Open Source Decoder

Architektur

Dokumentenpflege: Stefan Krauß
Hauptautor: Stefan Krauß
Mitautoren: -

Inhaltsverzeichnis

1 Überblick

1.1 Motivation und Ziel

Dieses Dokument beschreibt die Architektur der Firmware des Open Source Decoders (OSD). Wichtig ist, dass die wichtigsten Konzepte, die Aufteilung in die wichtigsten Module und die Bedeutung der Module und deren Schnittstellen erklärt wird. Die spätere Implementierung kann im Detail davon abweichen.

Es soll erreicht werden, dass sich über strukturelle Fragen im Vorfeld genügend Gedanken gemacht werden, um eine tragfähige Struktur im Code zu etablieren. Das gemeinsame Verständnis für diese Struktur soll durch das Dokument entstehen. Viele Details werden erst in der Umsetzung festgelegt werden, die von der Kreativität und Geschicklichkeit der beteiligten Programmierer abhängt. Außerdem muss davon ausgegangen werden, dass im Laufe des Projekts erst wichtige Punkte erkannt und durchdrungen werden sowie verschiedene Lösungswege erarbeitet und ausprobiert werden. Eine zu weitgehende Festlegung im Vorfeld wäre daher kontraproduktiv – ein Open-Source-Projekt muss atmen und wachsen können.

Ziel des Dokuments ist daher die grundlegende Strukturierung der Software und ein einheitliches Verständnis, wie die Module zusammenarbeiten.

1.2 Hardware-Basis

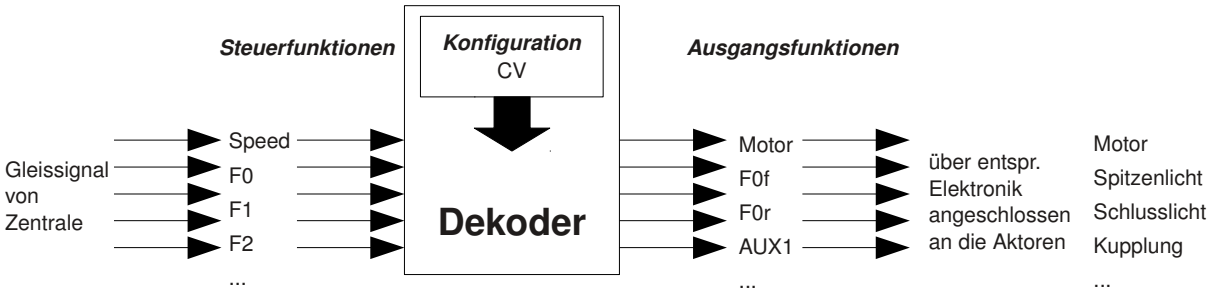
Grundsätzlich wäre es wünschenswert, wenn die OSD-Firmware vollständig HW-unabhängig und auf völlig unterschiedlichen Dekoder-Schaltungen lauffähig wäre. Dies ist in diesem Ausmaß allerdings nicht möglich (vgl. Kap. 4.1). Das vorliegende Design soll aber trotzdem dafür sorgen, dass die Firmware flexibel auf unterschiedliche Dekoder angepasst werden kann. Gegebenenfalls müssen dabei HW-abhängige Module ausgetauscht werden. Es wird im gesamten Code aber davon ausgegangen, dass es eine gemeinsame Basis gibt, die folgende Merkmale besitzt:

- Prozessor ARM Cortex-M0, Familie Nuvoton M051

Damit ist es dann möglich, mit wenig Aufwand in der Firmware, Dekoder mit unterschiedlichen Funktionsumfängen und HW-Ausstattungen zu erstellen.

1.3 Modell der Steuerungsabläufe und Begriffe

Von Außen betrachtet setzt der Dekoder bestimmte Eingangssignale in Ausgangssignale um. Die Umsetzung ist über Konfigurationsvariablen (CV) konfigurierbar. Die Eingangssignale kommen in der Regel kodiert über das Gleis, die Ausgangssignale steuern über eine geeignete Elektronik die angeschlossenen Aktoren (Motor, Lampen, LEDs, Lautsprecher, …). Dies gilt abstrakt für alle Dekoder.

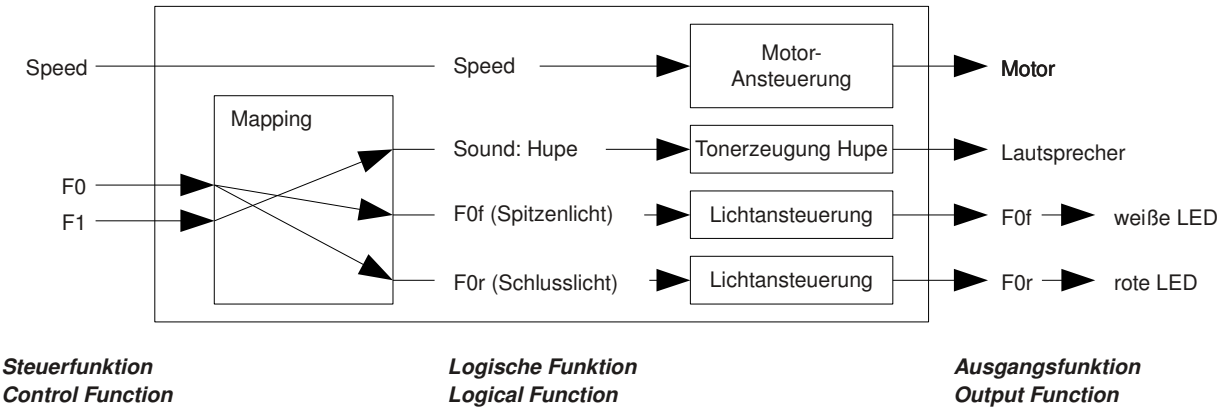


Die von der Zentrale über das Gleis kommenden Befehle (also insbesondere die Fahrstufe und die Funktionen F0 … Fn) nennen wir Steuerbefehle oder *Steuerfunktionen* (engl. *Control Function*).

Die elektrischen Ausgänge des Dekoder (also insbesondere die Motoranschlüsse und die Funktionsausgänge F0f, F0r, AUX1 … AUXn) nennen wir *Ausgangsfunktionen* (engl. *Output Function*).

Bemerkung: F0f und F0r stehen für die beiden Ausgangsfunktionen für die Stirnbeleuchtung vorne (front) und hinten (rear). Die Bezeichnungen sind der aktuellen Normung (insb. NEM658, NEM660) entliehen.

Für die interne Umsetzung der Funktionalität verwenden wir im Dekoder die folgende Struktur:



Die Ansteuermodule für die Ausgangsfunktionen werden durch CVs so konfiguriert, dass sie eine bestimmte *logische Funktion* an diesem Ausgang zur Verfügung stellen. Zum Teil ist dies eindeutig (z.B. Motor), meist können die Ausgänge aber auf verschiedene Aufgaben programmiert werden. So kann z.B. AUX1 sowohl eine Führerstandsbeleuchtung wie auch eine elektrische Kupplung ansteuern. Welche logische Funktion das Ansteuermodul zur Verfügung stellt, bestimmt die Konfiguration.

Die logischen Funktionen der Ausgangsfunktionen sind von der Konfiguration abhängig, wenn man es genau nimmt. In der praktischen Programmierung spielt dies jedoch keine Rolle. Die Ansteuermodule haben eine feste Anzahl an Eingängen, die die jeweilige logische Funktion abbilden. Das Ansteuermodul für den Ausgang AUX1 im obigen Beispiel hat also immer ein einziges Eingangssignal zum „Einschalten“ und „Ausschalten“. Ob in der konkreten Programmierung eine Kupplung oder eine Licht geschaltet wird, spielt für das Programm keine Rolle.

Das Mapping bildet die Steuerfunktionen, soweit sinnvoll, auf die logischen Funktionen ab. Das Mapping wird inhaltlich wie die Ansteuermodule durch die Konfiguration bestimmt. Die Steuerfunktionen können auch fest 1:1 mit den logischen Funktionen verbunden sein.

