

Erfahrungen mit der Methode UML für 16-Bit Projekte

In nahezu keiner Branche ist der Wandel und das Wachstum so schnelllebig, wie in der Software Industrie. Bei 70% der in Deutschland produzierten Produkte liegt der Hauptanteil der Wertschöpfung inzwischen in der Software. Und jeder, der in dieser Branche tätig ist, kennt Moore's law (alle 18 Monate verdoppelt sich die Anzahl der Transistoren auf einem Chip).

Eines haben wir in den letzten Jahrzehnten gelernt: höhere Komplexität kann mit vertretbarem Entwicklungsaufwand nur auf einem höheren Abstraktionsgrad beherrscht werden, und in diesem Zusammenhang setzt sich die formale Notation UML bei grösseren Systemen zunehmend durch.

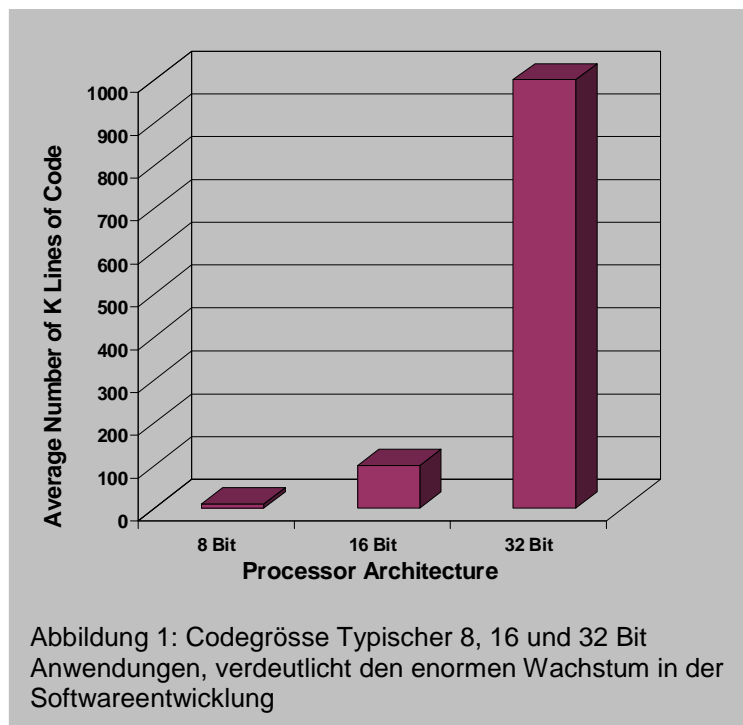


Abbildung 1: Codegrösse Typischer 8, 16 und 32 Bit Anwendungen, verdeutlicht den enormen Wachstum in der Softwareentwicklung

Nun haben grosse Systeme das Wachstum nicht für sich allein gepachtet. Auch die Anforderungen an die ,kleineren' Embedded Systeme wachsen mit grosser Geschwindigkeit, und der gleichzeitig steigende Time To Market Druck zwingt zur Wiederverwendung von Software. Das wiederum ist nur möglich, wenn Qualität, Verstehbarkeit und Änderbarkeit der Software Wiederverwendung überhaupt zulassen. Hier scheint die OOP der Stein der Weisen zu sein. Gleichzeitig bietet die derzeit bevorzugte Sprache C jedoch keine idealen Voraussetzungen für die OOP und ein Umstieg auf C++ führt nicht

zwangsläufig zu Objekt orientierter Programmierung und oft sind für 16-Bit Mikrocontroller keine C++ Compiler verfügbar.

UML mit automatischer Codegenerierung für C kombiniert mit einem RTOS, entpuppt sich zunehmend als eine ideale Möglichkeit für den Einstieg in die OOP auch für kleinere Systeme. (16 Bit CPU, min. 8K RAM und 128 K ROM)

Aber UML, das bedeutet auch den Einsatz von sogenannten CASE Tools. Und Vielen liegen Erfahrungen mit dem Thema CASE noch schwer im Magen. Genau mit diesen Erfahrungen aus der Vergangenheit, bezogen auf heutige UML Lösungen, beschäftigt sich dieser Artikel.

UML, eine Modeerscheinung?

