AddOn-Software einbinden in BiDiBOne

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.

Konflikte

Da es durchaus denkbar und sogar erstrebenswert ist, mehrere AddOn-Komponenten gemeinsam zu verwenden, wird eine Konvention eingeführt, die Konflikte zwischen den einzelnen Teilen verhindern soll.

Verwendete Portpins und Timer werden in der entsprechenden Header-Datei wie folgt bekanntgegeben:

```
#ifndef  I_NEED_PORTA0
  #define I_NEED_PORTA0
#else
  #error Port conflict A0
#endif
```

(Hier am Beispiel Port A, Pin 0)

Ebenso muss die Verwendung der schnellen Semaphoren bekannt gemacht werden:

```
#ifndef  I_NEED_SEMAPHORE_C6
  #define I_NEED_SEMAPHORE_C6
#else
  #error Semaphore conflict C6
#endif
```

(Hier am Beispiel Semaphore Typ C, Bit 6)

Da es sich „nur“ um eine Konvention handelt, ist hier die Disziplin des Einzelnen gefordert!

Sollte in der Basis ein neues Flag verwendet werden, führen wir eine entsprechende Deklaration ein. Eine Doppelbenutzung durch ein AddOn, das der Konvention folgt, deckt der Compiler dann schnell auf.

AddOn-Hooks

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

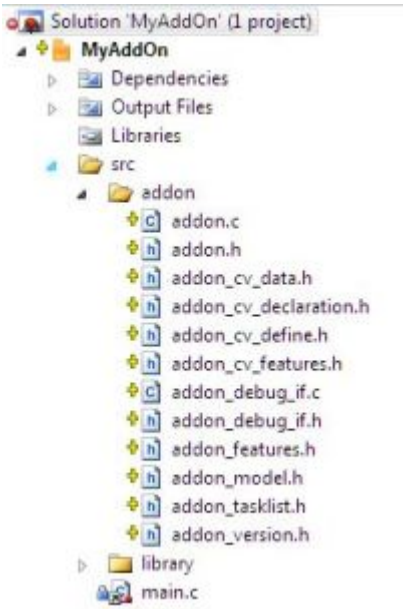
Im Repository der Gruppe BiDiB im Projekt **AddOnStub** steht ein Muster mit allen notwendigen Dateien zur Anbindung eines eigenen AddOns an den BiDiBus zur Verfügung.

Die entsprechenden Quellen sind im Unterverzeichnis **src/addon** zu finden.

Zusätzlich liegt im Unterordner **conf** (configuration) das Muster einer Beschreibungsdatei für die Tools BiDiB-Monitor und BiDiB-Wizard vor.

Der Unterordner **env** (environment) ist für Hilfsmittel zur Entwicklung vorgesehen und enthält u.A. eine Projektdatei für das AtmelStudio 6.

Tipp: Kopiert man diese Projektdatei in den MyAddOn-Ordner, kann man durch Doppelklicken das AtmelStudio starten. Das sieht dann etwa so aus:



(Im Bild zu sehen der **Link** auf das main-Modul der BiDiBone-Basis-FW. Die anderen Links auf Module in der Basis verstecken sich im Ordner: library.)

Baugruppen-Modell

Das Modell einer Baugruppe wird in der Headerdatei `addon_model.h` definiert.

Für die eindeutige Erkennung am BiDi-Bus sind die folgenden Angaben für die Geräte-ID 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.

