Bachelorarbeit

Integration der rollentypischen Abläufe für Embedded Software Entwicklung und entwicklungsnahe Tests in die Application Lifecycle Management Plattform von Microsoft (Team Foundation Server und Visual Studio)

Zenker, Alexander

geb. am 02.02.1991 in Zwickau

Studiengang Informatik
Studienrichtung Systementwicklung

Westsächsische Hochschule Zwickau (FH)
Fachbereich Physikalische Technik/Informatik
Fachgruppe Informatik

Betreuer, Einrichtung: Prof. Dr. Wolfgang Golubski, WH Zwickau (FH)
Dipl. Ing. Thomas Schäfer, conplement AG

Abgabetermin: 05.01.2013

Selbständigkeitserklärung gem. § 22 Absatz 5 BPO

Hiermit versichere ich, Alexander Zenker, dass ich die vorliegende Bachelorarbeit mit dem Titel „Integration der rollentypischen Abläufe für Embedded Software Entwicklung und entwicklungsnahe Tests in die Application Lifecycle Management Plattform von Microsoft (Team Foundation Server und Visual Studio)“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

.....	.....
Ort, Datum	Unterschrift

Autorenreferat

Diese Arbeit befasst sich mit der Analyse der von Visual Studio vorgegebenen Schnittstellen, um einen externen Debugger zu integrieren. Im Anschluss wird exemplarisch eine Integration am Trace32 durchgeführt. Auftretende Schwierigkeiten und theoretische Aspekte, die für diese Integration gelöst wurden, werden analysiert und das Vorgehen begründet.

Die praktische Arbeit beschränkt sich dabei auf die Erstellung einer eigenen Debugg-Engine, die als Kommunikationsbrücke zum externen Debugger dient. Unterstützente Features sind das Starten des Debuggvorgangs, das Stoppen des Debuggvorgangs, das Erstellen eines Breakpoints sowie das Fortführen des unterbrochenen Programmablaufs.

Vorwort

Das Thema der Arbeit wird von der conplement AG betreut. Die conplement AG ist eine Softwareentwicklungsfirma, welche im Dienstleistungssektor tätig ist. Sie unterstützt Kunden beim konzeptionellen Entwurf über die Entwicklung bis hin zur Betreuung von Softwareprojekten. Es betreute mich seitens der Firma Thomas Schäfer und Karsten Kempe. Mein Betreuer an der Westsächsischen Hochschule ist Prof. Dr. Wolfgang Golubski.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ALM	Application Lifecycle Managment
API	Application Programming Interface
GUID	Globally Unique Identifier
SDK	Software Development Kit
UI	User Interface

1 Einleitung

Das Unternehmen Microsoft verstärkt sein Repertoire bezüglich Application Lifecycle Managment (ALM) Systeme. Das ist auch verständlich, da der Trend der Softwareindustrie zu zentralen „All-in-One“- Tools geht. Ein beispielhafter Vertreter für „All-in-One“-Tools ist an dieser Stelle das SAP-System. Der Trend ist darin begründet, dass durch die Verwendung von nur einem Werkzeug die gesamte Produktivität gesteigert wird **[Sch12]**. Da der Softwareentwickler seine Aufgaben mit einem Werkzeug erledigen kann, muss er nicht ständig zwischen verschiedenen Anwendungen hin und her wechseln. Neben diesem entfallenden Overhead ist durch teils individuellen Bedienkonzepten auch ein gewisses Maß an Umdenkzeit zwischen den Anwendungen notwendig. Diese entfällt bei einer Einzellösung. „Darüber hinaus ist eine deutliche Konsolidierung („Weniger ist mehr“) spürbar, was [zu] einer zunehmenden Abkehr von dezentralen Einzellösungen“ **[Amb10]** in der Softwareentwicklung führt. Ein ALM-System beinhaltet Werkzeuge, welche es erleichtern ein Produkt in seinem Lebenszyklus zu überwachen, angefangen bei der Idee über die Entwicklung, bis hin zum Ausliefern und nachträglichem Supporten des Produktes. Der größte Vorteil eines solchen Systems ist, dass jeder, der an dem Projekt beteiligt ist, mit demselben Werkzeug arbeitet. Dadurch kann jeder unabhängig ob Kunde, Projektleiter, Entwickler oder Tester sehen woran die Anderen derzeit arbeiten, ohne sich mit dem, für den Bereich zum Teil recht spezifischen Werkzeugen, auseinandersetzen zu müssen. Ebenfalls muss man sich nicht um die Beschaffung einer gültigen Lizenz beschäftigen sowie die Installationen der einzelnen Werkzeuge durchführen **[Cha12]**.

Visual Studio 2012 ist zurzeit die neueste Entwicklungsumgebung von Microsoft. Genau wie Visual Studio 2010 ist sie hauptsächlich für die Windows-Entwicklung optimiert beziehungsweise ausgestattet. So erzeugt der hausinterne C-/C++-Compiler nur auf x86 und x64 Prozessoren lauffähige Programme **[Mic12]**. Dies macht Visual Studio im Embedded-Bereich unbrauchbar, da hier zum Beispiel häufig 8-Bit-Architekturen zum Einsatz kommen. Mit *„39% Anteil* [ist] *die 8-Bit-Familie immer noch die größte Gruppe“* **[Ele06]** der Mikrocontroller. Allerdings bietet dafür Visual Studio Möglichkeiten, um zusätzliche Architekturen zu integrieren an. Möglich sind zum Beispiel eigene Projektvorlagen, eigene Werkzeug-Fenster sowie die Integration einer komplett eigenen Programmiersprache.

Die Motivation dieser Arbeit ist, mit Visual Studio ein „All-in-One“-Tool zu konzipieren, mit dem der Embedded-Entwickler seine Aufgaben erfüllen kann und dabei von allen Vorteilen des ALM-Systems profitiert.

Schaut man sich den Lebenszyklus eines Softwareproduktes detailliert an, kann man diesen in sechs große Bereiche einteilen: Erstellen der Anwendung, Installieren der Anwendung, Testen der Anwendung, Debuggen der Anwendung, Entwickeln der Anwendung sowie automatisch generierte Berichte erstellen zum Status der Anwendung. Diese sechs Gebiete sind in Abbildung 1 dargestellt.

