Steuerung einer Modellbahnanlage mit dem Echtzeitbetriebssystem QNX

Falko Menge, Johannes Passing, Michael Perscheid

Hasso-Plattner-Institut für Softwaresystemtechnik

***Zusammenfassung*— Ziel der vorliegenden Arbeit ist zunächst, eine kurze Einführung in das Echtzeitbetriebssystem QNX zu geben sowie auf die Probleme des bestehenden Steuerungssystems der Eisenbahnanlage einzugehen. Es wird daraufhin das Ergebnis der von unser Projektgruppe durchgeführten Portierung und Neugestaltungdes Systems auf Basis von QNX vorgestellt. Schließlich werden die Resultate des durchgeführten Performanz-Vergleiches zwischen neuer und bestehender Implementierung gezeigt.**

***Stichworte*— QNX, Echtzeitbetriebssysteme, C++**

I. Einführung

DAS Projektseminar „Nicht-funktionale Eigenschaften eingebetteter Systeme“ beschäftigt sich mit verschiedenen Aspekten der Steuerung der DCL [1] Modellbahnanlage. Die vorliegende Ausarbeitung befasst sich mit der Schnittstelle zwischen dem verwendeten Märklin Digital-System und einem Steuerungs-PC und beschäftigt sich hierbei insbesondere mit der Problematik der Vermeidung von langen Reaktionszeiten beim Senden von Befehlen sowie bei Status-Rückmeldungen.

Aufgrund der etwas älteren Märklin Hardware besteht grundsätzlich nur ein geringer Daten-Durchsatz zwischen PC und dem Digital-System. Es sollte daher gewährleistet werden, dass die Bandbreite möglichst effizient genutzt wird und Befehle nicht zu lange auf ihre Ausführung warten müssen.

Bisher wurde die Anlage mit einem Dual-Pentium III-System mit Windows 2000 unter Einsatz einer in C# erstellten, .Net Remoting-basierten Steuerungssoftware betrieben.

Trotz dieser vergleichsweise leistungsfähigen Hardware ist beim Einsatz des bestehenden Systems eine hohe zeitliche Latenz bei der Ausführung von Befehlen feststellbar.

Noch deutlicher treten die Schwachstellen der Implementierung bei Betrachtung der Bearbeitung von Statusabfragen hervor. Um Statusmeldungen ermitteln zu können, muss die Steuerungs-Software in regelmässigen Abständen die in S88-Modulen vorgehaltenen Statusinformationen auslesen. Je häufiger dies geschieht, desto kürzer können die Reaktionszeiten auf Statusänderungen ausfallen. Das eingesetzte System ermöglichte jedoch nur relativ wenige solcher Abfragen pro Sekunde, welches zur weiteren Verschlechterung der Präzision der Steuerung führte.

Grund für die genannten, recht hohen Latenzen bei Befehlsverarbeitung und Statusabfragen sowie insbesondere der hierbei deutlich hervortretenden Varianz dieser Zeiten dürften nicht zuletzt der bei .Net eingesetzte Garbage Collector sowie weitere nebenläufig ausgeführte Windows-Dienste sein, die aufgrund der Tatsache, dass es sich bei Windows 2000 um kein Echtzeitsystem handelt, eine unverzügliche Befehlsarbeitung verhindern können.

Bei der Begutachtung der alten Implementierungen (C# .Net, C/C++) fiel es ferner schwer, einen roten Faden durch die Gesamtwerke zu finden. Neben zahllosen Magic Numbers[1] und der wenigen Dokumentation wirkten die Projekte durch geringe Abstraktionen sehr gestückelt und verteilt.

All diese Punkte ließen uns schließlich zu verschiedenen Lösungsansätzen kommen. Einfachster, zugleich aber teuerster Ansatz ist das Austauschen der Märklin Hardware, um leichter größere Durchsatzraten zu erzeugen. Diese Idee hat jedoch keine Relevanz, da sie neben den Kosten auch nicht alle Probleme zufriedenstellend löst, beispielsweise ist die Verringerung der primär Software-bedingten Reaktionszeiten fragwürdig.

So bleibt nur der Austausch der Software – einerseits wird das Betriebssystem zugunsten eines Echtzeitbetriebssystems gewechselt, andererseits die eigentliche Steuerungssoftware neu geschrieben. Echtzeitbetriebssysteme gelten als optimales Mittel, gleichmäßig kurze Reakionszeiten zu verwirklichen. Die mögliche Auswahl zwischen Windows CE und QNX wurde zu Gunsten des Letzteren entschieden. Da nun eine neue Grundlage für das System besteht, wird auch die Steuerungssoftware neu entwickelt. Dabei soll das Hauptaugenmerk auf einer möglichst effizienten Befehlsverabeitung und Status-Rückmeldung liegen, ohne dabei die Kompatibilität zu anderen Programmen zu verlieren, d.h. die API bleibt nach außen zumindest semantisch gleich.

II. QNX - Ein Echtzeitbetriebssystem

Im folgenden Abschnitt wird ein kurzer Blick auf das Echtzeitbetriebssystem QNX [2] geworfen. Von Echtzeitsystemen (Realtime Systems) spricht man, wenn auf ein Ereignis innerhalb eines festgelegten Zeitintervalls garantiert reagiert wird. Somit können auftretende Probleme durch stark variierende Reaktionszeiten von Befehlsausführungen weitestgehend vermieden werden. Man unterscheidet hierbei die „weiche“ (Die Einhaltung von Zeitanforderungen ist statisch definiert) und die „harte“ (Das Einhalten von Zeitpunkten ist garantiert) Latenzzeit.

A. QNX - Einführung

QNX ist ein Echtzeitbetriebssystem, welches von der kanadischen Firma QNX Software Systems Ltd. für verschiedene

[1] Bitbefehle für die Anlage

Plattformen entwickelt wird. Es ist ein 32-Bit Multiuser-System und unterstützt Multitasking, schnelle Kontextumschaltung und prioritätsgesteuertes, verdrängendes Scheduling. Ebenso wird der komplette POSIX Standard implementiert, wodurch die Portierung von Unix-basierter Software recht einfach zu vollziehen ist.

Die äußerst flexible Architektur erlaubt es ferner, sowohl große, verteilte Anwendungen, als auch sehr kleine, auf nur wenigen Kernelmodulen basierende Programme zu schreiben. Durch die zwei wesentlichen Konzepte (s. II-C), der Mikrokernel-Architektur und der nachrichtenbasierten Kommunikation, erreicht QNX eine hohe Effizienz, Modularität und Einfachheit.

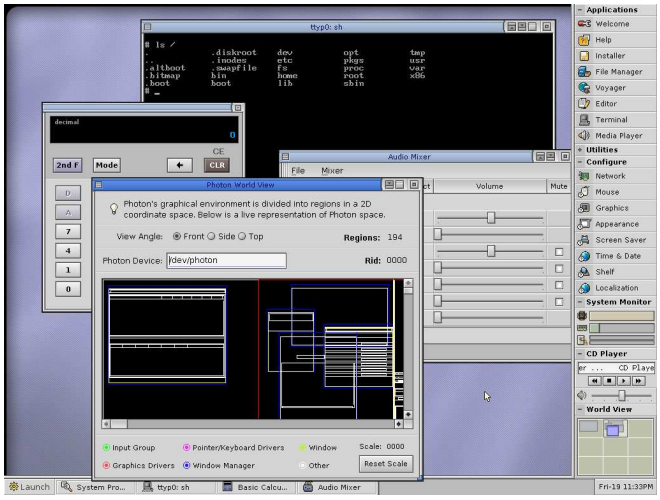

Abb. 1. QNX - Desktop

Abb. 1 zeigt die grafische Benutzeroberfläche (Photon microGUI s. II-C.6) von QNX.

B. QNX - Geschichte

Die Studenten der University of Waterloo Gordon Bell und Dan Dodge erschufen 1980 ihr eigenes Echtzeitbetriebssystem mit Mikrokernel. Mittels ihrer Firma „Quantum Software“ brachten sie 1982 die erste Version *QUNIX* für Intel 8088 CPUs heraus. Später wurde dieses System in *QNX* umgetauft.

Seit 1984 erhielt das Betriebssystem durch den Einsatz von namhaften Firmen (u.a. Hewlett-Packard, Siemens, Sony) nicht nur in Forschung und Lehre Einzug, sondern wurde zusätzlich im Markt für *Embedded Systems* bekannt.

Da der Markt sich Ende der 90er immer weiter an den POSIX-Modellen ausrichtete, wurde der Kernel neu überarbeitet. Das Ergebnis war QNX Neutrino (2001). Diese Version beinhaltet eine integrierbares GUI (Photon microGUI s. II-C.6), eine Entwicklungsumgebung (Momentics, basierend auf Eclipse) sowie Internetsoftware (Browser, Webserver). Dieser Meilenstein sorgte für den Namenswechsel der Firma in QNX Software Systems.

Neutrino wurde im Verlauf der Zeit auf viele Plattformen (z.B. x86, MIPS, PowerPC, SH-4), die im Embedded Markt Anwendung finden, portiert.

