Warp 3 zwischen allen Kernen

Entwicklung einer schnellen und sicheren Multicore-RTE

Thomas Barth, Prof. Dr.-Ing. Peter Fromm (Hochschule Darmstadt)

Multicore Mikrocontroller bringen aufgrund ihrer Komplexität besondere Herausforderungen, wie die Inter-Core Kommunikation und den Schutz von Ressourcen vor unerlaubtem Zugriff mit sich. Zudem ist die Parametrisierung und Nutzung immer leistungsfähigerer und umfangreicherer Peripherie komplex und fordert den Anwender somit zusätzlich.

Die Programmierung von Multicore Mikrocontrollern stellt Entwickler vor eine große Herausforderung. „Für die Umstellung auf die Multicore Technologie wird, trotz ihrer Vorteile, ein drastisch steigender Entwicklungsaufwand erwartet. So wird mit einer Steigerung der Entwicklungskosten um den Faktor 4.5 und einer Steigerung des Personalbedarfs um den Faktor 3 gerechnet“ [1] . Um diesem entgegen zu wirken, müssen intelligente sowie effiziente Tools und Tool-Ketten eingesetzt werden, die den Entwickler im gesamten Entwicklungsprozess von der Design- über die Entwicklungs- bis hin zur Testphase unterstützen.

Um den Entwicklungsaufwand bzw. die Fragilität und Komplexität des Systems aus Sicht des Applikationsentwicklers zu verringern, wurde in Kooperation mit einem Industriepartner eine innovative Multicore Laufzeitumgebung entwickelt. Die Laufzeitumgebung kombiniert hohe Performance mit guter Usability und ermöglicht eine konsequente Trennung der Runnables sowohl in der Speicher- als auch in der Zeitdomäne. Damit ist es möglich, die Entwicklung der applikativen Software von der Integration zu trennen und somit die Systemkomplexität vor den Applikationsentwicklern zu verbergen. Außerdem kann mit einer solchen Architektur eine Zertifizierung von sicherheitskritischen Funktionen unterstützt werden. In Erweiterung zu existierenden Lösungen, wie dem AUTOSAR Virtual Function Bus, wird die direkte Anbindung und Skalierung von Peripheriesignalen und Kommunikationsprotokollen unterstützt.

Multicore-Controller zur Kapselung von Applikationen

Für die Entwicklung der Laufzeitumgebung wurde ein Infineon AURIX Gen.1 TC27x Multicore Mikrocontroller als Referenzsystem gewählt, da automotive Controller häufig interessante Sicherheits-Features bereitstellen und die Derivate über lange Zeiträume verfügbar bleiben. Der Aufbau eines Multicore Mikrocontrollers am Beispiel Infineon AURIX TC27x ist in Abbildung 1 zu sehen.

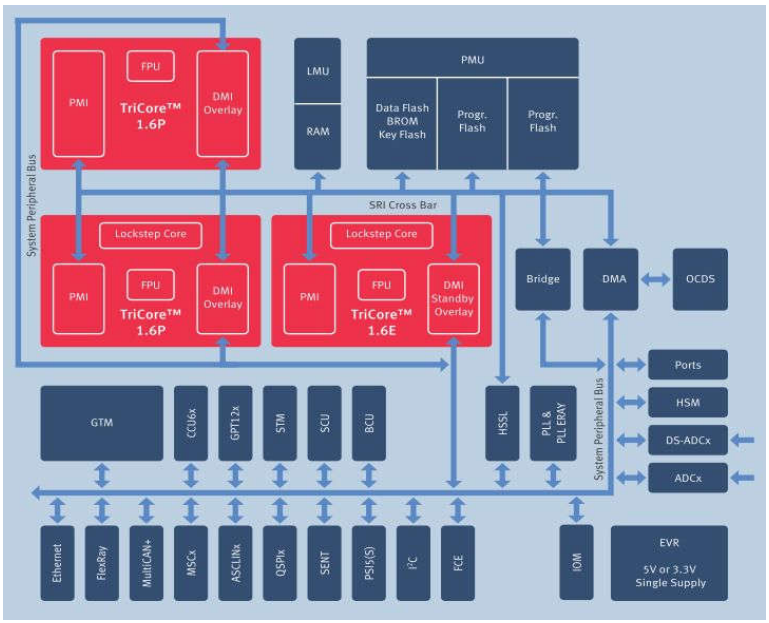


Abbildung 1 Infineon AURIX TC27x [5]

