Implementierungsvorschriften AUTOSAR

17. Juni 2015

Kurzer Überblick

1 Coding-Rules

1.1 Runnables

Jeder Teil des Systems wird in Runnables aufgeteilt. Eine Runnable ist dabei nur eine Funktion, die vom Task, dem die Runnable zugeordnet ist, angesprungen wird(Im besten Fall per `#define RUNABLE (CODE)`). In die Runnable gehört NUR der reine Applikationscode. D.h. sobald eine Kommunikation mit einer anderen Komponente erfolgt, sind entsprechende **RTE**-Funktionen aufzurufen(Dokumentation: siehe AUTOSAR.pdf). Beispiel:

```
//Falsch:
TASK(Output) // definition siehe Komponentendiagramm
{
        myPrintln("three"); // mit wie in den Ubungen definierter
            Funktion myPrintln
        TerminateTask();
}
// Problem: Applikationscode nicht in Runnables aufgeteilt und direkt
    eine OS-Funktion aufgerufen

//RICHTIG:

void myRunnable(char *s)
{
        RTE_Write_OutputPort_out(s); // bei der Codegenerierung
            entscheiden wir dann, welcher Codeschnipsel fuer diese
            Funktion eingefuegt wird
}

TASK(Output)
{
        myRunnable("three");
        TerminateTask();
}
```

Funktionen um Runnables übersichtlicher zu gestalten folgen im Punkt 1.2. Runnables zeichnen sich außerdem aus durch:

- keinen Rückgabewert
- keine übergebenen Parameter
- keine `while(true)` Schleifen (??)

1.2 Funktionen

Es gibt zwei Arten von Funktionen, die aus der Applikation heraus aufgerufen werden können:

- RTE-Funktionen
- nicht RTE-Funktionen, diese sind nur lokal für eine Runnable nötig

1.2.1 RTE-Funktionen

Alle RTE-Funktionen sind im Dokument AUTOSAR.pdf definiert. Es sind nur diese Funktionsköpfe zu nutzen. Es darf keine individuellen Anpassungen geben. Sollte jemand eine Änderung an einer Funktion benötigen(für Applikationscode) ist erst ein *Issue* auf github zu schreiben. Eventuell wird dann eine andere RTE-Funktion zusätzlich verfasst oder eine Änderung vorgenommen. Wenn jemand RTE-Funktionen einfach so ändert, ist er für die Seiteneffekte verantwortlich!

1.2.2 nicht RTE-Funktionen

diese Funktionen zeichnen sich dadurch aus, das sie nichts mit der übrigen Applikation zu tun haben und nur der Übersichtlichkeit innerhalb einer Runnable dienen. Es gibt zwei Möglichkeiten damit umzugehen:

1.) lokal innerhalb der Runnable definieren. Beispiel:

```
void myRunnable()
{
        uint32_t returnbigger(uint32_t a, uint32_t b){
                if(a < b)
                        return b;
                else
                        return a;
        }

        ...

        bigger = returnbigger(a,b);
}
```

2.) global im speziellen file (Input filename), welches dann inkludiert werden muss(eindeutiger Funktionsname mit Beschreibung). Im besten Fall die Funktion durch `#define XX` definieren(Achtung: siehe 1.3). Beispiel:

```
//im File funktionen.h
#define returnbigger(a,b) (((a>b) ? a : b))

//im Runnable
void myRunnable()
{
        bigger = returnbigger(a,b);
}
```

1.3 Defines

Für die allermeisten Fälle in unserem Projekt reicht es Funktionen durch `#define` zu deklarieren(Außerdem wird dann die Codegenerierung tendentiel leichter). Punkte, die zu beachten sind:

- Bei der Deklaration auf die Konkrete Notation achten! siehe Notation
- keine terminierendes ; am Ende des `#define`! Wenn sich die Berechnung über mehrere Zeilen zieht oder einen komplett abgeschlossenen Codeblock darstellt, müssen natürlich die entsprechenden ; vorhanden sein. Unter Umständen ist dann auch ein abschließendes ; nötig. Dies ist deutlich kenntlich zu machen und ordentlich zu dokumentieren!
- Ein `#define` ist GLOBAL. Bitte bei der Namensgebung dann darauf achten, das dieses `#define` eindeutig ist. Wenn lokale Funktionen `#define`t werden, dann den Namen so lange und eindeutig wählen, dass zu 100% keine Seiteneffekte auftreten können!
- `#define`s bitte **IMMER** mehrzeilig deklarieren. Dadurch werden die Makros erheblich leichter zu lesen! Beispiel:

```
        //Falsch:
        #define foo(value) (myfunc(value);0;)
        //Richtig:
        #define foo(value)\
                myfunc(value);\
                0;
```

- Da das Makro eine reine textuelle Übersetzung ist, ist der 'Rückgabewert', auch wenn es eigentlich keinen gibt, der Wert der ersten Anweisung im Funktiosblock! Beispiel wenn man über ein Makro den Wert 0 'zurückgeben' will:

```
        //Falsch:
        #define mymak(value)\
                expression1(value);\
                0;
```

```
//Richtig:
#define mymak(value)\
        0;\
        expression1(value);

//im Code beim Aufruf des Codes
...
int error = mymak(2);
```

1.4 Kommentare/Doxygen

Alle Funktionen sind zu dokumentieren(auch Runnables). Es ist sich an die bekannte Doxygen zu halten. Beispiele können sich bereits bei den RTE-Funktionen genommen werden. Es sind mindestens folgende Attribute zu dokumentieren