C. QNX - Architektur

QNX basiert auf der Grundidee, den Großteil des Systems in Usermode-Tasks auszugliedern. Die Mikrokernel-Architektur (s. II-C.1) erlaubt ferner dem Entwickler, nicht benötigte Tasks wegzulassen, wodurch ein sehr schlankes, ressourcensparendes System geschaffen werden kann. Die Module ergänzen je nach Bedarf den Mikrokernel z.B. um einen Prozess-Manager (s. II-C.2), Geräte-Manager (s. II-C.4), verschiedene Filesystem-Manager (s. II-C.3), Netzwerk-Manager (s. II-C.5), Photon microGUI (s. II-C.6). Alle Module, mit Ausnahme des Prozess-Managers, können während der Laufzeit gestartet oder beendet werden.

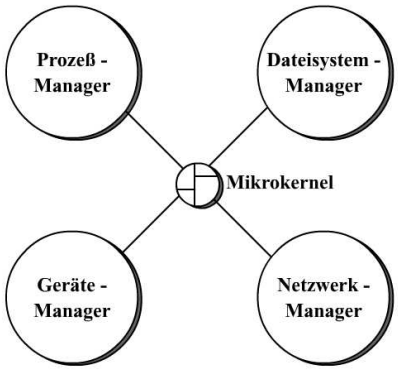

Abb. 2. QNX - Architektur

1) Mikrokernel: Der Mikrokernel ist hauptsächlich für die folgenden Aufgaben zuständig:

- Interprozesskommunikation – Austausch von Nachrichten zwischen Prozessen
- Low-Level Netzwerkkommunikation – Austausch von Nachrichten mit anderen Rechnern
- Scheduling – Zuteilung der Prozesse zur Ausführung
- First-Level Interruptbehandlung – Entgegennahme und Weiterleitung von Interrupts an einzelne Prozesse

Alle weiteren Dienste werden von eigenständigen Systemprozessen erbracht (s. Abb. 2 und folgende Abschnitte). Dabei besteht nahezu kaum ein Unterschied zu Anwendungsprogrammen, d.h. es gibt keine versteckten Schnittstellen und das Betriebsystem ist leicht erweiterbar. Somit verschwimmen die Grenzen zwischen Anwendung und Betriebssystem; lediglich einen zwingenden Unterschied gibt es: Betriebssystemdienste können Ressourcen für die Anwendungsprozesse verwalten.

Die einzige zulässige Möglichkeit, Daten zwischen Prozessen auszutauschen, besteht in der Interprozesskommunikation (IPC). Direkte Speicherzugriffe sind durch den Speicherschutzmechanismus nicht erlaubt. QNX unterstützt dabei Messages (synchrone Kommunikation), Signals (asynchrone Kommunikation) und Proxies (spezielle Form von Messages für Interrupts).

Der Scheduler wählt den nächsten zur Ausführung kommenden Prozess aus. Dies passiert immer dann, wenn ein Prozess den Status „blocked“ verlässt, der Zeitslot des aktuellen Prozesses verbraucht ist oder ein Prozess mit höherer Priorität den anderen Prozess verdrängt. Er entscheidet sich dann an

Hand des Status „ready“ und des Prozesses mit der höchsten Priorität. QNX unterstützt FIFO, Round-Robin und Adaptive Scheduling.

2) Prozess-Manager: Die Aufgaben des Prozess-Managers umfassen das Erzeugen, Laden, Verwalten der Ressourcen, Ausführen und Beenden von Prozessen. Um diese grundlegenden Dienste des Betriebssystems bereitzustellen, arbeitet der Prozess-Manager eng mit dem Mikrokernel zusammen. Zur Kommunikation mit anderen Prozessen benutzt er, wie üblich, das Message Passing des Mikrokernels. Dies hat beispielsweise den Vorteil, dass man über Rechnergrenzen hinweg eine Nachricht zum anderen Prozess-Manager schicken kann, um ein Programm zu starten.

3) Dateisystem-Manager: Der Dateisystem-Manager erlaubt mithilfe weiterer Prozesse das Verwalten von Dateien auf Disketten, Festplatten, Bandlaufwerken, Flash-EPROMS und der RAM-Disk. QNX unterteilt den Raum für Pfadnamen in Regionen mit Autoritäten. Jeder Prozess mit Datei-orientierten I/O-Diensten muss ein Präfix beim Prozess-Manager definieren, welches den Teil des Namespaces, den er verwalten möchte, benennt. Mittels der Präfixe wird ein Baum erstellt, der im Speicher eines jeden QNX Rechners existiert.

Sobald ein Prozess eine Datei öffnet, wird der Pfadname der Datei gegen den Präfixbaum geprüft, um so die `open()` Funktion an den entsprechenden I/O-Ressourcen-Manager zu senden.

4) Geräte-Manager: Mit dem Geräte-Manager ist das Ansprechen der Konsole und externer Peripherie möglich. Er bildet somit die Schnittstelle zwischen Prozessen und der Hardware. Diese Geräte befinden sich im I/O-Namespace /dev. QNX greift mit den üblichen Funktionen (`read()`, `write()`, `open()` und `close()`) darauf zu. Da für QNX ein Gerät nur einen Datenstrom darstellt, kann man sagen, dass der Geräte-Manager den Datenfluss zwischen einer Anwendung und dem Gerätetreiber steuert.

5) Netzwerk-Manager: Der Netzwerk-Manager erweitert das Nachrichtensystem, indem er direkt mit dem Mikrokernel kommuniziert und die IPC-Fähigkeit auf entfernte Rechner erweitert. Zu seinen Besonderheiten zählen erhöhter Durchsatz durch Lastausgleich der Kommunikationspfade, Fehlertoleranz durch redundante Verbindungen und Überbrückung zwischen QNX-Netzwerken.

6) Fenster-Manager: Beim Fenster-Manager handelt es sich um die Photon microGUI. Diese gilt selbst als ein grafischer Mikrokernel. Normale Fenstersysteme sind in Embedded Systems nicht praktikabel, da sie meist sehr viele Systemressourcen benötigen. So haben die Entwickler die Idee des Microkernels auf die GUI abgebildet, wofür aber die IPCs so klein und effizient wie möglich sein müssen. Nachdem nun der eigentliche Kernel um diese Funktionen erweitert wurde, konnte man einen grafischen „Mikrokernel“-Prozess entwickeln, welcher selbst nur fundamentale Schnittstellen anbietet. So werden alle weiteren grafischen High-Level Funktionalitäten in kooperative Prozesse ausgegliedert und mittels der performanten IPC-Mechanismen verbunden.

D. QNX - Echtzeit-Performance

Der Begriff „Echtzeit“ impliziert, dass auf ein Ereignis *sofort* reagiert wird. Jedoch muss auch ein Echtzeitbetriebssystem eine gewisse Rechenleistung und somit Zeit für ein auftretendes Ereignis für sich beanspruchen. Diese Latenzzeiten sind jedoch bei QNX auffallend gering. So erlaubt der Neutrino Kernel auf einem Pentium III-System die kontrollierte Steuerungszeit (Interruptverarbeitung) von 0,55 μs, in der Ereignisse registriert werden. Im Vergleich dazu reagieren Timesharing-Betriebssysteme (z.B. Windows NT 4.0) erst nach ca. 10 ms auf Ereignisse [4].

E. QNX - Zusammenfassung

Das Thema QNX ist ein wesentlich komplexeres Thema als es hier behandelt werden könnte. Darum sei für weiterführende Informationen auf [2] verwiesen. Auf dieser Seite ist es auch möglich, eine Version zur nicht-kommerziellen Nutzung herunterzuladen. Jedoch ist hier die Entwicklungsumgebung *Momentics* auf 1 Monat Nutzungsdauer beschränkt. Zusammenfassend lässt sich sagen, dass QNX ein sehr flexibles und effizientes Echtzeitbetriebssystem ist, welches neben den vielen schon genannten Vorteilen auch eine leichte Installation und flexible Konfiguration bietet. Somit ist es gerade in der Industrie für Embedded Systems eine hervorragende Grundlage.

III. Architektur des neuen Steuerungssystems

Um eine spätere Integration mit anderen Systemen zu erleichtern, wurde die zu erstellende Applikation in 3 Schichten zerlegt. Basierend auf QNX dient die unterste Schicht, als Schicht 0 bezeichnet, der Hardwareabstraktion. Sie nimmt Befehle von der darüberliegenden Schicht entgegen und bietet dieser via Callbacks die Möglichkeit an, über Statusänderungen notifiziert zu werden.

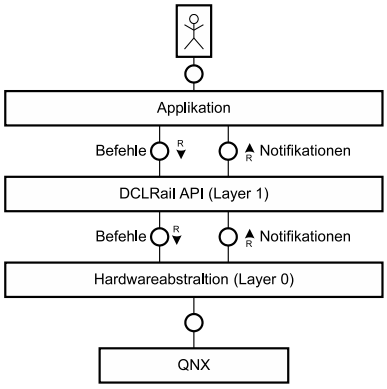

Abb. 3. Schichtenarchitektur

Auf Schicht 0 aufbauend, dient die Schicht 1 in erster Linie dazu, vom Märklin-Befehlsformat zu abstrahieren und eine einfach zu verwendende API bereitzustellen. Ferner ist diese Schnittstelle semantisch mit dem bestehenden System kompatibel, um eine eventuelle Migration zu erleichtern. Auch hier wird das Prinzip der Callbacks genutzt, um die darüberliegende Applikation über Statusänderungen zu informieren.