Mit den allgemeinen Definitionen können bestimmte Zusatzfunktionen aus der Basis-Firmware aktiviert und das Standardverhalten eingestellt werden:

```
//========================================================================
// General defines (CHECKED!)
//========================================================================
#define SERVO_ENABLED                FALSE    // TRUE: standard Servo Operation (like LightControl)
#define SERVO_OVER_CURRENT_ENABLED   FALSE    // TRUE: check over current of servos

#define BLINK_MORSE_ENABLED          FALSE    // FALSE|TRUE: blinking Morse code no|yes

#define NUM_OF_INPUTS                  0      // our Input channels
#define NUM_OF_POWER_OUTPUTS           0      // number of power outputs
#define NUM_OF_SPORTS                  0      // our Switch channels
#define NUM_OF_LIGHTS                  0      // our LED/DMX channels

#define NUM_OF_SERVOS                  0      // our Servo channels
#define NUM_of_GBM16T                  0      // Number of GBM16T modules

#define NUM_OF_MACROS                 32      // Standard set
#define NUM_OF_SAVED_MACROS           32
#define NUM_OF_ITEMS_PER_MACRO        32

#define NUM_OF_ACCESSORY              20      // Standard set
#define NUM_OF_ASPECTS_PER_ACCESSORY   8

#define NUM_OF_FEATURES               30      // Standard: so many features possible in EEPROM
```

In der Basis-Firmware wird geprüft, ob diese notwendigen Definitionen gemacht wurden und gfls. ein Compilerfehler angezeigt.

Versionsdaten

Die Versionsinformationen werden über den BiDiBus an den Host übertragen. Außerdem fließen sie in den Namen der Firmware-Dateien ein. Fehlen sie, wird ein Compiler-Fehler erzeugt.

Die Versionsdaten für das AddOn werden in der Header-Datei: `addon_version.h` definiert.

```
//===========================================================================
//
// purpose:   central version file - to be handled with gawk
//
//===========================================================================
// this file defines the following;
// these strings are not allowed to show up again in this file!
// file is parsed with awk or gawk!

#define MAIN_VERSION                0
#define SUB_VERSION                 01       // attention: octal reading by C
#define COMPILE_RUN                 01

#define BUILD_YEAR                  13
#define BUILD_MONTH                 6
#define BUILD_DAY                   22
```

Hier am Beispiel der originalen `version.h`.

Durch die feste Struktur lassen sich diese Versionsdaten für alle BiDiBOne-AddOns gleichermaßen z.B. mit einem Skript lesen und die Firmware-Datei eindeutig kennzeichnen.

Baugruppen-Fähigkeiten (Features)

„... Über features teilt die Hardware dem PC-Programm ihre Fähigkeiten mit (also z.B. 'ich kann Lokadresse UND Richtung erkennen'). Das kann eine Eigenschaft der Hardware sein (wer kann, der kann), das kann aber auch explizit abgewählt werden - weil eben z.B. eine Dreileiteranlage angeschlossen ist, wo die Richtungsinformation des Belegtmelders wertlos ist. Features sind in der Norm auf bidib.org erläutert...“ (siehe BiDiB, ein universelles Steuerprotokoll für Modellbahnen)

Die Features werden in `addon_features.h` definiert und müssen den Fähigkeiten der Baugruppe angepasst werden.

```
#define FEATURE_CTRL_INPUT_COUNT_idx    0
...
#define FEATURE_ACCESSORY_NEU_idx       7
...
#define FEATURE_STRING_SIZE_idx         16
#define FEATURE_table_size              17

t_feature feature[] =
{ // index                          num (see bidib_messages.h)   value         min     max     notify_function (if changed)
  [FEATURE_CTRL_INPUT_COUNT_idx] = { FEATURE_CTRL_INPUT_COUNT, NUM_OF_INPUTS,    0,     NUM_OF_INPUTS, NULL},  //  0
...
  [FEATURE_ACCESSORY_NEU_idx]    = { FEATURE_ACCESSORY_NEU,    3,               0,     10,  NULL},            //  7
...
  [FEATURE_STRING_SIZE_idx]      = { FEATURE_STRING_SIZE, BIDIB_STRING_MAX, BIDIB_STRING_MAX, BIDIB_STRING_MAX, NULL}, // 16
};
```

Dort werden sinnige Indices für die in `bidib_message.h` vorgegebenen Features definiert. Die Tabelle feature[] enthält den aktuellen Wert des Features und seine Grenzwerte (min/max). Für eine weitere Verarbeitung kann eine `notify_function` angegeben werden. (Im Beispiel gibt es keine.)