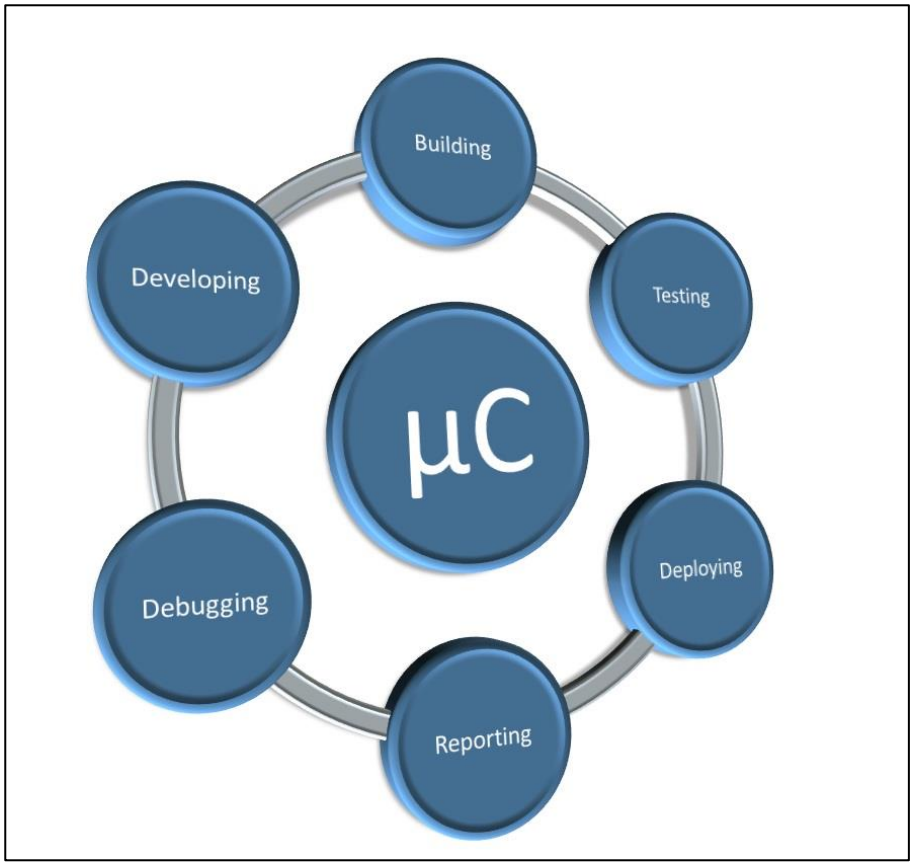

Abbildung 1: ALM-Kreis

Das „µC“ steht hier hierbei für Mikrocontroller und soll als Symbol für jeden im Embedded-bereich eingesetzten Mikrocontroller stehen.

„Building“ ist die Rubrik, welche sich mit dem Erstellen der Anwendung befasst. Abhängig vom verwendeten Mikrocontroller muss der entsprechende Compiler aufgerufen werden. Anschließend werden die einzelnen kompilierten Objektdateien mit dem Linker zu einer Binärdatei zusammengefasst. Je nach Mikrocontroller können jetzt weitere Tools notwendig

sein, um eine für das Ziel-System ausführbare Anwendung zu erhalten. Dieser Prozess ist in Abbildung 2 dargestellt.

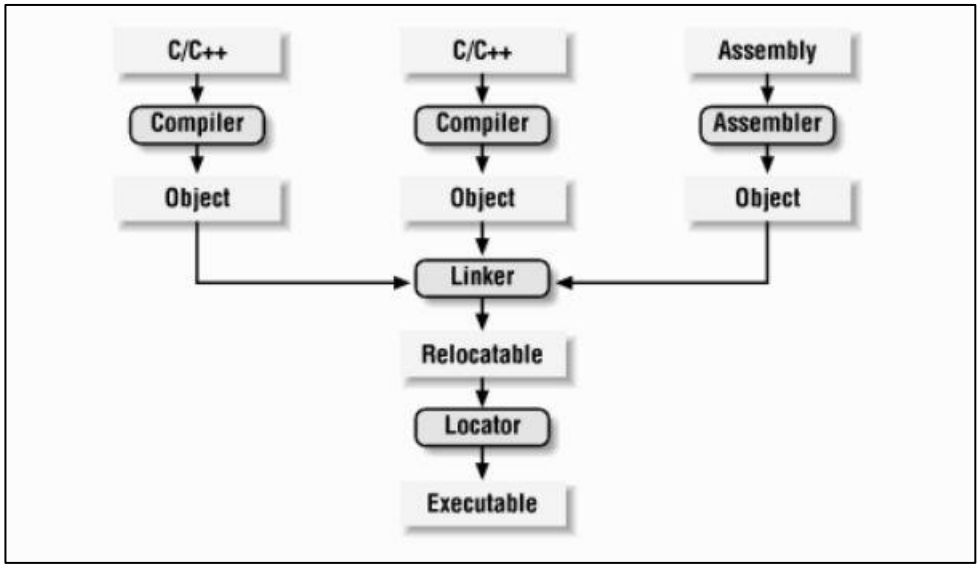

Abbildung 2: Embedded Software Prozess [Bar99]

Beim **„Testing“** bzw. Testen, wird die Qualität einer Anwendung gemessen. Mit automatisch ausführbaren Tests kann zu jedem Zeitpunkt festgestellt werden, ob die Anwendung den gestellten Anforderungen genügt **[Mic121]**. Mit Code-Coverage kann man dann die Qualität der Tests kontrollieren **[Kos04]**.

Mit dem Wort **„Deploying“** wird die Installation des Programms vom Entwicklungs-PC oder Server auf ein Ziel-System verstanden. In dieser Arbeit wird als Ziel-System immer ein Embedded-Controller gemeint sein. In diesem Bereich könnte man die fertige Anwendung zum automatischen Testen an Emulatoren weiterreichen oder auf echte Embedded-Hardware überspielen und diese ebenfalls automatisiert testen.

Beim **„Reporting“** geht es um die Berichterstellung mit denen der Projektleiter den Status des Projektes überwachen kann **[Mic123]**. In dieser Rubrik werden die Berichte erstellt, die je nach eingesetztem Softwareentwicklungsprozess variieren können. Bei einem Scrum-Prozess benötigt man zum Beispiel einen „Sprint Burndown“ oder einen „Build Summary Report“ **[Mic124]**. Sollte man andere Entwicklungsprozesse einsetzen, kann man individuelle Berichte hinzufügen.

Beim **„Debugging“** handelt es sich um den Bereich, wo die Softwareentwickler aktiv ihre erstellte Anwendung testen können und dabei diverse Möglichkeiten bereitgestellt bekom-

men, den derzeitigen Zustand ihrer Anwendung zu analysieren. Durch diese zusätzlichen Informationen können vorhandene Ursachen für Fehler schneller gefunden werden. So bietet Debugging dem Softwareentwickler zusätzlich eine einfache Möglichkeit die Anwendung manuell zu testen.

Beim **„Developing“** bzw. Entwickeln kann man viele nützliche Werkzeuge bereitstellen, die das Leben eines Entwicklers einfacher machen. So könnte man zum Beispiel neben C/C++, welches von Visual Studio supportet wird, auch die mikrocontrollerspezifischen Assembler-Sprachen integrieren. Dadurch erleichtert man das Entwickeln in der jeweiligen Sprache und kann weitere Werkzeuge, die den Arbeitsprozess erleichtern, bereitstellen. Zusätzlich sind Refactoring-Werkzeuge für C/C++ ebenfalls eine gute Erweiterung, da solche Werkzeuge nicht im Visual Studio enthalten sind. Hier kann man aber auch auf bereits vorhandene Produkte zurückgreifen. Zum Beispiel bietet Visual AssistX für C/C++ einige Refactoring-Features, wie das Umbenennen von Methoden, das Erstellen von Deklarationen und noch vieles mehr **[Who12]**.

Im Kapitel zwei werden die Grundlagen zu dieser Arbeit erläutert sowie benötigte Begriffe definiert und erklärt. Weiterhin wird in diesem Kapitel auf den aktuellen Stand der Technik eingegangen.

