LARUS

Interne Strukturen und Abläufe

Autor : Horst Rupp

1 Einleitung

Der Autor der **LARUS** - FrontEnd-SW ist fast 75 Jahre alt und möchte den Code einem nachfolgenden „Hüter und Weiterentwickler“ in die Hand 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 wieviel leichter es einem Softwerker fällt, Code zu verstehen, dessen Autor er nicht ist, wenn ihm ein klein wenig Hilfestellung gewährt wird und ihm die Knackpunkte erkärt 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. 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.

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 diese HW 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 -** All-In-One-Modul genannt.

Es ist erfahrungsgemäß[1] eine Herausforderung, ein Dokument wie das vorliegende zu schreiben, einmal, weil Dinge, Themen, Zusammenhänge 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, PrgNav_Lib.c, Generic_FieldDescriptor.c, Generic_Const.h und Generic_Types.h)***

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

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

Das AD57 mit einer der neu programmierten Menue-Seiten ist im Folgenden abgebildet.

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

4 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[2].

Der Kern `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 im Wesentlichen mit dem applikationsnahen „Kern", eine 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.

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

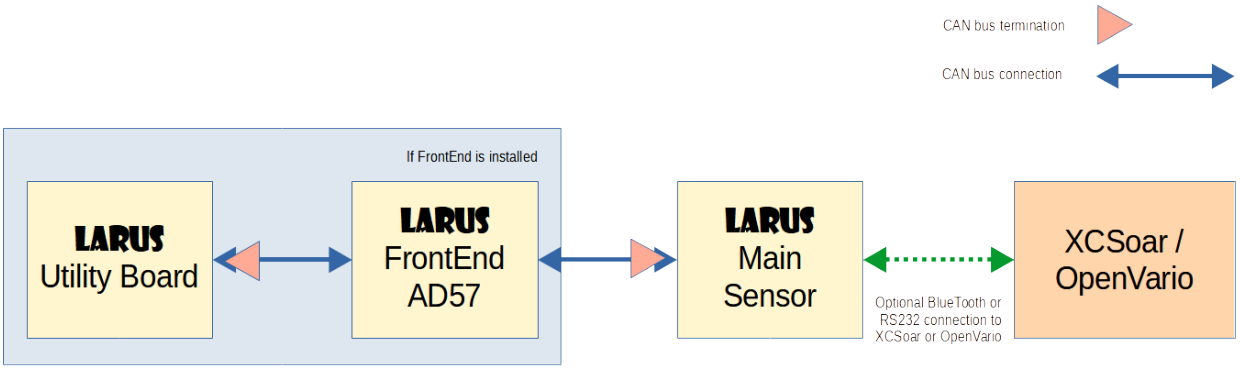
Genau diese Eigenschaft macht es didaktisch schwierig, ein einfach lesbares, sofort erkennbar klar gegliedertes Papier über ein solches System zu schreiben. Man kann das nur ganzheitlich und iterativ inhalieren. Also Geduld ! Hinweise, irgend etwas an diesem Handbuch zu verbessern, werden sofort angenommen.

NV_Main erzeugt zunächst verschiedene Queues und Semaphore (System-Ressourcen) und startet dann ein Konglomerat von interagierenden Tasks mit folgenden Aufgaben (grob geschnitzt) :

- → `task_BackGround` (setzt Default-Werte, koordiniert 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)
- → `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.)
- → `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)

2 *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 in keinem Verhältnis zum Nutzen steht. Konsequenterweise hat es Klaus Schäfer dann auch unterlassen, das von Horst Rupp geschriebene Handbuch für das Frontend zu lesen. Und das hat aber nur 50 Seiten.*

5 Die Laufzeitstruktur



Der Sensor versorgt das FrontEnd über den CAN-Bus zyklisch mit 100 Hz mit neuen Daten. Die Daten werden in `task_CAN_Bus_Receiver` entgegengenommen und in lokalen Datenstrukturen (`g_SensorData` und `g_FrontEndData`) gespeichert.

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 (im Menue oder auf einer graphischen Seite). Der `task_ImageBuilder` verzweigt in entsprechende Routinen (zB `BuildHorizonPage` *oder* `BuildTextPage`).

In einer weiteren nicht-synchronisierten Schleife übernimmt der `task_AudioController` die Daten und erzeugt daraus die Kommandos, die an das Audio gesendet werden, in `task_CAN_Bus_Sender`.

5.1 Die graphische Seiten

Für den Fall, dass die Darstellung dieser Daten in graphischer Form geschieht (Variometer-Bild, Horizont-Bild, Wendezeiger-Bild, Wind-Anzeige), bauen 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.