Im Gegensatz zu CVs sind Features eineindeutige Standardobjekte und damit genehmigungspflichtig. D.h. eine CV kann man definieren, wie man lustig ist, ein Feature nicht. Das ist der Grund, weshalb nur die in bidib_messages.h abgesprochenen und standardisierten Werte erlaubt sind.

Zum Lesen der Features stehen die zwei Funktionen aus der Header-Datei `features.h` zur Verfügung:

```
/**
 * \brief Returns the table index of the feature with the given number.
 *
 *  We check, if the feature is member of the feature table. If we find
 *  it, we return the table index otherwise we return -1.
 *
 *  You can use the result of this function as argument for function
 *  get_feature_index_value to be sure that the index is valid!
 *
 * \param fnum      number of feature
 *
 * \return index of table or -1
 * \retval >=0      index of table for feature
 * \retval -1       requested feature is invalid or unknown
 */
char get_feature_index(unsigned char fnum);
```

Mit Hilfe dieser Funktion erhält man über die in `bidib_messages.h` vorgegebenen Features den Index auf die interne Tabelle. Ist das Feature ungültig bzw. steht es nicht in der tabelle, wird -1 geliefert, andererseits der Index.

```
/**
 * \brief Returns the value of the internal feature table with the
 *        given index.
 *
 *  If index is out of bound, we return 0.
 *  You can use this function with the result of the function
 *  get_feature_index for performed access.
 *
 * \param index     index of internal feature table
 *
 * \return value of feature
 * \retval >0       value of feature
 * \retval 0        value 0 of feature or invalid value!
 */
unsigned char get_feature_index_value(unsigned char index);
```

Mit dem oben erhaltenen Index wird mit dieser Funktion der Wert des Features gelesen. Diese Funktion sollte nur aufgerufen werden, wenn der oben erhaltene Index größer oder gleich 0 ist.

Wenn ein Host-System eine Nachricht vom Typ: `MSG_LC_CONFIG_SET` abschickt, wird die Funktion `config_sport_addon` aufgerufen:

```
/**
 * \brief Sets configuration of the given SPORT with the given structure.
 *        The function is called after a MSG_LC_CONFIG_SET message from BiDiB.
 *
 * \param t_bidib_sport_cfg   configuration to set
 *
 * \return uint8_t  answer
 * \retval          0: out of range/no answer possible
 * \retval          1: struct is set
 */
unsigned char config_sport_addon(t_bidib_sport_cfg*);
```

Soll die Anfrage unterstützt werden, sind die entsprechenden Informationen des SPORT-Accessorys mit den übergebenen Informationen zu setzen und 1 zu liefern. Anderenfalls reicht die Rückgabe einer `0`.

Analog wird verfahren, wenn ein Host-System mit der Nachricht: `MSG_LC_CONFIG_GET` die SPORT-Eigenschaften anfordert, dann wird die Funktion `addon_sport_query` aufgerufen:

```
/**
 * \brief Sets configuration of the given SPORT with the given structure.
 *        The function is called after a MSG_LC_CONFIG_SET message from BiDiB.
 *
 * \param t_bidib_sport_cfg   configuration to set
 *
 * \return uint8_t  answer
 * \retval          0: out of range/no answer possible
 * \retval          1: struct is set
 */
unsigned char config_sport_addon(t_bidib_sport_cfg*);
```

Soll die Anfrage unterstützt werden, sind die angeforderten Angaben mit den entsprechenden Informationen des SPORT-Accessorys auszufüllen und 1 zu liefern. Anderenfalls reicht die Rückgabe einer `0`.

