Modellbasierte Code-Generierung für heterogene FPGA-SoC Systeme

Kreativität für Kernfunktionalität einsetzen, alles andere (sauber) generieren lassen

Alexander Wirthmüller, MPSI Technologies GmbH

Massiv gesteigerte Leistung von Einplatinenrechnern, insbesondere solchen, die Linux-fähige Rechenkerne und programmierbare Logik auf einem FPGA-SoC Chip vereinen, ist für Embedded Software Entwickler Motivation und Herausforderung zugleich, da zahlreiche neue Technologien zu beherrschen sind. Zeit für einen umfassenden Ansatz, der repetitives manuelles Kodieren eliminiert.

Spätestens mit Erscheinen von low-power Vielkern-CPU‘s zu Beginn der 2010er Jahre ist die klare Grenze aufgehoben, die zuvor zwischen low-level Embedded Software für FPGA‘s und MCU‘s einerseits, und high-level Anwendungssoftware für CPU‘s andererseits, verlief. Zunehmend werden „sowohl als auch“-Systeme eingesetzt, wobei nach Kriterien wie Echtzeitfähigkeit, Leistungsbudget, Flexibilität und verfügbarem Programmier-Know-How, die Umsetzung der Software-Funktionalität jeweils einer der beiden Ebenen zugeteilt wird.

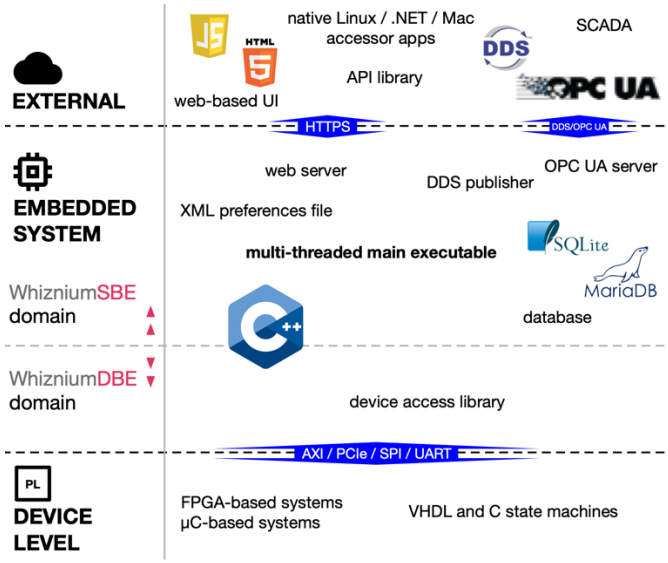

Abbildung 1: Übersicht FPGA-SoC Softwareprojekte mit WhizniumDBE & SBE

Gleichzeitig stärken Trends wie IIoT und Edge Computing die Vernetzung und Datenverarbeitung nahe am Geschehen. Damit rückt auch für Embedded-Software-Entwickler die high-level Ebene ins Sichtfeld, auf der ein Betriebssystem, oft Linux, Zugang zu einem reichhaltigen Angebot an Programmbibliotheken bietet. Für derartig umfassende Aufgaben (Abb. 1) ist der klassische Ansatz, Projekte vollständig innerhalb der Entwicklungsumgebung des jeweiligen Chip-Herstellers zu realisieren, ungeeignet. Hingegen kann der Aufwand, Programmiersprachen und -konzepte jenseits von VHDL und C zu erlernen und in guter Qualität umzusetzen, beträchtlich sein.

Es stellt sich also die Frage, welcher Typ Werkzeug einem Entwickler hier Arbeit abnehmen kann, insbesondere in Hinblick auf sich von Projekt zu Projekt wiederholende Funktionalität. Nicht ersetzt werden sollten die Hersteller-spezifischen low-level Synthese- und Placement-Werkzeuge (für den FPGA-Bereich) bzw. Compiler (für den MCU-/CPU-Bereich), welche die bestmögliche Performance aus der Ziel-Hardware herausholen. Auch eine neue „one-size-fits-all“ Programmiersprache würde der Aufgabe, heterogene Embedded-Software-Projekte zu realisieren, nicht gerecht. Zu unterschiedlich sind die vorkommenden Aspekte und die existierenden Sprachen haben sich nicht zuletzt deswegen durchgesetzt, weil sie für ihren spezifischen Anwendungsaspekt optimierte Ergebnisse garantieren.