Der Infineon AURIX TC27x ist ein heterogener Multicore Mikrocontroller mit zwei TriCore 1.6 Performance Cores und einem TriCore 1.6 Efficiency Core. Jeder Core verfügt über eigenen Programm und Daten RAM, zudem teilen sich die Cores einen gemeinsamen ROM und einen gemeinsamen RAM. Um der Anforderung nach Rückwirkungsfreiheit gerecht zu werden ist sicherzustellen, dass die Verwendung von gemeinsamen Ressourcen, wie dem RAM, nicht zu Inkonsistenzen/gegenseitiger Beeinflussung führt, siehe z.B. [2]. Diese Trennung muss durch geeignete Architekturmuster sichergestellt werden, welche durch Hardware-Features unterstützt werden sollten.

Das vorgestellte, beispielhafte Architekturmuster ist auf die strikte Trennung von Ressourcen ausgelegt, was eine Qualifizierung und Zertifizierung erleichtert. Jeder der Rechenkerne wird einer Sicherheitsdomäne zugewiesen und erhält spezifische Aufgaben.

Core 0 ist der „Safety Core“, welcher exklusiven Zugriff auf sicherheitsrelevante Peripherie erhält, die Signale aus dem „Application Core“ auswertet und weiterleitet, selbst jedoch keine applikative Software ausführt. Core 0 wird für diese Aufgabe gewählt, da er der einzige Rechenkern ist welcher automatisch nach einem Reset des Controllers gestartet wird.

Core 1 führt applikative Software aus, hat jedoch keinen Zugriff auf die Peripherie. Alle ein- und ausgehenden Signale werden vom „Safety Core“ bereitgestellt und empfangen.

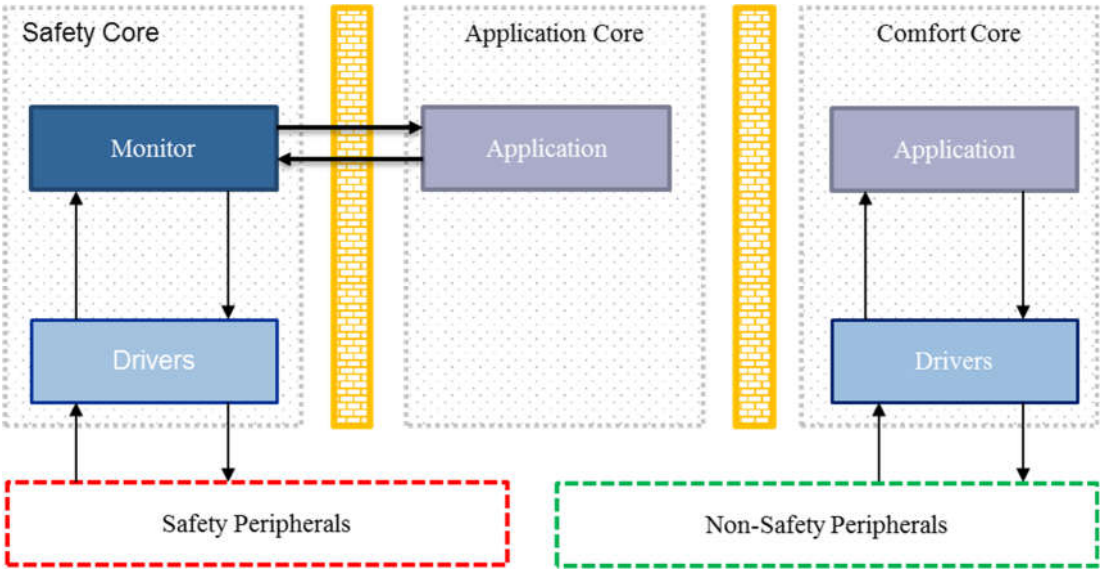


Abbildung 2 Konzept zur Kapselung von Applikationen auf Multicore [3]

Der Vorteil dieser Architektur ist die Möglichkeit applikative Komponenten ohne Änderungen am „Safety Core“ auszutauschen und damit den Aufwand einer Requalifizierung deutlich zu verringern.