EEPROM (CV-Daten)

Die so genannten CV-Daten werden im EEPROM des Bausteins abgelegt. Für die Organisation im AddOn sind vier Dateien zuständig:

- **addon_cv_define.h** ⇒ definiert die Variablen
- **addon_cv_declaration.h** ⇒ deklariert die Variablen für die notwendigen Strukturen
- **addon_cv_data.h** ⇒ legt den Inhalt im EEPROM fest
- **addon_cv_features.h** ⇒ legt die Features im EEPROM fest

Beispiele aus der OneControl:

addon_cv_define.h (Variablendefinition):

```
#ifndef ADDON_CV_DEFINE_H_
#define ADDON_CV_DEFINE_H_


//!< \brief Bit defines for IO-Pin L9822E:
#define CVbit_PwrConf_Feedback        0   //!< \brief feedback: 0=false, 1=true
#define CVbit_PwrConf_TurnOffSwitch   1   //!< \brief turn off switch: 0=false, 1=true
#define CVbit_PwrConf_PulseMode       2   //!< \brief pulse mode: 0=false, 1=true

//!< \brief Structure for Power output L9822E:
typedef struct
{
  unsigned char pwrConf;                  //!< \brief mode see Bit defines for IO-pin L9822E
  unsigned char pwrTicks;                 //!< \brief 0, 1...255 * 20 ms ticks/pulse time ticks
  unsigned char reservePwr1;              //!< \brief reserve1
} t_l9822e_cv;

//!< \brief Bit defines for IO-Pin MCP23S08:
#define CVbit_McpConf_Direction       0   //!< \brief direction of pin: 1=input, 0=output
#define CVbit_McpConf_Polarity        1   //!< \brief active level of pin: 1=LOW-, 0=HIGH-active
#define CVbit_McpConf_PulseMode       2   //!< \brief pulse mode: 0=false, 1=true

//!< \brief Structure for IO-expander MCP23S08:
typedef struct
{
  unsigned char mcpConf;                  //!< \brief control bits see: Bit defines for IO-Pin
  unsigned char mcpTicks;                 //!< \brief 0, 1...255 * 20 ms pulse time ticks
  unsigned char reserveMcp1;              //!< \brief reserve1
} t_mcp23s08_cv;


#endif /* ADDON_CV_DEFINE_H_ */
```

addon_cv_declaration.h (Variablendeklaration):

```
#ifndef ADDON_CV_DECLARATION_H_
#define ADDON_CV_DECLARATION_H_


t_l9822e_cv   lc9822e_cv[NUM_OF_POWER_OUTPUTS];
t_mcp23s08_cv mcp23s08_cv[NUM_OF_SPORTS - NUM_OF_POWER_OUTPUTS];


#endif /* ADDON_CV_DECLARATION_H_ */
```

addon_cv_data.h (CV-Daten im EEPROM):

```
#ifndef ADDON_CV_DATA_H_
#define ADDON_CV_DATA_H_

// Content          Name       CV  -alt  type    comment
  {
    { // ------------------- Power Output 0 -------------------
      //!< \brief mode bits see: Bit defines for IO-Pin:
      ( 1 << CVbit_PwrConf_Feedback      ) |          //!< \brief feedback: 0=false, 1=true
      ( 1 << CVbit_PwrConf_TurnOffSwitch ) |          //!< \brief turn off switch: 0=false, 1=true
      ( 0 << CVbit_PwrConf_PulseMode     ),           //!< \brief pulse mode: 0=false, 1=true
      0x0A,                                           //!< \brief 0, 1...255 * 20 ms ticks
      0x00,                                           //!< \brief reserve1
    },


    { // ------------------- IO-Expander Pin 15 -------------------
      //!< \brief control bits see: Bit defines for IO-Pin:
      ( 1 << CVbit_McpConf_Direction   ) |            //!< \brief direction of pin   : 1=input, 0=output
      ( 1 << CVbit_McpConf_Polarity    ) |            //!< \brief polarity of pin: 1=LOW-, 0=HIGH-active
      ( 0 << CVbit_McpConf_PulseMode   ),             //!< \brief pulse mode: 0=false, 1=true
      0x00,                                           //!< \brief 0, 1...255 * 20 ms ticks
      0x00,                                           //!< \brief reserve1
    },
  },
```