Basierend auf der von Schicht 1 bereitgestellten API bietet eine Shell-artige Konsolenapplikation auf Schicht 2 die Möglichkeit, mit der Eisenbahnanlage zu interagieren. Beweggrund zur Erstellung dieser Applikation war zunächst die Möglichkeit, das System interaktiv testen zu können. Jedoch hat sich die Shell als praktikables und komfortables Mittel zur Bedienung der Anlage herausgestellt, welches insbesondere bis zur weiteren Integration mit den bestehenden Werkzeugen nützlich ist.

Allen Schichten gemein ist die Erstellung in der Sprache C++, wobei die Schnittstellen zwischen den Schichten rein prozedural und C-kompatibel ausgelegt sind. Im Folgenden wird nun genauer auf jede der drei genannten Schichten eingegangen.

IV. Schicht 0 - Hardwareabstraktion

A. Aufgaben

Aufgabe dieser Schicht ist die Kommunikation mit der Eisenbahnanlage über eine RS-232 Schnittstelle mit Hilfe des von Märklin definierten Digital-Protokolls. Die von dieser Schicht bereitgestellte API nimmt Befehle im Märklin-Format entgegen und stellt, wie erwähnt, der darüber liegenden Schicht mittels Callbacks Statusinformationen zur Verfügung. Die Implementierung sollte dabei so gestaltet sein, dass sie von sämtlichen Implementationsdetails abstrahiert, welche die Kommunikation mit der seriellen Schnittstelle betreffen. Die Bearbeitung der einzelnen Befehle und Rückmeldungen sollte asynchron erfolgen, um das Blockieren der darüber liegenden Schichten zu verhindern. Ferner sollte die gleichzeitige Nutzung dieser API von mehreren Threads aus möglich sein.

Die Kommunikation zwischen dem Märklin Interface-Modul, welches die Schnittstelle zum Rest des Märklin Digital-Systems herstellt, und dem PC geschieht über ein einfaches, von der Firma Märklin definiertes binäres Protokoll, welches bei der relativ geringen Datenübertragungsrate von 2400 Baud verwendet wird. Zur Flusskontrolle wird der von der seriellen Schnittstelle bereitgestellte CTS-Mechanismus verwendet – sobald das Interface zum Empfang eines Befehls bereit ist, wird dies über das Setzen von CTS (Clear To Send) auf High mitgeteilt. Da Befehle sequentiell abgearbeitet werden und daher stets nur ein einziger in Bearbeitung sein kann, wird unmittelbar nach Empfang eines Befehls CTS auf Low gesetzt. Ist das Gerät zum erneuten Empfang eines Befehls bereit, wird dies durch erneutes Setzen von CTS auf High signalisiert.

Aufsetzend auf dieser Flusskontrolle wird ein einfach strukturiertes Protokoll verwendet. Vom PC aus werden bis zu zwei Byte lange Befehle an das Digital-System gesendet. Hierbei kann zwischen zwei Arten von Befehlen unterschieden werden – zum einen existieren Befehle, welche eine konkrete Aktion veranlassen, bspw. das Regeln der Geschwindigkeit einer Lokomotive oder das Stoppen des gesamten Systems. Diese Befehle enthalten stets die Information, welche Aktion zu erbringen ist sowie, falls erforderlich, welches Modul bzw. welche Lokomotive diese Aktion zu erbringen hat. Gemein ist diesen Befehlen ferner, dass sie weder eine Bestätigung noch eine Antwort auslösen; die Kommunikation verläuft in diesem Fall also unidirektional.

Die zweite Art von Befehlen bilden Status-Abfragen. Die auf der Anlage anfallenden Statusinformationen, wie etwa das Überfahren eines Kontaktgleises, werden in S88-Modulen vermerkt und zur Abfrage vorgehalten. Jedes Modul kann dabei bis zu 16 Statusbits aufnehmen. Mit Hilfe der Abfrage-Befehle lassen sich diese Informationen von einem oder mehreren solcher S88-Module ermitteln. Das Senden eines solchen Befehls löst eine Antwort aus, welche aus dem Speicherinhalt der abgefragten Module besteht – je Modul sind dies 16 Bit. Die Kommunikation erfolgt bei solchen Befehlen also bidirektional.

B. API

Um das erneute Auftreten von Magic Numbers vorwegzunehmen, enthält Schicht 0 eine Anzahl Makros, welche die Formatierung dieser Steuer-Befehle vornehmen. Als Beispiel soll hier der Zug mit der Nummer 2 auf Geschwindigkeit 5 beschleunigt und gleichzeitig die eingebaute Licht-Funktion aktiviert werden. Unter Verwendung jener Makros lässt sich das entsprechende Befehlswort bilden (s. Listing 1).

```
1 DWORD command = COMMAND_LOC_SETSPEED( 2, 5 )
2   | COMMAND_LOC_FLAG_FUNCTION;
```

Listing 1. Layer 0 - Beispiel zur Makro-Verwendung

Die darauf aufbauende, an höhere Schichten bereitgestellte API (Listing 2) beinhaltet eine Struktur (Zeile 1), zwei Typdefinition (Zeile 9 und 13) sowie vier Funktionen (Zeile 17, 24, 26 und 32), auf welche im Folgenden eingegangen wird.

```
1  typedef struct _STATUS {
2    union
3    {
4      WORD rgwStatus[ MAX_S88_COUNT ];
5      BYTE rgbStatus[ MAX_S88_COUNT * 2 ];
6    };
7  } STATUS, *PSTATUS;
8
9  typedef void ( *COMMAND_CALLBACK )(
10   IN RCSTATUS rcsCommandStatus,
11   IN DWORD dwCookie );
12
13 typedef void ( *STATUS_CALLBACK )(
14   IN STATUS newStatus,
15   IN STATUS oldStatus );
16
17 RCSTATUS RailInitialize(
18   IN DWORD dwIoAddress,
19   IN DWORD dwReserved,
20   IN DWORD dwMaxPollInterval,
21   IN DWORD dwS88count,
22   IN STATUS_CALLBACK pfnStatusCallback );
23
24 RCSTATUS RailFinalize();
25
26 RCSTATUS RailEnqueueCommand(
27   IN DWORD dwCommand,
28   IN BOOL fPutInFront,
29   IN OPTIONAL COMMAND_CALLBACK pfn,
30   IN OPTIONAL DWORD dwCallbackCookie );
31
32 RCSTATUS RailClearQueue(
33   OUT OPTIONAL DWORD *pdwElemCleared );
```

Listing 2. API Layer 0

Um größtmögliche Flexibilität zu wahren, ist die Schnittstelle von Schicht 0 rein prozedural und C-kompatibel, während die interne Implementierung objektorientiert aufgebaut ist. Entsprechend wird statt der Verwendung von Exceptions stets als Rückgabewert ein RCSTATUS verwendet, welcher bei

Werten bis zu 0xFFFF errno-Werten entspricht, bei größeren Werten anwendungsspezifische Fehlercodes enthält. Ein Wert von 0 (`RCSTATUS_SUCCESS`) indiziert jeweils die erfolgreiche Bearbeitung.

Die Initialisierung erfolgt über den Aufruf der Funktion `RailInitialize`, welcher die I/O-Adresse des zu verwendenden seriellen Ports, sowie die Anzahl der auf der Anlage installierten S88-Module übergeben werden muss. Da Schicht 0 die Statusermittlung übernimmt und dies Protokoll-bedingt durch Polling geschieht, kann durch Angabe des Parameters `dwMaxPollInterval` der maximal gewünschte Zeitraum in Millisekunden festgelegt werden, der zwischen zwei solcher Polls verstreichen darf. Ein geeigneter Erfahrungswert ist hierbei 100 ms. Je kleiner der gewählte Wert, desto geringer ist die Reaktionszeit auf Statusänderungen. Durch das vermehrte Polling kommt es jedoch bei Werten um etwa 50 ms zu dem Effekt, dass für das Verarbeiten sonstiger Befehle keine Bandbreite mehr zur Verfügung steht, wodurch die Anlage nicht mehr steuerbar wird. Je nach Einsatzzweck kann die Performanz des Systems also mit diesem Parameter an die Bedürfnisse angepasst werden. Im Parameter `pfnStatusCallback` wird ein Pointer auf eine Funktion übergeben, welche nach Durchführung eines Status-Polls aufgerufen wird. Dabei werden der alte und neue Status jeweils in einer STATUS-Struktur als Argumente übergeben. Neben der Konfiguration werden durch Aufruf von `RailInitialize` ferner Hilfs-Threads gestartet, auf die im Folgenden noch eingegangen wird. Analog werden durch Aufruf von `RailFinalize` Ressourcen freigegeben und die Hilfs-Threads beendet.