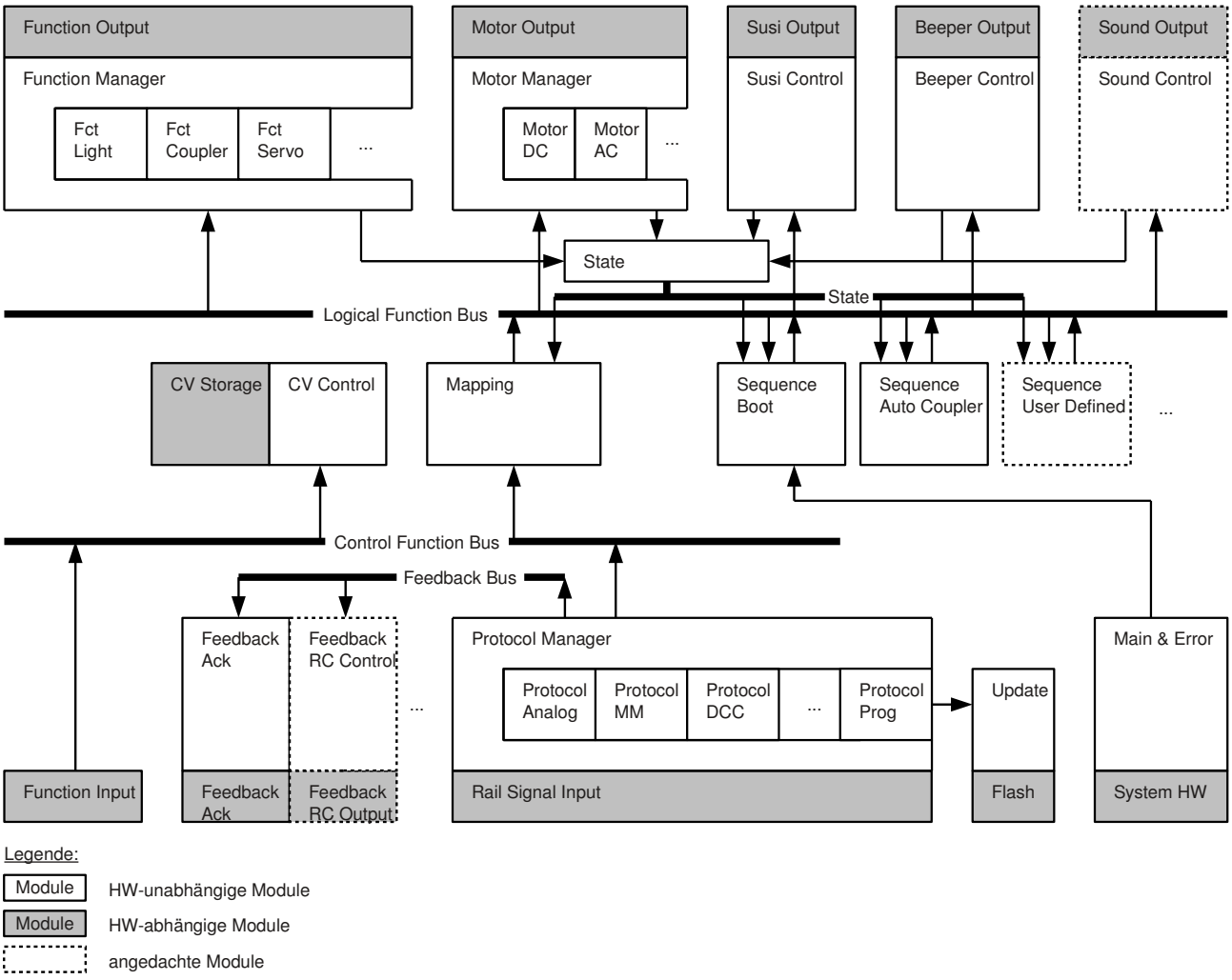
Diese interne Struktur ist zunächst ein Gedankenmodell. Es definiert zentrale Begriffe und schlägt in erster Linie auf die Konfiguration durch. Die Implementierung könnte grundsätzlich anders strukturiert sein. Dafür gibt es aber keinen Grund, deshalb baut die vorliegende Architektur ebenfalls auf dieser Struktur auf.

Moderne Dekoder bauen in der Regel ebenfalls auf diesem Modell auf. Dort sind auch die CVs entsprechend aufgebaut und auch die komfortableren Konfigurationsoberflächen der Zentralen orientieren sich daran. Deshalb ist es eine gute Idee, auch den OSD so zu strukturieren.

2 Modulstruktur

2.1 Übersicht

Die folgende Grafik zeigt die Modulstruktur und die wichtigsten Datenflüsse zwischen den Modulen im Überblick. Die einzelnen Bestandteile werden in den folgenden Unterkapiteln genauer erklärt.



2.2 Grundkonzepte

Module

Module im Sinne des Design-Dokuments sind zusammengehörige Code-Teile mit einer klaren Funktionsbeschreibung und einer sinnvollen Schnittstelle. Die Beschreibung der Funktionalität und der Schnittstelle definieren das Modul.

Module werden in der Regel als eigenständige Module im Sinne von C implementiert, daher in einer eigenen C-Datei ausprogrammiert und die Schnittstelle in einer C-Header-Datei definiert. Von dieser Regel kann aber aus programmiertechnischen Gründen abgewichen werden. So kann ein Modul auch nur aus einer einzigen Header-Datei oder aus mehreren C- und Header-Dateien bestehen. Mehrere Module können auch zusammenfallen, wenn es sich herausstellt, dass die Aufteilung der Funktionalität nur Overhead produziert und sich nicht lohnt.

Das Design-Dokument beschreibt die Funktionalität und Schnittstelle der Module nur informell. Die konkrete Ausgestaltung geschieht während der Implementierung und ist dem Code zu entnehmen. Auch die Namensgebung der Module unterscheidet sich im Design-Dokument geringfügig von den Namen im Code (wg. Lesbarkeit werden längere Namen mit Leerzeichen verwendet, allerdings werden schon die englischen Begriffe verwendet, die auch im Code verwendet werden sollen).

HW-unabhängige, HW-abhängige Module

Bei allen Modulen wird zwischen HW-abhängigen und HW-unabhängigen Modulen unterschieden. Für viele Aufgaben, wie z.B. die Susi-Schnittstelle, ist der Code deshalb in zwei Teile aufgeteilt, nämlich einen HW-unabhängigen und einen HW-abhängigen.

HW-abhängig heißt, dass das Modul auf konkrete HW zugreift und von der konkret verfügbaren HW abhängt. Das HW-abhängige Modul stellt dabei nicht unbedingt nur eine abstrakte Sicht auf die HW dar, sondern erledigt den HW-abhängigen Teil einer Aufgabe. Im Susi-Beispiel heißt das: Das HW-abhängige Susi-Modul führt Befehle wie „Sende Susi-Befehl“ aus (führt also einen konkreten Teil der Susi-Funktionalität aus, die aber von der verfügbaren HW abhängig ist) und stellt nicht nur eine allgemeine Funktion wie „Sende Byte auf einer seriellen Schnittstelle“ bereit.

Die HW-abhängigen Teile stützen sich in der Regel auf Zugriffsbibliotheken (z.B. CMSIS) des Prozessor- oder Compiler-Herstellers. Eine Umstellung auf einen anderen Prozessor wäre damit in der Regel einfach machbar. Trotzdem kann das HW-abhängige Modul alle Besonderheiten des vorliegenden Prozessors verwenden. Gegebenenfalls muss das HW-abhängige Modul durch ein komplett neues ersetzt werden, das die gleiche Funktionalität und damit Schnittstelle auf einer ganz anderen HW und vielleicht auch auf eine ganz andere Art und Weise zur Verfügung stellt (vgl. 4.1).

Busse

In der Zeichnung sind „Busse“ eingezeichnet. Diese bündeln zum besseren Verständnis bestimmte Datenflüsse zwischen Modulen. In der Implementierung sind diese Busse, wie alle anderen Datenflüsse auch, als einfache Funktionsaufrufe implementiert. Anders als die Darstellung nahelegt, sind die Verbindungen fest und von Anfang an bekannt. Es werden also keine dynamischen Verbindungen über die Busse verwaltet.