Datenfluss und Features der entwickelten Laufzeitumgebung
Bei der Entwicklung der Laufzeitumgebung standen folgende Anforderungen im Vordergrund:

- Hohe Performance: Zykluszeiten < 1ms.
- Zwischenpufferung von Signalen
- Direkte Anbindung von Treibern über möglichst generische Schnittstelle
- Unterscheidung von Roh- und Applikationsdaten, sowie die Möglichkeit zur Wandlung zwischen Roh- und Applikationsdaten.
- Möglichkeiten zum Schutz der Signale vor unerwünschter Änderung

Die Laufzeitumgebung besteht aus zwei Hauptkomponenten:

- Der Signallayer stellt Datenklassen und Objekte zur Kommunikation zwischen Softwarekomponenten zur Verfügung.
- Die eigentliche Laufzeitumgebung ist für die Realisierung des Zeitverhaltens zuständig.

Kernelemente des Signallayers sind Datenklassen, die den Zugriff auf die Signalobjekte über eine einheitliche API definieren. Im Gegensatz zu der Autosar RTE werden pro Signal nicht nur die Nutzdaten gespeichert, sondern zusätzlich Rohdaten des Treibers und Konfigurationsdaten. Die Datenklassen der Signale bestehen somit aus drei Aggregaten:

- Die Applikationsdaten beschreiben das Signal in einer einfach zu nutzenden Form, z.B. in [SI] Größen. Die Applikationssoftware greift nur auf diese Daten zu.

- Über einen an das Signal gekoppelten Treiber können die Daten direkt über eine Schnittstelle eingelesen oder übertragen werden. Hierzu wird das applikative Signal in ein treiberspezifisches Rohdatenformat umgewandelt.
- Die Konfiguration der Treiber sowie die genutzten Skalierungsfunktionen werden pro Signal in einem Konfigurationsdatensatz hinterlegt

Als Beispiel soll das Einlesen einer Spannung über einen 10bit ADC-Treiber beschrieben werden.

Als Rohdatenformat wird ein uint16 Typ verwendet. Der gewünschte ADC Treiber und Port wird über den Konfigurationsdatensatz mit dem Signal verbunden. Über eine Skalierungsfunktion wird der uint16 Wert z.B. in eine Fließkommazahl umgerechnet und der Applikation zur Verfügung gestellt.

Bei der Implementierung der Signalklassen standen die Designziele Usability und Performance im Vordergrund. Die Usability wird durch ein objektorientiertes Design und intuitiv nutzbare getter und setter-Funktionen sichergestellt, wie das folgende Beispiel verdeutlicht:

```
float myVoltage = VOLTAGE_read(&so_in_voltageBattery);
```

Die Performance wird durch eine konsequente Nutzung von Inline Funktionen realisiert.

Die Signale in der Signalschicht werden in einem globalen Datenpool gehalten und durch Hardware- und Betriebssystemmechanismen vor unerlaubtem Zugriff geschützt. Greifen mehrere Tasks auf ein Signal zu, so müssen die Zugriffsrechte klar definiert sein um die Integrität des Signals zu schützen. Um eine performante Lösung unter Nutzung der CPU-MPUs zu realisieren, wird das Messaging-Konzept des verwendeten RTOS PxROS der Firma HighTec genutzt. Die Kommunikation zwischen Tasks erfolgt über die Weitergabe von Zugriffsrechten. Der Sendertask definiert einen Speicherbereich der „versendet“ werden soll und der Empfängertask erhält temporären Zugriff auf diesen Bereich. Aufgrund der zugrundeliegenden Hardware lohnt die genauere Betrachtung des Datenflusses, um das Signal in einen für den gegebenen Anwendungsfall günstigen Speicher zu allokieren.