Intern werden Befehle in einer First In First Out (FIFO) Queue abgelegt, bevor sie in sequentieller Folge über die serielle Schnittstelle geleitet werden. Das Queuing dient dabei insbesondere dem Zweck der asynchronen Verarbeitung – der aufrufende Thread veranlasst durch Aufruf von `RailEnqueueCommand` lediglich die Erzeugung eines Queue-Eintrags. Die eigentliche Abarbeitung des Befehls geschieht jedoch auf einem der Hilfs-Thread. Der ggf. mithilfe der erwähnten Makros erstellte Befehl wird der Funktion über den Parameter `dwCommand` übergeben. Bei hoch-prioren Befehlen, wie etwa einem Nothalt, kann durch Setzen von `fPutInFront` das Einreihen des Befehls an die Spitze der Queue erzwungen werden, wodurch die Wartezeit des Befehls minimiert wird. Im Regelfall sollte auf die Benutzung dieser Möglichkeit jedoch verzichtet werden. Optional kann ein Callback-Funktionspointer übergeben werden. Wurde ein Befehl übermittelt, so wird, falls vorhanden, die hier übergebene Funktion zusammen mit dem übergebenen `dwCookie`-Wert aufgerufen.

Schließlich kann durch `RailClearQueue` die interne Queue vorzeitig geleert werden.

Als Beispiel (Listing 3) soll die Initialisierung des Systems, sowie das Stellen einer Weiche dienen:

```
// Initialisierung und Konfiguration
RCSTATUS rcs = RailInitialize(
   COM1_ADDRESS,        // I/O-Adresse
   0,                   // Reserviert
   100,                 // Poll-Intervall
   S88COUNT,            // # S88-Module
   StatusCallback );    // Callback
if ( RCSTATUS_SUCCESS != rcs )
{ /* ... */ }

// Weichenstell-Befehl absetzen
rcs = RailEnqueueCommand(
  COMMAND_SWITCH_CURVE(1), // Befehl
  FALSE,                   // Ende der Queue
  NULL,                    // kein Callback
  0 );                     // kein Callback
if ( RCSTATUS_SUCCESS != rcs ) { /* ... */ }

sleep( 5 );

// Ressourcen freigeben
rcs = RailFinalize();
if ( RCSTATUS_SUCCESS != rcs )
{ /* ... */ }
```

Listing 3. Layer 0 - Beispiel

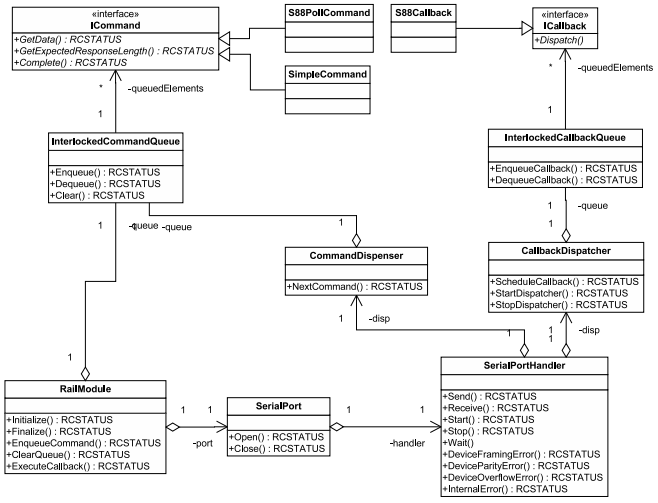

Abb. 4. Implementierung des Layer 0

C. Befehls-Vearbeitung

Sämtliche Befehle werden intern durch Objekte verkapselt, welche die Schnittstelle ICommand[2] implementieren. Die Klasse SimpleCommand wird für jene genannte Art von Befehlen verwendet, welche keine Rückantwort auslösen. Für Statusabfragen wird hingegen die Klasse S88PollCommand verwendet.

Wurde für ein durch Aufruf der Funktion `RailEnqueueCommand` übergebenen Befehl ein solches Command-Objekt erzeugt, wird dieses in der durch die Klasse InterlockedCommandQueue implementierten Queue abgelegt. Um den gleichzeitigen Zugriff von mehreren Threads zu erlauben, wird intern ein Mutex zur Synchronisation verwendet.

Da Status-Abfragen nicht durch den Benutzer der API veranlasst werden, sondern in regelmäßigen Abständen durch das System selbstständig durchgeführt werden, befinden sich keinerlei S88PollCommand-Objekte in der Queue. Um die regelmäßige Ausführung dieser Befehle sicherzustellen, existiert eine weitere Klasse CommandDispenser, von der, ebenso wie der Queue, nur eine einzige Instanz verwendet wird. Ist das System zur Ausführung eines Befehls bereit, so wird die

[2]Es werden Klassen mit ausschließlich pure abstract methods verwendet

Methode NextCommand() des CommandDispensers aufgerufen. Die Implementierung dieser Methode prüft daraufhin, ob seit dem letzten Statusabfrage-Befehl bereits das zulässige Zeitfenster verstrichen ist. Ist dies der Fall, wird der Dispenser ein S88PollCommand-Objekt erzeugen und zurückgeben, um eine Statusabfrage zu veranlassen. Ist der Zeitraum noch nicht verstrichen, wird der Queue ein SimpleCommand-Objekt entnommen und dieses dem Aufrufer zurückgegeben. Wenn die Queue jedoch leer ist, so wird die Zeit für eine erneute Status-Abfrage genutzt – es wird also ein S88PollCommand zurückgegeben.

D. Callback-Verarbeitung

Analog zu Befehlen wird je durchzuführendem Callback intern ein die Schnittstelle ICallback unterstützendes Objekt erzeugt. Während für Callbacks anlässlich Befehlsabarbeitungen die Klasse SimpleCallback verwendet wird, wird bei Status-Callbacks die Klasse S88Callback angewendet. Zur Abarbeitung der Callbacks wird ein separater Thread verwendet, um das Blockieren jenes Thread zu verhindern, der die Kommunikation mit der seriellen Schnittstelle abwickelt. Um diese asynchrone Abwicklung zu erlauben, existiert für Callbacks eine separate Queue, welche durch die Klasse InterlockedCallbackQueue implementiert wird.

Steht ein Callback zur Durchführung an, so wird, bspw. durch ein S88PollCommand veranlasst, ein Callback-Objekt in die Queue eingereiht. Auch hier wird durch Verwendung eines Mutexes der sichere parallele Zugriff mehrerer Threads ermöglicht. Die eigentliche Ausführung der Callbacks geschieht in der zentralen Instanz der Klasse CallbackDispatcher. Diese verwendet einen separaten Thread, der sequentiell der Queue ein Callback-Objekt entnimmt und die enstprechende Callback-Funktion mit den im Objekt abgelegten Parametern aufruft. Zur Unterstützung dieser Arbeitsweise bietet die Klasse InterlockedCallbackQueue in der Methode `DequeueCallback()` die Möglichkeit, den aufrufenden Thread (Thread des CallbackDispatchers) zu blockieren, bis entweder ein Callback bereitsteht, oder ein Timeout abgelaufen ist. Intern wird dies durch eine Semaphore realisiert. Durch Aufrufen der Methoden `CallbackDispatcher::StartDispatcher()` bzw. `CallbackDispatcher::StopDispatcher()` lässt sich der Hilfs-Thread jederzeit starten oder stoppen.

E. I/O

Die eigentliche Kommunikation über die serielle Schnittstelle ist in den Klassen SerialPort und SerialPortHandler implementiert. Deren Implementierung macht intensiven Gebrauch der durch die Klassen CommandDispenser und CallbackDispatcher bereitgestellten Dienste.

Eine frühzeitig getroffene Entwurfsentscheidung bzgl. der I/O-Kommunikation war der Verzicht auf einen Gerätetreiber, zugunsten der von QNX gebotenen Möglichkeiten, auch im Usermode auf Geräteregister zugreifen zu können. Entsprechend arbeitet der hier besprochene Teil der Schicht 0 direkt mit den vom Seriellen Port-Controller bereitgestellten Registern [6], sowie den hierzu von QNX bereitgestellten Funktionen, zu denen insbesonder `mmap_device_io()`, `in8()` und `out8()` zählen. Aufgabe der Klasse SerialPort ist das Öffnen und Schließen des Ports, sowie dessen Konfiguration mit Hilfe der verschiednenen Konfigurations-Register. Ferner wird ein separater Thread verwendet, der die eigentliche I/O-Verarbeitung übernimmt. Bis zu seiner Beendigung durch Aufruf von `SerialPort::Stop()`, implementiert dieser Thread die CTS-basierte Flusskontrolle, sowie den Empfang und das Senden der Daten.

Im Zusammenhang mit der CTS-Flusskontrolle stellte sich das Auslesen der Antworten, deren Länge vom gesendeten Befehl abhängen, mit Hilfe einer auf den Funktionen `InterruptAttachEvent()` und `InterruptWait()` basierten Implementierung als problematisch dar, sodass diese zugunsten eines einfachen Mechanismus ersetzt wurde.