Im Kapitel drei wird der Entwurf der Arbeit aufbereitet. Es wird auf die Ziele der Arbeit, die verwendeten Werkzeuge sowie auf die Ausgangslage der Arbeit eingegangen. Ebenfalls werden grundlegende Entwurfsentscheidungen der Arbeit reflektiert.

Kapitel vier behandelt die Probleme der Arbeit. Es werden die Lösungsansätze zu den Problemen dargestellt. Behandelt werden Probleme vom Anpassen der Projekteigenschaften über das Integrieren eines externen Debuggers bis zum Erstellen eines Breakpoints.

Im Kapitel fünf wird das Ergebnis der Arbeit bewertet sowie eine Kompatibilitätsübersicht zu der in der Arbeit entwickelten Lösung dargestellt.

Im abschließenden Kapitel sechs gibt es eine Zusammenfassung dieser Arbeit sowie einen Ausblick hinsichtlich Erweiterbarkeit oder Tiefgang in bestimmten Teilproblematiken.

2 Grundlagen und Stand der Technik

2.1 Debugger

Ein Debugger ist ein Programm das hilft, Ursachen von Fehlern oder Bugs in einer programmierten Anwendung zu finden. Es stellt dabei Werkzeuge zur Verfügung, die den Ablauf des Programms überwachen. Beispiele für diese Werkezuge sind Anzeigen von Werten der Variablen, den aktuellen Stack-Frame oder das Anzeigen von Auszügen aus dem Speicher-Dump. **[Wil02]**. Ein Debugger kann sowohl Software wie auch Hardware sein. In jedem Fall existiert zwischen dem Debugger und dem zu debuggenden Mikrocontroller eine physische Verbindung. Die Mikrocontroller selbst müssen das Debuggen unterstützen und entsprechende Schnittstellen für das Debuggen anbieten. Manche Mikrocontroller können einen Debugger sogar Onboard haben. Diese können dann Informationen dem Entwickler direkt zur Verfügung stellen und benötigen dadurch keine zusätzliche Software. **[The12]**. Eine gute Einführung in das systematische Debuggen findet man in diesem Buch: **[Zel06]**.

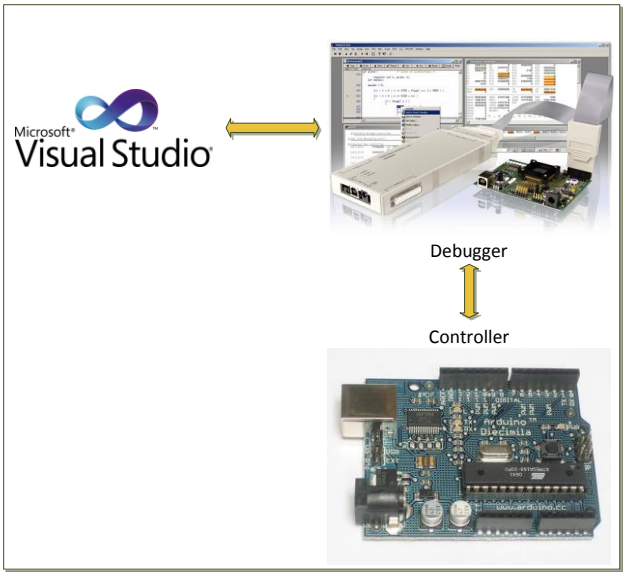

Abbildung 3: Visual Studio, Debugger & Mikrocontroller Beziehung ([Zum1230], [Tri10], [Gee12]**)**

Ein Beispiel für einen Debugger ist der Lauterbach Debugger. Dieser besteht sowohl aus Software, welche man Trace32 nennt, wie auch Hardware. Der Trace32 unterstützt insgesamt über 50 Prozessoren-Architekturen und wird in über 80.000 Entwicklerplätzen eingesetzt. Der Lauterbach Debugger wird von der Lauterbach GmbH hergestellt. Das Unternehmen wurde 1979 gegründet und ist zurzeit der führende Markthersteller im Bereich von „Embedded-Hardware-Debugg“-Werkzeugen **[Lau12]**.

2.2 Erweiterungsmöglichkeiten von Visual Studio

Um Visual Studio zu erweitern, wird das für die Version entsprechende Software Development Kit (SDK) benötigt. Das SDK besteht aus einer Sammlung von Klassen und Objekten. Diese stellen ein Interface zum Visual Studio bereit, über welches man zusätzliche Funktionalitäten integrieren kann. Eine solche Erweiterung kann man nicht mit allen Visual Studio Edi-

tionen erstellen. In Visual Studio 2010 benötigt man die Ultimate Edition, während bei Visual Studio 2012 bereits die Premium Edition ausreicht.

Eine solche Erweiterung kann in Form eines Add-Ins in Visual Studio integriert werden. Ein Add-in ergänzt ein bestehendes Programm um zusätzliche Funktionalitäten. In diesem Zusammenhang sei auf folgende Literatur hingewiesen: **[Mue09]**. Diese gibt einen recht guten Einblick in das Automatisierungsmodell von Visual Studio und wie man ein eigenes Add-In schreibt.

An dieser Stelle macht sich ein Blick auf das Automatisierungsmodell von Visual Studio recht gut, welches im Anhang (Abbildung 21, Seite: 54) dargestellt ist.

Ein Teil des SDK von Visual Studio ist das Debugger-SDK. Dieses beinhaltet die Debugg-API von Visual Studio. Mit Hilfe der Debugg-API erhält man ein Interface, welches Zugriff auf alle Visual Studio Debugg-Features bietet, wie zum Beispiel das Setzen von Breakpoints, Aufbau eines Stack Frames sowie Thread Überwachung.

2.3 Projekt-Eigenschaften

Visual Studio verwaltet spezielle projektabhängige Einstellungen in den Projekt-Eigenschaften. Dort können interne Werkzeuge entsprechend den Anforderungen konfiguriert werden. Zum Beispiel kann der Name der zu erstellenden ausführbaren Datei angegeben werden. Zusätzlich werden das Ziel-System der Anwendung oder auch einfach spezielle Optionen für den Compiler oder Linker in den Projekt-Eigenschaften gespeichert.

Visual Studio besitzt für das Erstellen solcher Projekte die „Microsoft Build Engine“ (MSBuild). Dieses Programm ist dafür zuständig, die erfoderlchen Werkzeuge mit den richtigen Parametern, welche in den Projekt-Eigenschaften geschrieben wurden, aufzurufen. Mit MSBuild wurde ein neues XML-basiertes Projektdateiformat eingeführt, das einerseits leicht verständlich und erweiterbar ist, und andererseits von Microsoft vollständig unterstützt wird. Entwickler können mit MSBuild beschreiben, wie der Erstellungsvorgang unter verschiedenen Plattformen und Konfigurationen auszuführen ist **[Mic125]**. Erst mit Visual Studio 2010 wurde das alte VCBuild System von MSBuild abgelöst **[Mic126]**.

In Abbildung 4 sieht man ein Ausschnitt einer Projekt-Eigenschaftsseite. Der gelb markierte „Local Windows Debugger“ ist der standardmäßig eingestellte Debugger für Visual Studio

C++-Projekte. Hier kann man auswählen, welchen Debugger man verwenden möchte und anschließend kann man diesem einige Parameter mitgeben. Das ist im rot umrahmten Teil des Bildes zu sehen. Als Beispiel kann man das Arbeitsverzeichnis des Debuggers hier einstellen. Das entspricht in der Abbildung „Working Directory“, der als Parameter das Projektverzeichnis übergeben bekommt.