Die Busse sind nur logische Gebilde, keine FIFOs oder irgendwelche dynamischen Strukturen. Ich weiß z.B., dass ein Steuerbefehle "F-Taste" immer vom Gleis und damit von der Protokoll-Erkennung kommt und dem Mapping übergeben werden muss, das den Befehl verarbeitet. Implementiert wird diese „Bus-Kommunikation“ also nur als einfacher Funktionsaufruf von der Protokoll-

Erkennung ans Mapping. Gäbe es weitere Abnehmer, würden eben zwei oder drei (gleichartige) Funktionsaufrufe erzeugt. Im Code kann man das geschickt über Makros machen, so dass man nur noch in einem Header eintragen muss, welche "Buseingänge" auf welche "Busausgänge" zu legen sind.

Bibliotheken

Die verwendeten Bibliotheken des Prozessor- oder Compiler-Herstellers sind in der Zeichnung nicht dargestellt.

Vollständigkeit der Module

Die Zeichnung soll einen Überblick über die wichtigsten Module und deren Schnittstellen und Abhängigkeiten geben. Sie erhebt aber nicht den Anspruch auf Vollständigkeit. In der Implementierung können bestimmte Module aus mehreren Einzelmodulen bestehen oder bei Bedarf weitere Hilfsmodule eingefügt werden. Diese Freiheiten sind den Implementierern zugestanden, so dass diese die insgesamt übersichtlichste und funktional beste Struktur wählen können.

2.3 Module: Allgemeine Funktionen

Die allgemeinen Funktionen enthalten im Wesentlichen den Code zum Booten des Prozessors und zur Initialisierung der Hardware, der Timer, der Interrupts usw. Die Module sind damit auch der Ursprung für alle Prozesse (siehe Kap. 3) und die Hauptschleife.

`Main & Error`

Das Modul enthält die Hauptschleife, die zunächst alle Initialisierungsroutinen aufruft und dann zyklisch alle Aufgaben abarbeitet, die nicht Interrupt-gesteuert sind.

Ebenso ist die globale Fehlerüberwachung und -verarbeitung hier implementiert. Es wird insbesondere der Stromverbrauch des Moduls bzw. der Funktionsausgänge zyklisch gemessen und bei Überlastung die Ausgänge abgeschaltet.

Die Fehlerbehandlung folgt einer einfachen Strategie: Sobald ein Fehler auftritt, geht der Dekoder in einen sicheren Zustand, der in der Regel heißt, alle Ausgangsfunktionen abzuschalten. Ist die Störung beseitigt, wird in den letzten Zustand zurückgekehrt. Im Zweifel bleibt der Dekoder im sicheren Zustand, bis er wieder neu gestartet wird.

In diesem Modul ist die C-Main-Funktion enthalten, die automatisch nach Start des Dekoders aufgerufen wird. Hier beginnt also die Ausführung.

`System HW`

Die Initialisierung der Prozessor-Hardware, Timer etc. ist HW-abhängig und daher in einem eigenen Modul untergebracht. Es geht hier nur um die Initialisierung der Prozessor-Ressourcen, nicht um die Initialisierung der verschiedenen Peripherie-Einheiten oder Funktionalität. So wird z.B. die Susi-Schnittstelle im Susi-Modul initialisiert und dabei ggf. auch erste Susi-Befehle verschickt, die wiederum angeschlossene Susi-Komponenten initialisieren.

Ein Modul wie das Modul Susi Control hat deshalb eine Init-Funktion, die von Main aufgerufen wird. Susi Control wird wiederum ein Init im HW-spezifischen Susi-Modul Susi Output aufrufen, um die Schnittstelle zu initialisieren (auch die im Prozessor eingebaute Peripherie, hier also z.B. die SPI-Schnittstelle, falls die für die Susi-Kommunikation verwendet wird). Alle zur Susi-Schnittstelle

gehörenden Routinen inklusive HW-Initialisierung und -Ansteuerung sind so lokal in den Susi-Modulen gekapselt. Dieses Schema gilt für alle Modulgruppen und wird im Weiteren nicht mehr gesondert erwähnt.

Das Modul enthält auch die HW-Zugriffsfunktionen, die für die (globale) Fehlererkennung notwendig sind, also insbesondere die Routine zum Messen der Stromaufnahme.

Interface: Die Funktion „Init“ zum Aufruf der Initialisierung. Dazu die für die Fehlererkennung notwendigen Abfragefunktionen.

2.4 Module: Gleissignalerfassung und Protokollauswertung

Die Gleissignalerfassung misst die Spannung am Gleis (Hardware), filtert diese und gibt sie als Datenstrom von positiven und negativen Impulsen (auch Pausenzeiten ohne Gleisspannung und analoge Spannungen) an die Module der Protokollauswertung weiter. Diese gibt es n-fach, sie enthalten die Protokollerkennung und die dafür notwendige Zustandsmaschine. Als Ergebnis senden sie über den Control Function Bus (siehe unten, Abschnitt 2.9) die erkannten Befehle wie Geschwindigkeit ändern oder Funktion schalten. Diese Befehle sind unabhängig vom Protokoll und für alle Protokolle gleich. Auch werden bereits in der Protokollerkennung die CV-Nummern auf die intern verwendete, einheitlichen CV-Identifier (ebenfalls eine Nummer) abgebildet und nur diese über den Control Function Bus gesendet.

Allerdings gibt es auch protokollspezifische Aktionen, z.B. das Versenden der Dekoder-ID über die Rückmeldung. Diese Aktionen werden nicht über den Control Function Bus gesendet und sind auch nicht abstrahiert. Die jeweilige Protokollauswertung ruft z.B. das Rückmeldemodul direkt auf.

`Rail Signal Input`

Liest das Gleissignal mit Hilfe der vorhandenen Hardware ein, filtert es und generiert daraus eine Folge von relevanten Gleisereignissen (z.B. „100 µs positiv“). Welcher Algorithmus für die Erfassung, Berechnung und Störungsunterdrückung verwendet wird, ist nicht bestimmt.

Das Modul ist in zweierlei Hinsicht HW-abhängig: Zum Einen greift der Code aus Performance-Gründen möglichst direkt auf die HW zu. Zum Anderen ist das eingesetzte Erkennungsverfahren von den zur Verfügung stehenden HW-Möglichkeiten abhängig (z.B. ob vom Gleissignal einer oder mehrere verschiedene Interrupts generiert werden können, ob Polling angewandt wird, wie die Zustände des Gleissignals erfasst werden usw.).

Das Gleissignalerfassungsmodul kann je nach Hardware und gewünschtem Algorithmus ausgetauscht werden, jedoch nur zur Compile-/Programmier-Zeit. Damit sind verschiedene Experimente möglich, außerdem eine leichte Anpassung an die jeweilige Dekoder-HW.

Schnittstelle: Ausgabe von Gleisereignissen, insbesondere sind das

- positives Signal für x µs
- negatives Signal für x µs
- kein Wechsel seit x ms (x ist ein fest eingestelltes Timeout; der Befehl wird benötigt, um das Ende eines Befehls zeitnah erkennen zu können)
- kein Gleissignal (0 V)

- Analogmodus erkannt, Gleissignal beträgt zur Zeit x Volt
- PWM-Signal erkannt, Duty Cycle beträgt zur Zeit x % (die Erkennung von PWM-Signalen wird unter Umständen aber auch erst in einem entsprechenden PWM-Protokoll-Auswertemodul realisiert, dann wird auch dort erst der Duty Cycle berechnet)

Im PWM und Analogmodus wird der aktuelle Zustand zyklisch übertragen. Die Zykluszeit sollte im Programm leicht geändert werden können, so dass man nach ein paar Versuchen einen optimalen Wert vorgeben kann.

Die Gleissignalerfassung und die Auswertung laufen in zwei unterschiedlichen Prozesskontexten, insbesondere läuft die Gleissignalerfassung im Interrupt. Zur Entkopplung kann eine (kurze) Queue zur Übergabe der Gleisereignisse verwendet werden.