In einer Schleife wird nun geprüft, ob das Line State Register mittels des Data Available-Bits anliegende Daten signalisiert. Ist dies der Fall, werden diese dem SerialPortHandler übergeben. Andernfalls wird durch Abfragen des CTS-Bits des Modem Status Registers, sowie des Empty Transmitter Holding Register-Bits des Line Status Registers die Sendebereitschaft abgefragt. Besteht diese, wird der SerialPortHandler angewiesen, Daten zu senden. Besteht weder Empfangs- noch Sendebereitschaft werden verschiedene Fehler-Register geprüft. Um einen busy wait zu verhindern, wird nicht sofort mit der nächsten Iteration begonnen, sondern, abhängig von einer Variable, eine gewisse Anzahl Millisekunden gewartet. Nach jeder Iteration, bei der Empfangs- oder Sendebereitschaft indiziert wurde, wird diese Variable auf 0 gesetzt, welches ein Warteintervall von 0 ms veranlasst. Wurde jedoch keine Bereitschaft festgestellt, so wird das Warteintervall um 1 ms erhöht, bis ein festgelegter Höchstwert erreicht wird.

Wie bereits angedeutet, wird die Abwicklung des Protokolls, also das Senden von Befehlen, Auslesen der Antworten, sowie Behandlung von Gerätefehlern von der Klasse SerialPortHandler übernommen. Die Sende-Funktionalität bedient sich hierbei insbesondere des CommandDispensers, um den nächsten zu sendenden Befehl zu ermitteln. Zu jeder Zeit hält der SerialPortHandler einen Pointer zu dem sich derzeit in Bearbeitung befindlichen ICommand-Objekt. Da Protokoll-bedingt je SerialPort-Iteration nur ein einziges Byte verarbeitet werden kann, ist dieses Objekt meist über mehrere Iterationen hinweg gültig. Wurden alle Befehls-Daten übertragen, sowie ggf. die Antwort vollständig empfangen, so wird die Methode `ICommand::Complete()` aufgerufen. Im Falle eines SimpleCommands wird dies, wenn gewünscht, einen Aufruf der bei `RailEnqueueCommand()` angebenen Funktion erwirken, welcher durch den CallbackDispatcher abgewickelt wird. Handelt es sich um einen S88PollCommand, so wird dessen Implementierung von `Complete()`, einen Aufruf der bei `RailInitialize()` angegebenen Status-Callback-Funktion in analoger Weise erwirken.

V. SCHICHT 1 - DCLRAIL-API

Zur Kapselung der Komplexität und Wahrung der Kompatibilität zur alten Programmierschnittstelle wurde, wie einführend erwähnt, die Zwischenschicht 1 entwickelt. Ihr Sinn ist es vor allem, die Details des Märklin-Protokolls zu verstecken. Darunter fallen die Abbildungen der Bus-Befehle auf

Funktionen und die eigentliche Ausführung mehrere Befehle eines logischen Zusammenhangs (z.B. Weiche stellen, Listing 4). Ein weiterer wichtiger Punkt ist die Auswertung der von Schicht 0 gemeldeten Fehler, welche entweder direkt behandelt oder codiert an die darüberliegende Schicht gereicht werden.

Bei Abfrage des Status werden die in Schicht 1 verwendeten Bitmasken auf für Programmierer freundlichere, zwei-dimensionale Arrays abgebildet. Dies geschieht in der von Schicht 0 regelmäßig aufgerufenen Callback-Prozedur (`internalCallBack()`), welche ferner die einzelnen Statuswerte nach S88 und Portstelle ordnet. Applikationen der nächsten Schicht können entweder wiederum eine Callback-Prozedur für Statusänderungen registrieren oder alternativ durch Aufruf der Funktion `getState()` den Inhalt des genannten Arrays abfragen, um die Daten entsprechend weiter verarbeiten zu können.

Listing 4 zeigt die Implementation der `setElement()` Funktion. Hierbei wird leicht erkenntlich, warum eine Kapselung sinnvoll ist. So ist es nun einfach möglich, über die ID, die Richtung und die Priorität der Anfrage eine Weiche zu stellen. Intern muss dazu die Priorität auf eine Queue-Position umgewandelt werden, d.h. jede Priorität > 0 wird an den Anfang der Befehlsqueue eingereiht, alle anderen am Ende. Das alte System sah an dieser Stelle mehrere Queues vor – um die Schnittstelle jedoch nicht zu verändern, wurde hier diese Abbildung vorgenommen. Als nächstes wird entsprechend der gewählten Stellrichtung der passende Weichenstell-Befehl (Zeile 12) abgesetzt. Dazu werden auf dieser Schicht die C-Makros und die zusätzlichen Parameter entsprechend gesetzt. Wichtig beim Stellen einer Weiche ist es, darauf zu achten, dass der Strom am Ende wieder ausgestellt wird. Dies geschieht hier automatisch (Zeile 24) und ohne Priorität, so wird gewährleistet das der Befehl immer nach dem Umschalten ausgeführt wird.

Am Ende (Zeile 27) findet, wie schon beschrieben, noch die Befehlsauswertung statt und wird als Boolean nach oben gereicht.

```
1  bool setElement(int elementid,
2                  bool straightOrGreen, int priority = 0)
3  {
4    unsigned int railControlStatus(0);
5    bool putInFront(false);
6
7    if (priority > 0) putInFront = true;
8
9    if (straightOrGreen == true)
10   {
11     railControlStatus =
12       RailEnqueueCommand(COMMAND_SWITCH_STRAIGHT(elementid),
13                          putInFront, NULL, NULL);
14   }
15   else
16   {
17     railControlStatus =
18       RailEnqueueCommand(COMMAND_SWITCH_CURVE(elementid),
19                          putInFront, NULL, NULL);
20   }
21
22   // Always turn off at the end of the queue
23   railControlStatus +=
24     RailEnqueueCommand(COMMAND_SWITCH_RESET,
25                        false, NULL, NULL);
26
27   if (railControlStatus == 0)
28   {
29     return true;
30   }
31   else
32   {
33     return false;
34   }
35 }
```

Listing 4. Layer 1 - setElement

Die Namen und Parameter der Funktionen orientieren sich am bestehendem System, darum sei mit dem Listing 5 noch einmal die komplette Schnittstelle beschrieben. Mit diesen Funktionen ist es nun möglich, alle notwendigen Befehle zur Steuerung abzusetzen und die alten Programme der darüber liegenden Schichten leicht zu portieren.

```
1  bool initDCLRail(
2      unsigned long dwS88count,
3      DCLRAIL_STATUS_CALLBACK pfnDCLRailStatusCallback = NULL,
4      unsigned long dwMaxPollInterval = 100,
5      unsigned long dwIoAddress = 0x3f8)
6
7  bool closeDCLRail()
8
9  bool go(int priority = 0)
10
11 bool stop(int priority = 0)
12
13 bool setResetMode(bool on = true, int priority = 0)
14
15 DCLRAIL_STATUS getState()
16
17 bool setElement(int elementid,
18                 bool straightOrGreen, int priority = 0)
19
20 bool setTrain(int trainID, int speed,
21               bool function_on = true, int priority = 0)
22
23 bool changeTrainDirection(int trainID,
24                           bool function_on = true,
25                           int priority = 0)
```

Listing 5. Layer 1 - Schnittstelle

VI. Schicht 2 - interaktive Steuerung mit der DCLRail-Shell

Mit Schicht 1 steht nun eine komfortable Programmierschnittstelle zur Verfügung, um Steuerungssysteme für die Eisenbahnanlage unter QNX zu entwickeln. Doch man möchte nicht immer gleich ein Programm schreiben und kompilieren müssen, um die Anlage mit dem Computer zu steuern. Schnell entsteht der Wunsch, auch interaktiv auf das Geschehen Einfluss zu nehmen. Dazu entwickelten wir eine Kommandozeilen-Shell für QNX, mit der man direkt Steuerbefehle an die Züge und Weichen senden kann.

Die Shell basiert auf der von Schicht 1 bereitgestellten Programmierschnittstelle und nutzt zusätzlich die GNU Readline Bibliothek [8], welche die Eingaben des Benutzers von der Konsole empfängt und dabei viele zusätzliche Funktionalitäten wie zum Beispiel eine Befehls-Historie oder automatische Vervollständigung bereitstellt.

