Softwareorganisation mit BiDiBOne

oder: Wie hänge ich mein AddOn in BiDiB ein?

Die Basissoftware des BiDiBOne enthält neben den Funktionen zur Kommunikation mit dem BiDi-Bus auch das Tasksystem Cortos und einige andere Grundfunktionen sowie die Debug-Schnittstelle.

Darauf sollen die verschiedenen AddOns mit möglichst einfachen Mitteln aufbauen können.

- Motto: Klare Trennung zwischen Basis- und AddOn-Software.

Solution und Projekte

Für eine bessere Übersicht und Wartung sind die Quellen für Basis und AddOn(s) in verschiedenen Projekten unter einer Solution organisiert.

Es gibt eine Solution: **BiDiBOne** mit dem Projekt **Basis** und weiteren AddOn-Projekten (Details siehe unten).

Das Zusammenspiel der einzelnen Projekte wird in den Properties der Solution geregelt.

Dazu ist im Solution-Explorer die Solution-Zeile anzuwählen und im Kontextmenü (z.B. Rechtsklick) der Menüpunkt *Properties* zu wählen.

Im Atmel Studio 6 hat man die Auswahl zwischen einem einzelnen oder mehreren Startup Projekten. In jedem Falle kann man aber beim BiDiBOne nur einen Startpunkt wählen. Das sollte das AddOn-Projekt sein:

Die Abhängigkeiten unter den einzelnen Projekten sind im Absatz *Project Dependencies* einzustellen (dazu jedes Projekt einzeln in der Drop-Down-Box aufrufen):

Die Abhängigkeiten regeln die Reihenfolge, in der die einzelnen Projekte gebaut werden. Die zeigt Atmel Studio 6 u.A. im Punkt *Startup Projects*.

Basis-Projekt

Die Basis enthält alle zum Betrieb notwendigen Funktionen. Zusätzlich werden viel gebrauchte Hilfsfunktionen zur Verfügung gestellt.

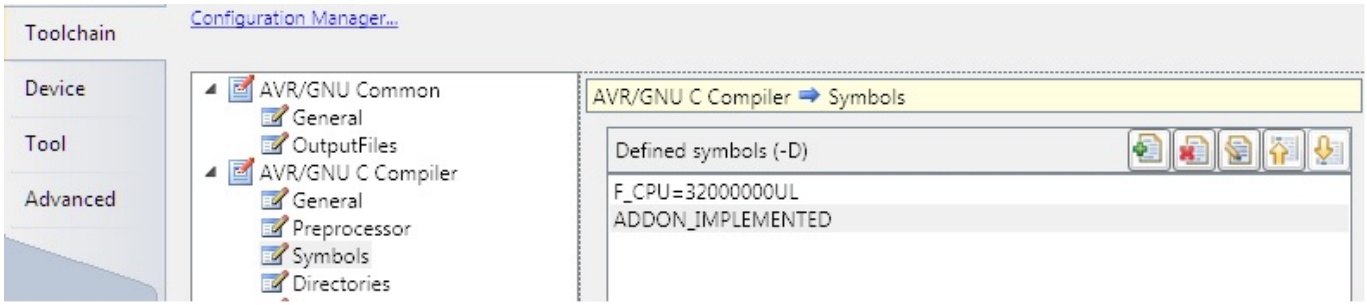
Die Quellen im Basis-Projekt sollten in keinem Falle direkt geändert werden. Sollten dennoch Anpassungen notwendig sein, die nicht allgemeingültig sind das unter AddOn-Basisersatz-Projekt beschriebene Verfahren zu verwenden.

Dieses Vorgehen dient der klaren Trennung und leichteren Wartbarkeit der Software.