addon_cv_features.h (Features im EEPROM):

```
#ifndef ADDON_CV_FEATURES_H_
#define ADDON_CV_FEATURES_H_


  // Feature data
  NUM_OF_INPUTS,                // [FEATURE_CTRL_INPUT_COUNT_idx]
  1,                            // [FEATURE_CTRL_INPUT_NOTIFY_idx]
  NUM_OF_SPORTS,                // [FEATURE_CTRL_SPORT_COUNT_idx]
  NUM_OF_SERVOS,                // [FEATURE_CTRL_SERVO_COUNT_idx]
  2,                            // [FEATURE_CTRL_MAC_LEVEL_idx]
  NUM_OF_SAVED_MACROS,          // [FEATURE_CTRL_MAC_SAVE_idx]
  NUM_OF_MACROS,                // [FEATURE_CTRL_MAC_COUNT_idx]
  NUM_OF_ITEMS_PER_MACRO,       // [FEATURE_CTRL_MAC_SIZE_idx]
  0,                            // [FEATURE_CTRL_MAC_START_MAN_idx]
  1,                            // [FEATURE_CTRL_MAC_START_DCC_idx]
  NUM_OF_ACCESSORY,             // [FEATURE_ACCESSORY_COUNT_idx]
  1,                            // [FEATURE_ACCESSORY_SURVEILLED_idx]
  NUM_OF_ASPECTS_PER_ACCESSORY, // [FEATURE_ACCESSORY_MACROMAPPED_idx]
  1,                            // [FEATURE_FW_UPDATE_MODE_idx]

#endif /* ADDON_CV_FEATURES_H_ */
```

Initialisierung und Shutdown

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

- Nach Abschluss der Basisinitialisierung und **vor** Freigabe der **Interrupts** die Funktion **`init_addon()`**
- Beim Herunterfahren die Funktion **`close_addon()`**
- Nach Abschluss der gesamten Initialisierung sowie **nach** Freigabe der **Interrupts** und innerhalb einer Task die Funktion **`power_up_addon()`**
- Während der Power-Up-Phase wird der Zustand der Initialisierung in der Funktion

init_finished_addon abgefragt. Die Initialisierung wird genau dann fortgesetzt, wenn hier 0 geliefert wird!

- Jedesmal ~~nach Initialisierung der BiDiB-Komponente, also sowohl nach einem Neustart als auch~~ nach dem Befehl, den Knoten zurückzusetzen, wird die Funktion **restart_addon()** aufgerufen.*(seit V.00.06)*

Die Funktionen sind in 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!

```
/**
 * \brief Answers if initialization of AddOn is finished.
 *
 * \return state of initialization
 * \retval 1      initialization finished
 * \retval 0      initialization still running
 */
uint8_t init_finished_addon()
{
  return init_finished_turnout_manager();
}
```

Zur Synchronsierung mit dem Basissystem wird die Funktion `init_finished_addon` aufgerufen. Das Verfahren ist notwendig, wenn ein AddOn erst im weiteren Programmablauf (z.B. während einer eigenen länger dauernden Task) die Initialisierung fertig stellen kann.

In manchen Fällen ist ein Neustart der Anwendung notwendig, wenn sich z.B. die Struktur des Knotens durch Umkonfiguration geändert hat. Wenn das AddOn dieser BiDiB-Befehl erreicht (oder auch nach einem Neustart der Baugruppe), wird die Funktion: `restart_addon()` aufgerufen.

```
/**
 * \brief Restarts AddOn application.
 *
 * Call all your own functions which have to restart after running.
 *
 */
void restart_addon(void)
{
  restart_turnout_manager();

  // TODO: continue restarts ...
}
```

Dort können alle Neu-Initialisierungen organisiert werden, die einen Neustart des Knotens notwendig machen.

Hier am Beispiel der OneControl, bei der bei Umkonfigurierung der Ein- und Ausgänge des GPIO-Bausteins ein Neustart nötig wird.

Weitere so genannte Callback-Funktionen im Modul `addon.c` werden weiter unten in den Kapiteln: Accessory-Behandlung, Makroausführung und Hilfseingabe erläutert.

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

Die BiDiB-Nachrichten werden in der Basissoftware abgehandelt. Allerdings kann für die abschließende Benachrichtigung die Bewertung des AddOns notwendig sein.

