungsfeld erscheinen soll
    uint8_t   Explanation2[c_size_Explanation];«««« Text der im Erklärungsfeld erscheinen soll
    uint8_t   FieldType;                      «««« Typ des Feldes
    uint8_t   ValidInFlight;                  «««« = 1 wenn dieser Wert veränderlich ist
    uint8_t   UnitSpec;                       «««« Index in ein Array von Strings
    uint8_t   Decimals;                       «««« Anzahl der darzustellenden Dezimalstellen

    uint32_t* iPtrToValue;                    «««« Pointer auf Integer-Wert
    int16_t   iLowLim;                        «««« untere Grenze des Wertes
    uint16_t  iUpLim;                         «««« obere Grenze des Wertes
    uint16_t  iStepWidth;                     «««« Schrittweite der Veränderung

    float*    fPtrToValue;                    «««« Pointer auf Float-Wert
    float     fLowLim;                        «««« untere Grenze des Wertes
    float     fUpLim;                         «««« obere Grenze des Wertes
    float     fStepWidth;                     «««« Schrittweite der Veränderung

  }  LongFieldDescriptorItem_t;
```

Jede Menue-Seite wird in der Datenstruktur

```
ROM  ProgramDescItem_t    Programs[c_Max_Menues_ExpertMode]
```

durch einen Eintrag vom Typ

```
typedef  struct
  {
    uint8_t               NoOfFields;  // including the menue line of current program
    FieldInProgramItem_t  FieldInProgram[c_max_no_of_fields_any_program];  // = 20
  }  ProgramDescItem_t;
```

beschrieben. Hier ein Beispiel :

```
{ 11,                                          // program 0
  {
    { c_Basic_Setup,            c_NeverMod },
    { c_MacCready,              c_AlwaysMod },
    { c_Ballast,                c_AlwaysMod },
    { c_PilotMass,              c_AlwaysMod },
    { c_Bugs,                   c_AlwaysMod },
    { c_Vario_Volume,           c_AlwaysMod },
    { c_SC_Volume,              c_AlwaysMod },
    { c_Brightness,             c_AlwaysMod },
    { c_Time_Selector,          c_AlwaysMod },
    { c_DarkThemeOn,            c_AlwaysMod },
    { c_NormalMode,             c_AlwaysMod }
  }
}
```

Die Datenstruktur für ein Programm besteht demnach aus einer Liste von maximal 20 Indices (im Beispiel sind es 11), die auf Einträge in der Datenstruktur **<FieldDescriptors>** zeigen..

Jede Zeile definiert mit ihrem Index die Feldbeschreibung, die zum Aufbau einer Zeile der Menue-Seite herangezogen werden. Der zweite Wert jeder Zeile beschreibt, ob dieser Wert im Menue-Kontext veränderlich ist oder nur angezeigt werden kann. Auch der Name des Menue s wird in einem Eintrag abgelegt, der der o.g. Typdefinition entspricht. Bitte inspizieren sie dazu die ersten Einträge der langen Liste von **<FieldDescriptors>**.

7.2.2 Rendering der [Vario-Values]-Seite

Auf der **[Vario-Values]**-Seite sind (ähnlich wie in den Menues) Textfelder zu sehen. Das Rendering dieser Felder erfolgt analog zum Rendering der Menue-Felder ab.

Die hier erscheinenden Felder sind in dem Sinne frei programmierbar, als dass sie in einem Konfigurationsvorgang ausgewählt werden können. Dabei werden zwei Sets von Feldern definiert, jeweils passend für die Modi „Kurbeln“ oder „Gleiten“ („Sollfahrt“ oder „Steigen“).

8 Audio and Utility Driver

Als Einstieg hier das Pflichtenheft der Entwicklung aus dem April 2020, überarbeitet im April 2021

8.1 Ausgabe eines in Tonhöhe und Lautstärke regelbaren Tones.

Tonhöhe steuerbar über einen linearen Parameter mit der Range -2.0 bis +2.0 .

Das Audio selbst definiert die maximale und die minimale Frequenz, die es ausgibt.

Der Wert -2 entspricht der minimalen Frequenz, der Wert +2 entspricht der maximalen Frequenz.

Der Wert 0 entspricht der Mittenfrequenz fm = sqrt (fmin * fmax)

Die Steuerung der Frequenz soll einen physiologisch wahrnehmbar gleichmäßigen Frequenzanstieg erzeugen.

(Forderung Klaus)

Das Audio soll in der Lage sein, den „Ziehen“-Ton aus dem Vario-Kontext durch einen „Langsamer“-Ton im Sollfahrt-Kontext zu ersetzen.

Der bekannte Standard bei E-Varios ist der mit größer werdendem Steigen in der Frequenz anschwellende Ton, der in Intervallen zerhackt wird in mit dem Steigen anschwellender Frequenz. → Beep-Beep

Beim Sollfahrt-Fliegen soll der Ton für „Ziehen“ nicht eine gleichmäßige Frequenz sein, sondern in sich anschwellen. → huit-huit

Lautstärke steuerbar zwischen Ton-Aus und dem Maximum, was die Aussteuerung hergibt.

Die Steuerung der Lautstärke soll den linearen Anstieg des Eingangswert zwischen 0 und 100 (%) abbilden auf einen physiologisch wahrnehmbar gleichmäßigen Anstieg der Lautstärke.

Wenn aus der Ansteuerungslogik heraus das Audio getaktet wird (Mute oder Intervall), dann soll das Audio in der Lage sein, den Volume-Sprung (zB 0 auf 50 oder von 60 auf 90) zu glätten.

Analog sollen auch Frequenzsprünge geglättet werden, weil sie sonst das Hörempfinden deutlich stören.

8.2 Tonsignale

Das Vario ist in seiner internen Arbeitsweise abhängig von äußeren Einflüssen (zB der Steigen-Gleiten-Umschaltung aus dem Sensor, durch die automatisch angepasste Skalierung des Variometers, etc.) ändern kann, ohne dass der Pilot in diese Veränderung involviert ist, ist es notwendig, den Piloten durch ein Tonsignal auf diese Veränderung aufmerksam zu machen.

Deshalb soll das Audio, getrennt von den Tonsignalen für Vario und Sollfahrt, in der Lage sein, eine kleine Zahl von Tonsignalen autonom zu erzeugen.

8.3 Umsetzung auf Seite des Vario-FrontEnds

Asynchron zur Data Acquisition and Rendering – Schleife erzeugt **task_AudioController** die Daten zur Steuerung des Tongebers im Utility-Board.

Die Schleife wird alle 20 msec durchlaufen. Die Tonsteuerungsdaten werden in eine Q geladen (siehe die Struktur **AudioQItem_t).**

Analog werden, verteilt über die gesamte Applikation, völlig asynchron, an definierten Stellen im AD57-Code SignalQItem-s in eine weitere Q geladen (ein Beispiel : Tonsignal für „Ende der Liste erreicht“).

Der **task_CAN_Bus_Sender** liest beide Qs aus und sendet die Daten per CAN-Bus an das Utility-Board weiter, wo dann die eigentliche Tongeneration erfolgt.