`Protocol Manager`

Dieses Modul nimmt die Gleisereignisse entgegen und verteilt sie an alle aktiven Protokolle. Eine Auswertung der Gleisereignisse findet dabei nicht statt, dies ist Aufgabe der verschiedenen Protokollauswertungsmodule (wg. Lokalität). Allerdings kann nicht jedes Protokollauswertungsmodul alle Ereignisse entgegen nehmen (z.B. die Analogspannung). Das Modul sorgt dafür, dass nur die Protokollauswertungsmodule angesprochen werden, die auch eingeschaltet sind. Damit werden bei abgeschalteten Protokollen die Rechenbelastung im Dekoder verringert und Fehlinterpretationen ausgeschlossen.

Die Protokollauswertungsmodule werden nicht dynamisch verwaltet und müssen auch keine vollständig abstrahierte Schnittstelle haben. Das Manager-Modul sucht also nicht beim Start nach verfügbaren Protokollauswertemodulen. Diese sind aus Performance-Gründen zur Compile-Zeit fest mit dem Manager-Modul „verbunden“. Auch kann das Manager-Modul mit leicht unterschiedlichen Schnittstellen der Protokollauswertemodule umgehen, falls diese Unterschiede benötigt werden. Schließlich wird jedes Protokollauswertemodul fest eincompiliert.

Das Manager-Modul umschließt sozusagen die Protokollauswertemodule. Es nimmt daher deren Ausgabe, die erkannten Befehle, entgegen und gibt diese über den Control Function Bus weiter. Es kann dabei die verschiedenen Ausgaben synchronisieren. So kann z.B. eine Priorisierung von Protokollen stattfinden.

Schnittstelle: Die Eingabe besteht aus dem Lesen der Gleisereignisse, die Ausgabe sind Schreiboperationen auf den Control Function Bus.

`Protocol Analog, Protocol MM, Protocol DCC, ...`

Diese Module sind die eigentlichen Protokollauswertemodule. Sie dekodieren aus dem Strom von Gleisereignissen die Gleisbefehle im jeweiligen Protokoll. Die Protokolle werden dabei parallel zueinander ausgewertet. Die Toleranzen für die Eingangssignale können so sehr großzügig und stark überlappend gewählt werden. Im Laufe der Befehlserkennung werden dann Fehlinterpretationen sehr schnell erkannt und verworfen.

Die Protokollauswertemodule sind vergleichsweise aufwändige Module. Sie verarbeiten die Gleisereignisse zu logischen Bitfolgen, erkennen darin die Frames des Protokolls, dekodieren die darin enthaltenen Befehle und führen ggf. noch eine entsprechende Zustandsmaschine mit (z.B. um Befehlsfolgen zu erkennen). Außerdem wird eine Übersetzung von protokollspezifischen CV-Nummern (in welcher Form auch immer CVs im jeweiligen Protokoll identifiziert werden) in die intern

verwendeten, für alle Protokolle einheitlichen CV-Speicherplätze vorgenommen.

Auch die unterschiedlichen Bremsmodi werden als eigene Protokolle betrachtet und erhalten jeweils ein eigenes Protokollauswertemodul. Sie erzeugen Bremsbefehle auf dem Control Function Bus. Diese ähneln den Fahrbefehlen und wirken wie diese auf die Motor-Steuerung.

Die Protokollauswertemodule haben im Wesentlichen nur Schnittstellen zum Protokoll-Manager-Modul. Ggf. wird noch ein Rückmeldemodul angesprochen.

`Feedback Ack Control`

Das DCC-Protokoll kennt eine einfache Rückmeldung durch Bestätigungsimpulse (Acknowledge). Für diese Rückmeldung gibt es wie für alle anderen Arten von Rückmeldungen ein eigenes, HW-unabhängiges Steuermodul.

Dieses bedient sich einer eigenen Rückmelde-HW, die über das zugehörige, HW-abhängige Modul angesteuert wird, oder einer vorhandenen Einrichtung wie dem Motor. In letzterem Fall wird direkt der Motor-Manager zu einer entsprechenden Aktion aufgefordert. Diese Verbindung ist in der Übersicht nicht eingezeichnet. Wichtig ist jedoch, dass dies eine eigene Funktionalität der Motor-Ansteuerung ist, also nicht über Pseudo-Fahrbefehle o.ä. simuliert wird.

Auch der Funtionsmanager könnte die Funktionalität Acknowledge anbieten, falls die Rückmeldung über eine an einem Funktionsausgang angeschlossene Elektronik implementiert wird. Welche Art von Rückmeldung verwendet wird, wird im Dekoder nicht konfiguriert, sondern schon während der FW-Erstellung fest einprogrammiert. Denn die Dekoder-HW gibt die Rückmeldemöglichkeiten eigentlich fest vor. Man kann den Quellcode natürlich über Makros schon so ausrüsten, dass leicht zwischen verschiedenen Rückmelde-Konfigurationen umgeschaltet werden kann.

`Feedback Ack Output`

Das HW-abhängige Modul enthält die Ansteuerung für eine eigene Acknowledge-Rückmelde-HW, falls es eine solche überhaupt gibt.

`Feedback RC Control, ...`

Auch für die Rückmeldung über das zu DCC gehörende RC wird eine eigenes Modulpaar (HW-abhängig und HW-unabhängig) benötigt, wenn diese Rückmeldetechnik implementiert werden soll. Dies gilt auch für alle anderen denkbaren Rückmeldetechniken.

Die Informationen, die zurückgemeldet werden, bestehen z.B. aus bestimmten CV-Werten oder Status-Informationen. Dazu holt sich das Rückmeldemodul diese Informationen aus den entsprechenden Modulen (hier also z.B. dem CV-Steuermodul oder dem Zustandsmodul). Die angesprochenen Module müssen dafür auch entsprechende Schnittstellen bereithalten. Diese Verbindungen sind in der Skizze nicht eingezeichnet.

Die Schnittstelle besteht in der Regel aus Funktionen der Art „Sende Information X“.

`Feedback RC Output, ...`

Das HW-abhängige Rückmeldemodul enthält die Ansteuerung der zur Rückmeldung notwendigen Hardware.

Die Schnittstelle ist von der Art der Rückmeldung abhängig.

`Protocol Prog`

Sollte ein spezielles Programmierverfahren im Dekoder implementiert werden, so wird dieses wie ein eigenständiges Protokoll behandelt. Es wird daher als Protokollauswertemodul implementiert und in den Protokoll-Manager eingehängt. Die z.B. in DCC integrierten Befehle zum Schreiben/Lesen von CVs fallen jedoch nicht darunter, sie sind integraler Bestandteil des DCC-Protokolls. Sound- und Firmware-Flash-Routinen verwenden hingegen unter Umständen ein eigenes Protokoll und werden durch dieses Protokoll-Modul implementiert.

Anmerkung: Auch eine Update-Technik, die auf den vorhandenen Protokollen aufsetzt, ist denkbar (z.B. durch wiederholtes Schreiben in eine CV oder der NMRA-Vorschlag einer Streaming-Erweiterung des DCC-Protokolls). In diesem Fall ist natürlich kein eigenes Protokoll-Modul notwendig.

Das Programmier-Modul generiert keine internen Befehle über den Control Function Bus, sondern führt selbst die Neuprogrammierung durch und steuert dabei direkt das Update-Modul an.

`Update`

Da der Flash-Speicher nicht neu programmiert werden kann, wenn das Programm aus dem Flash ausgeführt wird, sind die benötigen Programmteile aus einem nicht zu programmierenden Speicherbereich auszuführen. Dies kann das RAM oder ein eigener Flash-Bereich sein.

Im RAM: Das Update-Modul kopiert vor Start der Programmierung die noch benötigten Routinen ins RAM. Ab diesem Zeitpunkt ist dann nur noch der Flash-Vorgang möglich, alle anderen Module und Fähigkeiten des Dekoders sind inaktiv (schon allein deshalb, weil die benötigten Routinen nicht komplett ins RAM passen würden). Nach dem Neustart ist der Flash-Vorgang beendet und der Dekoder arbeitet wieder normal. Dies hat den großen Nachteil, dass ein Unterbrechen des Flash-Vorgangs den Dekoder in einem undefinierten Zustand zurücklassen könnte, in dem auch kein erneutes Flashen mehr möglich wäre.