5.2 Die Menue-Seiten

Für den Fall der Darstellung von alphanumerischen Menue-Seiten übernimmt eine generische Routine (*BuildTextPage.c)* diese Aufgabe. Die Erzeugung der unterschiedlichen Menues wird gesteuert durch Interpretation einer großen Datenstruktur (*Generic_FieldDescriptor.c*). Das zentrale ausführende Organ der Tabellensteuerung ist dabei die Routine *PrgNav_Lib.c*.. Sie erzeugt ihrerseits eine neue Datenstruktur (*TheMenueLines[c_max_no_of_fields_in_program]*), die dann ohne weitere Informationszufuhr (wie eine graphische Seite) 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[3]. Die Diversität der datentypen und ihre Darstellbarkeit werden gewährleistet durch *PrgNav_Lib.c* als ausführendes Organ und *Generic_FieldDescriptor.c.*

5.2.1 Generic_Common_Org.c / Generic_Const.c / Generic_FieldDescriptor.c

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.*

g_ProgramData enthält den aktuellen logischen Zustand des Varios (welches Programm ist aktiv, auf welcher Zeile steht der Cursor, ist das Feld unter dem Cursor veränderbar, ist es „offen“,…)

g_ConfigData enthält alle Informationen, die den nahtlosen Wiederanlauf des System sicherstellen.

g_SensorData enthält alles, was aus den Sensoren kommt, und alles, was aus diesen Werten direkt abgeleitet wird (Mittelwerte). Diese Daten sind volatil mit eine Frequenz von 100 Hz. Die Routine *CAN_Data_Digester* übernimmt die Sensor-Daten über *CAN_Bus-Receive*r vom Sensor und verarbeitet und speichert sie primär in *g_SensorData,* nachgeordnet auch in *g_FrontEndData.*

g_FrontEndData enthält alle Daten, die nicht aus *g_SensorData* abgeleitet werden, und alle die, die zwar aus *g_SensorData* abgeleitet werden/wurden, aber eine Transformation erfahren müssen, bevor sie dargestellt werden können (zB LocTime, Zeiger).

Generic_FieldDescriptor.c enthält (u.a.) die Beschreibungen der darstellbaren Menue-Felder.
Hier zunächst der DatenTyp eines Eintrages in *Generic_FieldDescriptor.c* :

```
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];     «««« ..
    uint8_t   Explanation2[c_size_Explanation];     «««« ..
    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
```

3 *Neben der Routine CAN_Data_Digester gehören die Routine PrgNav_Lib.c . und der task_MMI zu den komplexesten Bestandteilen des Systems.*

```
    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;
```

Notabene :

Die angegebene Schrittweite der Veränderung wird benutzt beim Ändern des Wertes über den äußeren Knopf/Encoder. Bei Veränderung des Wertes über den inneren Knopf wird die Schrittweite mit 10 multipliziert.

Die Datenstruktur, in der diese Typdefinition `c_Max_Menues_ExpertMode` mal instanziiert wird, heißt `ROM  ProgramDescItem_t  Programs[c_Max_Menues_ExpertMode] =`

- Jeder dieser Einträge beschreibt ein darstellbares Feld in einem Menue.
- Jeder dieser Einträge hat einen eindeutigen Index `Idx_Field_Descr`.
- *`Generic_Const.c`* enthält (u.a.) eine Liste der Indices aller darstellbaren Menue-Felder.

Wichtig : Die Reihenfolge der Indices in `Generic_Const.c` bestimmt auch die Reihenfolge der Feldbeschreibungen in `Programs`. Beim Programmstart wird dieser Zusammenhang überprüft. Der Programmstart gelingt nur, wenn die Reihenfolgen übereinstimmen.

Die Sammlung der darstellbaren Menues steckt in der Datenstruktur

```
ROM  ProgramDescItem_t    Programs[c_Max_Menues_ExpertMode] =   // TODO
    {
      { 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 erste Zeile entspricht dem Namen dieser Menue-Seite/Programm, die weiteren Zeilen definieren mit ihren Indices die Feldbeschreibungen, die zum Aufbau der Menue-Seite herangezogen werden. Der zweite Wert jeder Zeile beschreibt, ob dieser Wert im Menue-Kontext veränderlich ist oder nur angezeigt werden kann. Je Menue/Programm sind maximal `c_max_no_of_fields_in_program` Zeilen möglich (im Beispiel ist die aktuelle Zahl von Werten = 11).

6 Die Tasks en detail

Alle Tasks sind in ihrer Struktur zwei-geteilt. Es gibt immer eine Initialisierungsphase, die auch praktisch leer sein kann, und eine Schleife. Der Code der Tasks ist intern gut detailliert und kommentiert und sollte sofort verständlich sein. In den folgenden Abschnitten werden nur solche Dinge und Zusammenhänge erläutert, die sich nicht sofort erschließen, bzw. solche Zusammenhänge, die Task-Grenzen überschreiten.

6.1 task_BackGround

```
g_Fake_TakeOff
Put signature into EEProm to facilitate further start operations
//  On CONDITION : Obtain g_ConfigData from SD_Card
Fill dir_list
```

6.2 task_ImageBuilder

6.3 task_IO_FrontEnd

6.4 task_AudioController

6.5 task_Buzzer

6.6 task_CAN_Bus_Senser

6.7 task_CAN_Bus_Receiver

6.8 task_MMI

7 Kommunikation mit dem Sensor