Nach Abarbeitung eines Accessorys bittet die Basissoftware das AddOn mit dem Aufruf der Funktion **`accessoryStateAddOn(…)`** um ein „Schlusswort“. Das kann eine Gutmeldung einfacher Art oder mit Zusatzinformationen laut BiDiB-Protokoll sein. Aber auch die Meldung eines Fehlers während der Bearbeitung des angefragten Accessorys ist möglich. Hat das AddOn nichts zu sagen, wird eine Standardantwort an das Host-System geschickt.

```
/**
 * \brief Fills the given accessory state for the given accessory according to BiDiB protocol:
 *        4.6.4. Uplink: Nachrichten für Accessory-Funktionen - MSG_ACCESSORY_STATE
 *        The function is called after finishing a macro.
 *
 * The given accessory corresponds to accessory given in performAddOnMacro.
 *
 * \param accessory         corresponding accessory
 * \param accessory_state   pointer to accessory state see BiDiB protocol
 *
 * \return uint8_t  state of delivery
 * \retval 1        state filled
 * \retval 0        no state available
 */
uint8_t accessoryStateAddOn(uint8_t accessory, t_bidib_accessory_state* accessory_state)
{
  return accessory_state_turnout(accessory, accessory_state);
}
```

Am Beispiel der OneControl. Der Weichenmanager schickt eine vom angeforderten Accessory abhängige Information in der übergebenen Struktur zurück und quittiert mit „1“. Dadurch wird seine Nachricht an den Host verschickt.

Makroausführung

Nach Erkennen eines Accessory-Set-Befehls wird 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 OneControl, die an dieser Stelle wird nur den Typ: BIDIB_OUTTYPE_SPORT unterstützt. Die Argumente sind:

- lstate ⇒ Schaltbefehl
- lvalue ⇒ Wert des Makropunktes
- accessory ⇒ Nummer des Accessorys für das „Schlusswort“

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.

Hilfseingabe

Die Basis-Firmware unterstützt Eingänge in zwei Arten:

- Meldung über den BiDi-Bus nach einer MSG_LC_KEY_QUERY-Nachricht
- Starten von Makros über externe Eingänge

In beiden Fällen wird die Callback-Funktion **`check_input_addon(…)`** aufgerufen. Sie muss den Zustand des übergebenen Eingangs liefern. Hat das AddOn diese Fähigkeit nicht, muss FALSE geliefert werden.

```
/**
 * \brief Checks the input with the given number and returns result.
 *        This function is called by the macro engine to start a macro.
 *        If there is no input on the given number, we return FALSE.
 *
 * \param number    number of input (0...7)
 *
 * \return uint8_t  result of check
 * \retval          TRUE  input is active
 * \retval          FALSE input is not active
 */
uint8_t check_input_addon(uint8_t number)
{
  return check_input_turnout_manager(number);
}
```

Beachte: Derzeit werden für die Makroausführung nur die Zustände der ersten 8 Eingänge abgefragt.

Spontane Keyboard-Ereignisse werden mit dem integrierten **Event-System** realisiert.

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:

```
#ifndef ADDON_DEBUG_IF_H_
#define ADDON_DEBUG_IF_H_

#include <stdbool.h>


#define ASCII_PARSE_TAB_ADDON \
    /* Command / Call                implemented / simulated / tested / comment */ \
    {{"OCL"},      PA_OneControlList },             /* i / s / t   list all dmx values  */ \
    {{"OCA"},      PA_OneControlAll },              /* i / s / t   all on, all off      */ \
    {{"OCD"},      PA_OneControlDirect },           /* i / s / t   set value            */ \


/*! \brief Initializes the AddOn. Called by main.
 *
 */
bool PA_info_addon(void);

/*! \brief Closes the AddOn e.g. should close all own used interrupts!
 *
 */
void PA_help_addon(void);


void PA_OneControlAll(void);
void PA_OneControlList(void);
void PA_OneControlDirect(void);


#endif /* ADDON_DEBUG_IF_H_ */
```

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:

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

/*! \brief Sends the caption of AddOn.
 *
 */
bool PA_info_addon(void)
{
  pc_send_string_PM("OpenDCC OneControl ");
  return true;
}