Um diese Frage zu beantworten, möchte ich Sie kurz in das Jahr 1985 entführen. Erinnern

Sie sich? Für Embedded Systeme werden überwiegend 8-Bit Mikrocontroller eingesetzt. Der Stand der Softwareentwicklung in diesem Bereich basierte auf der Assembler Sprache.

Aber in einigen fortschrittlichen Projekten, überwiegend mit 16-Bit Architekturen, wird bereits in sogenannten Hochsprachen programmiert. Basic, Fortran, Pascal, PL/M. Alles Basis-Sprachen mit verschiedenen Ausprägungen und einem grossen Nachteil: Die Portierung eines Projektes auf eine andere Sprache oder auch nur einen anderen Dialekt ist mit grossem Aufwand verbunden. Aber es gibt eine sich schnell verbreitende neue Sprache : „C“. Der entscheidende Vorteil ist die genaue Festlegung der Notation der Sprache zur Vermeidung von Dialekten durch die Erschaffer Kernighan & Ritchie. Wenige Jahre später wird die C-Notation in einer etwas überarbeiteten Version als ANSI C offiziell standardisiert. Einer der Hauptgründe, warum die Sprache C so erfolgreich wird. Mit der Standardisierung der Notation werden Investitionen in Tools, Ausbildung und erstellte Software auf Jahre gesichert.

Gegenüber der Softwareentwicklung in Assembler gibt es jedoch noch ein kleines Problem. Die Übersetzung der C-Programme in Assembler-Code mit Hilfe eines Compilers ist mit Overhead verbunden. Für 16-Bit Mikrocontroller nicht so relevant, aber bei den stark eingeschränkten Ressourcen der 8-Bit Systeme ist dieser Overhead nicht akzeptierbar. So glaubte man damals zumindest. Aber kommen Sie mit mir zurück in die Gegenwart. Fallende Preise in der Hardware, leistungsfähigere 8-Bit Mikrocontroller und verbesserte Technologien der C-Compiler haben inzwischen dafür gesorgt, dass heute auch 8-Bit Systeme fast ausschließlich in C programmiert werden.

Eines haben wir inzwischen gelernt: Komplexer werdende Software ist durch die Programmierung auf einem höheren Abstraktionsgrad besser beherrschbar. Der Knackpunkt scheint die Frage zu sein, ob der damit verbundene Aufwand in Form von Toolkosten, Einarbeitungszeit und Performance-Overhead vertretbar ist. Solange für eine neue Technik die Investitionen nicht

über Jahre gesichert sind, wird die Entscheidung sicher häufig negativ ausfallen. Das ist eine grosse Problematik der proprietären CASE Tools.

Und nun kommen wir zum Thema UML. Die damalige Entwicklung der Hochsprache C und die jetzige Entwicklung der Notation UML besitzen erstaunliche Parallelen. Wie Ende der 80er Jahre die Sprache C, ist auch die UML

UML: Unified Modelling Language

Die UML ist das geistige Kind von Grady Booch, James Rumbaugh und Ivar Jacobson. Diese auch ‚die drei Amigos‘ genannten Herren entwickelten in den 80er Jahren jeder seine eigene Methode zur Softwareentwicklung. Ihre Konzepte setzten sich gegenüber vielen anderen immer weiter durch und zeigten deutliche Parallelen. In den 90er Jahren beschlossen Sie den logischen Schritt zu vollziehen und Ihre Arbeit gemeinsam weiter zu führen. Es entwickelte sich die UML. Aus heutiger Sicht entscheidend, für Realtime Applikationen, war die Einführung der Semantik für Verhaltensbeschreibung, basierend auf den Statecharts von Prof. David Harel. So entstand die UML, wie sie derzeit eingesetzt wird. Sie setzte sich schnell durch und ist heute in der Softwareindustrie zu einem Standard geworden. Resource

eine Notation die sich zum Standard entwickelt hat. Für diese Notation sind inzwischen verschiedene Werkzeuge unterschiedlicher Hersteller verfügbar. Schauen Sie sich um, inzwischen behaupten nahezu alle CASE-Tools von sich UML-tauglich zu sein. Welch eine stürmische Entwicklung hat da stattgefunden? Und genau das ist der Unterschied zu allen vorherigen CASE-Ansätzen.