```
//
// ***************************************************************************
//
// Audio Queue definition
//
typedef struct
{
  int16_t   NormedFrequency;      [-2.0 - +2.0]
  uint16_t  Interval;             [in climb-mode  : 20 - 200 msec     ]
                                  [in cruise-mode : fix 200 msec      ]
  uint8_t   Audio_Volume;         [in climb-mode  = Vario-Volume 0-100]%
                                  [in cruise-mode = SC-Volume    0-100]%
  uint8_t   Duty_Cycle;           [in climb-mode  = 50                ]%
                                  [in cruise-mode = 10 - 80           ]%
  uint8_t   ClimbMode;            [0 - cruising,  2 - climbing         ]
                                  [0 - huit-huit, 2 - beep-beep       ]

} AudioQItem_t;

#define    c_MaxElementsAudioQ   25
#define    c_Size_AudioQItem     sizeof ( AudioQItem_t )
//
// ***************************************************************************
//
// SignalQueue definition
//
typedef struct
{
  uint16_t  Signal_Id;            [siehe CAN_SIGNAL_IDs]
  uint16_t  Signal_Volume;        [0 - 100]%

} SignalQItem_t;

#define    c_MaxElementsSignalQ   25
#define    c_Size_SignalQItem     sizeof ( SignalQItem_t )
```

Mit den Konfigurationsvariabeln aus dem UI

```
g_ConfigData.Signal_Mute                    [0/1]
g_ConfigData.Signal_Volume                  [0-100]     %

g_ConfigData.SC_Mute                        [0/1]
g_ConfigData.Total_Mute                     [0/1]

g_ConfigData.Vario_Volume                   [0-100]     %
g_ConfigData.SC_Volume                      [0-100]     %

g_ConfigData.Vario_MuteWinUL                [+1 - +20]  cm/s
g_ConfigData.Vario_MuteWinLL                [-1 - -20]  cm/s
g_ConfigData.SC_MuteWinUL                   [+5 - +20]  km/h
g_ConfigData.SC_MuteWinLL                   [-5 - -20]  km/h
```

werden diese Datentransfers gesteuert.

8.4 Umsetzung auf Seite des Audios (Prototype-Utility-Board)

Das Audio selbst definiert

```
#define c_max_frequency          5000.0F
#define c_min_frequency          300.0F
#define c_mid_frequency          1225.0F   // ca. sqrt ( min * max )
```

und

```
enum CAN_SIGNAL_IDs
  {
    cNoSignal,
    cAutoChange,
    cInvAutoChange,
    cAlarm,
    cTransfer,
    cClick,
    cBeep,
    cmaxSigId,
  };
```

Die digitale Tonerzeugung nach Pflichtenheft erfolgt im Modul **task_AudioController** auf dem Utility-Board. Lautstärke und Tonfrequenz werden dabei über den DAC im STM32F4 Controller gesteuert.

8.5 Test des Tongebers

Diese Funktion lässt sich erfolgreich nur im Experten-Modus ausführen.

Verstreut über verschiedene Module enthält der Code des FrontEnds Einsprengsel von dieser Art