Ein entwicklerfreundlicher Ansatz, der dennoch massive Zeitersparnis unter Achtung existierender Programmiersprachen und Aspekt-spezifischer *best practices* verspricht, ist die hier vorgestellte modellbasierte Code-Generierung mit Hilfe des Open Source Werkzeugs Whiznium.

Software-Design-Prozess

Das Ziel des Software-Design-Prozesses mit Whiznium ist es, das Werkzeug in die Lage zu versetzen, einen synthetisier- bzw. kompilierfähigen Quellcode-Baum zu schreiben, welcher alle Projektaspekte abdeckt. Hierzu wird Whiznium in Form von Aspekt-spezifischen Modelldateien eine Software-Beschreibung mitgeteilt, deren Detailtiefe mit Projektfortschritt üblicherweise zunimmt. Entsprechend ist die Ausgabe von Whiznium zu Projektbeginn auf einen Rohbau an Quellcode-Dateien beschränkt, der durch manuelle Ergänzungen, klassisches Debugging und Iteration der Modellinformation, gekoppelt mit wiederholten Whiznium-Aufrufen, gefüllt wird, bis alle Projektanforderungen erfüllt sind.

Whiznium ist unterteilt in WhizniumDBE (Device Builder‘s Edition [1]) für FPGA- und MCU-Code, und WhizniumSBE (Service Builder‘s Edition [2]) für Vielkern-kompatible Embedded Linux Software. Beide Werkzeuge (… sind WhizniumSBE-Projekte, ein Hinweis auf die Universalität des WhizniumSBE-Konzepts, und …) folgen jeweils ihren eigenen Sequenzen um schrittweise Projektinformation, von grob zu fein, in einer SQL-Datenbank anzusammeln.

Beginn und Ende der WhizniumDBE-Sequenz (Abb. 2 links) sind von FPGA-Projekten gut bekannt, werden hier aber in abgewandelter Form zusätzlich auch auf MCU-Projekte angewandt: von einer hierarchischen Modul-Struktur geht es über Zwischenschritte wie Befehle, die vom *processing system* (PS) in *programmable logic* (PL) ausgelöst werden, sowie parallelisierbare Algorithmen, zur Feinstruktur aus *ports*, *signals* und *processes*.

Abbildung 2: Sequenz der Import-/Generierungsschritte von WhizniumDBE & SBE

Ein wesentliches Merkmal von Whiznium ist die eingebaute Fähigkeit – neben manuellen Importen – Elemente der nächstfeineren Ebene automatisch abzuleiten. Ein Beispiel hierfür ist die automatische Generierung von Handshake-Feinstruktur für Befehlsaufrufe zwischen FPGA-Modulen. Zudem können eigene Modulvorlagen eingearbeitet werden, die dank des Zugriffs auf die SQL-Datenbank parameterbasiert in die WhizniumDBE-Sequenz eingreifen können – in erweiterbarem, offenem, C++ Code. Diese Möglichkeit ist deutlich leistungsfähiger als die reine Verwendung von *generics* in VHDL-Modulen.

Die WhizniumSBE-Sequenz (Abb. 2 rechts) hat mehr Aspekte zu berücksichtigen als ihr WhizniumDBE-Gegenstück. Begonnen wird mit Informationen zu Zielplattformen und -komponenten (s.u.), der Struktur einer zu Grunde liegenden relationalen SQL-Datenbank (zumeist SQLite [3] für Embedded-Projekte) und einer groben Gliederung des User Interface (UI). Die nächstfeinere Ebene beschreibt etwa die Verteilung von Rechenoperationen auf separate Executables oder den Aufbau des UI bis auf das Niveau von einzelnen *controls*. Schließlich beschreibt der hierarchische *job tree* eine Struktur aus C++ Klassen, deren Objekte zur Laufzeit für einzelne Hardware-, Algorithmus- oder UI-Aspekte zuständig sind. Hier können auch Zustandsautomaten, *inter-job*-Kommunikation sowie XML-/JSON-Datenblöcke für das MVC-konforme UI spezifiziert werden.

(Teil-)automatisch generierte Komponenten

Nachdem alle relevanten Projektinformationen in der WhizniumDBE bzw. WhizniumSBE SQL-Datenbank vorhanden sind, werden in einem einfachen aber robusten Verfahren Quellcode-Bäume generiert. Dabei wird auf eine Vielzahl an Vorlage-Dateien zurückgegriffen, welche mit Platzhaltern und Einfügepunkten für Codebausteine versehen sind. Im Fall von WhizniumDBE können solche auch spezifisch für Modulvorlagen angelegt werden.