Zusätzlich zu den eigentlichen Daten enthält ein Signal Metainformationen über das Alter und den Status eines Signals, was die Entwicklung von Monitoring-Funktionen erleichtert.**Fehler! Verweisquelle konnte nicht gefunden werden.** Die folgende Abbildung zeigt den schematischen Aufbau der Laufzeitumgebung mit Zugriff auf Signale durch Runnables.

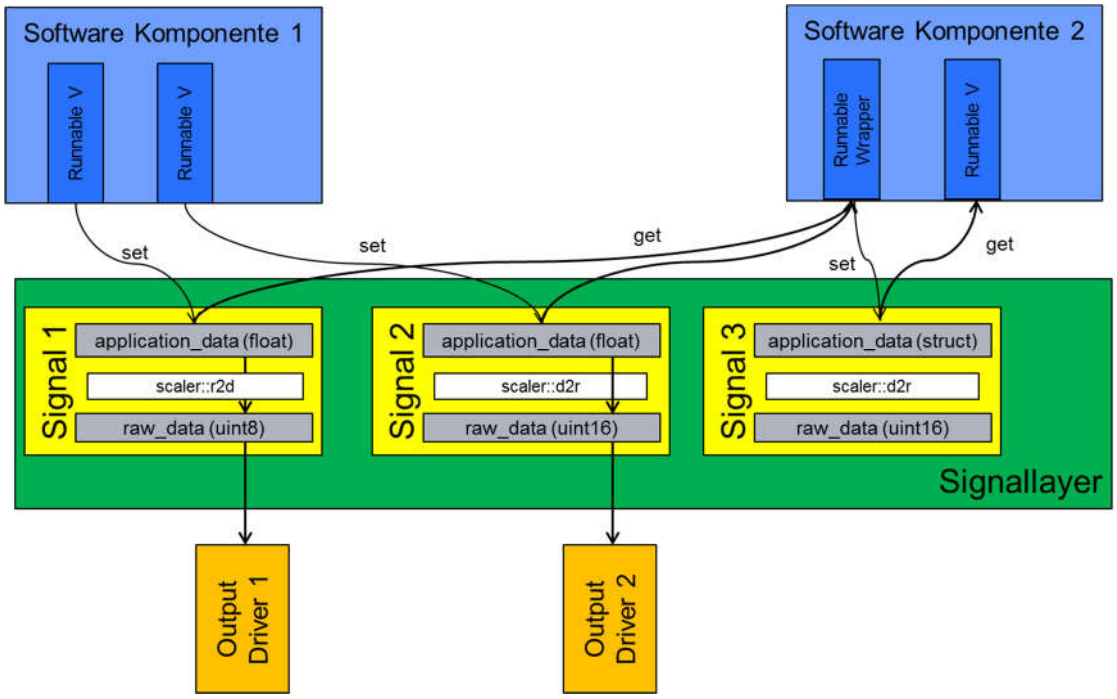


Abbildung 3 Schematischer Aufbau Laufzeitumgebung [4]

Die Laufzeitumgebung realisiert das Zeitverhalten durch Nutzung gut lesbarer Konfigurationstabellen, welche durch Events oder zeitgesteuert aufgerufen werden.

Eine besondere Herausforderung ist hierbei die Synchronisierung von Runnables zwischen den Rechenkernen. Zur Verdeutlichung der Problematik soll ein EVA (Eingabe, Verarbeitung, Ausgabe) -Zyklus angenommen werden, bei dem eine inter-Core Kommunikation genutzt wird.

Zyklischer Aufruf: Der Zyklus kann rein zyklisch ablaufen. E, V und A-Task werden unabhängig voneinander nacheinander aufgerufen. Der Vorteil dieses Konzeptes liegt im strengen zeitlichen Determinismus der Runnables innerhalb eines Cores. Bei Aufteilung des Zyklus auf mehrere Kerne ist die zeitliche Konsistenz nicht mehr sichergestellt. Zwar kann bei einem zyklischen Aufruf der Status eines zu verarbeitenden Signals geprüft werden, aber ein nicht aktualisiertes Signal würde in diesem Fall erst im nächsten Zeitslot berücksichtigt werden, was das gesamte Zeitverhalten verlangsamen würde und zu einem starken Jitter der Signallaufzeiten führt.