A. Konfigurationsmöglichkeiten

Um sich an die aktuell gesteuerte Modellbahnanlage anpassen zu können benötigt die Shell einige technische Informationen. Dazu zählen zunächst die E/A-Adresse des seriellen Ports, über den die Anlage mit dem Computersystem verbunden ist, und die Anzahl der S88-Controller für Sensormeldungen. Diese Informationen können mit den Kommandozeilenparametern

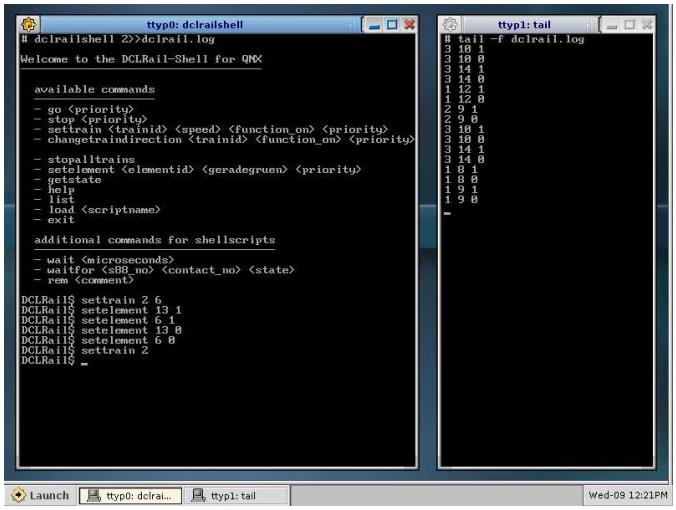

Abb. 5. DCLRail-Shell für QNX

`-i` und `-s` beim Starten der Shell übergeben werden, die sie dann über Schicht 1 nach unten zu Schicht 0 weiterreicht. Weiterhin bieten es die unteren Schichten an ein Zeitintervall anzugeben in dem periodisch zusätzliche Statusabfragen gemacht werden. Diese Funktionalität wird jedoch für die Shell nicht benötigt, weil sie nur sequentiell Befehle sendet und beim Warten auf einen Sensor mit `waitfor` (siehe Abschnitt VI-B) ohnehin die Queue in Schicht 0 leer bleibt. Daher ist der Standardwert für den dazugehörigen Parameter `-p` auf 0 festgelegt. Mit dem Kommandozeilenparameter `-f` kann ein nicht verwendete Zug-ID angegeben werden um die Sperre der IDs nach der Verwendung in der Shell wieder aufzuheben. `-s` weist die Shell dazu an, Sensormeldungen in die Standardfehlerausgabe zu schreiben. Eine typische Nutzung dieser Informationen besteht darin, diesen Stream mit `dclrailshell -s 2>> dclrail.log`in eine Logdatei umzuleiten. Diese kann dann sehr einfach mit einem zweiten Terminal via `tail -f dclrail.log` verfolgt werden, wie auch in Abb. 5 gezeigt wird. Detailierte Informationen zu allen Konfigurationsmöglichkeiten erhält man, indem man die Shell mit `dclrailshell -h` aufruft.

B. Shell-Kommandos

Zu dem Steuerungssystem unter Windows gibt es einen Simulator, welcher über eine ähnliche, wenn auch nicht so mächtige, Shell verfügt. Bei der Entwicklung der Shell für das QNX-Steuerungssystem wurde darauf geachtet, dass die Kommandos abwärtskompatibel zu denen der Simulatorshell sind (siehe Listing 6). Dies macht den Umstieg für die Nutzer beider Systeme einfacher und ermöglicht prinzipiell sogar Shell-Skripte mit dem Simulator zu testen und dannach auf der echten Hardware auszuführen.

```
LOAD scriptname [scriptname]
GO priority
STOP priority
SETRESETMODE on pririty
GETSTATE
SETELEMENT elementid geradegruen priority
SETTRAIN trainid speed function_on priority
CHANGETRAINDIRECTION trainid function_on priority
INSERTTRAIN trainid elementname
REMOVETRAIN trainid
REMOVEALLTRAINS
```

Listing 6. Befehle der bestehenden Simulatorshell

Zum Vergleich die Übersicht der Kommandos beim Starten der Shell für QNX:

```
Welcome to DCLRail-Shell for QNX
-------------------------------

  current configuration
  ---------------------
  serial port I/O addess: 0x3f8
  S88 controllers:        4
  polling intervall:      disabled
  print sensor changes:   enabled
  free train id:          80

  available commands
  ------------------
- settrain <trainid> <speed> <function_on> <priority>
- changetraindirection <trainid> <function_on> <priority>
- setelement <elementid> <straightorgreen> <priority>
- stop <priority>
- go <priority>
- stopalltrains
- getstate
- getconfig
- list
- load <scriptname>
- save <scriptname>
- help
- exit

  additional commands for shellscripts
  ------------------------------------
- wait <milliseconds>
- waitfor <s88_no> <contact_no> <state>
- rem <comment>

DCLRail$ _
```

Listing 7. Befehle der Shell für QNX

Im folgenden werden die einzelnen Kommandos die in der Shell zur Verfügung stehen vorgestellt. Bei einigen werden auch technische Lösungen erklärt, die beim Entwurf eigener Steuerungssysteme auf Basis der DCLRail API wiederverwendet werden können.

1) settrain <trainid> <speed> <function_on> <priority>: Das wohl wichtigste Kommando für die Steuerung der Eisenbahnanlage ist `settrain`. Es basiert auf der von Schicht 1 angebotenen Funktion `setTrain()` und mit ihm wird der Zug mit der ID `<trainid>` auf die Geschwindigkeit `<speed>` beschleunigt. Mögliche Werte für die Geschwindigkeit sind ganze Zahlen von 0 bis 15, wobei der Wert 15 den Zug anhält und die Fahrtrichtung umkehrt und damit äquivalent zum Kommando `changetraindirection` ist. Ist der dritte Parameter `<function_on>` auf den Wert 1 gesetzt, wird die eingebaute Zusatzfunktion des Zuges (typischerweise Licht) aktiviert. Bei einem `<priority>`-Wert größer als Null, wird das Kommando an erster Position in die Queue von Schicht 0 eingereiht, um vor allen anderen Befehlen, die noch in der Warteschlange stehen, abgearbeitet zu werden. Alle vier Parameter des Befehls sind optional. Je nach dem wie viele der Parameter vom Benutzer weggelassen werden, trifft die Shell unterschiedliche Annahmen über das gewünschte Verhalten. Der Standardwert für `<priority>` ist Null. Die Zusatzfunktion wird standardmäßig aktiviert, wenn `<function_on>` nicht explizit angegeben wird. Man kann kann einen Zug anhalten indem man die Geschwindigkeit auf

0 oder sie einfach weglässt. Dabei wird die Zusatzfunktion des Zuges deaktiviert. Gibt man keinen der Parameter von `settrain` an, so werden alle Züge angehalten, wie beim Kommando `stopalltrains`.

2) changetraindirection <trainid> <function_on> <priority>: `changetraindirection` hält den Zug mit der ID `<trainid>` an und kehrt seine Fahrtrichtung um. Danach muss der Zug mit `settrain` wieder beschleunigt werden. Die Parameter `<function_on>` und `<priority>` haben die gleiche Bedeutung wie bei `settrain` mit der Ausnahme, dass der Standardwert für `<function_on>` hier 0 ist. Der Befehl nutzt die API-Funktion `changeTrainDirection()`.

3) setelement <elementid> <straightorgreen> <priority>: Mit `setelement` kann die Weiche oder das Signal mit der ID `<elementid>` in den Zustand geschaltet werden, der durch den Parameter `<straightorgreen>` beschrieben wird. Der Wert 1 bedeutet für Weichen die Stellung „gerade“, während 0 den Zustand „rund“ identifiziert. Ein Signal wird bei 1 auf grün und bei 0 auf rot geschaltet. Nach dem Umschalten des Elementes muss mit einem zweiten Hardwarebefehl der Strom wieder abgeschaltet werden. Dies erledigt ebenfalls schon die Funktion `setElement()` die aus Schicht 1 verwendet wird. Da jedoch zur Zeit in Schicht keine atomare Operation vorgesehen ist um zwei Hardwarebefehle gleichzeitig in die Queue einzufügen, hat der Parameter `<priority>` bei diesem Shell-Kommando keine Funktion. Er existiert nur aus Gründen der Kompatibilität mit dem entsprechenden Befehl der Simulatorshell.

4) stop <priority>: `stop` sendet einen Nothaltbefehl an die Eisenbahnanlage der den kompletten Strom abschaltet. Bei einem `<priority>`-Wert größer als Null, wird das Kommando vor allen anderen Befehlen in der Warteschlange abgearbeitet.

5) go <priority>: Das Kommando `go` gibt nach einem Nothalt die Anlage wieder frei und reaktiviert die Stromversorgung.

6) stopalltrains: Mit `stopalltrains` können auf einfache Weise alle Züge angehalten werden ohne gleich einen Nothalt auslösen zu müssen. Da alle 80 Zug-IDs sequentiell einen `setttrain`-Befehl gesendet bekommen, dauert dies allerdings länger als mit einem einzigen `stop`-Befehl das System anzuhalten. Im Notfall sollte also `stop` bevorzugt werden.

7) getstate: Diese Kommando existiert hauptsächlich aus Kompatibilitätsgründen. Es gibt den aktuellen Zustand der Statusbits aller angeschlossenen S88-Controller auf die Konsole aus. `getstate` ist für die echte Hardware relativ nutzlos, da die Controller mehrmals pro Sekunde von Schicht 0 abgefragt werden und damit die Sensor-Meldungen nur extrem kurz zu sehen sind. Für die Beobachtung der echten Eisenbahnanlage sei der Kommandozeilen-Parameter `-s` empfohlen, mit dem Sensormeldungen direkt über die Standardfehlerausgabe ausgegeben werden können.