Investitionen in die Notation UML sind nicht von einem proprietären Hersteller abhängig,

sondern Investitionen in einen inzwischen weit verbreiteten und von der OMG (Object Management Group) gesicherten Standard. Ausbildung in die Mitarbeiter ist auf Jahrzehnte gesichert, UML-Kenntnisse können zukünftig bei Personalentscheidungen als Standard Programmierkenntnisse herangezogen werden. Damit sind auch Investitionen in ein Softwaredesign auf mehrere Jahre gesichert. Auch die kommende Generation von Entwicklern wird UML verstehen und in UML programmieren. Für 32-Bit Systeme ist die Frage überwiegend zu Gunsten der UML entschieden. Keine Kunst, wo Speicher in Mbyte gerechnet wird, spielen einige paar KByte mehr oder weniger an Overhead keine so grosse Rolle. Aber dieser Artikel wendet sich an die Entwickler, die Speicher noch in KByte messen.

UML und wie viel Overhead?

Ein typisches Projekt mit einem 16-Bit Mikrocontroller hat zwischen 8 und 200 KByte RAM und 128 und 512 KByte ROM zur Verfügung. Und da ist er, einer der unverdauten CASE-Brocken, der sicher

einigen von Ihnen noch schwer im Magen liegt. Die Codegenerierung: Der Overhead der Codegeneratoren ist zu gross für kleinere und mittlere Systeme. Daraus folgt gleich noch ein dickerer Brocken, der noch schwerer im Magen liegt, der Bruch zwischen Design und Implementation. Nachdem die Analyse und ebenso das Design vorbildlich durchgeführt wurden, wird nun kräftig codiert, und die Zeit vergeht wie im Fluge, da steht schon die Messe vor der Tür und der erste Prototyp kann nur unter extremem Einsatz aller Programmierressourcen erstellt werden. Diejenigen Leser unter Ihnen, die bereits mit einem CASE-Tool ohne Codegenerierung gearbeitet haben wissen nun, wovon ich spreche. Das Modell ist auf der Strecke geblieben. Design-Modell und Code haben sich derart auseinander entwickelt, dass nur noch der Code am Leben erhalten werden kann. Die gehofften Investitionen in bessere Dokumentation, Verstehbarkeit und Wartbarkeit sind grossen teils verloren, der eigentliche Sinn in vielen Fällen verfehlt.

Können Sie sich noch an die ersten C-Compiler erinnern? Aus den speicherfressenden Monstern der 80er Jahre sind Werkzeuge geworden, die an Optimierung inzwischen die meisten Assemblerprogrammierer glatt in den Schatten stellen. Erweiterungen machen es möglich, auch spezielle Hardwareigenschaften der Mikrocontroller effizient zu programmieren, und niemand wird ernsthaft anzweifeln, dass 8-Bit Projekte heute sehr effizient in C programmiert werden können.

Eine analoge Entwicklung hat in den letzten Jahren im Bereich Codegeneratoren stattgefunden und so war es nicht verwunderlich, dass sich Kundenanfragen nach einer UML Lösung für die Infineon C166 Familie bei uns häuften. Also begannen wir nach einer passenden Lösung Ausschau zu halten.

UML für 16 Bit, ist das möglich?

Wir stellten die Anforderungen an eine UML basierende Entwicklungsumgebung, für eine typischen 16 Bit Applikation, wie folgt zusammen.

Hauptziel ist eine langfristige hohe Qualität, Verstehbarkeit, Änderbarkeit, Portierbarkeit und Dokumentation der Software auf hohem Abstraktionsgrad zu sichern. Dazu müssen das UML-Modell und der C-Code absolut konsistent gehalten werden. Dafür sind notwendig:

- Die Verhinderung des Bruchs zwischen Analyse, Design und Implementation.
- Daraus resultiert die Forderung nach einer Codegenerierung, sowie
- Möglichkeiten um durchgeführte Änderungen an den C-Quellen wieder in das Modell zu übernehmen (auch als Roundtrip Engineering bezeichnet).

Weiterhin setzten wir folgende Anforderungen bezüglich der Codegenerierung als voraus:

- Der generierte Code sollte gut verständlich sein
- der Overhead durch die Codegenerierung sollte unter 50 KByte bleiben
- Laufzeitverluste sollten durch ein effizientes Handling von Interrupt-Routinen weitestgehend vermieden werden.