```
  if ( g_TuneTest )
{
    g_SensorData.Sensor_CR  = g_TuneTest_CR;
}
```

Die globale Variable <g_TuneTest> wird auf 1/true gesetzt, wenn in einem der Programme [Vario_Values] oder [Vario_Wind] <Button_5> gedrückt wird. Erneutes Drücken dieser Taste oder Wechsel des Programm setzt <g_TuneTest> wieder auf 0/false.

Wenn <g_TuneTest> auf 1/true gesetzt ist, dann leuchten die LEDs unter den Tasten.

Bei gesetzter Variable <g_TuneTest> kann im Programm [Vario_Values] mit dem kleinen Knopf der Vario-Zeiger (weißes Dreiecke am inneren Skalenrand) verschoben werden. Abgesehen von einem kleinen Stumm-Fenster (einstellbar in [Audio_Features]) sollte dann der Ton hörbar werden.
Dazu muss der Drei-Wege-Schalter (wenn vorhanden) auf „off“ stehen (Steigen – off – Gleiten).

Bei gesetzter Variable <g_TuneTest> kann im Programm [Vario_Wind] mit dem kleinen Knopf der Sollfahrt-Balken (weißer Balken auf der Skala) verschoben werden. Passend zu dem eingestellten Stumm-Fenster und der eingestellten Lautstärkerampe (einstellbar in [Audio_Features]) sollte dann der Ton hörbar werden.
Dazu muss der Drei-Wege-Schalter (wenn vorhanden) auf „off“ stehen (Steigen – off – Gleiten).

9 User Interface

Das zyklische Programm des FrontEnds hat einen Kern : die Datenstruktur *Generic_Common_Org.c*, und in dieser Datenstruktur enthalten die Common-Blöcke *g_ProgrammData* und *g_ConfigData* sowie *g_SensorData* und *g_FrontEndData.*

10 Einbau und Inbetriebnahme, Änderung von Sensor-Parametern

Der Einbau der Sensoreinheit bedarf einiger Sorgfalt : Die eigene drei-dimensionale Einbaulage in Bezug auf die Flugzeuglängsachse (drei Winkel) muss der Sensor-SW bekannt sein, damit sie sinnvolle Werte liefern kann. Da könnte der Horizont zu hoch stehen oder er steht schief oder auf dem Kopf, der Kompass zeigt falsch an.

Das folgende Bild zeigt den Einbau in 0-Lage an : Pfeil nach vorn, Yaw = 0°, Nick = 0°, Roll = -180 ° (auf dem Kopf). Der Roll-Winkel ergibt sich aus der Einbaulage des Sensors auf der Platine.

Es ist fast unmöglich, die Abweichungen dieser Winkel von der 0-Lage beim Einbau ganz exakt zu bestimmen. Allerdings reichen in einem ersten Schritt auch die ungefähren Winkel aus. Sie müssen vor Inbetriebnahme des Sensors in einem neuen Flugzeug in einer Text-Datei in einem vorgegebenen Format notiert werden. Diese Text-Datei muss sich auf uSD-Karte im Sensor befinden, wenn der Sensor eingeschaltet wird. Der Sensor richtet sich im Software-Start nach diesen Werten aus.

Beispiel :