Im Flash: Wenn möglich sollte daher eine kleine „Flash-FW“ in einem Flash-Block, der nicht geupdated werden kann, abgelegt werden. Dieser Teil enthält alles, was zum Flashen benötigt wird, auch die Gleissignalerfassung. Allerdings wird dieser Teil nur für den Update-Vorgang verwendet. Außerdem startet dieser Teil (im Sinne einer „Default Boot FW“), wenn der Update-Vorgang nicht vollständig erfolgreich war. Dies wird an einem Byte erkannt, das als erstes gelöscht und als letztes geschrieben wird. Nach einem abgebrochenen Programmiervorgang kann so der Dekoder immer wieder neu programmiert werden. Da der Flash-Speicher meist in Segmenten aufgeteilt ist, wäre das ein Segment, das nicht neu programmiert werden kann. Die Update-Routine selbst könnte so aus Sicherheitsgründen auch nicht geupdatet werden. Das wäre sicher verschmerzbar.

Die Flash-FW würde man als weiteres Modul implementieren, das dann vom Programmierprotokoll aufgerufen wird.

Manche Protokolle benötigen eine eindeutige Seriennummer. Diese ist so im Dekoder unterzubringen, dass sie nicht überschrieben werden kann. Das wäre also in der beschriebenen festen Boot-FW. Alternativ könnte die Update-Routine dafür sorgen, dass die Seriennummer beim Update nicht verändert wird. Dazu wird während des Flash-Vorgangs die Seriennummer im zu flashenden Code durch die bereits im Dekoder gespeicherte ersetzt.

Hinweis: Da ohne die Boot-FW nur mit Hilfe des Programmier-Adapters und entsprechender Software die FW auf den Dekoder aufgespielt werden kann, sollte zumindest die Boot-FW auf Dekodern vorhanden sein, die an Endkunden verkauft werden.

`Flash`

Soll über ein Programmierprotokoll Sound oder Firmware programmierbar sein, so wird dazu eine HW-spezifische Zugriffsroutine auf den Flash-Speicher benötigt. Diese ist in diesem Modul implementiert.

2.5 Module: Configuration Variables und Mapping

`CV Control`

Das CV-Modul stellt einen einfachen Zugriff auf die verschiedenen Variablen (CV) des Dekoders zur Verfügung.

Schnittstelle: Die Variablen können über den Control Function Bus mit Schreib- und Leseoperationen über das Gleisprotokoll programmiert und ausgelesen werden. Beim Auslesen werden die Daten an die Rückmeldemodule weitergegeben.

Alle abgespeicherten Werte werden von den unterschiedlichsten Modulen des Dekoders benötigt, zum Teil auch geschrieben. Dies geschieht über eine zweite Schnittstelle, die einen effizienten Zugriff auf die Werte erlaubt. Diese Zugriffspfade sind in der Übersichtszeichnung oben nicht dargestellt!

`CV Storage`

CV Storage kapselt den EEPROM-Bereich oder einen anderen nicht-flüchtigen Speicher zum Abspeichern der CVs.

Die Schnittstelle stellt Lesen und Schreiben von Speicherbereichen zur Verfügung.

`Mapping`

Das Mapping ordnet Befehlen des Control Function Busses einem oder mehreren Befehlen auf dem Logical Function Bus zu. Das Mapping ist über CVs konfigurierbar. Das Mapping entspricht damit einer großen Zuordnungstabelle (sie wird so zum Teil auch in den Programmier-Tools und Zentralen dargestellt). Die Zuordnung ist auch vom Zustand des Dekoders abhängig, insbesondere die aktuelle Fahrtrichtung.

Es gibt einige Befehle (z.B. die Fahrbefehle), die nicht verändert werden. Hier besteht also eine feste 1:1-Zuordnung. Trotzdem werden auch diese Befehle über das Mapping-Modul vom Control Function Bus auf den Logical Function Bus durchgereicht.

Das Mapping arbeitet synchron und kann keine Sequenzen ausführen. Dafür gibt es eigene Module (vgl. 2.8). Das Mapping kann allerdings aus einem Eingangsbefehl mehrere Ausgangsbefehle erzeugen (z.B. auf F0 die beiden logischen Funktionen Stirn- und Schlusslicht einschalten).

Das Mapping-Modul ist auch dafür zuständig, dass auf dem Logical Function Bus nur die wirklich notwendigen Befehle aufgerufen werden. Über das Gleissignal werden fast alle Befehle laufend wiederholt. So wird der Zustand der Funktionen zum Beispiel regelmäßig übermittelt. Dies führt jeweils zu einem Schaltbefehl auf dem Control Function Bus (z.B. „F0 = ein“). Das Mapping erzeugt nun auf dem Logical Function Bus nur dann einen Befehl (z.B. „Stirnlicht einschalten“), wenn die Funktion vorher nicht schon aktiv war. Das Mapping speichert dazu also auch den aktuellen Zustand der Steuerfunktionen.

Widersprechen sich Funktionen durch mehrfaches Mapping auf eine logische Funktion, ist dies zunächst kein Problem für den Dekoder. Das Einschalten einer Steuerfunktion führt zu einem Einschaltbefehl für die logische Funktion, unabhängig davon, ob sie schon von einer anderen Funktion eingeschaltet wurde. Ebenso beim Ausschalten. Die beobachtbare Reaktion wird dabei natürlich auch von der Tatsache beeinflusst, dass nur bei einem Zustandswechsel der Steuerbefehle neue Befehle auf dem Logical Function Bus erzeugt werden.

Das Mapping soll eine weitere Funktion erhalten, die es z.B. erlaubt, das Abschalten der dem Zug zugewandten Beleuchtung im Mapping zu definieren. Dazu können jeder Steuerfunktion auch logische Funktionen zugeordnet werden, die „gesperrt“ sind, wenn die Steuerfunktion aktiv ist. Gesperrt heißt, dass die gesperrte logische Funktion nicht mehr angesprochen wird, so lange die sperrende Steuerfunktion aktiv ist. Außerdem wird die gesperrte logische Funktion beim Aktivieren der sperrenden Steuerfunktion explizit ausgeschaltet und beim Deaktivieren der sperrenden Steuerfunktion wieder in den alten Zustand zurückversetzt.

Bestimmte Zustandswechsel könnten als Pseudo-Steuerkommandos Eingangssignale für die Mapping-Tabelle darstellen. Ein solcher Zustand kann der Wechsel in den Analog-Modus sein, in dem man über die Mapping-Tabelle bestimmte (logische) Funktionen automatisch einschalten könnte. Ebenso wäre es auf diese Weise möglich, bestimmte logische Funktionen beim Anhalten (Wechsel in den Status Stillstand) auszulösen. Für diese Funktionen wäre es gut, wenn das Mapping die Funktion auch negieren könnte. Wenn also beim Einschalten einer Steuerfunktion das Ausschalten einer logischen Funktion bewirkt werden könnte.

2.6 Module: Funktionsausgänge und Motoransteuerung

Die Ausgabefunktionen werden über verschiedene, aufgabenspezifische Ausgabe-Module implementiert. In der Regel wird eine Funktionalität über ein HW-unabhängiges Steuerungsmodul und ein HW-abhängiges Ausgabe-Modul implementiert.

`Function Manager`

Das Modul ist für die Ansteuerung der „normalen“ Funktionsausgängen des Dekoders zuständig. Diese Ausgänge sind universell einsetzbar und können auf ganz unterschiedliche logische Funktionen programmiert werden. Es gibt relativ viele solche Ausgänge, die die gleiche Funktionalität bereitstellen. Aus diesem Grund wird für alle Ausgänge ein gemeinsames Modul zur Ansteuerung verwendet.