Weiter sollte eine genaue Anlehnung an den UML Standard und eine umfassende Unterstützung des Standards den Return of Invest langfristig sichern.

Als Letztes hatten wir noch die Forderung, vorhandene C-Sourcen weiterhin nutzen zu können (auch als reverse Engineering bezeichnet).

Mit diesen Forderungen haben wir uns dann verschiedene UML Umgebungen angesehen, und festgestellt, das es eine derartige Lösung für 16 Bit Systeme nicht gab. Was konnten wir tun? Ein eigenes UML basierendes Tool zu entwickeln war realistisch nicht möglich, eher schon ein vorhandenes Tool auf unsere Anforderungen anzupassen.

Da das Tool Rhapsody von I-Logix (heute Telelogic) bereits von einigen unserer Kunden in die engere Wahl gezogen wurde und es auch von allen verglichenen Tools am dichtesten an unseren Anforderungen lag, nahmen wir Kontakt mit der Firma I-Logix auf. Es stellte sich heraus, dass Rhapsody die notwendige Flexibilität besitzt um die eigenen Erweiterungen durchzuführen. Es schien sogar mit wenigen Mannwochen Aufwand realisierbar zu sein.

Die Codegenerierung unter Rhapsody setzt einen Scheduling Mechanismus voraus, und so benötigten wir noch ein RTOS. Da in unserem Hause gute Erfahrungen mit CMX-RTX gemacht wurden, es ist ein kleines unkompliziertes Standard RTOS und mehr



Abbildung 2:

Wird der Ressourcen-Bedarf von Software nach dynamischen und statischen Gesichtspunkten betrachtet, dann stellt sich heraus, dass i.d.R. 80% der Software nur 20% der CPU Auslastung bewirken und umgekehrt. Überwiegend die Interrupt Funktionen sind vom Umfang (statisch) sehr klein, bewirken aber (dynamisch) eine hohe Laufzeitauslastung.

Da 10 % mehr oder weniger Speicher bei vielen Applikationen heute nicht so sehr ins Gewicht fallen, ist es möglich auch bei kleineren Systemen auf einem Höheren Abstraktionsgrad Software zu entwickeln, wenn es auf die 80% der nicht laufzeitkritischen Bereiche beschränkt bleibt. Wird die Laufzeitkritische Treiber- und Interrupt-Schicht herkömmlich in C oder wenn notwendig sogar in Assembler programmiert, entstehen keine negativen Auswirkungen auf die Performance des Gesamtsystems.

Durch diese Aufteilung der Software wird eine sehr gute Portierbarkeit, Änderbarkeit und dadurch eine lange Lebensdauer der eigentlichen System- bzw. Applikations-Software erreicht.

wurde für diesen Zweck nicht benötigt, war auch dort die Wahl schnell getroffen.

Erfreut stellten wir fest, dass es auch schon eine Anpassung des Frameworks für dieses RTOS gab und innerhalb weniger Tage erhofften wir ein lauffähiges System zu haben, das unseren Anforderungen schon sehr nahe kam, und damit fingen die Probleme an. Nach und nach stellten wir fest, dass die UML-Welt ursprünglich eben doch für die 32-Bit Prozessoren geschaffen wurde. Angefangen bei Pointer Size, über erwartetes dynamisches Kreieren von Tasks, bis hin zum Starten einer Timer Task für jeden Prozess, wurde uns Schritt für Schritt klarer, was es noch alles für Implementierungen bei der Codegenerierung gab, die in dieser Form für 16-Bit Systeme nicht akzeptabel waren.

Zum Glück hatten wir vorher keine Ahnung, sonst hätten wir das Projekt wahrscheinlich gar nicht erst angefangen. Nur der Umstand, dass wir uns ständig kurz vor dem Ziel gesehen haben, hat uns dann vier Monate adaptieren, programmieren und nach kreativen Ideen

suchen lassen, und so stehen wir nun vor einem Framework, das die gestellten

Anforderungen inklusive Codegenerierung erfüllt.

Das Ergebnis:

Der Basis-Overhead der Codegenerierung inklusive RTOS liegt weit unter 30 KByte ROM und ca. 100 Byte RAM. Dazu müssen ca. 5% grösserer Code durch die OO-Programmierstrukturen gerechnet werden. Das ist jedoch stark abhängig vom Programmierstil des jeweiligen Entwicklers. Eine saubere sicherheitsbewusste Programmierung benötigt in etwa die gleiche Grösse.