Eventbasierter Aufruf: Bei der eventbasierten Verarbeitung von Signalen wird ein Event an den nächsten Task gesendet, sobald ein Signal verfügbar ist. Dies erlaubt eine schnellstmögliche Verarbeitung von Signalen, allerdings wird das gesamte Zeitverhalten sehr „dynamisch“ und im Fehlerfall schwerer zu debuggen.

Die Überwachung des zeitlichen Verhaltens, der Vergleich des Entwurfs und Debugging auf dem realen System sind komplexe Aufgaben, welche ohne geeignetes Tooling händisch kaum zu beherrschen sind. Ob ein zyklischer oder eventbasierter Aufruf von Tasks zu wählen ist, ist von den Echtzeitanforderungen eines

Signalpfades abhängig und muss im Einzelfall entschieden werden. Auch sind hybride Architekturen möglich, bei denen E und A Tasks zyklisch aufgerufen werden, während V-Tasks eventbasiert durch den E-Task aufgerufen werden.

Hardware Features für die Separierung von Ressourcen
In der Standardkonfiguration des AURIX ist jeder Speicher und jedes Peripheriemodul von jedem Core aus erreichbar. Um die Integrität des vorgestellten Architekturmusters auch hardwareseitig abzusichern, bietet der AURIX einige Features zur Separierung von Ressourcen:

CPU-MPUs: Jeder Core verfügt über eine eigene CPU-MPU, welche ausgehende Zugriffe überwacht. Diese CPU-MPUs werden vom verwendeten RTOS PxROS der Firma HighTec dynamisch konfiguriert. Aus Sicht des Users müssen je Task Speicherbereiche „freigeschaltet“ werden, für welche zudem die Art des Zugriffs (lesen, schreiben, ausführen) festgelegt werden kann. Dieser Mechanismus eignet sich sowohl für Speicher als auch für Peripheriemodule. Soll ein Task auf ein Signal der Signalschicht zugreifen, muss dieses Signal explizit freigeschaltet werden.

BUS-MPUs: Alle RAMs verfügen über eine BUS-MPU, welche definiert von welchem Core aus schreibend auf definierbare Speicherbereiche zugegriffen werden darf. Hiermit lassen sich ganze RAM-Speicherbereiche schützen, indem sie explizit zugewiesen werden.

RAP (Register Access Protection): Der schreibende Zugriff auf Peripheriemodule kann über die RAP kontrolliert werden, welche es erlaubt einzelnen Cores Schreibrechte zuzuweisen. Hierdurch können ganze Peripheriemodule vor unerlaubtem Zugriff durch nicht berechtigte Cores geschützt werden.

Mit den vorgestellten Features lässt sich die Separierung der Sicherheitsdomänen in Hardware realisieren. Sollte es aufgrund fehlerhafter Programmierung oder während eines Angriffes auf das System zu einer Schutzverletzung kommen, so verweigert das System den Zugriff und bietet Möglichkeiten, die Verletzung zu analysieren und entsprechend zu reagieren.

Anbindung von Kommunikations-Stacks an die Laufzeitumgebung
Die Flexibilität der Laufzeitumgebung erlaubt die Anbindung von Kommunikations-Stacks, was eine Erweiterung der Laufzeitumgebung über Gerätegrenzen hinweg ermöglicht. Aus Sicht des Entwicklers ist es somit irrelevant ob sich Signal-Quellen und -Senken auf der lokalen ECU oder auf einem entfernten Gerät befinden, solange Latenz und Jitter der Kommunikation berücksichtigt werden. Durch die Entkopplung zwischen Signalen und Geräten lassen sich so auch flexible Restbus- und Diagnosetools, beispielsweise auf einem PC, einsetzen. Soll ein Signal von einem entfernten Gerät aktualisiert werden, wird das gleiche Schnittstellenkonzept wie bei einem Zugriff auf Peripheriemodule genutzt.