Die verschiedenen logischen Funktionen wie Blinklicht oder automatische Kupplung werden in eigenen Funktionsmodulen implementiert. Damit können weitere Funktionen leicht hinzugefügt werden. Das Manager-Modul „instanziert“ das jeweilige Funktionsmodul für jeden Ausgang, der auf diese Funktion vom Benutzer programmiert wird. Das Funktionsmodul gibt es also nur einmal, es kann aber mehrfach gleichzeitig für verschiedene Ausgänge benutzt werden. Die Koordination wird vom Manager-Modul erledigt.

Die Schnittstelle des Manager-Moduls ist eine (generische) logische Funktion für jeden elektrischen Ausgang (also jede Ausgangsfunktion). Je nach Konfiguration des Ausgangs stellt die logische Funktion eine Lichtfunktion, eine Kupplung oder sonst etwas dar. Das ist für das Programm aber nicht wichtig, es gibt immer genau eine logische Funktion pro Ausgang. Deshalb die Bezeichnung als generische logische Funktion.

`Function Light, Function Coupler, Function Servo, ...`

Die Funktionsmodule implementieren jeweils eine Funktion und können wie bei Manager-Modul beschrieben mehrfach instanziert werden (aber für jeden Ausgang höchstens einmal).

`Function Output`

Dieses Modul enthält die HW-abhängigen Teile für die Ansteuerung der einfachen Funktionsausgänge. Die Ausgänge können damit ein- und ausgeschaltet werden. Im eingeschalteten Zustand können die Ausgänge ein PWM-Signal ausgeben, Frequenz und Puls-Pausen-Verhältnis (duty cycle) sind programmierbar.

`Motor Manager`

Die Ansteueralgorithmen für die verschiedenen Motortypen funktioniert ähnlich wie bei der Funktionsausgabe. Allerdings gibt es hier immer nur maximal eine aktive Motor-Steuerung, die dann auch die Hardware-Ansteuerung übernimmt. Der Motor-Manager hat deshalb nur die Aufgabe, das aktive Motor-Steuermodul auszuwählen.

Das Modul implementiert außerdem die Anfahr-/Bremsverzögerung bzw. Geschwindigkeitskennlinien (genauer: die „Massensimulation“). Vom Logical Function Bus kommt die Anforderung („Befehl“), eine bestimmte Geschwindigkeit zu fahren. Die ABV setzt dies stufenweise in entsprechende Geschwindigkeitsvorgaben für die Motor-Ansteuerung um.

Ebenso könnte in diesem Modul die Lastregelung implementiert werden. Allerdings wird davon ausgegangen, dass die Lastregelung sehr Motortyp-spezifisch ist und daher in den Motor-Ansteuermodulen untergebracht werden muss.

Die Motor-Ansteuerung könnte im Prinzip auch zwei Motoren bedienen. Motor-Manager, die konkrete Motor-Ansteuerung und die HW-abhängigen Motor-Module müssten entsprechend erweitert werden. Würde ein Motor über eine Schnittstelle indirekt angesteuert (z.B. über SPI), so müsste das HW-abhängige Motor-Modul eben SPI „sprechen“.

`Motor DC, Motor AC, ...`

Die Module enthalten die Ansteueralgorithmen für verschiedene Motoren (DC-Motor mit/ohne Lastregelung, 3/5-Poler-AC-Motor, AC-Motor mit Lastregelung, Glockenanker-Motor, …). Durch Austausch der Module könnten auch verschiedene Algorithmen für die gleiche Motorart ausprobiert werden.

Der Dekoder kann zwar mehrere Motor-Ansteueralgorithmen enthalten, aktiv ist jedoch zu jedem Zeitpunkt maximal eine (konfiguriert über CVs).

`Motor Output`

Das Modul enthält den HW-abhängigen Teil der Motoransteuerung. Dieser besteht vor allem aus der elektrischen Motor-Ansteuerung (H-Brücke) und dem Messen der Höhe des Gegen-EMK (A/D-Wandler; Spannung am drehenden Motor in kurzen Ansteuerlücken). Letzteres wird für die Lastregelung benötigt.

Die Idee ist, die HW-abhängigen Teile in einem Modul zu kapseln und die verschiedenen Motor-Ansteueralgorithmen in jeweils eigenen Modulen unterzubringen. Die HW-Anpassung wäre so sehr einfach. Dazu wäre aber eine abstrakte HW-Schnittstelle für die Motor-Elektronik notwendig,

die auf die unterschiedlichsten Dekoder-Typen anwendbar ist. Aus Effizienzgründen und um die vorhandenen HW-Eigenschaften verschiedener Dekoder-Elektroniken wirklich nutzen zu können, könnte es aber sinnvoll sein, die Schnittstelle an die tatsächlichen Gegebenheiten anzupassen. In diesem Fall könnten die Module mit den Ansteueralgorithmen nicht mehr einfach ausgetauscht werden. Trotzdem wären die Module vielleicht mit sehr wenig Aufwand anpassbar.

Abhängig von den Motor-Ansteueralgorithmen müsste hier noch über die konkrete Schnittstelle entschieden werden (ggf. nach Versuchen).

2.7 Module: Spezielle Funktionen wie Sound, Susi und Funktionseingänge

Auch alle „besonderen“ Funktionen werden in entsprechenden Modulen realisiert. Diese werden an den Control Function Bus angeschlossen, wenn es sich um Eingabefunktionen handelt, und an den Logical Function Bus, wenn es sich um Ausgabefunktionen handelt. Diese beiden Busse bilden damit die Schnittstellen dieser Module.

Zu den aufgeführten Funktionalitäten können leicht weitere hinzugefügt werden, z.B. für einen speziellen Zugbus. Diese werden nach dem gleichen Muster strukturiert und an den Control Function Bus oder den Logical Function Bus angeschlossen.

`Function Input`

Das Modul kümmert sich um das Einlesen der zusätzlichen Funktionseingänge. Es ist HW-abhängig, weil es direkt auf die HW zugreift. Die Ausgabe sind Steuerbefehle auf dem Control Function Bus analog zu denen, die über das Gleisprotokoll von der Zentrale gesendet werden. Es sind also ebenfalls Steuerfunktionen.

`Susi Control`

Die Susi-Schnittstelle leitet interne Befehle an extern angeschlossene Susi-Komponenten weiter. Sie bildet damit im Wesentlichen eine Kommunikationsschnittstelle. Das Control-Modul implementiert das Protokoll und übernimmt auch ggf. zyklisches Polling von Angaben aus der Susi-Komponente. Das Modul ist HW-unabhängig.

`Susi Output`

Das Versenden der Susi-Befehle über die entsprechende HW-Schnittstelle übernimmt das Output-Modul, das deshalb auch HW-abhängig ist.

`Beeper Control`

Dieses Modul erzeugt einfache Sound-Ausgaben, also im Wesentlichen synthetisch erzeugte Geräusche wie Horn, Pfeife o.ä. Diese Funktionalität erlaubt eine einfache Sound-Ausgabe für Nicht-Sound-Dekoder, die keinen Speicher für das Abspielen von aufgenommenen Sound-Samples haben.

Das Control-Modul wird wie eine Funktion angesprochen und hat eine entsprechende Schnittstelle zum Logical Function Bus.

`Beeper Output`

Das zum Beeper-Modul gehörende HW-abhängige Ausgabe-Modul.

Ob sich die beiden Module wirklich trennen lassen, ist noch unklar. Evtl. hängen die Sound-Möglichkeiten auch zu stark von der vorhandenen HW ab. In diesem Fall werden beide Beeper-Module zusammengefasst. Das Modul ist dann HW-abhängig.

`Sound Control`

Dieses Modul steht für die echte Sound-Ausgabe bei Implementierung eines waschechten Sound-Dekoders. Es entspricht bezüglich Einbindung, Aufgabe und Schnittstellen den einfachen Beeper-Modulen.

Erweiterung für die Zukunft. Es soll nur gezeigt werden, dass die Architektur entsprechend vorbereitet ist.

`Sound Output`

HW-spezifisches Ausgabe-Modul für die Sound-Ausgabe eines Sound-Dekoders. Hier könnte z.B. die Ausgabe an einen D/A-Wandler oder die Ansteuerung einer digitalen Class D Verstärker-Ausgabe implementiert werden.