Zur Laufzeit lässt sich sagen, dass ein objektorientiertes System, basierend auf preemptiven Multitasking, sehr effizient läuft und wir keine Laufzeiteinbußen feststellen konnten. Die sehr guten Laufzeitverhalten von preemptiv designeten Systemen erhöhen jedoch den RAM-Bedarf um einige 100 Byte.

Lediglich die Interruptservice-Routinen sollten weiterhin herkömmlich programmiert werden. Die Prioritäten für die wichtigsten Interruptservice-Routinen werden dann über dem RTOS und dem UML Framework angelegt. Damit wird die Interrupt Latency nicht beeinträchtigt und die volle Performance steht

weiterhin zur Verfügung. In diesem Fall kann natürlich auf den Komfort des RTOS und der OO- Eigenschaften innerhalb der Interrupt Routinen nicht zugegriffen werden.

Aber nicht alle Projekte eignen sich für den Einsatz von Codegeneratoren. Wir sehen bei Single Chip Lösungen mit weniger als 4 K RAM derzeit die Grenze, an der die Ressourcen zu knapp sind, um Systeme auf Basis von UML und der Generierung von Code zu designen, es sei denn, es werden gezielt auf bestimmte UML Konstrukte verzichtet.

Abschliessend bleibt uns zu sagen, dass wir inzwischen von der Möglichkeit, die UML auch für 16-Bit Systeme einzusetzen, überzeugt sind. Wir gehen noch einen Schritt weiter, und behaupten, die Vorteile sind so gross, dass in den kommenden Jahren kein Entwickler an UML vorbei kommen wird. UML ist eine Droge, sind die ersten Klippen der Einarbeitung überwunden, ist eine Softwareentwicklung im herkömmlichen Stiel nicht mehr denkbar. Besonders im Vergleich zu C++ bietet der Einstieg in die OOP mit UML einen gravierenden Vorteil, UML führt von sich aus zu einer Objekt Orientierten Denkweise. Das haben uns unsere Entwickler und unsere Kunden mehrfach bestätigt.

Jetzt fragen Sie sich vielleicht **„ob Sie es sich leisten können in eine neue Technologie, wie die UML, derzeit zu investieren“** Diese Frage ist ungünstig, denn haben Sie jemals erlebt, dass Sie in einem Folgeprojekt mehr Zeit hatten, oder die Randbedingungen für Investitionen in neue Technologien günstiger waren als in dem vorherigen Projekt? Fragen Sie sich besser **„wie lange Sie mit der jetzt**

eingesetzten Technik noch effizient genug Software entwickeln können“.

Entsprechend einer Studie des Wissenschaftsministeriums liegt die eigentliche Wertschöpfung bei 70% aller in Deutschland entwickelten Produkte nicht mehr in der Hardware, sondern in der Software. Wie sieht es bei Ihnen aus? Entspricht der Werteanteil der Software an dem Produkt auch der Stellung der Softwareabteilung in Ihrer Firma oder wird auch bei Ihnen die Software gratis entwickelt? Lassen Sie sich nicht dadurch blenden, dass bei der Hardwareproduktion jedes Bauteil exakt kalkuliert und jedem einzelnen Stück bis auf den Cent genau zugeordnet werden kann. Kennen Sie die Produktionskosten Ihrer Software pro Stück? Was ist der anteilige Wert Ihrer Software an dem Endprodukt? Irgendwann kommt der kritische Punkt, an dem die eigentliche Wertschöpfung zum überwiegenden Anteil durch die Software generiert wird. Spätestens dann kann es existenzbedrohlich werden, wenn die Softwareentwicklung und damit die eigentliche Wertschöpfung uneffizient wird. Denn dann kann die Konkurrenz leicht davon ziehen.

Der Zahn der Zeit nagt und der Übergang von Hardware zur Software schreitet unaufhaltsam voran. Das ist an sich weder gut noch schlecht, entscheidend ist wie wir damit umgehen, und in wie weit wir bereit sind, früh genug die richtigen Technologien einzusetzen.

Andreas Willert (awillert@willert.de)
Willert Software Tools GmbH
(www.Willert.de)