AddOn-Projekt

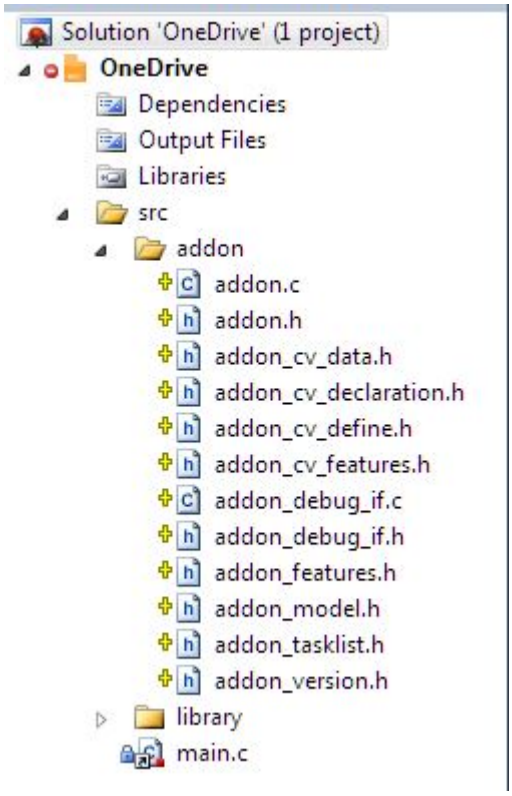
Das AddOn-Projekt enthält die eigentliche Funktionalität für die Zusatzhardware.

Um die vordefinierten Funktionen dem Basis-Projekt zugänglich zu machen, ist in den *Properties* des AddOn-Projektes im Kapitel: *ToolChain - AVR/GNU C Compiler* bei *Symbols* das Symbol **ADDON_IMPLEMENTED** einzutragen:



Das AddOn-Projekt liegt parallel zum Basis-Projekt. Es enthält unterhalb des Verzeichnisses `src` mindestens die Verzeichnisse `addon` und `core`. Soll die main-Funktion angepasst werden, muss sie direkt in das `src`-Verzeichnis kopiert werden.

Das AddOn-Projekt im Solution-Explorer am Beispiel OneControl:



Im `addon`-Verzeichnis befinden sich die Quellen, die die Anbindung an das Basis-System ermöglichen.

Sie werden direkt durch „**Add**“ eingebunden. Details finden sich weiter unten.

Im `core`-Verzeichnis befinden sich alle notwendigen Dateien aus dem Basis-Projekt. Sie werden als „**Add As Link**“ eingebunden und werden so nicht physisch kopiert sondern verbleiben ausschließlich im Basis-Projekt. Die Quellen werden mit einem Verknüpfungssymbol angezeigt.

Alle für das AddOn-Projekt notwendigen Verzeichnisse und Quellen können jetzt nach Bedarf ergänzt werden. Es empfiehlt sich aber aus Gründen der klaren Abgrenzung eigene Quellen in Verzeichnisse unterhalb des `addon`-Verzeichnisses zu legen.