```
01 SensTilt_Roll        = -3.141592
02 SensTilt_Nick        = 0.0
03 SensTilt_Yaw         = 1.570796
04 Pitot_Offset         = -7.000000e0
05 Pitot_Span           = 1.030975e0
06 QNH-delta            = 0.0
30 Vario_TC             = 4.577637e-3
31 Vario_Int_TC         = 3.051757e-3
32 Wind_TC              = 1.525879e-3
33 Mean_Wind_TC         = 3.051757e-3
40 GNSS_CONFIG          = 1.0
41 ANT_BASELEN          = 2.030000e0
42 ANT_SLAVE_DOWN       = 2.630000e-1
43 ANT_SLAVE_RIGHT      = -6.000000e-2
```

Da die Winkel-Werte nur UNGEFÄHR richtig sind, müssen folgende Parameter noch nachjustiert werden:

- Einstellung der Winkel Yaw, Nick und Roll
- Einstellung der Deklination und der Inklination

Weiterhin gibt es im Sensor noch eine Reihe weiterer Parameter, deren Änderung aus Sicht des Benutzers zur Laufzeit des Systems sinnvoll sein kann :

- Einstellung der Zeitkonstante des schnell veränderlichen Winds im Gleiten
- Einstellung der Zeitkonstante des langsam veränderlichen Winds im Gleiten
- Einstellung der Zeitkonstante des schnell veränderlichen Winds im Kreisen
- Einstellung der Zeitkonstante des langsam veränderlichen Winds im Kreisen
- Einstellung der Hysterese beim automatischen Umschalten zwischen Steigen und Gleiten

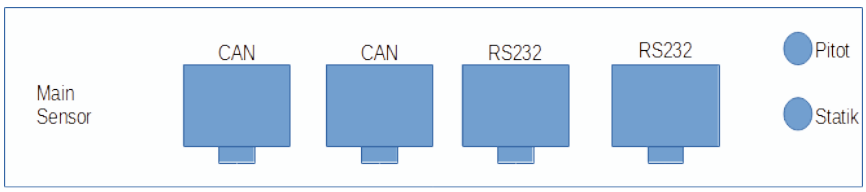
Wenn *KEIN* AD57-FrontEnd an diesem Sensor angeschlossen ist, können fehlerhafte Werte nur durch (wiederholtes) Editieren der o.g. Text-Datei korrigiert werden.

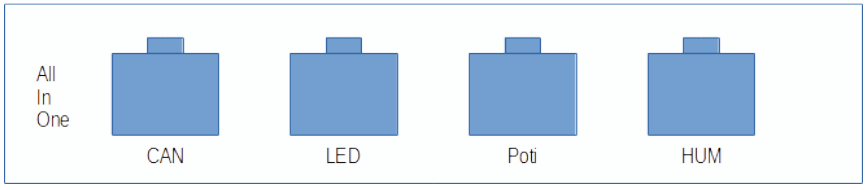
Wenn aber ein AD57-FrontEnd an diesem Sensor angeschlossen ist, können diese Parameter interaktiv verändert werden, im Menue *<Sensor SetUp>* . Die Veränderung dieser Werte erfolgt im Prinzip genau so wie die Veränderung von Parametern des FrontEnds, allerdings ist die Wirkung einer Veränderung verzögert. Der Benutzer sieht zunächst zwar seine gewünschte Änderung, aber erst wenn das Menue *<Sensor SetUp>* wieder verlassen wird, werden die neu eingestellten Werte an den Sensor gesendet. Der Sensor speichert dann diese Werte und muss eventuell einen Neustart seiner SW veranlassen. Wenn dieser Neustart ausgeführt ist, sendet der Sensor die dann neu eingestellten Werte. Ein Neustart der SW ist nur für die o.g. Winkel notwendig, die neu eingestellten Zeitkonstanten werden sofort zurückgemeldet.

Nachdem dieses Menue nach Veränderung von Parametern verlassen wurde, sollte eine kurze Wartezeit eingelegt werden, bevor das Menue erneut angewählt wird. In dieser Periode wird dem Sensor Zeit gegeben, die neu übertragenen Werte zu verarbeiten. **Der Sensor wird dabei auch die Text-Datei auf der uSD-Karte neu mit den korrigierten Werten beschreiben.** Wenn ein Neustart der Sensor-SW notwendig wird, erscheint für eine kurze Zeit die Meldung „**SENSOR OFFLINE**“.

Notabene :

Die Parameter der Deklination und der Inklination müssen dann nachgestellt werden, wenn das Flugzeug an einen entfernten Ort gebracht wird, z.B. von Frankfurt nach Sisteron. Dies ist notwendig, weil das LARUS zZ noch nicht über ein internes Modell der magnetischen Missweisung verfügt.





11 Isolierte Themen

11.1 Variometer – Mittelwert

Es stehen zwei Variometer-Mittelwerte zur Verfügung :

- ein Mittelwert aus dem Sensor[7], der beim Kurbeln das mittlere Steigen des jeweils letzten Kreises abbildet, beim Geradeausfliegen das mittlere Steigen der letzten 30 Sekunden
- den „wahre“ Mittelwert, der dadurch ermittelt wird, dass der Höhengewinn oder -verlust seit der letzten Umschaltung Gleiten-Steigen oder Steigen-Gleiten geteilt wird durch die Zeit seit diesem Umschaltzeitpunkt

11.2 Wind-Anzeigen

Es gibt zwei Windanzeigen[8], die sich in der Größe und in der Notation des Windes unterscheiden.

Auf der **Seite** *[Vario-Values]* drehen sich die kleinen Dreiecke und

- zeigen **die Windrichtung immer relativ zur Flugzeuglängsachse** an, auch im Kurbeln.

Auf der **Seite** *[Vario-Wind]* drehen sich die großen Dreiecke und

- zeigen im Kurbeln **die Windrichtung immer relativ zu Nord** an.
- zeigen im Gleiten **die Windrichtung immer relativ zur Flugzeuglängsachse** an.

7 Im Steigen mittelt der Sensor gleitend das Steigen über den letzten Kreis. Wenn der Sensor autonom aus dem Modus „Gleiten“ in den Modus „Steigen“ gegangen ist, kann demnach dieser Wert vor Beenden des ersten Kreises nur näherungsweise korrekt sein.
Im Geradeausflug mittelt der Sensor gleitend das mittlere Steigen/Fallen über die letzten 30 Sekunden.
Wenn der Benutzer* den automatischen „Steigen/Gleiten“-Modus manuell übersteuert, weiß der Sensor das nicht. Die vom Sensor gelieferten Werte sind dann aus der Sicht des Piloten* für ca. 30 Sekunden nur näherungsweise korrekt.

8 Sehr oft weicht der aktuelle Wind beim Kreisen innerhalb eines Bartes vom allgemeinen mittleren Wind (dem Gradientenwind) ab. Für die Endanflugrechnung wird deshalb sinnvollerweise der Gradientenwind (der Wind außerhalb des Thermikschlauchs) benutzt.