8) getconfig: `getconfig` gibt die aktuelle Konfiguration der DCLRail-Shell auf die Konsole aus.

9) list: Den Inhalt der Befehlshistorie kann man sich mit `list` anzeigen lassen.

10) save <scriptname>: `save` speichert den Inhalt der Befehlshistorie in die Datei `<scriptname>`. Wird der Parameter weggelassen wird der zuletzt mit `load` oder `save` verwendete Dateiname verwendet. Für die Eingabe des Pfades wird eine automatische Vervollständigung über die Tabulatortaste angeboten. Eine solche Datei kann mit `load` wieder eingelesen und abgearbeitet werden.

11) load <scriptname>: `load` liest Shell-Befehle zeilenweise aus der Datei `<scriptname>` oder, falls kein Pfad angegeben, aus der zuletzt mit `load` oder `save` verwendeten Datei und führt sie aus. Auch hier gibt es die automatische Vervollständigung des Dateinamens über die Tabulatortaste. Besonders interessant für solche Shell-Skripte sind die Anweisungen `wait`, `waitfor` und `rem`.

12) wait <milliseconds>: Das `wait`-Kommando ist sinnvoll für Shell-Skripte, denn damit kann die Ausführung des nächsten Befehls um `<milliseconds>` Millisekunden verzögert werden.

13) waitfor <s88_no> <contact_no> <state>: `waitfor` wartet mit der Abarbeitung des nächsten Befehls eines Shell-Skriptes solange, bis der Sensor `<contact_no>` am S88-Controller `<contact_no>` den Wert `<state>` angenommen hat. Um diese Funktionalität zu ermöglichen registriert die Shell bei Schicht 1 beim Aufruf von `initDCLRail()` eine Callback-Funktion, welche dann über alle Sensormeldungen informiert wird. Beim Aufruf von `waitfor` wird das durch die Parameter definierte Sensor-Ereignis in globalen Variablen gespeichert und auf ein Semaphor gewartet, das von der Callback-Funktion freigesetzt wird, sobald das gewünschte Ereignis eingetroffen ist. Dieser Mechanismus kann auch für die Ereignisbehandlung in eigenen Steuerungssystemen verwendet werden. Dabei wäre es dann sinnvoll eine Liste von erwarteten Ereignissen zu verwenden, um auf mehrere Sensoren gleichzeitig warten zu können. Für die Shell war dies noch nicht notwendig, da sie nur sequentiell Befehle abarbeiten kann und keine Verzweigungen im Kontrollfluss unterstützt.

14) rem <comment>: Dieser Befehl hat keine Funktion für die Steuerung der Anlage. Er kann benutzt werden um Shell-Skripte mit Kommentaren zu versehen oder um darin Befehle vorübergehend zu deaktivieren.

15) help <command>: Mit `help` kann man die Online-Hilfe zu einem Befehl `<command>` aufrufen. Eine Kurzübersicht der Befehle und ihrer Parameter erhält man, wenn man den Parameter `<command>` weglässt. Ein Aliasname für `help` ist `?`.

16) exit: Das Verlassen der Shell ist mit `exit`, `quit`, `q` oder der Tastenkombination `Strg + D` möglich. Dabei wird die Verbindung zur Hardware mit `closeDCLRail()` beendet.

C. Praxiserfahrungen mit der Shell

Mit diesem Befehlssatz können alle Funktionen der Eisenbahnanlage genutzt werden und die realisierten Komfortfunktionen stehen typischen Betriebssystemshells in nichts nach. Vor allem der Einsatz von Shell-Scripten ermöglicht es sehr schnell einfache Steuerungsalgorithmen zu entwickeln

und interaktiv testen, denn es ist keine Kompilierung der Steuerungsprogramme nötig. Ein Beispiel für ein einfaches Steuerungsskript zeigt Listing 8.

```
rem  initial assumtions:
rem  train 1 is at sensor 0-7/9
rem  current direction of the train is clockwise

changetraindirection 1
settrain 1 12
waitfor 3 15 1
settrain 1 0 0 1

wait 10000
changetraindirection 1
settrain 1 12
wait 5000
settrain 1
```

Listing 8. Shell-Skript für QNX

Trotz des verwendeten Interpreterprinzips ergaben sich verglichen mit Programmen auf Basis von Schicht 0 oder 1 keine großen Performance-Verluste, welche die Echtzeitfähigkeit des Systems hätten gefährden können. Einziger Nachteil bleibt der begrenzte Sprachumfang der nur Stapelverarbeitung von Befehlen zulässt. Es allerdings auch nicht das Ziel eine strukturierte Programmierung in die Sprache einzuführen. Um strukturierte Programmierung auf Basis eines Interpreters zu ermöglichen, sollte eher eine bestehende Skriptsprache mit einer DCLRail-Bibliothek erweitert werden.

VII. Performance Analyse

Nachdem wir nun ein komplettes Steuerungssystem auf der Basis eines harten Echtzeitbetriebssystems realisiert hatten, stellte sich die Frage, wie gut das neue System den Echtzeitanforderungen gerecht wird. Vor allem der Vergleich zu der auf .NET basierenden Lösung war von besonderem Interesse. Um dies zu untersuchen, entwickelten wir ein Experiment, das die beiden Steuerungssysteme bezüglich ihrer Reaktionszeit bei der Prozess-Steuerung evaluieren sollte.

A. Versuchsaufbau

Bei einem Bremstest sollten beide Computersysteme zeigen, wie schnell sie auf eine Sensormeldung mit einem Steuerbefehl reagieren können. Der Versuchsaufbau bestand darin, einen Zug mit Höchstgeschwindigkeit über einen Sensor fahren zu lassen. Die Computersysteme hatten jeweils die Aufgabe den Zug anzuhalten, sobald die Sensormeldung bei ihnen eintraf. Das C++-Programm, das für die Tests mit QNX verwendet wurde, ist in Listing 9 gezeigt. Ein Shell-Skript mit einer etwas erweiterten Funktionalität war bereits in Listing 8 zu sehen.

```
#include <dclrailcontrol.h> // Layer0
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define S88COUNT 4
#define COM1_ADDRESS  0x3f8

void TestStopOnContactStatusCallback(
    STATUS status,
    STATUS oldStatus
)
{
  if ( status.rgwStatus[ 3 ] & 4 )
  {
    RailEnqueueCommand(
        COMMAND_LOC_SETSPEED(1,0),
        FALSE,
        NULL,
        0
    );
    RailEnqueueCommand(
        COMMAND_LOC_SETSPEED(12,0),
        FALSE,
        0,
        0
    );
  }
}

void TestStopOnContact(int speed)
{

  RailInitialize(
    COM1_ADDRESS,
    0,
    70,
    S88COUNT,
    TestStopOnContactStatusCallback );

  RailEnqueueCommand(
    COMMAND_LOC_SETSPEED(1,speed),
    FALSE,
    NULL,
    0 );

  sleep( 15 );

  RailFinalize();
}

int main(int argc, char *argv[])
{
  if (argc == 2) {
    TestStopOnContact( atoi( argv[1] ) );
  } else {
    TestStopOnContact( 10 );
  }
  return 0;
}
```

Listing 9. C++-Programm für die Tests mit QNX (Als Kommandozeilen-Parameter kann die Geschwindigkeit übergeben werden)

Getestet wurde mit zwei verschiedenen Geschwindigkeiten (12 und 14). Auf der .NET-Seite wurde zum einen direkt im Server ein Test-Programm integriert und anderen in der über Remoting angebundenen Client-Anwendung. Es wurden jeweils zehn Messungen mit dem gleichen Zug auf dem gleichen Sensor durchgeführt.

B. Ergebnisse

In Tabelle I sind die Testergebnisse gegenübergestellt.