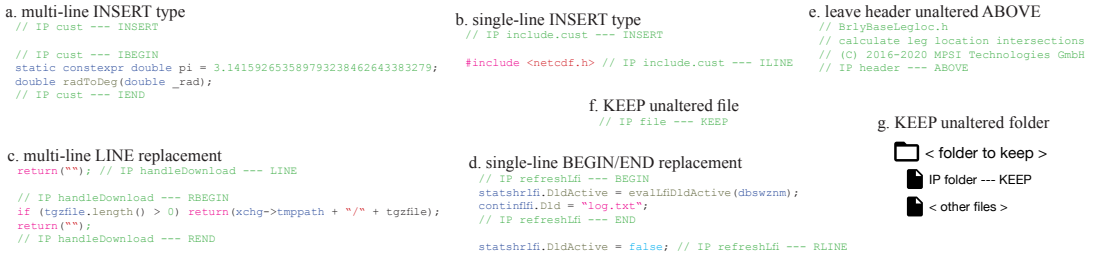

Abbildung 3: Illustration des Prinzips von Einfügepunkten

Wie zuvor erwähnt, existiert manuell geschriebener Code symbiotisch in automatisch generierten Quellcode-Bäumen. Daher wird ab dem zweiten Aufruf von Whiznium für ein Projekt bzw. bei der ersten Modelliteration manueller Inhalt des existierenden Quellcode-Baumes aus Einfügepunkten (Kommentare in der jeweiligen Programmiersprache, Abb. 3) extrahiert, um in den neuen Quellcode-Baum wieder eingefügt zu werden. Umfassende Syntax-Überprüfungen sichern diesen Prozess ab.

WhizniumDBE ist in der Lage, folgende Komponenten zu generieren:

- *device access library*: C++ Code für das PS, welcher *non-blocking* den Aufruf von PL-Befehlen sowie Datentransfers von/zur PL ermöglicht
- *easy model device access library*: C++ Code für vorstehende Funktionalität, allerdings *serial*
- *FPGA code*: VHDL Code und *constraints* für PL, modular aufgeteilt, mit PS-aufrufbaren Befehlen und Datentransfers. Optional mit FPGA-Befehlscache
- *MCU code*: auf minimale Größe optimierter C Code, welcher die Event-basierte Logik von PL emuliert

WhizniumSBE kann diese Komponenten generieren:

- *main engine*: Vielkern-kompatibler C++ Code mit SQL-Datenbankanbindung, welcher einerseits zuverlässig auf Hardware-Subsysteme zugreift, und andererseits Sessions für den Zugriff von außen verwaltet. Für diese Kommunikation sind ein libmircohttpd-basierter HTTPS-Server sowie optional ein OPC UA Server und ein DDS *publisher* enthalten
- *operation engines*: C++ Code für Satelliten-Executables, die *remote procedure calls* (RPC‘s) ausführen können. Anbindung an die *main engine* über HTTP
- *combined engine*: eine Kombination der Funktionalitäten von *main engine* und *operation engines* wenn keine Cluster-Architektur bzw. heterogene Vielkern-Architektur vorhanden ist
- *database access library*: C++ Code zum komfortablen Zugriff auf die Projekt-MySQL-/PostgreSQL-/SQLite-Datenbank
- *web app user interface files*: HTML5 und JavaScript Code für vielsprachiges, web-basiertes UI. Feingliedrige Zugriffsverwaltung nach Userlevels
- API *library*: C++ Code um basierend auf der Kommunikation des UI *machine-to-machine* Steuerung zu ermöglichen. Dies vereinfacht Integration in native Windows-/macOS-/Linux-Anwendungen

- Java API *package*: Java Code für vorstehende Funktionalität

Beispielprojekt

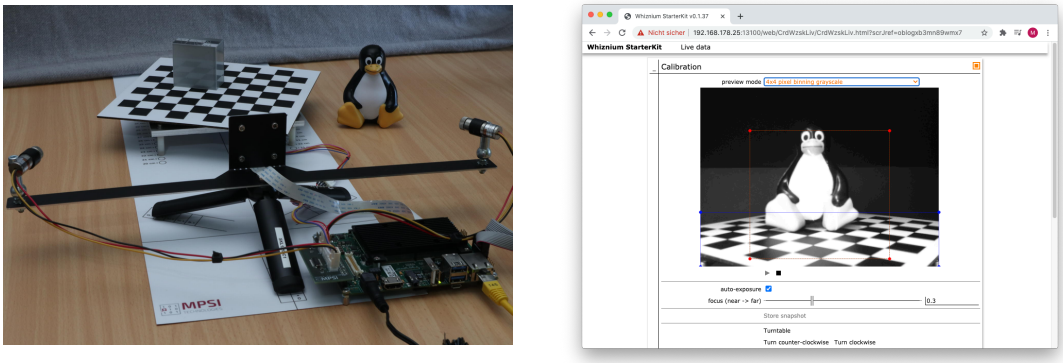

Abbildung 4: Kompakter 3D Laserscanner mit Software (kleiner Auszug)