11.3 Die Hintergrundfarbe innerhalb des Skalenkreises

Die Hintergrundfarbe des inneren Vario-Bereiches ist im Normalfall **schwarz oder weiß (je nach Thema)**, kann jedoch andere Farben annehmen, wenn bedrohliche Flugzustände (Alarm-Situationen) bestehen, wie

- **grün** Annäherung an die oder Unterschreiten der Mindestfluggeschwindigkeit V_{min}
- **rot** Annäherung an die oder Überschreiten der Manövergeschwindigkeit V_{RA}, der absoluten Maximalgeschwindigkeit V_{NE} oder der dynamischen Lastvielfache (Envelope)
- **magenta** Überschreiten der zugelassenen Geschwindigkeiten für die jeweils aktuelle Klappenstellung
- **gelb** Zustand Bremsklappen ausgefahren bei gleichzeitig NICHT ausgefahrenem Fahrwerk

Hier ist von „Annäherung“ die Rede. Gemeint ist, dass die oben genannten Grenzen nicht exakt erreicht werden müssen, um den Alarm auszulösen. Vielmehr setzen die Alarme schon ein, wenn der kritische Wert sich bis auf eine einstellbare Marge dem Grenzwert genähert hat. Der Parameter, der diese einstellbare Marge definiert, heißt ***<SafetyMargin>***. Ergibt an, bis auf welche prozentuale Differenz sich der kritische Wert der Grenze nähern kann. Beispiel : Marge 10 %, Grenzwert 200 km/h, Alarm bei 180 km/h.

Diese Alarm-Situationen werden einmal, wie hier beschrieben, visuell kenntlich gemacht. Sie werden aber auch von Warntönen (im Audio und im Buzzer) begleitet. Das kann nervig sein, wenn z.B. im Flug aus Sicherheitsgründen permanent die Klappen ausgefahren bleiben. Das System wertet dies als Alarmsituation „Beginn der Landung ohne ausgefahrenes Fahrwerk“. Wenn solche Daueralarme auftreten, können sie durch einen Tastendruck auf ***{ESC}*** stillgelegt werden. Dieses „Stilllegen“ wird aber zeitbegrenzt, damit der Alarm nicht vergessen geht. Die Zeit, für die ein Tastendruck auf ***{ESC}*** als Belegung des Alarms wirkt, ist einstellbar (konfigurierbar) über den Parameter ***<Danger_Delay>***. Der Parameter hat in der Fabrikeinstellung den Wert 5.

11.4 Handhabung der uSD-Karte

Das Prozedere für die Nutzung des µSD-Karte als Konfigurationsspeicher, um die Konfiguration über den Update hinweg zu retten :

- Das Vario-System ist gestartet worden mit gesteckter µSD-Karte. Die Anzeige noSD auf den Vario-Seiten ist nicht sichtbar.
- Auf der o.g. Menue-Seite *[Config LoadSave]* wird der Konfigurationsschalter *[ForceSDSave]* auf 1 gesetzt.
- Mindestens 20 Sekunden warten. Die Konfigurationsdaten sind dann auch auf der µSD-Karte gespeichert.
- Das System ausschalten.
- ---------- (eventuell Update)
- Das System wieder einschalten.
- Wie bei jedem Starten des System wird zunächst die Standard-Konfiguration (Master-Default) ins Vario geschrieben. Danach wird im EEProm nach einem "gültigen" Konfigurationsdatensatz gesucht, aber es wird kein gültiger Konfigurationsdatensatz gefunden. Der aktuelle Datensatz im EEProm wird mit einer anderen Versionsmummer ausgestattet sein als der Update - :) sonst wäre ja kein Update notwendig gewesen.
- Auf der o.g. Menue-Seite *[Config LoadSave]* wird der Konfigurationsschalter *[ForceSDSave]* auf 1 gesetzt.
- 20 Sekunden warten. Die neuen Konfigurationsdaten sind jetzt auch EEProm gespeichert.
- Das System ausschalten.
- ---------- (KEIN Update)
- Das System wieder einschalten.
- Wenn jetzt der Schalter *[ForceSDSave]* mit Wert 1 gefunden wird – wie hier provoziert – , dann werden auch auf der µSD-Karte Konfigurationsdaten gesucht.
- Wenn die Konfigurationsdaten auf der µSD-Karte gefunden werden und gelesen werden können, dann werden alle vorher eingestellten Konfigurationsdaten (die Master-Default und die Daten aus dem EEProm) überschrieben.
- In jedem Fall wird dann im EEProm der Konfigurationsschalter *[ForceSDSave]* auf 0 zurück gesetzt.

*Wenn eine µSD-Karte im Slot steckt und die Daten gelesen werden, **könnten** diese Konfigurationsdaten "ALT" sein, d.h. geschrieben worden sein mit einer älteren Version der Vario-FrontEnd-SW. Das kann dazu führen, dass ausgealterte, obsolete Konfigurationsparameter ignoriert werden. Der Nutzer merkt davon idR nichts. Gültige Konfigurationsparameter werden übernommen.*

Auf diese Art und Weise ist es möglich, Konfigurationen über Systeme hinweg auszutauschen. Und es ist möglich eine bewährte Konfiguration über den SW-Update zu retten.

Während des normalen Betriebes erkennt das System autonom, wenn, wann und ob konfigurationsrelevante Parameter geändert wurden. Wenn es eine solche Veränderung sieht, wird es 15 Sekunden warten, bis es die veränderten Konfigurationsparameter in das EEProm schreibt. Diese Wartezeit ist deshalb sinnvoll, weil erfahrungsgemäß die meisten Veränderungen nicht alleine kommen, sondern immer gleich mehrere Parameter geändert werden. Damit werden Speichervorgänge im EEProm[9] gespart. Jede Änderung eines konfigurationsrelevanten Parameters startet die Wartezeit neu.