Im Folgenden soll die exemplarische Anbindung der Laufzeitumgebung an einen CANopen Stack dargestellt werden. CANopen ist ein auf CAN basierendes Kommunikationsprotokoll, welches hauptsächlich in der Industrieautomatisierung Anwendung findet. CANopen organisiert die über das Netzwerk zugänglichen Daten als „Objekte“ in einem sog. „Objektverzeichnis“. Ähnlich TCP/IP-Sockets wird eine Kombination aus Geräte- und Service-ID verwendet um auf das Objektverzeichnis zuzugreifen. Der Datentyp der CANopen Objekte ist frei definierbar und zudem sind Zugriffberechtigungen und Grenzwerte einstellbar. Ein Signal, welches über CANopen angebunden werden soll, benötigt somit lediglich Informationen über die Netzwerkadresse und die Position der Daten im Objektverzeichnis.

Der Betrieb im Client/Server Modus(SDO) erlaubt eine flexible Kommunikation, erzeugt jedoch Overhead, da die Objekte im Objektverzeichnis explizit adressiert werden müssen. Eine Alternative hierzu ist der Sender/Receiver Modus (PDO). Hierbei werde einzelne Objekte des Objektverzeichnisses mit eindeutigen Message-IDs („Sockets“) assoziiert und entweder zyklisch oder eventbasiert versendet. Da die Zuordnung durch die Message-ID erfolgt, entfällt der Adressierungs-Overhead. Es kann aufgrund der Anzahl der verfügbaren Message-IDs nur eine begrenzte Anzahl von Objekten auf diese Weise versendet werden.

Neben der Wahl des geeigneten Transfermodus ist bei der Betrachtung der Kommunikation auch der Zeitpunkt der Signalaktualisierung zu beachten. Es ergeben sich folgende Möglichkeiten:

Blocking: Soll ein Signal erneuert werden, so blockiert die Funktion die Applikation mit einem Spinlock, welcher erst freigegeben wird, sobald eine Antwort erhalten wurde oder ein Timeout auftritt. Dieser Ansatz eignet sich für synchrone Kommunikationsszenarien, in denen ein streng deterministischer Ablauf der Kommunikation gefordert wird. Aufgrund des „Busy Waitings“ sind die Kosten in Bezug auf die Rechenzeit sehr hoch.

Handler: Die Applikation erhält Zugriff auf einen Handler, welcher den Status der Übertragung repräsentiert. Der Handler wird z.B. zyklisch abgefragt und das assoziierte Signal entsprechend aktualisiert. Dieser Ansatz eignet sich für Signale die zyklisch abgefragt werden, da so die Zeitpunkte zwischen Anfrage und Verarbeitung entsprechend synchronisiert werden können.

Callback: Das Signal verfügt über eine Callback-Funktion, welche aufgerufen wird sobald die Übertragung abgeschlossen ist oder ein Timeout auftritt. Diese Art der Übertragung eignet sich besonders für Signale die eventbasiert und möglichst unmittelbar aktualisiert werden sollen. Es ist jedoch zu beachten, dass die Callback-Funktionen das zeitliche Verhalten der Applikation verändern.

Referenzsystem

Die mit dem Industriepartner entwickelte Laufzeitumgebung wird genutzt, um hochschulintern ein Versuchsfahrzeug (Abbildung 4) mit Multicore-ECU aufzubauen. Die Besonderheit des Fahrzeuges liegt in der Art des Antriebes, welcher Rotation um die eigene Achse und seitliche Bewegungen erlaubt. Das Fahrzeug verfügt unter anderem über einen LIDAR Sensor, welcher in einer späteren Ausbaustufe für ADAS Funktionen genutzt werden soll.