AddOn-Basisersatz-Projekt

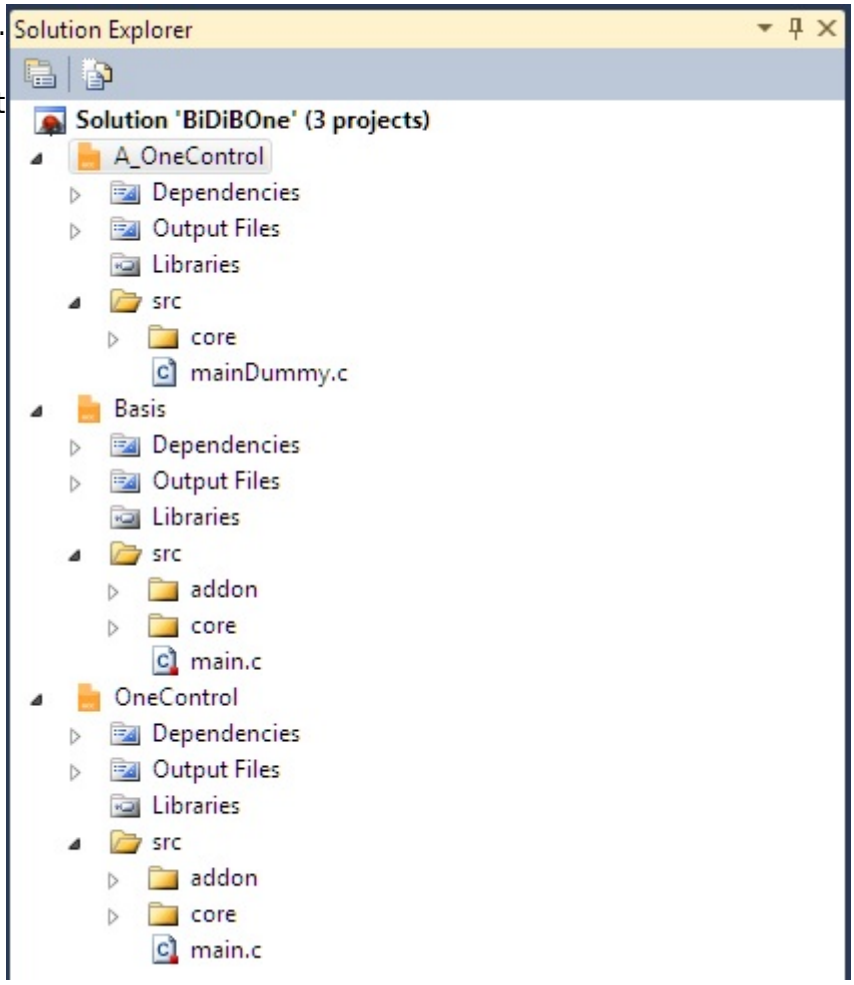
Es kann Fälle geben, in denen man Quellen aus dem Basis-Projekt zeitweilig oder auch dauerhaft für sein AddOn anpassen muss. Und die Anpassungen sollen nicht allgemeingültig sein.

Diese Maßnahme sollte aber die Ausnahme sein, da die in den eigenen AddOns angepassten Quellen nicht bei einem Update des Basis-Projektes nachgeführt werden!

Die anzupassenden Quellen müssen dann in ein eigenes Projekt kopiert werden. Dort können sie angepasst werden ohne Einfluß auf das Basis-Projekt selber oder gar andere AddOns zu nehmen. Hierbei sind die im Folgenden beschriebenen Besonderheiten des Atmel Studio 6 zu beachten.

Das Atmel Studio erstellt aus den Quellen für jedes Projekt einer Solution ein **makefile**. Quellen, die in anderen Projekten liegen und über eine Verknüpfung referenziert werden, bindet Atmel Studio ebenso ein. Das entsprechende **makefile** wird aber abhängig von der **alphabetischer Reihenfolge**

der internen Projektnamen aufgebaut. Da unsere angepasste Quellen anstatt der eigentlichen aus dem Basis-Projekt eingebunden werden sollen, müssen sie in einem Projekt mit einem günstigen Namen angelegt werden; hier empfiehlt sich der Name A_<*AddOn-Projektname*>.



Die Verknüpfung („Add As Link“) im AddOn-Projekt bezieht sich jetzt auf dieses Hilfsprojekt anstatt auf das Basis-Projekt.

Das funktioniert sicher mit C-Quellen. Bei Header-Dateien kann man auf die speziellen Header hinweisen. Allerdings beziehen sich Quellen aus dem Basisverzeichnis immer noch auf ihre Header dort. Bei ungeschickter Konstellation kann das aber zu Inkonsistenzen führen.

Zum korrekten Bauen muss das Hilfsprojekt eine Datei mit einer main-Funktion haben (im Beispiel: mainDummy.c).

(Dieses Verhalten von Atmel Studio lässt sich im *makefile* überprüfen.)