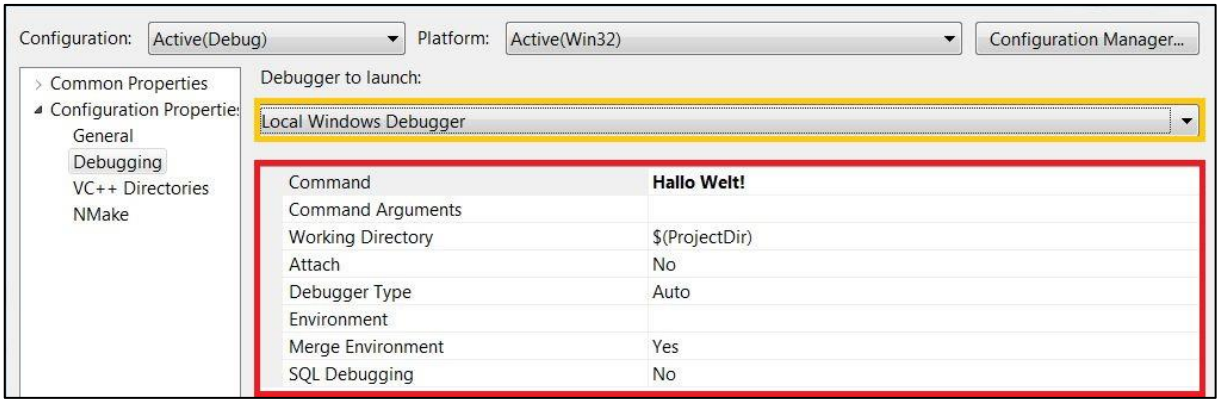

Abbildung 4: Standart Debugg Projekteigenschaftsseite (Screenshot: Visual Studio®)

2.4 Debug Engine

Eine Debug Engine ist ein Programm, welches mit Visual Studio über ein bestimmtes Interface kommuniziert. In Abbildung 5 wird die Beziehung zwischen Visual Studio mit seinen internen Komponenten und der externen Debug Engine dargestellt. Die Debug Engine übernimmt dabei die Rolle des Debuggers und versorgt den Session Debug Manager von Visual Studio mit Informationen zum aktuellen Debugg-Vorgang. Neben den Visual Studio internen Debug Engines kann man über das Visual Studio SDK bereitgestellte Interface eigene Debug Engines erstellen.

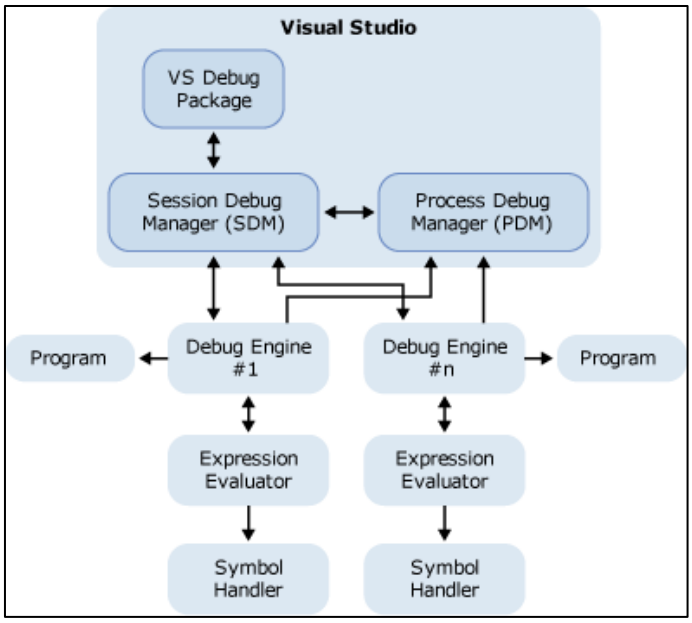

Abbildung 5: Visual Studio Aufbau Debugger Schnittstelle ([Mic1229])

2.5 Debugger Launch Extension

Eine Debugger Launch Extension ist ein spezielles Add-In für Visual Studio. Diese wird ausgeführt, wenn der Benutzer das Debuggen starten möchte. Um eine solche Erweiterung zu erstellen, benötigt man die „Visual C++ Debugger Launch Extension“ von Microsoft, die man am besten mit dem Extension Manager von Visual Studio installieren kann. Damit bekommt man eine neue Projekt-Vorlage, um eine Debugger-Launch-Extension erstellen zu können.

Ein solcher Debugger Launcher ist dazu da die von MSBuild bereitgestellten Projekteigenschaften auszuwerten und eine passende Debugger Engine anhand dieser Werte zu instanzieren und benötigte Parameter an diese weiter zu geben.

2.6 Modul

Unter einem Modul werden in dieser Arbeit die einzelnen kompilierten Quelldateien des Programms verstanden. Jede Quellcodedatei wird vom Compiler zu einem Modul kompiliert. Diese Module werden dann vom Linker zu einem Programm verlinkt.