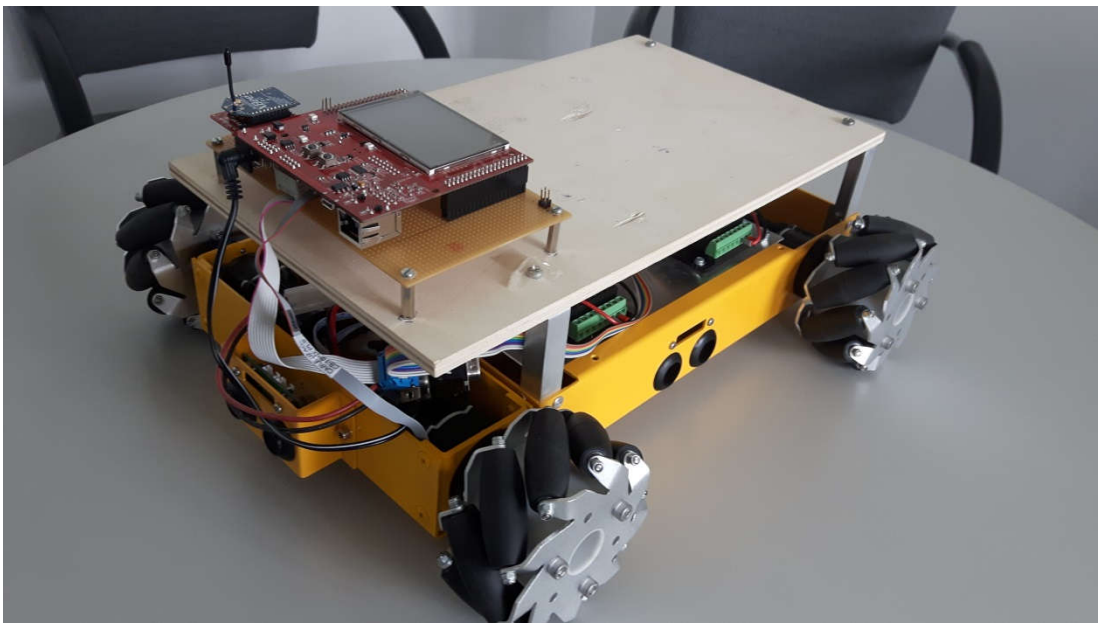


Abbildung 4 Versuchsfahrzeug Multicore – Hochschule Darmstadt

Neben der Schulung von Studenten zum Thema Multicore, dient der Versuchsaufbau der ständigen Evaluierung und Weiterentwicklung der Laufzeitumgebung. So haben im Sommersemester 2016 30 Studenten in 5 Teams in parallelen Projekten an dem Fahrzeug gearbeitet. Abbildung 5 zeigt einen Ausschnitt aus dem Signalfluss des Versuchsaufbaus. Durch die strenge Kapselung von Funktionen und die klare Definition der Signalschnittstellen konnten die verschiedenen Sub-Projekte mit vertretbarem Aufwand in das Gesamtsystem integriert werden.

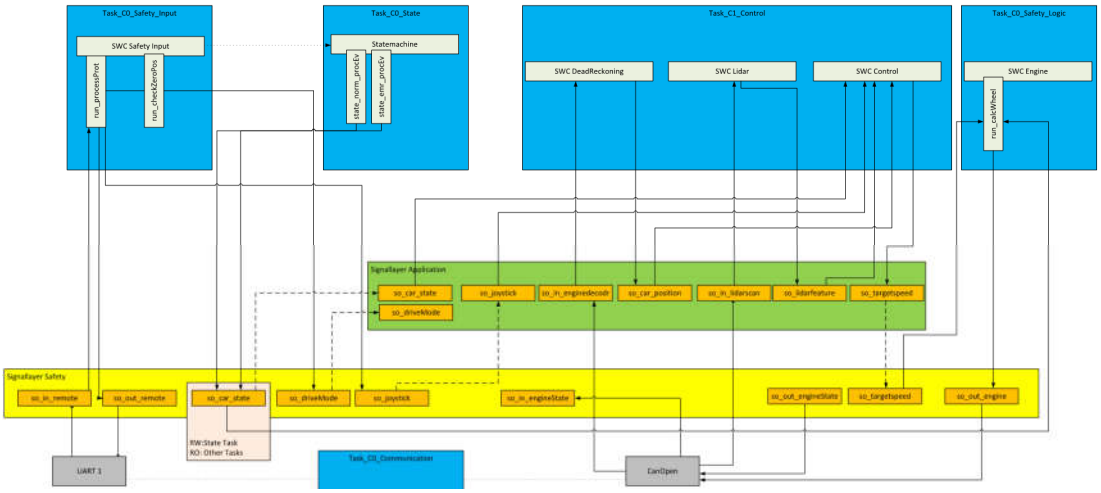


Abbildung 5 Signalfluss StudentCar

Erfahrungen, Fazit, Ausblick

Bedingt durch die Komplexität und den damit verbundenen Entwicklungsaufwand auf Multicore Mikrocontrollern, hat sich die Entwicklung und Verwendung einer Laufzeitumgebung als notwendig erwiesen. Während der Industriepartner die Laufzeitumgebung bereits in ersten Projekten im Feld nutzt, war es im Hochschulumfeld möglich, eine große Anzahl von studentischen Teams ohne Multicore-Erfahrung in parallelen Projekten einzusetzen und die Teilergebnisse mit vertretbarem Aufwand in das Gesamtsystem zu integrieren.