2.8 Module: Sequenzsteuerung

Für bestimmte Funktionalitäten muss der Dekoder Abläufe (Sequenzen) ausführen können, z.B. um einen sogenannten „Kupplungswalzer“ zu vollziehen. Diese Sequenzen stellen sich als logische Funktionen dar und werden als solche gestartet (also in der Regel über das Mapping einer Steuerfunktion wie einer F-Taste oder einem Eingangssignal zugeordnet). Die Sequenz steuert wiederum selbst logische Funktionen an, eigene Hardware-Ausgänge oder -Funktionen bedient sie nicht. Welche logische Funktionen angesprochen werden, ist teilweise programmierbar.

`Sequence Boot`

Das Modul implementiert die Start-Sequenz. Diese sorgt dafür, dass alle Ausgänge nach dem Starten im gewünschten Zustand sind. Die Sequenz kann z.B. eine gespeicherte Richtung und Geschwindigkeit wiederherstellen.

Wie weit das Modul tatsächlich benötigt wird, muss sich erst erweisen. Es ist aber zumindest einmal vorgesehen.

`Sequence Auto Coupler`

Das Modul implementiert den sogenannten „Kupplungswalzer“. Die Lok entkuppelt hierbei und setzt ein kurzes Stück zurück. Mit Auslösen des Kupplungswalzers ist also eine komplexere Fahr- und Schaltsequenz verbunden.

Auf welcher generischen logischen Funktion (und damit auf welcher Ausgangsfunktion) die automatische Kupplung liegt, kann über CVs konfiguriert werden. Auch andere Parameter wie Geschwindigkeit, Dauer etc. können ggf. konfiguriert werden.

`Sequence User Defined`

Abläufe könnten nicht nur von den FW-Herstellern, sondern auch von Benutzern programmiert werden. Diese werden in einer einfachen Sprache definiert und im Dekoder interpretierend ausgeführt. Da sich solche Sequenzen kaum in ein paar CVs speichern lassen, werden die benutzerdefi-

nierten Sequenzen ähnlich wie Sounds in den Dekoder geladen und dort gespeichert.

Zur Definition der Sequenzen würde man vermutlich ein Programm für den PC brauchen, das die Sequenz bereits in eine binäre Zwischensprache übersetzt. Diese wird dann in den Dekoder geladen und vom Interpreter ausgeführt.

Dieses Sequenz-Modul implementiert diese Funktionalität, enthält also im Wesentlichen den Interpreter und den Speicher für die vom Benutzer definierten Abläufe.

Für dieses Modul müssen zunächst die Zwischensprache und die Ein- und Ausgangssignale, auf die die Sprache zugreifen kann, definiert werden. Außerdem ist die Implementierung des Moduls nur sinnvoll, wenn auch die notwendige Infrastruktur erstellt wird. Dazu gehört eine Sprache für den Benutzer und ein entsprechendes Umsetzungstool auf dem PC.

2.9 Busse: Control Function Bus (C-Bus)

Der Control Function Bus verteilt die vom Gleis oder einer anderen Quelle kommenden „Befehle“ (Steuerungsfunktionen) an die verschiedenen Datensenken für ungemappte Befehle.

Gedanklich kann man sich das so vorstellen: Auf dem Gleis wird ein Befehl von einer der verschiedenen Protokollauswerter erkannt. Dieser dekodiert den Befehl, z.B. einen Fahrbefehl, und gibt ihn in protokollneutraler Form über den Control Function Bus weiter. Alle interessierte Module holen sich diesen Befehl vom Control Function Bus und reagieren entsprechend. In diesem Fall würde er über das Mapping und den Logical Function Bus weitergeleitet zur Motor-Steuerung, die die Geschwindigkeit anpassen würde. Gegebenenfalls würde der Geschwindigkeitsbefehl über den Logical Function Bus an andere Module weitergegeben (z.B. die Sequenz-Module), wenn diese die Geschwindigkeitsbefehle benötigen. Allerdings darf man das nicht mit dem Zustand verwechseln (siehe 2.12).

2.10 Busse: Logical Function Bus (L-Bus)

Der Logical Function Bus verteilt in gleicher Weise wie der Control Function Bus die Steuerbefehle für die verschiedenen logischen Funktionen. Die logischen Funktionen werden in der Regel vom Mapping oder den Sequenz-Modulen aufgerufen (vgl. 2.8).

Der Aufruf einer Funktion erfolgt also in etwa wie folgt: Zunächst wird über die Gleissignalerfassung und Protokollauswertung das Schalten einer Funktion im Sinne der Zentrale erkannt und über den Control Function Bus an das Mapping gemeldet. Dieses bildet die Funktion auf eine oder mehrere logische Funktionen ab und erzeugt dementsprechend einen oder mehrere Schaltbefehle für logische Funktionen auf dem Logical Function Bus. Über diesen bekommt zum Beispiel der Function Manager den Auftrag, eine Lichtfunktion zu schalten.

2.11 Busse: Feedback Bus

Der Feedback Bus (Rückmelde-Bus) gibt die Aufforderung, eine Rückmeldung zu senden, an die entsprechenden Rückmeldemodule weiter. In der Regel ist dies lediglich ein Aufruf vom Protokoll-Erkennungsmodul an das zu diesem Protokoll zugehörige Rückmeldemodul. Dieser Aufruf steuert auch den Zeitpunkt der Rückmeldung, der meist durch die Zentrale und damit das Signal am Gleis vorgegeben wird.

Prinzipiell wäre es aber auch denkbar, dass bestimmte Rückmeldesignal asynchron vom Dekoder

versendet werden. Damit könnten Ereignisse zeitnah mitgeteilt werden. Ebenso wäre es möglich, dass von einem Gleisbefehl eines Protokolls die Rückmeldung in einem anderen Protokoll verwendet wird, also z.B. MM-Befehle per DCC-Acknowledge quittiert werden. Aus diesem Grund wurden die Rückmelde-Aufforderungen als Bus modelliert. Zunächst wird es sich aber im Wesentlichen um 1:1-Verbindungen zwischen Protokoll-Erkennung und zugehöriger Rückmeldung handeln.

2.12 Module/Busse: State

Das Mapping, aber auch die Sequenzen sind vom aktuellen Zustand des Dekoders abhängig. Das kann z.B. die Fahrtrichtung oder die tatsächliche Geschwindigkeit (Achtung: über den Logical Function Bus werden Befehle zur gewünschten Geschwindigkeit übertragen, der Zustand hingegen gibt die tatsächliche Geschwindigkeit an) sein.

`State`

Technisch werden die Zustände in globalen Variablen in diesem Modul abgelegt. In diesem Fall legen die Manager-Module dort die entsprechende Information ab, die von den interessierten Modulen bei Bedarf gelesen werden können.

Falls notwendig, kann auch ein „Push-Betrieb“ implementiert werden. In diesem Fall wird eine Änderung durch einen Funktionsaufruf direkt an die Abnehmer weitergegeben.

3 Prozesse

3.1 Übersicht

Als Prozesse werden hier die verschiedenen „Ausführungsfäden“ bezeichnet, die mehr oder weniger unabhängig voneinander ausgeführt werden. Das sind im Wesentlichen die Hauptschleife, also die eigentliche Programmausführung des Prozessors, und die verschiedenen Interrupts.

Ein Betriebssystem oder eine andere Art von Scheduler wird nicht verwendet, auch keine Threads oder Prozesse im Sinne von Betriebssystemen.

Es werden die folgenden Prozesse verwendet:

- **Hauptschleife** (Hauptprogramm)
- **Gleissignalerfassung** (Interrupt): Einen eigenständigen Prozess gibt es nur, wenn die Gleissignalerfassung über einen oder mehrere Interrupts gesteuert wird. Bei einem Polling-Ansatz geschieht die Gleissignalerfassung im Timer-Interrupt.
- **Timer-Interrupt**
- **Interrupt der Motor-Steuerung**
- **Susi-Interrupt:** Der Interrupt wird nach Ende einer Byte-Ausgabe von der SPI-Einheit des Prozessor ausgerufen, falls für Susi die SPI-HW des Prozessors verwendet wird. Es wird dann das nächste, anstehende Byte ausgegeben. Dieser Prozess entfällt, wenn die Susi-Ausgabe synchron im Hauptprogramm stattfindet (z.B. über eine reine Digital-Pin-Ansteuerung) oder im Timer erledigt wird.