2.7 Stand der Technik

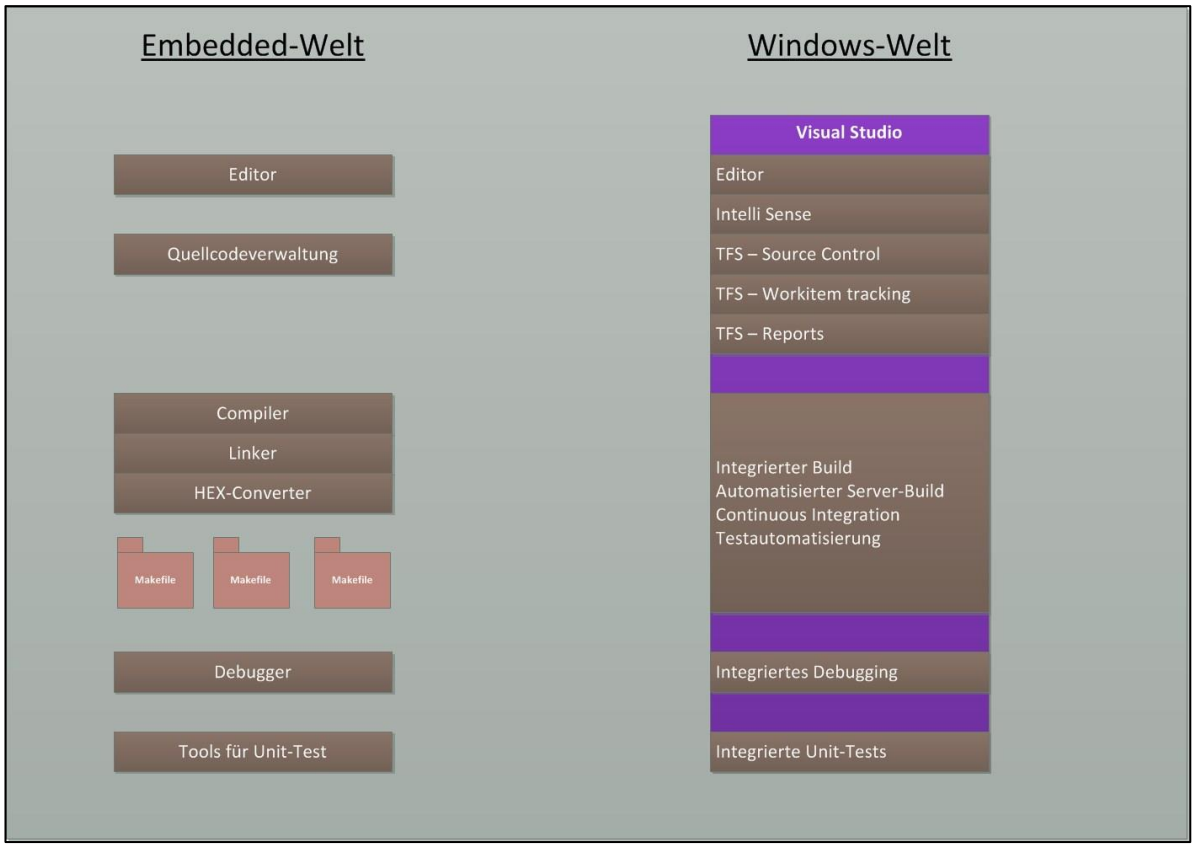

Abbildung 6: Vergleich Softwareentwicklung

In Abbildung 6 ist ein Vergleich zwischen den Prozesslandschaften der Embedded-Welt und der Windows-Welt dargestellt.

Im Embedded-Umfeld ergibt sich durch die Verwendung unterschiedlichster Mikrocontroller von diversen verschiedenen Herstellern, dass sich dann eine Vielzahl an Werkzeugen ansammelt, die man im alltäglichen Entwicklungsprozess verwenden muss. In der Regel verwendet man zum Schreiben des Mikrocontroller-Programms einen Editor. Mit einem weiteren Werkzeug stellt man dann den Quellcode für das Entwicklerteam in ein Quellcodeverwaltungsprogramm zur Verfügung. Um aus dem Quellcode dann ein ausführbares Programm zu machen, müssen diverse Werkzeuge, wie Compiler, Linker oder ein HEX-Converter, zum Einsatz kommen. Ein HEX-Converter ist ein Werkzeug. Dieses Werkzeug erstellt aus der generierten Datei des Linkers ein ausführbares Programm. Um diesen Erstellungsprozess nicht immer wieder neu konfigurieren zu müssen, werden Makefiles erzeugt, damit man den Erstellungsprozess automatisieren kann. Ein Makefile ist eine Textdatei, die Informationen enthält, wie und was ein bestimmtes Werkzeug ausführen soll. Jedes Werkzeug bringt dabei eine eigene Palette an Konfigurationsmöglichkeiten mit sich. Wenn man

dieses erstellte Programm dann debuggen möchte, wird als zusätzliches Werkzeug ein Debugger benötigt. Sollen dann noch Tests gemacht werden, benötigt man noch weitere Werkzeuge. Der Embedded-Entwickler arbeitet dadurch täglich mit einer Reihe von Werkzeugen, die er alle beherrschen muss und die außerdem durch teils herstellerspezifische Eigenheiten ein ständiges Umdenken zwischen den Werkzeugen erfordern.

Auf der anderen Seite gibt es die Windows-Welt mit der Entwicklungsumgebung Visual Studio. Dort werden alle diese einzelnen Werkzeuge in einem einzigen Werkzeug zusammengefügt. Dadurch kann ein Windows Entwickler seine täglichen Aufgaben schneller und effektiver erfüllen. Er muss nicht immer zwischen den verschiedenen Kontexten hin und her wechseln. Ein solches „All-in-One“-Tool erleichtert auch das Anpassen an diverse Unternehmens-Prozesse und erhöht damit die Produktivität des Softwareentwicklers.

Allerdings gibt es einen eingeschränkten Umfang an unterstützten Mikrocontroller-Architekturen in Visual Studio. Bereits in der Einleitung wurde darauf verwiesen, dass Visual Studio nur x86- und x64-Prozessoren unterstützt. Dadurch kann im Moment Visual Studio nicht im Embedded-Umfeld eingesetzt werden.

3 Entwurf

3.1 Allgemeine Ziele

Im Kapitel Stand der Technik (Seite 15) wird die momentane Prozesslandschaft zwischen einer Embedded-Welt und der Windows-Welt dargestellt. Der Kerngedanke der Arbeit ist es, dass der Embedded-Entwickler genau wie in der Windows-Welt alle Werkzeuge in einem „All-In-One“-Tool bereitgestellt bekommt. Der Embedded-Entwickler bekommt damit ein Werkzeug, in welchem er seinen Quellcode entwickeln kann. Zusätzlich soll der Embedded-Entwickler direkt aus der Entwicklungsumgebung heraus den Quellcode mit einem Quellcodeverwaltungsprogramm abgleichen können. Es sollen für einen Mikrocontroller die entsprechenden ausführbaren Dateien direkt in der Entwicklungsumgebung erstellt werden. Im Anschluss daran kann der Embedded-Entwickler diese ausführbaren Dateien debuggen. Außerdem sollen Unit-Tests automatisiert innerhalb der Entwicklungsumgebung durchführbar sein.

Beim Betrachten der Ziele wird schnell klar, dass es unmöglich ist, alle Ziele mit einer Arbeit zu erfüllen. Die Probleme, die hinter den Zielen stehen sind einfach zu komplex. Diese Arbeit baut darum auf die Arbeit des vorangegangen Praxissemesters auf.

3.2 Verwendete Werkzeuge

Die Motivation der Arbeit besteht darin, alle Embedded-Werkzeuge in Visual Studio zu integrieren. Prinzipiell wäre jede andere Entwicklungsumgebung auch geeignet, aber die conplement AG ist zertifizierter Partner von Microsoft und Spezialist in der Microsoft-Produktlinie. Dadurch wird Visual Studio als zentrale Entwicklungsumgebung für diese Arbeit verwendet.

Allerdings auf diesen Punkt muss man noch etwas genauer eingehen, da mehrere Versionen von Visual Studio existieren. Interessant sind hierbei nur die neuesten Versionen also Visual Studio 2010 sowie Visual Studio 2012. Visual Studio 2012 erschien offiziell am 12.9.2012 im Handel. Für MSDN und TechNet-Abonnenten war es bereits seit dem 15.8.2012 verfügbar **[Bau12]**. Da zu Beginn der Bachelorarbeit noch kein SDK für Visual Studio2012 vorhanden war, wird in dieser Arbeit nicht Visual Studio 2012, sondern Visual Studio 2010 Ultimate verwendet.

Das zweite Werkzeug, was eine wichtige Rolle spielt, ist der Debugger. Der Trace32 wird fast ausschließlich von der Embedded-Abteilung der conplement AG verwendet und auch bei den

Kunden kommt dieser Debugger zum Einsatz, zum Beispiel bei der Preh GmbH. Aus diesen Gründen und der Tatsache, dass auf vorhandene Strukturen aus dem Praxissemester aufgebaut werden kann, wird in dieser Arbeit der Trace32 als externer Debugger verwendet.