AddOn-Hooks

Die Basissoftware enthält definierte „Hooks“, um die Software der AddOns einzubinden.

Baugruppenkennzeichnung

Für die eindeutige Erkennung am BiDi-Bus sind die folgenden Angaben (in `addon.h`) zu machen:

```
//========================================================================
// Device-ID
//========================================================================
#define   BIDIB_VENDOR_ID    13   // vendor-ID
#define   BIDIB_PRODUCT_ID  116   // product ID, see http://www.opendcc.de/elektronik/bidib/opendcc_bidib.html
#define   CLASS_ACCESSORY     1   // 0: no accessory messages, 1: accessory messages included
#define   CLASS_OCCUPANCY     0   // 0: no occupancy messages, 1: occupancy messages included
#define   CLASS_SWITCH        1   // 0: no switch messages,    1: switch messages included
```

Hier am Beispiel einer **Eigenbaugruppe** (`BIDIB_VENDOR_ID=13`) und der Produkt-ID 116.

Die Produkt-IDs werden an der angegebenen Adresse (http://www.opendcc.de/elektronik/bidib/opendcc_bidib.html) vergeben.

Initialisierung und Shutdown

Zur Initialisierung ruft das Basissystem Funktionen des AddOns an drei Stellen im Modul `addon.c` auf:

- Nach Abschluss der Basisinitialisierung und **vor** Freigabe der **Interrupts** die Funktion **`addon_init()`**
- Nach Abschluss der gesamten Initialisierung sowie **nach** Freigabe der **Interrupts** und innerhalb einer Task die Funktion **`addon_power_up()`**
- Beim Herunterfahren die Funktion **`addon_close()`**

Die Funktionen sind din der Quelle `addon.c` vorbereitet.

```
//------------------------------------------------------------------------
// Functions:
//------------------------------------------------------------------------

/*
 * Central initialization for AddOn application.
 * Call all your own initialization functions.
 */
void init_addon(void)
{
  init_addon_hardware();  // initializes all devices
  init_power_output();    // initializes power output tasks

  // TODO: continue initialization ...
}

/**
 * Central close function.
 */
void close_addon(void)
{
  close_addon_hardware();
  close_power_output();

  // TODO: continue close procedure ...
}

/*! \brief Initializes all task based parts. Called by task scheduler.
 *
 * To finish the task return -1 .
 *
 */
t_cr_task power_up_addon(void)
{
  return power_up_addon_hardware();
}
```

Von dort aus können alle weiteren Initialsierungen und „Shutdowns“ aufgerufen werden.

Hier am Beispiel OneControl in einer frühen Version.

Besondere Beachtung ist der `power_up_addon`-Funktion zu zollen. Sie wird zu Beginn der Taskverwaltung aufgerufen und ist wie eine „normale“ Task zu programmieren. Beendet wird sie erst, wenn die Anwendung `-1` zurückliefert.

Während die Power-Up-Tasks noch laufen, startet das Tasksystem alle angemeldeten Tasks, die als `ready` gekennzeichnet wurden! Um Initialisierungskonflikte zu vermeiden, dürfen die vom AddOn abhängigen Taks erst zum Schluss der Power-Up-Task freigegeben werden!

Taskverwaltung

Zur Einbindung in die Taskverwaltung `cortos` sind in der Headerdatei: **addon_tasklist.h** die folgenden Einträge erforderlich:

- **Prototypdefinition `ADDON_PROTOTYPES`**

- **Task-ID ADDON_TASKENUMS**
- **Taskdefinition ADDON_TASKLIST**
- **FIFO-Definitionen ADDON_FIFOS**