Die Aufgaben der Prozesse ergeben sich aus dem Namen. Sie laufen in der Regel im entsprechenden Modul ab. Wenn sie Aufgaben in anderen Modulen aufrufen, so müssen die entsprechenden Funktionen in der Regel nicht re-entrant programmiert werden, weil diese Funktionen ausschließlich von diesem Prozess genutzt werden.

Beispiel: Der Timer-Interrupt ruft z.B. eine Funktion im Modul „Function Light“ auf, um dort ein Blinklicht zu realisieren. Diese vom Timer aufgerufene Funktion wird in „Function Light“ nur vom Timer-Interrupt aufgerufen, nicht aber z.B. vom Hauptprogramm. Sie muss deshalb nicht re-entrant sein.

4 Konzepte

4.1 HW-Unabhängigkeit und Portierbarkeit

Ein effizientes embedded System wie ein Lok-Dekoder muss aus den vorhandenen Ressourcen möglichst viel herausholen. Eine komplette HW-Abstraktion ist daher nicht möglich. Die Architektur unterscheidet jedoch sorgfältig zwischen HW-abhängigen und HW-unabhängigen Teilen. Letztere werden so klein wie möglich gehalten, sollten dabei aber einen sinnvollen Funktionsumfang abdecken. Bei einer Portierung der Firmware auf eine andere Dekoder-HW müssen die HW-abhängigen Module angepasst oder ausgetauscht werden.

Bietet die neue HW ähnliche Möglichkeiten wie die alte, reicht meist eine einfache Anpassung im Modul aus. Es werden dann meist nur die konkreten Ansteuerungen für die HW an die neue Plattform angepasst. Beispiel: Die SPI-Schnittstelle der unterschiedlichen Controller wird zwar über verschiedene Register konfiguriert, funktioniert im Grunde aber gleich. Deshalb muss im Modul Susi Output ggf. nur die konkrete Registerbelegung im Controller geändert werden.

Bietet die neue HW-Plattform aber weiter gehende oder weniger Möglichkeiten, muss der gesamte Ansatz gegen einen anderen, passenden ausgetauscht werden. Das Dimmen von Funktionsausgängen ist zum Beispiel sehr einfach über eine Änderung des PWM-Puls-Pausen-Verhältnisses zu realisieren, wenn der Controller über entsprechende PWM-Ausgänge verfügt. Muss die Funktionsausgabe aber über einen einfachen Digital-Pin erledigt werden, muss z.B. in der Firmware über den Timer-Prozess eine PWM nachgestellt werden. Das entsprechende Ausgabemodul ist dafür weitest gehend neu zu erstellen. Auch ein Mischbetrieb (ein Teil über PWM-HW, der andere Teil über eine Software-PWM) ist möglich. Dann würde man aber die unterschiedlichen Ansteuerungen auch in zwei unterschiedlichen Modulen unterbringen.

In der Entscheidung, welche Art von Ansteuerung man verwendet (z.B. HW-basiert oder in der SW emuliert), ist man weitest gehend frei. Allerdings sollte das Ziel sein, die verfügbare HW aus Performance-Gründen (und wenn es nur um eine gewisse Performance-Reserve für spätere Erweiterungen geht) soweit wie möglich zu nutzen.

4.2 Modularität und Erweiterbarkeit

Es gibt verschiedene Möglichkeiten, Funktionalitäten zu verändern und den Dekoder um neue, auch ganz eigene, zu erweitern:

1. Die wesentlichen Funktionen sind in eigene Module mit klarer Aufgabe und Schnittstelle gepackt. Diese Module kann man problemlos gegen andere Module ersetzen. So ließe sich z.B. das eigentlich vorgesehene, komplexe Mapping-Modul gegen ein ganz einfaches Map-

ping-Modul (festes 1:1-Mapping) für einen Anfängerdekoder ersetzen. Solche Einfachstmodule werden auch während der Entwicklung verwendet, wenn die komplexen noch nicht zur Verfügung stehen oder noch nicht die erforderliche Reife haben. Umgekehrt könnte man natürlich auch ein ganz anderes Mapping-Modul einsetzen, wenn jemand dazu eine interessante Idee hat.

2. Für bestimmte Funktionalitäten wie die Protokoll-Erkennung, die Rückmelde-Kanäle, die Funktionseffekte (Licht, Servo, Kupplung, …), die Motor-Ansteuerung usw. sind in der Architektur bereits mehrere Alternativen vorgesehen, die über CVs konfiguriert werden. Auch diese Module haben eine weitest gehend einheitliche Schnittstelle. Man kann auf einfache Weise weitere Module programmieren und in den Dekoder integrieren. Als Beispiel sei ein weiterer Motor-Ansteueralgorithmus genannt.

3. Die beiden „Befehlsbusse“ Control Function Bus und Logical Function Bus erlauben es, weitere Ein- oder Ausgabe-Module zur programmieren und in den Dekoder zu integrieren. Dabei müssen zwar ggf. auch andere Module wie das Mapping angefasst werden (um z.B. ein neues Ausgabe-Modul als Aktionssenke im Mapping zu etablieren). Dies sollte aber relativ einfach und geradlinig am Beispiel der vorhandenen Module erfolgen können.

4.3 Performance

Die Dekoder-Architektur ist ein Kompromiss zwischen Allgemeinheit (HW-Unabhängigkeit, Modularität, …) und Performance (Ausnutzung der vorhandenen HW-Ressourcen, Ablaufgeschwindigkeit durch direkte Funktionsaufrufe, …).

5 Technische Festlegungen

5.1 Programmiersprache

Die Firmware wird komplett in C geschrieben. Assembler sollte nicht verwendet werden, auch nicht in Hardware-abhängigen Teilen.

Die Prozessor-Hardware wird über die CMSIS-Bibliothek des Prozessors angesprochen. Wenn diese nicht ausreicht, kann prozessorspezifischer Code verwendet werden.

Die Hardware-Schnittstellen, die in der Regel nicht über CMSIS angesprochen werden können, werden, wenn möglich, über vom Hersteller zur Verfügung gestellte Bibliotheken angesprochen. Diese können durch eigene Bibliotheken ersetzt werden, falls dies notwendig ist.

5.2 Entwicklungsumgebung und Compiler

Die Entwicklung erfolgt mit CoIDE, Version ≥ 1.5 (www.coocox.org) Es gibt aber keine Abhängigkeiten im Code von dieser Entwicklungsumgebung. Der Code kann bei Bedarf auch mit anderen Tools bearbeitet werden. Für ein entsprechendes Makefile etc. muss dann aber separat gesorgt werden.

Als Compiler kommt ARM GCC 4.6 (download über CooCox-Seite bei Installation der CoIDE) zum Einsatz.

Für Programmierung und Debugging werden die Programmier-Adapter CoLinkEx oder NuLinkMe

verwendet. Die Adapter werden von CoIDE und von CoFlash direkt unterstützt.

5.3 Bibliotheken

Die folgenden Bibliotheken werden verwendet:

- C Library (CooCox)
- CMSIS M0 Core Library (CooCox)
- CMSIS M051 Boot Library (CooCox)
- Peripheral.Nuvoton System Definition (CooCox)
- Peripheral.Nuvoton SYS (CooCox)
- Peripheral.Nuvoton GPIO (CooCox)
- Peripheral.Nuvoton ADC (CooCox)
- Peripheral.Nuvoton FMC (CooCox)
- Peripheral.Nuvoton PWM (CooCox)
- Peripheral.Nuvoton TIMER (CooCox)

Die Bibliotheken müssen nicht für die Ansteuerung der Prozessor-Hardware und -Peripherie verwendet werden. Die entsprechenden Register können im C-Code auch direkt manipuliert werden. Die betreffenden Stellen müssen hinreichend kommentiert werden. Andere oder zusätzliche Bibliotheken sollten nicht verwendet werden.