Die .NET Client-Anwendung erzielte katastrophale Testergebnisse, denn sie brachte den Zug durchschnittlich nach 76 Zentimetern zum stehen, was mehr als der doppelte QNX-Bremsweg von knapp 36 Zentimetern ist. Dazu kam jedoch noch eine extrem hohe Varianz der Ergebnisse. Die verschiedenen Werte lagen fast einen Meter auseinander und bei diesen zehn Messungen kam der Zug im Extremfall erst nach eineinhalb Metern zum Stehen. Der .NET Server kam zwar im Mittel näher an die Reaktionszeiten vom QNX-System heran, jedoch ist auch hier die Varianz sehr hoch. Die QNX-Lösung hingegen zeigte sich als äußerst präzises Steuerungssystem. Bei Höchstgeschwindigkeit wurde eine Lok, die selbst ca. 20 Zentimeter lang ist, nach durchschnittlich 40 Zentimetern zum Stehen gebracht. Vor allem war hierbei die Varianz sehr gering. Die Werte schwankten maximal in einem Bereich von

TABELLE I
PERFORMANCE-VERGLEICH

	.Net		QNX	
Messung	Remoting Speed = 12 / STOP	Server Speed = 14 / STOP	Speed = 12 / 0	Speed = 14 / STOP
1	68	36	35,7	39
2	81	56,5	35	39
3	104,5	50	36,5	38,5
4	48,6	42	34,3	40
5	61,5	57,8	34,5	39
6	69,5	45,5	36	41,5
7	64,6	39,9	37	40
8	142	41,8	36	43
9	49	48,9	35	38,5
10	69,5	51,5	36,5	41,5
Mittelwert	75,8	47,0	35,7	40,0
Max-Min-Abstand	93,4	21,8	2,7	4,5
Varianz	795,7	51,7	0,8	2,3

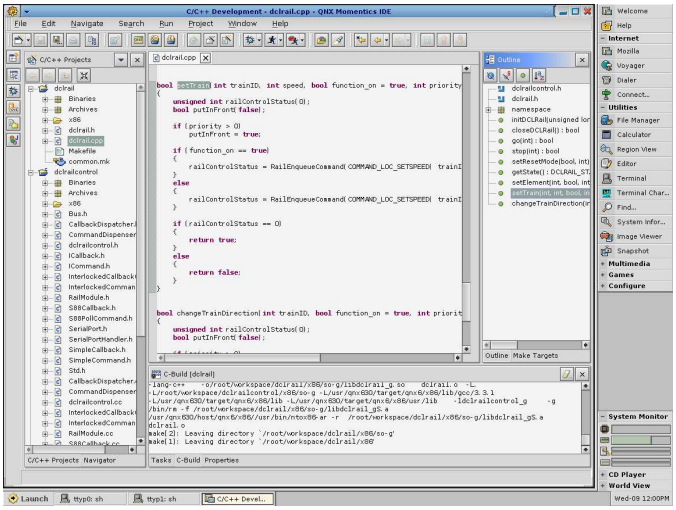

Abb. 6. Die QNX Momentics IDE

4,5 Zentimetern, was möglicherweise sogar auf mechanische Einflüsse zurückzuführen ist. Dabei ist zu beobachten, dass bei geringerer Geschwindigkeit die Varianz abnimmt und die Steuerung damit noch exakter wird. Unsere umfangreichen Praxiserfahrungen mit dem QNX-Steuerungssystem lassen vermuten, dass es sich bei den erreichten Werten um das Optimum handelt und dass die Reaktionsgeschwindigkeit nur noch durch die Märklin-Hardware beschränkt ist.

Zu den Ergebnissen muss man anmerken, dass die .NET-Steuerung hardwareseitig große Vorteile hatte, denn sie konnte symetrisches Multiprocessing auf einem Server mit zwei 1GHz-Pentium III Prozessoren nutzen, während das QNX-System nur über einen 800MHz-Prozessor verfügte.

VIII. ZUSAMMENFASSUNG UND AUSBLICK

QNX ist ein sehr interessantes Betriebssystem, das im Hinblick auf den Einsatz für eingebettete Systeme sehr komfortabel ist und einen relativ leichten Einstieg ermöglicht. Die Entscheidung für QNX zum Steuern der Modellbahnanlage hat sich als gute Wahl erwiesen. Das neue Steuerungssystem kann durch die besondere Architektur des Betriebssystems erstmalig die gestellten Echtzeitanforderungen erfüllen. Damit wurde das Ziel des Projektes, ein Computersystem zu realisieren, dass die Anlage präzise steuern kann, erreicht. Die entwickelte Lösung baut auf einer zukunftsfähigen Schichten-Architekur auf, die bereits die arbeitsteilige Entwicklung erleichtert hat und für zukünftige Projekte viele Erweiterungsmöglichkeiten anbietet. Die beiden unteren Schichten abstrahieren vollständig von der Hardware und lassen sich als DCLRail-API sehr einfach als dynamische Bibliotheken in eigene Steuerungssysteme integrieren. Mit der DCLRail-Shell ist ein sehr komfortables Werkzeug für interaktive Demonstrationen und schnelles Testen von Steuerungsalgorithmen entstanden.

ANHANG I
ANLEITUNG FÜR DIE QNX MOMENTICS IDE

Die integrierte Entwicklungsumgebung die QNX bereitstellt ist die Momentics IDE. Sie basiert auf Eclipse 2.0 und bietet umfangreiche Funktionen die speziell auf den Einsatzbereich des Echtzeitbetriebssystems ausgelegt sind.

Obwohl die Momentics-Projekte im Sourcecode-Release enthalten sind, möchten wir hier dennoch genau erklären wie die Projekte konfiguriert und miteinander integriert werden, um den Einstieg für die Anpassung oder Weiterentwicklung unseres Systems unter QNX zu erleichtern. Die beiden untersten Schichten sollen als dynamische Bibliotheken realisiert werden, um die Erweiterbarkeit und Wiederverwendbarkeit der einzelnen Komponenten zu erhöhen.

Für Schicht 0 wird dafür ein neues C++-Projekt mit dem Namen dclrailcontrol und vom Typ Shared library erstellt. Als Build-Varianten sollten in unserem Fall nur Debug und Release für die X86-Architektur gewählt werden. Als Compiler sollte die GNU Compiler Collection der Version 3.x gewählt werden, da unter QNX beim Linking dynamischer Bibliotheken mit GCC 2.x Probleme auftreten.

Schicht 1 benötigt ebenfalls ein C++-Projekt mit den Build Varianten Debug und Release für X86 als Shared library. Der Name dieses Projektes sollte dclrail sein, denn daraus wird auch der Name der Binary generiert. Auch diesmal sollte GCC 3.x gewählt werden. Als zusätzlicher Include-Pfad muss nun natürlich das Verzeichnis der Header-Dateien aus Schicht 0 angegeben werden. Dazu wird einfach das Projekt dclrailcontrol in die sogenannten Extra include paths bei den Compiler-Optionen des Projektes aufgenommen. Für den Linker sollte unter Extra libs ebenfalls auf das Projekt mit Schicht 0 verwiesen werden. Dabei sollte das Debug-Build gegen die Debug-Version der libdclrailcontrol und für das Release-Build gegen die Release-Version gelinkt werden.

Schicht 2 ist ein C++-Applikationsprojekt, das ebenfalls GCC 3.x nutzt. Als zusätzlicher Include-Pfad für den Compiler müssen die Projekte von Schicht 0 und Schicht 1 konfiguriert werden. Die Header-Dateien von der Readline-Bibliothek müssen nach /usr/qnx630/target/qnx6/usr/include/readline kopiert werden, damit der Compiler sie finden kann. Als Extra libs für den Linker müssen die dynamischen Bibliotheken readline, history und termcap ergänzt werden. Beim Build für die X86-Architektur sucht der Linker im Verzeichnis

/usr/qnx630/target/qnx6/x86/usr/lib nach den dynamischen Bibliotheken, daher müssen die libreadline.so und die libhistory.so aus dem Readline-Paket dorthin kopiert werden. Weiterhin muss auch hier analog zu Schicht 1 auf die Debug- bzw. Release-Version der dynamischen Bibliothek der darunterliegenden Schichten verwiesen werden, also jeweils auf dclrailcontrol und dclrail.

Zusätzlich müssen alle Schichten mit der Compiler-Option `-D_REENTRANT` übersetzt werden. Dazu kann man unter *Project → Properties → QNX C/C++ Projects → Compiler → Definitions* den Wert `_REENTRANT` eintragen.

Die dynamischen Bibliotheken müssen natürlich im System installiert werden, damit sie beim starten der Anwendung geladen werden können. Benötigt werden libreadline.so.4, libhistory.so.4, libdclrailcontrol.so.1 und libdclrail.so.1 sowie für Debug-Builds libdclrailcontrol_g.so.1 und libdclrail_g.so.1. Auf Entwicklungssystemen empfiehlt es sich für die dclrail*-Bibliotheken symbolische Links auf die Dateien in den Build-Verzeichnissen der Momentics-Projekte zu verwenden.

Die fertige Anwendung sollte in einem regulären Terminal gestartet werden, da die Readline-Bibliothek in der Konsole der Momentics IDE nicht die Einstellungen des Terminals ermitteln kann und damit einige Funktionen wie beispielsweise die Shell-Historie nicht zur Verfügung stehen.

REFERENZEN

[1] Operating Systems and Middleware Group *Distributed Control Lab* www.dcl.hpi.uni-potsdam.de, HPI, 2006
[2] *QNX*: www.qnx.com, 2006
[3] *F.* Sippach *QNX: Das Echtzeitbetriebssystem*: http://www.uni-weimar.de/ sippach1/uni/qnx/, Universität Weimar, 2001
[4] *www.operating-system.org*, http://www.operating-system.org/-betriebssystem/_german/bs-qnx.htm, 2005
[5] *Märklin*, http://www.maerklin.de/produkte/digital/
[6] *Interfacing RS-232*, http://www.beyondlogic.org/serial/serial.htm
[7] E. Gamma, R. Helm, R Johnson und J. Vlissides *Design Patterns* Addision-Wesley, 1995
[8] Chet Ramey, *The GNU Readline Library* http://tiswww.tis.case.edu/ chet/readline/rltop.html
[9] *Open Source applications for QNX* OpenQNX project, 2006. http://sourceforge.net/project/showfiles.php?group_id=21249