Als Beispielprojekt dient ein kompakter 3D-Laserscanner mit Drehteller einerseits und 5MP Kamera und Linienlasern andererseits (Abb. 4, [4]), dessen Logik auf einem FPGA-SoC [5] implementiert ist. FPGA-SoC‘s verkörpern die Verschmelzung von low-level und high-level Embedded Software ideal, da PS und PL auf einem Stück Silizium vereint sind. Whiznium eignet sich auch für reine PS/PL Systeme, sowie für solche bei denen PS und PL auf getrennten Chips oder Platinen zu finden sind.

Weitere Konzepte

Whiznium ist aus dem Entwickleralltag entstanden, entsprechend wird folgenden, für Entwickler relevante, Anforderungen besonders Rechnung getragen:

- Whiznium setzt auf wenige, aber gut etablierte und offene Technologien: libxml2, libmicrohttpd, [eine aus] MySQL/PostgreSQL/SQLite, [optional] Vue.js
- dank des „all code in plain sight“-Prinzips ist alle Funktionalität einfach nachvollziehbar
- mit Whiznium begonnene Projekte können unabhängig weiterentwickelt werden
- Whiznium ist kompatibel mit Versionierung über Git, speziell bei Quellcode-Baum-Iterationen
- generierter Code ist auf gute Lesbarkeit optimiert und modular aufgebaut. Dies erleichtert Testbarkeit und Pflege; die üblichen Debugging-Werkzeuge können eingesetzt werden
- WhizniumDBE und WhizniumSBE sind Open Source. Dies ermöglicht anwenderspezifische, auch Closed Source, Erweiterungen

Zusammenfassung und Ausblick

Es wurde eine Methodik vorgestellt, zeitsparend leistungsfähige Software für moderne Embedded Systeme zu entwickeln. Der modellbasierte Ansatz des Werkzeugs Whiznium deckt die meisten in FPGA-SoC Systemen vorkommenden Aspekte mit einer hohen Detailtiefe ab.

Whiznium ist ein lebendiges Projekt, welches dank seiner Offenheit Markt-Trends, seien es neue Hardware-Plattformen oder neue Programmieransätze, unkompliziert in seine Modellbeschreibung und Quellcode-Generierung aufnehmen kann. Beiträge von praktischer Relevanz aus der Community sind jederzeit gerne willkommen.

Referenzen

[1] Whiznium Device Builder's Edition auf GitHub: https://github.com/mpsitech/wdbe-WhizniumDBE
[2] Whiznium Service Builder's Edition auf GitHub: https://github.com/mpsitech/wznm-WhizniumSBE
[3] SQLite Projekt-Homepage: https://www.sqlite.org/index.html
[4] Whiznium StarterKit auf GitHub: https://github.com/mpsitech/wzsk-Whiznium-StarterKit
[5] Microchip PolarFire SoC FPGA's: https://www.microchip.com/en-us/products/fpgas-and-plds/fpgas/polarfire-fpgas/polarfire-soc-fpgas
(Alle Links aufgerufen am 17. Oktober 2021)

Autor

Alexander Wirthmüller ist Inhaber des Start-up's MPSI Technologies, welches er nach umfangreichen Erfahrungen in forschungsnahen Anwendungen 2016 in München gegründet hat, um Systementwicklern die Zeit zurückzugeben, die bei klassischer Programmierung durch repetitive Coding-Aufgaben verloren geht. Nach einem ersten kommerziellen Mikrocontroller-basierten Projekt als Schüler, konnte Alexander während seines Studiums sowie danach in diversen Forschungseinrichtungen zahlreiche Projekte, von Leistungselektronik, über verteiltes Rechnen, bis hin zu Photonik mit Anwendungen in der Robotik, bearbeiten bzw. begleiten. Hierbei spielte stets der Gedanke, mittels effizient programmierter Software zu besseren Ergebnissen zu kommen, eine wichtige Rolle. Alexander besitzt ein Diplom in Elektrotechnik von der ETH Zürich.

Kontakt

Internet: www.mpsitechnologies.com
Email: aw@mpsitechnologies.com