- `\brief` Beschreibung der Funktion, was sie macht
- `@param` alle Parameter, für was sie gebraucht werden, ob sie verändert werden(Call-by-Reference) (optional, für jede Variable eigener Punkt)
- `@return` Hier ist global im doxygen-file ein spezieller Errorcode definiert, sodass hier nichts anderes angegeben werden muss außer `@Errorcode`
- `@version` Die neuer Version. Angefangen bei 1.0 und dann, je nach Veränderung innerhalb der Funktion, langsam(1.01) oder schneller(1.1) hochzählen. Gibt es eine grundlegende Änderung(weil z.B. die Funktion inklusive Übergabeparameter geändert werden muss) ist eine neue Version einzuführen(2.0). Nach der Versionsnummer ist ausserdem eine Zusammenfassung der Änderungen zu vermerken(als Changelog)
- `@author` Für jede neue Version die geschrieben wurde den Autor dieser Version angegeben
- `@date` Das Datum der letzten Änderung

Außerdem gibt es noch eine Menge weiterer Tags, nützlich unter anderem:

- `@todo` Aufgaben die noch durchgeführt werden müssen
- `@warning`
- `@attention`

Ausserdem sind alle diese Attribute noch pro file zu deklarieren(gilt nicht für files, in der sich nur eine Funktion befindet).
Um alle files zu erfassen, muss (am besten direkt nach der einleitenden `#ifndef ATTR # define ATTR`) noch ein zusätzliches `@file FILENAME` befinden!

Es braucht/muss/soll nicht jeder nach jeder Änderung die ganze Dokumentation neu erzeugen. Allerdings sollte Zeitnah eine neue Dokumentation eingecheckt werden, sodass auch die anderen immer einen leichten Zugriff auf die Funktionen haben. Download Doxygen.
Tipp: Die meisten guten Editoren(geany,atom) unterstützen(von Haus aus oder per plugin) Unterstützung für Doxygen.

1.5 Applikationscode

Der Applikationscode ist in 'atomare' Strukturen zu zerteilen und entsprechend in Dateien aufzuteilen. Ein Beispiel/Vorlage befindet sich unter `/Code/examples/Runnable`(TODO). Pro File hat nur eine Runnable vorhanden zu sein. Diese ist zu dokumentieren und NUR darf NUR durch die definierten *RTE-Funktionen* mit der Aussenwelt kommunizieren. Diese Funktionsaufrufe können den Kontrollfluss unterbrechen!

2 Ordner- und Filestruktur

Hier folgt eine kurze Beschreibung welche Inhalte in welchen Ordner/Files zu finden sind:

- `/Code`
  hier liegt der ganze Applikations/RTE/Legosar-Runtime Code drin.
  - `/src` src-files
  - `/include` alle header-files die irgendwann im Projekt wichtig sind
  - `/examples` Beispiel zu runnables, RTE-Funktionen etc.
- `/Modell` Alle Dateien zum Modell(EMF und als UML-Datei)
  - `EMF` Dateien für Eclipse EMF inklusiver Test/Edit/Editor
  - `runtime-EclipseApplication`
- `/generated` generierter Src-Code: Pro konfiguriertem Brick gibt es einen Ordner, indem alle src-files, oil-files und binary-files für diesen Brick drinliegen. Außerdem gibt es noch global in diesem Ordner einzelne Dateien, die benötigt werden(Shell-Skripte etc.).

3 Wichtige Konventionen/Vereinbarungen

-

4 Ablauf für Kommunikation

4.1 Server-Client

4.1.1 Ablauf Applikationsprogrammierer

Der Applikationprogrammierer ruft nur die entsprechende RTE Funktion auf.(Server-Client hat das Verhalten eines Funktionsaufruf). (als !DEFINE!)

4.1.2 Ablauf Generierung

Fall 1: 2 Ports liegen auf gleichem Brick nichts besonderes zu beachten

Fall 2: 2 Ports liegen auf unterschiedlichem Brick

1. Commservice aufrufen, der dann die Nachricht wie folgt verschickt:
   .1 Funktionsname als string
   .2 Terminierungszeichen ',', nach dem letzten Parameter kommen zwei ',,'
   .3 alle Parameter wieder getrennt durch Terminierungszeichen
   .4 der Rückgabewert der Funktion wird getrennt nicht als string als letzter Parameter übergeben
2. WaitEvent auf ein Event das gesetzt wird, wenn der Rückgabewert vom anderen Brick kommt
3. Der Rückgabewert wird gelesen und an die Addresse geschickt

Sicht anderer Brick:

1. In der Codegenerierung:
   .1 Task erstellen, Taskname: T_Funktionsname, der gleich nach Einstieg auf Event wartet(WaitEvent)
   .2 Funktionsaufruf generieren mit Funktionsname(wobei Funktion an sich, eigener Teil des Applikationscodes ist)
2. im Commservice:
   .1 empfängt Nachricht
   .2 über ID zuordnung das jetzt Funktion aufgerufen werden muss
   .3 Name extrahieren aus string und in var speichern
   .4 Alle Parameter extrahieren und in Array speichern, dessen größe während der Codegenerierung festgelegt wurde(abhängig von anzahl an Parametern)
   .5 Event setzten das in 11 gesetzt wurde