Alle Teile werden an den betreffenden Stellen in der Taskverwaltung eingebunden.

```
#ifndef ADDON_TASKLIST_H_
#define ADDON_TASKLIST_H_


//------------------------------------------------------------------------------
//
// Add the list of _AddOn_ task (higher index = higher priority)
// Every task in the system must have a number.
//
// A task must return as soon as possible (cooperative multitasking), the return
// value determines the time gap until it is called again:
// return(-1) means: 'no longer ready'.
// return(5) means: 'ready again in 5 systick
//------------------------------------------------------------------------------
#define ADDON_PROTOTYPES \
  t_cr_task run_power_output(void);   \
  t_cr_task run_power_feedback(void); \
  t_cr_task run_variable_io(void); \


#define ADDON_TASKENUMS \
  TASK_ADDON_POWER_OUTPUT,      /* power output message from BiDiB */ \
  TASK_ADDON_POWER_FEEDBACK,    /* power output feedback handling */ \
  TASK_ADDON_VARIABLE_IO,       /* variable i/o handling */ \
  //TODO add AddOn task id defines ...

#define ADDON_TASKLIST \
  /* id                              ready, wakeup, call */ \
  [TASK_ADDON_POWER_OUTPUT]   = {   1,  0,    run_power_output}, \
  [TASK_ADDON_POWER_FEEDBACK] = {   1,  0,    run_power_feedback}, \
  [TASK_ADDON_VARIABLE_IO]    = {   1,  0,    run_variable_io}, \
  //TODO add AddOn task defines ...


#endif /* ADDON_TASKLIST_H_ */
```

Hier am Beispiel OneControl in einer frühen Entwicklungsphase. (Die FIFO-Definitionen schließen sich entsprechend an.)

Anmerkung: Es ist zu überlegen, ob das Basis-System verschiedene Prioritätsgruppen unterstützen sollte. Dann würden die einzelnen Blöcke entsprechend in die Taskliste der Basis eingefügt.

Accessory-Behandlung

Nach Erhalt einer **MSG_ACCESSORY_SET**-Nachricht vom BiDiBus wird die Funktion `bidib_msg_accessory_set_addon` im Modul `addon.c` aufgerufen.

Die Argumente sind:

- `unsigned char acc_num` = Nummer des Accessorys, beginnend mit 0, maximal NUM_OF_ACCESSORY-1
- `unsigned char aspect` = Nummer des Aspekts, beginnend mit 0, maximal NUM_OF_ASPECTS_PER_ACCESSORY-1

Da diese Aktionen während des Betriebs innerhalb einer Task laufen, gilt auch hier, dass bei komplexeren Funktionen die folgende Bearbeitung lediglich initiiert werden sollte. Die eigentliche Funktionalität sollte in einer entsprechenden Task durchgeführt werden.

... wird fortgesetzt ...

Makroausführung

Nach Erkennen eines Accessory-Set-Befehls wird aus der Makro-Task heraus die Funktion `performAddOnMacro` im Modul `addon.c` aufgerufen:

```
/**
 * \brief Performs macros of AddOns, starts macro to run and returns if performed or not.
 *
 * We need accessory number for emergency call to BiDiB after failure and for the
 * concluding remark after finishing the macro.
 *
 * \param lstate    pointer to lstate of current macro
 * \param lvalue    lvalue of current macro
 * \param accessory number of accessory, 255 if none
 *
 * \return state of execution
 * \retval 1      macro performed
 * \retval 0      no macro performed
 */
uint8_t performAddOnMacro(t_lstate *lstate, uint8_t lvalue, uint8_t accessory)
{
  switch(lstate->type)
    {
      case BIDIB_OUTTYPE_SPORT:               // standard port
          start_sequence_turnout(lstate, lvalue, accessory);
          return 1;                           // macro performed
      case BIDIB_OUTTYPE_LPORT:
          break;
      case BIDIB_OUTTYPE_SERVO:
          break;
      case BIDIB_OUTTYPE_SOUND:
          break;
      case BIDIB_OUTTYPE_MOTOR:
          break;
      case BIDIB_OUTTYPE_ANALOG:
          break;
      default:
          break;
    }
    return 0;                                 // macro not performed
}
```

```
typedef union
  {
    uint8_t byte;
    struct
      {
        uint8_t  cmd:4;
        uint8_t  type:4;
      };
  } t_lstate;
```

Hier am Beispiel der Standardmakro-Implementierung. Die Argumente sind:

- lstate ⇒ Schaltbefehl
- lvalue ⇒ Wert des Makropunktes

Die Unterscheidung des Accessory-Typs kann am Beispiel abgelesen werden.