9 Das EEProm verträgt eine begrenzte Anzahl von Speichervorgängen. Diese Zahl ist zwar sehr hoch und sollte in einer Systemlebenszeit von 30 Jahren nie erreicht werden, aber ….

Was heißt „konfigurationsrelevant“ ?

„Konfigurationsrelevant“ sind alle einstellbaren und alle gemessenen Werte, die notwendig sind, um nach einer Betriebsunterbrechung (Strom aus ein und Neustart) einen sauberen System-Neustart im alten Look&Feel zu gewährleisten.

Einfache Beispiele sind : Die MacCready-Einstellung, die Ballasteinstellung, die Mücken, der Flugzeugtyp.

Das System regelt das voll-automatisch, deshalb wird der Benutzer* von den komplexeren Details verschont :-).

Mit dem Schalter ***<Reset_Config>*** werden die Konfigurationsdaten auf Master-Default/Fabrikeinstellung zurückgesetzt[10].

Das sollte mit Vorsicht erfolgen, weil danach alle Personalisierungen wiederholt werden müssen, auch die Flugzeugauswahl. Aber diese Funktion ist ein probates Mittel, um eine völlig verkorkste Konfiguration wieder auf einen definierten Anfang zu stellen.

Kapitel 13.1 enthält eine vollständige Liste der einstellbaren Parameter. Die konfigurationsrelevanten Parameter sind mit „CR“ markiert.

Der Sensor des Varios kann autonom entscheiden, in welchem der beiden Zustände das Vario betrieben werden soll :

- Steigen – Kurbeln
- Gleiten

Erfahrungsgemäß ist diese automatische Moduswahl aus dem Sensor beim Flug an einem Hang oder beim Flug in der Welle oder im F-Schlepp aber nicht immer optimal. Der Sensor wählt „Gleiten“, wo aus Sicht des Piloten* „Steigen“ sinnvoller wäre.

Es gibt zwei Möglichkeiten, den Sensor zu übersteuern :

- Mikro-Schalter am Wölbklappengestänge, der ab einer eingestellten (sinnvollerweise, wenn die Klappen zwischen „Neutral“ und „Steigen“ stehen) Klappenstellung schließt → „Steigen“.
  Diese Funktion kann durch Konfiguration wirkungslos gesetzt werden (***<UseFlapsAboveNeutral>***). Hierzu sind Hardware-Voraussetzungen zu beachten.
- einem Ein-Aus-Ein-Schalter (Drei-Wege-Schalter) im Knüppel

Eingebaute Vorrangregelung:

Wenn der Drei-Wege-Schalter in Stellung AUS steht und der Mikroschalter im Wölbklappengestänge nicht als aktiv konfiguriert ist (***<UseFlapsAboveNeutral>*** nicht gesetzt), dann regiert die automatische Umschaltung aus dem Sensor den Variobetrieb.

Wenn der Drei-Wege-Schalter in Stellung AUS steht und der Mikroschalter im Wölbklappengestänge als aktiv konfiguriert ist, dann regiert dieser Schalter.

Wenn der Drei-Wege-Schalter NICHT in Stellung AUS steht, dann regiert der Drei-Wege-Schalter.

Unter der Anzeige der Spannungsversorgung wird als Piktogramm der derzeitige *ClimbMode* angezeigt :


für einen steigenden Gleitpfad im Geradeausflug


für einen ebenen Gleitpfad im Geradeausflug

10 Wenn dieser Parameter auf „ein == 1“ gesetzt wird, nimmt das System dies zwar wahr, setzt den Wert aber sofort wieder zurück auf „aus == 0“, so schnell, dass der Betrachter es nicht immer wahrnehmen kann.

↘ für einen fallenden Gleitpfad im Geradeausflug

und

● für Steigen im Kreisflug

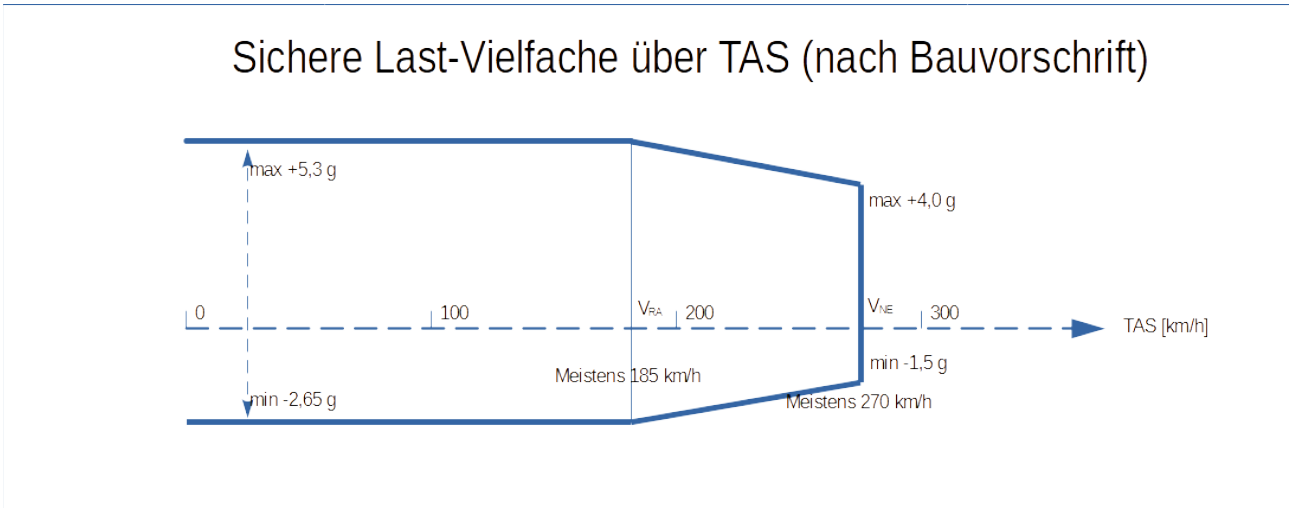
Wenn im Geradeausflug das **Sollfahrt-Kommando** den Piloten* veranlassen würde, eine Geschwindigkeit langsamer als die Geschwindigkeit des besten Gleitens anzustreben, dann wird das Sollfahrt-Audio-Signal für „Langsamer fliegen“ (huit-huit) ersetzt durch das Vario-Signal für Steigen (piep-piep). Dies wird gleichzeitig optisch kenntlich gemacht durch das rote Dreieck anstelle eines der o.g. Pfeile:

▲ Details dazu auch im Kap. AUDIO[11].

Dieses Feature ist nicht jedermanns Sache und kann in Menue ***<Features_Setup>*** mit dem Parameter ***<Allow_AudioClimbCruise>*** ein- und ausgeschaltet werden.

11.5 Turbulenz

Das Feld „Turb“ zeigt einen Wert für die Turbulenz an. Das ist zZ noch ein Experiment. Mit diesem Wert soll in Zukunft die Nervosität des Sollfahrtgebers gesteuert werden.



11 Die Erfahrung zeigt, dass diese Situation vor allem dann auftritt
- wenn unter einem großen CU die Stelle des besten Steigens gesucht wird
- wenn unter Wolkenstraßen geflogen wird
- wenn marginal geradeaus geflogen werden kann (Delphin)
- wenn am Hang oder in der Welle geflogen wird

Die Idee ist, den Piloten nicht dazu zu verleiten langsamer zu fliegen als die Geschwindigkeit, die er braucht, um bequem, sicher und zügig einkreisen zu können.

11.6 Konzept für Warnungen und Alarme

11.6.1 Alarmbelegung -Danger Delay

11.6.2 Margin

l_Safety_Margin_Low = 1.0 - (float) g_ConfigData.SafetyMargin / 100.0;
l_Safety_Margin_High = 1.0 + (float) g_ConfigData.SafetyMargin / 100.0;

11.6.3 Konzept auf LEDs

11.6.4 Konzept auf dem AD57 Schirm

11.7 Wind

Wind Mittelung im Gleiten wenn Turnrate < 3 °/sec

Wind Mittelung im Steigen wenn Turnrate > 1,5 °/sec
oder wenn im Startvorgang
oder im F-Schlepp
IAS > 120
Turnrate < 3 °/s

oder Welle
TAS > 100
Turnrate < 1 °/s
CR_Mittel > 0

am Boden alles = 0

Selector 0