3.3 Ausgangslage

Ein Teil der Ziele sind bereits in der Arbeit des Praxissemesters umgesetzt. Auf Grundlage des damaligen Projektes wird als prototypischer Mikrocontroller der C166 verwendet. Das ist ein 16-Bit-Microcontroller auf Basis der Von-Neumann-Architektur, welcher von Siemens/Infineon in den 80iger Jahren entwickelt wurde **[web12]**.

Es ist jetzt bereits möglich, mit Visual Studio für den Ziel-Mikrocontroller C166 das Projekt zu erstellen und so die ausführbaren Dateien zu erhalten. Dabei kann man die zum Erstellen benötigten Werkzeuge, wie Compiler, Linker, HEX-Konverter über ein Makefile konfigurieren.

Die Unit Tests sind ebenfalls aus der vorangegangen Praktikumsarbeit bereits integriert. Es wird der Trace32 verwendet, welcher dann jeden einzelnen Test automatisiert durchführt. Die Steuerung erfolgt über Test-Funktionen, welche in einer eigenen Test-Klasse bereitgestellt werden. Ob ein Test erfolgreich oder fehlgeschlagen ist, wird im Test-Explorer von Visual Studio dargestellt.

Diese Arbeit wird sich darum im Kern nur um die Integration eines externen Debuggers in Visual Studio befassen, da das der letzte fehlende Bereich ist.

3.4 Spezielle Ziele

Das Ziel dieser Arbeit wird demzufolge die Integration eines externen Debuggers in Visual Studio sein. Wie bereits im Kapitel Verwendete Werkzeuge (Seite 17) begründet, wird als externer Debugger der Trace32 verwendet.

Dieses Ziel wird noch einmal dahingehend eingeschränkt, dass es im Rahmen der Bachelorarbeit sich nur um einen Prototyp handeln kann. Das heißt, es wird der Trace32 nicht vollständig in Visual Studio integriert, sondern nur diverse Funktionalitäten, die aber ausreichend beweisen, dass eine Integration überhaupt möglich ist. Anhand dieses Prototyps eröffnet sich später die Möglichkeit, je nach Bedarf, ein vollwertiges Produkt zu erstellen.

In dieser Arbeit werden nur die Grundfunktionalitäten des Trace32 in Visual Studio integriert. Zu den Grundfunktionen, die auch jeder andere Debugger beherrscht, zählen das Setzen eines Breakpoints, das Stoppen eines Programms zu einem beliebigen Zeitpunkt sowie das Fortsetzen eines pausierten Programms. Der Trace32 soll intuitiv über das von Visual Studio vorgegebene User Interface (UI) gesteuert werden, um eine bestmöglichste User Experience zu geben.

3.5 Auswahl der Schnittstelle

Visual Studio muss erweitert werden, um einen externen Debugger integrieren zu können. Allerdings bietet Visual Studio sehr viele Schnittstellen. Es muss daher zunächst entschieden werden, welche Schnittstellen in dieser Arbeit verwendet werden.

Das SDK von Visual Studio bietet zum einen die in dem Kapitel Erweiterungsmöglichkeiten von Visual Studio (Seite 11) erklärte Debugg-API, um einen externen Debugger zu integrieren. Zu Beginn der Arbeit stand allerdings die Frage, ob man im Zweifel auch über ein normales Add-In eigene Buttons in Visual Studio integrieren kann. Diese müssen dann dieselbe Funktionalität bereitstellen, die auch mit der Debugg-API realisiert werden kann. Dahinter steckt die Idee mit den eigenen Buttons Zeit zu sparen. Zunächst muss analysiert werden, ob sich diese Idee überhaupt umsetzen lässt, das heißt es soll möglich sein, auch über andere Wege auf die Debugg-Elemente zuzugreifen. Darum müssen zunächst beide Schnittstellen hinsichtlich ihrer Komplexität analysiert werden. Komplex ist hierbei der Aufwand um eine solche Erweiterung einzubinden sowie der Aufwand zum Implementieren und Verstehen der vorgegebenen Interfaces. Neben der Komplexität muss jedoch auch betrachtet werden, ob nicht gegen die User Experience von Visual Studio verstoßen wird. Als letzter Bewertungspunkt soll hinterfragt werden, ob die Schnittstelle für den Anwendungszweck konzipiert wurde. Das heißt, dass durch ein Update oder Patch von Visual Studio die Schnittstelle immer noch kompatibel sein muss.

Mit einem normalen Add-In kann man auf die Debugger Elemente zu greifen. Dazu bietet Visual Studio ein sogenanntes DTE-Objekt, welches die Basis für das Visual Studio Automatisierungsmodell darstellt. Über dieses Modell kann man ebenfalls auf die Breakpoints zugreifen oder auf die Hilfsfenster von Visual Studio, wie den Call-Stack **[Mic1215]**, Watch-Fenster

[Fir12], etc. Es ist also prinzipiell machbar mit einem normalen Add-In die Debugger-Features zu implementieren.

Bezüglich der Komplexität müssen jetzt beide Seiten näher beleuchtet werden. Dazu schaut man sich den Ablauf in beiden Fällen genauer an. In beiden Fällen müssen die Projekt-Eigenschaftsseiten angepasst werden, so dass man über das Projektsystem von Visual Studio den Debugger konfigurieren kann. Für die Debugg-API benötigt man eine Debugger Launch Extension. Diese ist dafür zuständig, eine Debug Engine mit den entsprechenden Projekt-Eigenschaften aufzurufen. Die Debug Engine ist dann der zentrale Punkt, in dem die Logik implementiert werden muss. Notwendig ist die Steuerung des Debuggers und das Austauschen von Nachrichten zwischen Visual Studio und den Debugger. In Abbildung 7 wird dargestellt, wie die drei Elemente unter sich und mit Visual Studio zusammenhängen.

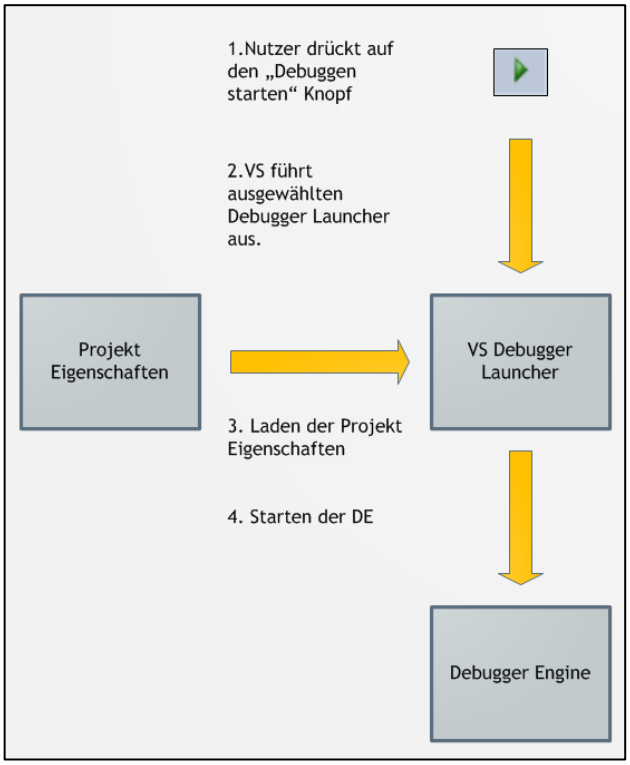

Abbildung 7: Ablauf starten des Debuggens

Der Aufwand für die Installation ist dabei recht hoch. Die Debugger Launch Extension kann zwar mit einem Klick installiert werden, die Debug Engine aber nicht. Diese muss speziell in der Windows Registry registriert werden. Dem gegenüber steht das Add-In, welches mit einem Klick installiert werden kann. Die Integration eines Add-In zur Steuerung des Debuggers ist also einfacher. Hinsichtlich des Aufwandes der zu implementierenden Interfaces ist die Debug Engine ebenfalls im Nachteil. Während diese mehrere Interfaces implementieren muss, kann man bei seinem eigenen Add-In bereits von Visual Studio implementierte Klassen verwenden. Allerdings erfordert dies das Hinzufügen einer eigenen Logik. Die Kommunikation zwischen dem eigenen Add-In und Visual Studio muss selbst erstellt werden. Dadurch ist der Zeitaufwand für die Implementierung der Logik ähnlich dem, der auch für die Implementierung mit der Debugg-API benötigt wird.