Da diese Aktionen während des Betriebs innerhalb einer Task laufen, gilt auch hier, dass bei komplexeren Funktionen die folgende Bearbeitung lediglich initiiert werden sollte. Die eigentliche Funktionalität sollte in einer entsprechenden Task durchgeführt werden.

Debug-Schnittstelle

Auch das `debug_if`-Modul enthält „Hooks“ zur Unterstützung der AddOns.

- **Aufruftabelle `ASCII_PARSE_TAB_ADDON`**
- **Überschrift `PA_info_addon()`**
- **Zusammenfassung `PA_help_addon()`**

Diese Teile werden ebenso an den entsprechenden Stellen im Basis-Projekt eingebunden.

Die Aufruftabelle muss in der vorbereiteten Headerdatei: **`addon_debug_if.h`** definiert werden.

Hier am Beispiel OneControl mit DMX-Vorgabe:

Zusätzlich müssen hier auch die Prototypen für die eigentlichen Debug-Funktionen definiert werden.

Die Texte für Überschrift und Zusammenfassung werden in der vorbereiteten Quelle **`addon_debug_if.c`** formuliert:

Anschließend werden die eigentlichen Debug-Funktionen programmiert.

Zusammenfassung

Vorgehen

1. AddOn-Projekt mit vorgegebener Verzeichnisstruktur erstellen
2. Projektattribute anpassen (i.e. ADDON_IMPLEMENTED)
3. In der Solution Projektreihenfolge und -abhängigkeiten festlegen
4. Benötigte Basis-Projekt-Dateien über „Add As Link“ in das `core`-Verzeichnis einbinden
5. In `addon.h` Versionsinformationen definieren
6. In `addon.c/h` Initialisierung und „Shutdown“ veranlassen
7. In `addon_tasklist.h` Tasks definieren
8. In `addon.c` Makrofunktionen initiieren
9. In `addon_debug.c/h` Debugfunktionen einbauen

Hooks

Definitionen

Baugruppenkennzeichnung

- **BIDIB_VENDOR_ID** vendor ID = 13
- **BIDIB_PRODUCT_ID** product ID see http://www.opendcc.de/elektronik/bidib/opendcc_bidib.html
- **CLASS_ACCESSORY** occurence of accessory messages (0/1=no/yes)
- **CLASS_OCCUPANCY** occurence of occupancy messages (0/1=no/yes)
- **CLASS_SWITCH** occurence of switch messages (0/1=no/yes)

Taskverwaltung cortos

- **ADDON_PROTOTYPES** Prototypdefinitionen
- **ADDON_TASKENUMS** Task-IDs
- **ADDON_TASKLIST** Taskdefinitionen
- **ADDON_FIFOS** FIFO-Definitionen

Funktionen

Start und Stopp

- **addon_init()** Funktionsaufruf zur Initialisierung **vor** Freigabe de Interrupts
- **addon_power_up()** ask zur weiteren Initialsierung **nach** Freigabe der Interrupts
- **addon_close()** Funktion zum Schließen des AddOns insbesondere der Interrupts

Betrieb

- **performAddOnMacro()** Ausführen der Makros eines AddoOns

Debug-Schnittstelle

- **ASCII_PARSE_TAB_ADDON** Aufruftabelle der einzelenen Debug-Befehle
- **PA_info_addon()** Überschrift
- **PA_help_addon()** Zusammenfassung