/*! \brief Sends short syntax description of AddOn debug commands.
 *
 */
void PA_help_addon(void)
{
  pc_send_answer_PM("OneControl Test");
  pc_send_answer_PM("   OCL: DMX all [0|1]: all off, all on");
  pc_send_answer_PM("   OCA: DMX direct");
  pc_send_answer_PM("   OCD: DMX list values");
}

//------------------------------------------------------------------------
// Special Debug-Functions for AddOn defined in ASCII_PARSE_TAB_ADDON:
//------------------------------------------------------------------------
void PA_OneControlAll(void)
{
  pc_send_answer_PM("Here comes the deep information for the requested debug command ... ");
}
```

Anschließend werden die eigentlichen Debug-Funktionen programmiert.

Zusammenfassung

Vorgehen

1. In `addon_model.h` Modell konfigurieren
2. In `addon_version.h` Versionsinformationen definieren
3. In `addon_features.h` Fähigkeiten definieren
4. EEPROM-Daten festlegen (CV)
    1. `addon_cv_define.h` Variablen definieren
    2. `addon_cv_declaration.h` Variablen deklarieren
    3. `addon_cv_data.h` Inhalte festlegen
    4. `addon_cv_features.h` Fähigkeiten festlegen
5. In `addon.c/h` Initialisierung (inkl. Power-Up) und „Shutdown“ sowie Neustart veranlassen
6. In `addon_tasklist.h` Tasks definieren
7. In `addon.c` „Schlusswort“ und Makrofunktionen sowie gfls. Eingangsschalter für Makros initiieren
8. In `addon_debug.c/h` Debugfunktionen einbauen

Hooks

Definitionen

Versionsdaten

- **MAIN_VERSION** Hauptversionsnummer
- **SUB_VERSION** Unterversionsnummer
- **COMPILE_RUN** Kompilierlauf

- **BUILD_YEAR** Herstellungsdatum: Jahr
- **BUILD_MONTH** Herstellungsdatum: Monat
- **BUILD_DAY** Herstellungsdatum: Tag

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)

Allgemeine Definitionen

siehe **`addon_model.h`**

- Aktiviert Zusatzfunktionen aus der Basis-Software, z.B. Servos (**SERVO_ENABLED**), LED-Blinken (**BLINK_MORSE_ENABLED**).
- Notwendige Angaben, z.B. über die Anzahl von Ein- und Ausgängen, z.B. **NUM_OF_INPUTS**, **NUM_OF_SPORTS**

Fähigkeiten

siehe **`addon_features.h`**

- Fähigkeiten des AddOns laut BiDiB, ein universelles Steuerprotokoll für Modellbahnen, z.B. **FEATURE_STRING_SIZE** oder **FEATURE_CTRL_INPUT_COUNT**.

Taskverwaltung cortos

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

CV-Werte

- EEPROM Version, Herstellerkennzeichnung, ... CV1-CV7
- (Reserviert CV8-CV69)
- Allgemeine Einstellungen CV70
- (Reserviert CV71-80)
- Servos (wenn konfiguriert) ab CV81
- Accessorys (wenn konfiguriert) ab CV209
- **AddOn spezifische Werte ab CV389**

Funktionen

Start und Stopp

- **init_addon()** Funktionsaufruf zur Initialisierung **vor** Freigabe de Interrupts
- **power_up_addon()** ask zur weiteren Initialsierung **nach** Freigabe der Interrupts
- **init_finished_addon()** Abfrage, ob AddOn mit INitialisierung fertig ist
- **restart_addon()** Funktion zum Neustart nach Knoten-Reset
- **close_addon()** Funktion zum Schließen des AddOns insbesondere der Interrupts

Betrieb

- **performAddOnMacro()** Ausführen der Makros eines AddoOns
- **accessoryStateAddOn()** „Schlusswort“ bei Beendigung des Accessorys
- **check_input_addon()** Zustandsabfrage der ersten 8 Eingänge zum Makrostart

Debug-Schnittstelle

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