```
g_SensorData.WS_Curr  = g_SensorData.Sensor_WS;          // Klaus current wind force
g_SensorData.WD_Curr  = g_SensorData.Sensor_WD;          // Klaus current wind dir
g_SensorData.WS_Mean  = g_SensorData.Sensor_WS_Average;  // Klaus mean wind force
g_SensorData.WD_Mean  = g_SensorData.Sensor_WD_Average;  // Klaus mean wind dir
```

Selector 1

```
g_SensorData.WS_Curr  = g_SensorData.Sensor_WS_Average;  // Klaus mean wind force
g_SensorData.WD_Curr  = g_SensorData.Sensor_WD_Average;  // Klaus mean wind dir
g_SensorData.WS_Mean  = l_WS_Mean;                       // Horst mean wind force
g_SensorData.WD_Mean  = l_WD_Mean;                       // Horst mean wind dir
```

11.8 Check sensor system status

11.9 Behandlung LogBook Zeitfortschreibung

11.10 Konzept Signal Gebung und Buzzer

11.11 Tau zu Null

Definition

```
if (  g_ConfigData.XXX_Tau == 0.0 )
  l_XXX_Tau  = 1.0;
else
  l_XXX_Tau  = 1.0 / ( g_ConfigData.XXX_Tau * Frequenz );
ll_XXX_Tau = 1.0 - l_XXX_Tau;
```

Anwendung