Die User Experience verbietet die Verwendung von eigenen Buttons, da die gewohnten Buttons verwendet werden sollen. Dies geschieht bei der Verwendung mit der Debugg-API. Es ist also nicht möglich mit einem eigenen Add-in dieselbe User Experience abzubilden.

Im letzten Punkt wird untersucht, ob die entsprechende Schnittstelle für die Integration eines zusätzlichen Debuggers ausgelegt ist. Dies ist bei der Debugg-API der Fall, jedoch nicht bei der Idee mit den eigenen Buttons.

	Debugger SDK	**Eigene UI Elemente**
Integration Trace32 möglich?	ja	ja
Mit einem Klick installierbar?	nein	ja
Wenig Implementierungsaufwand?	nein	nein
User Experience Visual Studio gleich?	ja	nein
Schnittstelle dafür vorgesehen?	ja	nein

Tabelle 1: Vergleich Debugger SDK und Automatisierungsmodell

In der Tabelle 1 werden nochmal alle Ergebnisse zu den Bewertungspunkten zusammengefasst. Summiert man die Bewertungspunkte gewinnt die Debugger-API vor der Add-in Variante. Ausschlaggebend ist am Ende der Punkt, dass diese Schnittstelle genau für diesen Zweck geschaffen wurde und dadurch ein gewisser Kompatibilitätsgrad impliziert wird.

3.6 Projekteigenschaften

Um das Debuggen mit dem Trace 32 zentral von Visual Studio steuern zu können, ist es notwendig sich, mit den erforderlichen Konfigurationsschritten des Trace32 auseinander zu setzen. In Abbildung 8 sind diese Schritte dargestellt. Wichtig ist es, die für die einzelnen Schritte erforderlichen Informationen festzuhalten. Das sind die veränderlichen Informationen, die der Nutzer im Projektsystem einstellen soll. Zunächst muss von Visual Studio der Trace32 gestartet werden. Dazu ist der Dateipfad zum Programm erforderlich. Zusätzlich kann man Trace32 über eine Konfigurationsdatei mit besonderen Einstellungen starten. Diese Datei ist wichtig, da in ihr zum Beispiel steht, welchen Port die Trace32 API verwendet. Über diesen Port wird später die Kommunikation zwischen der eigenen Debug Engine in Visual Studio und dem Trace32 stattfinden. Aus diesem Grund muss die benötigte Konfigurationsdatei, die verwendet wird, ebenfalls vom Nutzer ausgewählt werden können. Dies ist notwendig, da nicht auf jedem System der gleiche Port vom Trace32 verwendet werden kann.

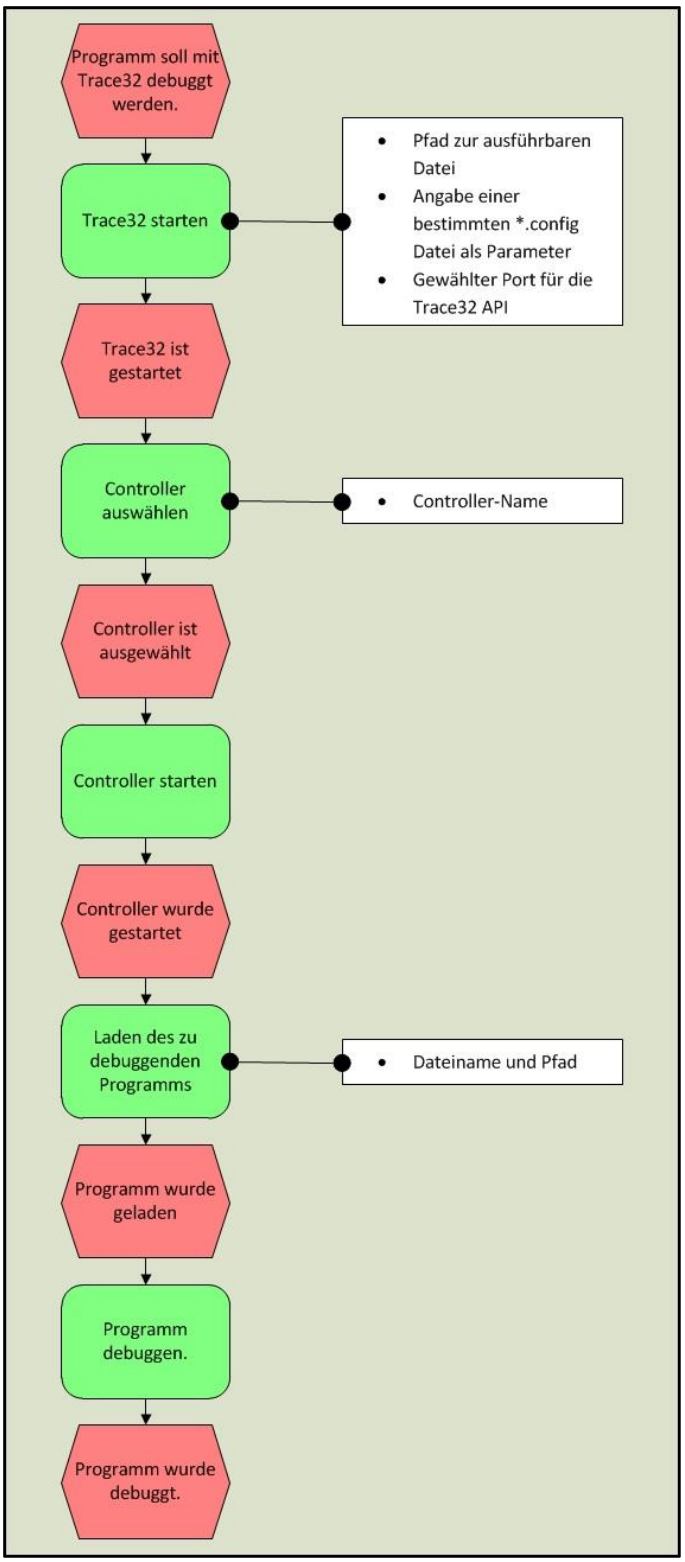

Abbildung 8: Startprozess des Trace32

Nach dem Start des Trace32 muss der Mikrocontroller, auf welchen debuggt wird, ausgewählt werden. Hier reicht es vom Nutzer den Name des Mikrocontrollers zu erfahren. Bei

einer kompletten Integration wäre das allerdings recht fehleranfällig. Da Erstens sichergestellt werden müsste, dass die Eingabe vom Nutzer ein gültiger Mikrocontroller-Name ist. Zweitens weiß man nicht, ob der Mikrocontroller überhaupt unterstützt wird.