Während sich die Komplexität aus Sicht des Anwendungsentwicklers bei Verwendung einer Laufzeitumgebung verringert, so führt immer komplexere (Multicore-)Hardware zu neuen Herausforderungen für System-Architekten, Basissystementwickler und Integratoren. Nicht selten sind die Herausforderungen so hoch, dass diese noch recht junge Technologie, trotz ihrer Vorteile, nicht genutzt werden kann. Aus diesem Grund arbeitet die Hochschule Darmstadt zusammen mit Partnern aus Industrie und Forschung an der Entwicklung von Entwurfswerkzeugen, welche Multicore-Mikrocontroller insbesondere für kleine und mittelständische Unternehmen nutzbar machen sollen.

Das zeitliche Verhalten von Multicore Mikrocontrollern ist alles andere als trivial. Während auf Singlecore-Controllern eine Parallelisierung hauptsächlich virtuell durch Multitasking erfolgte, so besteht auf Multicore Controllern eine echte Nebenläufigkeit welche durch Multitasking zusätzlich erweitert wird. Eine Überwachung des Timings und ein Abgleich mit dem Entwurf ist daher nur durch geeignetes Tooling sicherzustellen. Auch in diesem Bereich arbeitet die Hochschule Darmstadt mit Partnern zusammen um entsprechende Werkzeuge verfügbar zu machen.

Abkürzungsverzeichnis

ADAS - Advanced Driver Assistance Systems
MPU - Memory Protection Unit
PDO - Process Data Object
RAP - Register Access Protection
SDO - Service Data Object

Literaturverzeichnis

[1] S. Balacco and C. Rommel,
"Next Generation Embedded Hardware Architectures," in *VDC Research*, 2010.

[2] *DIN EN ISO 13849,*
Safety of machinery – Safety-related parts of control system, 2006.

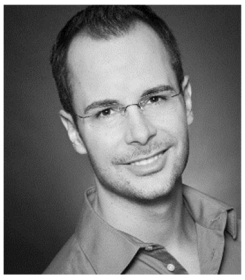
[3] T. Barth, "Functional Safety on Multicore Microcontrolles for Industrial Applications," in *Embedded World Conference*, Nürnberg, 2016.

[4] P. Prof. Dr. Ing-. Fromm,
Spezifikation Laufzeitumgebung, Hochschule Darmstadt, 2016.

[5] Infineon AG, "Infineon.com," [Online]. Available: http://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-tm-microcontroller/aurix-tm-family/aurix-tm-family-%E2%80%93-tc27xt/channel.html?channel=db3a30433cfb5caa013d01df64d92edc. [Accessed 14 10 2016].

Autoren

Thomas Barth ist Lehrkraft für besondere Aufgaben an der Hochschule Darmstadt und forscht an funktionaler Sicherheit auf Multicore-Microcontrollern. Herr Barth hat Elektro- und Informationstechnik mit dem Schwerpunkt embedded Systems in Darmstadt, Helsinki und Frankfurt studiert und war 7 Jahre bei der Firma Siemens in der Industrieautomatisierung tätig.



Prof. Fromm hat in Aachen promoviert und danach 10 Jahre bei der Firma Continental den Bereich Software und System Engineering geleitet. An der Hochschule Darmstadt vertritt er im Fachbereich Elektrotechnik und Informationstechnik das Gebiet Mikrocontroller und Informationstechnik. Daneben berät und unterstützt er mehrere Firmen im Bereich Embedded Software und System Engineering. Ein aktueller Forschungsschwerpunkt ist der Bereich Multicore Architekturen und Embedded Operating Systems.