```
float x_OAS = l_OAS_Error_Tau *
                Opt_CAS_For (
                                g_ConfigData.MacCready - g_SensorData.AMM,
                                    g_SensorData.TAS
                            );
g_SensorData.OAS = ll_OAS_Error_Tau * g_SensorData.OAS + x_OAS;
```

11.12 Detektion Start, Landung

Mdmömdfö

11.13 Konzept Freier Flug versus Startvorgang

xvcbcxbc

11.14 Single thread I2C Handler

xcvbbcv

11.15 Single thread SD Handler

cxvbcvbv

11.16 Scale adjustment

Die Skala des Variometers reicht von -5 m/s bis +5 m/s. In der Regel reicht diese Spanne in Mitteleuropa auch aus. Aber es gibt Ausnahmen : massive Bärte, riesiges Fallen zB bei gewittrigen Lagen, massives Steigen, endloses Fallen bei der Wellenfliegerei. In diesen Fällen passt sich das Vario-System an.

Wenn der mittlere Vario-Ausschlag bei mehr als ¾ des aktuellen Vario-Skala-Endwertes ist (im Fallen wie im Steigen), dann wird die Skala im 1 m/s erweitert. Diese neu eingeteilte Skala ist dann logarithmisch so geteilt, dass der Winkelausschlag für 1 m/s gleich bleibt. Dieser Vorgang kann sich wiederholen bis zu einem Skala-Maximal-Wert von +-25 m/s.

Wenn das mittlere Steigen/Fallen sich wieder normalisiert, wird auch der Skalen-Endwert wieder kleiner. Siehe Routine ***void Adjust_Scale (void);***

11.17 Handlung ConfigData

xcvbxcvb

11.18 Switch back

xcvbcvb

11.19 PopUpWindows

xcbcvbxcv

11.20 Brightness Control

xvbcvbxcv

11.21 Check connection forces other parameter to reset

xcvbnxcb

11.22 Interdependenzen zwischen FlapsSensConnect, UseFlapsSwitch, FlapsControl

xcbbbcv

11.23 Interdependenzen zwischen LoggingOn, g_Mounted, g_SD_Present, g_LogFile_Opened

xcbbbc

11.24 Interdependenzen zwischen Start ohne Logging, Start ohne SD

xcbvvxvb

11.25 Verhalten, wenn uSD im Betrieb gezogen wird

xcvbcbv

12 Blame-Board von LARUS

LARUS konnte nur entstehen durch die Zusammenarbeit und die Beiträge folgender Personen (in alphanumerischer Reihenfolge)

Betz, Max, Segelflieger, Software- und Hardware-Entwickler, ehemaliger Student von Prof. Dr. Klaus Schäfer an der FH Darmstadt

- Hardware- und Software-Entwicklung des Sensor-Boards

- Guru des GitHub-Auftritts

Foerderer, Marc, Segelflieger, CEO und Partner bei Air Avionics

- Unterstützung durch Material-Überlassung

Langer, Stefan, deutscher Spitzenpilot, baut und vertreibt das OpenVario System

- Alpha-Tester, Unterstützung durch Test und Feedback

Leutenegger, Stefan, Schweizer Spitzenpilot, Professor an der TU München, Schwerpunkt-Thema „Steuerung autonomer Systeme“, viele Jahre Erfahrung um Bau von Drohnensteuerungen. hat in 2011 die Windschätzung im Butterfly Air Glide S geschrieben

- Software-Entwicklung Windschätzung

Maier, Felix, Segelflieger, Maschinenbauingenieur (FH), QS-Ingenieur bei Continental

- 3-D-Druck

Rupp, Horst, Segelflieger, Dipl.-Inform., Nestor des Projekts, ehemals Software-Entwickler, dann IT-Manager und IT-Berater, viele Jahre Erfahrung im Bau von Variometern, hat auf die alten Tage Programmieren wieder neu lernen müssen

- Software-Entwicklung FrontEnd und BootLoader

- Hardware- und Software-Entwicklung Prototype Audio

Schäfer, Klaus, Segelflieger, Professor an der FH Darmstadt, Schwerpunkt-Thema “Microcontroller“, viele Jahre Erfahrung im Bau von Drohnensteuerungen und Variometern

- Software-Entwicklung Sensor, AHRS und Windschätzung

- Guru der Entwicklungsplattform STM32CUBEIDE

Simon, Winfried, Segelflieger, Dipl.-Ing. für E-Technik, Software-Entwickler, viele Jahre Erfahrung im Bau von Variometern

- Wichtigster Sparringspartner und Ideenreiniger im Team

- Vorverarbeitung und Darstellung von Flugzeugpolaren für die Nutzung im FrontEnd

Wahlig, Uwe, deutscher Spitzenpilot, Software-Entwickler und SAP-Berater

- Alpha-Tester, Unterstützung durch Test und Feedback