Dafür ist eine eigene Plattform für jeden Mikrokontroller notwendig. Wie gewohnt kann der Nutzer diese dann in den Projekteigenschaften aus der Liste der verfügbaren Plattformen auswählen. Das vollständige Erstellen einer solchen Plattform kostet allerdings Zeit, weshalb in dieser Bachelorarbeit keine vollständig eigene Plattform erstellt wird.

Nach der Auswahl des Mikrocontrollers ist es wichtig, dass die Debug Engine weiß, welches Programm debuggt werden soll. Der Nutzer soll dazu den Pfad zum Programm angeben.

Das Ziel der Anpassung ist es, den Trace32 in Visual Studio als Debugger zur Auswahl zu haben. An diesen sollen eigene Parameter übergeben werden. Die Parameter sind dabei die erforderlichen Informationen, die Visual Studio benötigt, um den Trace32 zu steuern. Das heißt, es wird der Pfad zum Trace32 benötigt, die erforderliche Konfigurationsdatei, der zu verwendete Port, der Mikrocontroller-Name sowie die Angabe des Pfades für das zu debuggenden Programm. Das Ergebnis in den Projekteigenschaften von Visual Studio sieht man in Abbildung 9.

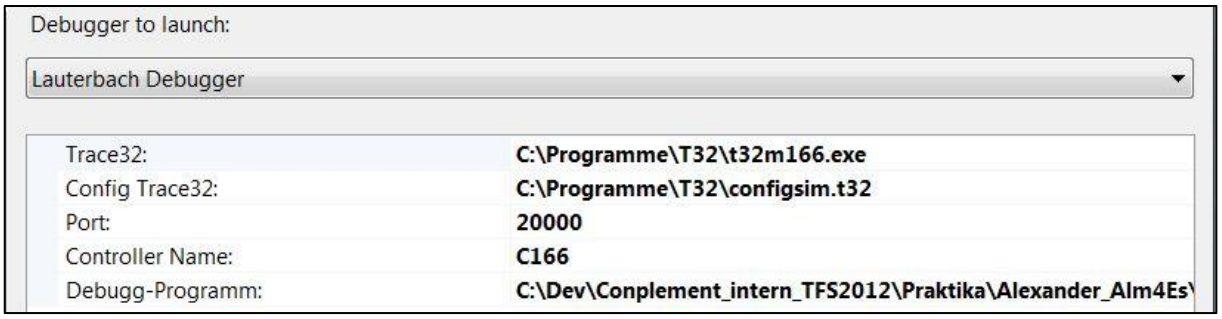

Abbildung 9: Ergebnis Projekteigenschaftsblatt (Screenshot: Visual Studio®)

3.7 Logik

Eine wichtige Fragestellung ist, wo man die Logik zum Ansteuern des Trace32 unterbringt. In Kapitel Auswahl der Schnittstelle (Seite 19) haben wir uns für den grundsätzlichen Aufbau der Implementierung entschieden. Die Debugg-API von Visual Studio erfordert das Erstellen

einer eigenen Debug Engine. Diese kommuniziert mit Visual Studio über vorgegebene Interfaces. Immer wenn der Nutzer von Visual Studio ein bestimmtes Debugg-Ereignis auslöst, wird die Debug Engine ebenfalls benachrichtigt. Sie ist dafür zuständig, dass dann die entsprechenden Debugg-Aktionen durchgeführt werden. Die Debug Engine ist also der zentrale Punkt in dem der externe Debugger integriert wird (Abbildung 10).

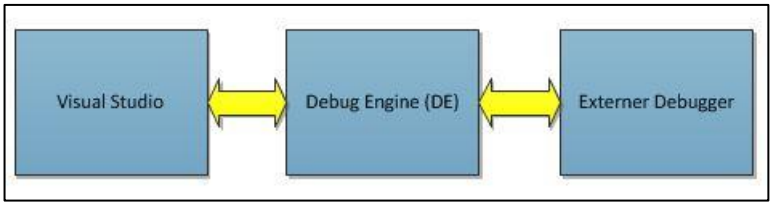

Abbildung 10: Visual Studio, Debug Engine, Externer Debugger Beziehung

3.8 Eventsystem

Visual Studio unterstützt zwei verschiedene Arten von Events. Die Einen sind synchrone Events. Diese werden von Visual Studio sofort zurück an die Debug Engine gesendet und müssen von einer zentralen Methode verarbeitet werden. Der zweite Typ sind asynchrone Events, bei denen der SDM von Visual Studio über dem Event die richtige Methode zuordnet und aufruft. **[Mic1211] [Mic1212]**.

Die synchronen Events haben den Nachteil, dass eine zusätzliche Methode implementiert werden muss. Das kostet Zeit. Der Vorteil dieser Events ist, dass man auf Ereignisse sofort reagieren kann. Da der Prototyp keine performancekritischen Operationen berücksichtigen muss, werden in der Arbeit nur asynchrone Events verwendet.

4 Implementierung

4.1 Anpassen der Projekteigenschaften

Um die Projekteigenschaften anzupassen, muss man im MSBuild-Verzeichnis eine Modifikation vornehmen. Hier hat man jetzt die Auswahl, ob man spezielle Anpassungen für eine oder für alle Plattformen vornimmt. Damit aus dem Prototyp ein vollwertiges Produkt wird ergibt sich die Notwendigkeit, für jeden Mikrocontroller eine neue Plattform zu erstellen. So können für jeden Mikrocontroller die Einstellungen der Werkzeuge in den Projekteigenschaften bereitgestellt werden. Zum Beispiel kann man nur kompatible Debugger zur Auswahl lassen.

Das Erstellen der Projekteigenschaften sowie das Integrieren der Werkzeuge in MSBuild kostet allerdings Zeit. Darum ist in dieser Arbeit der Trace32 Debugger für alle C-\C++-Projekte verfügbar.

Jede Projekteigenschaftsseite wird durch eine XML-Datei repräsentiert. Interessant für die Arbeit sind nur die Dateien, welche mit einem „debugger_“ im Namen beginnen. Zum Beispiel findet man hier die XML-Datei, die den „Local Windows Debugger“ beschreibt. Möchte man seinen eigenen Debugger der Liste hinzufügen, kopiert man sich einfach die „debugger_local_windows.xml“ und benennt die Kopie um. Diese Datei dient als Ausgangslage zum Anpassen. In diesem Dokument kann man die im Kapitel Projekteigenschaften (Seite 22) festgelegten Parameter einfügen. Jeder Parameter bekommt dabei einen Namen mit dem der Wert später vom System identifiziert werden kann und außerdem eine Beschreibung zum Anzeigen in den Projekteigenschaften.

Ein Auszug aus so einer Datei finden sie im Listening 1. Weiterführende Literatur zu diesem Thema finden Sie hier: **[Mic127]**.

```
<Rule Name="Trace32 Debugger" DisplayName="Lauterbach Debugger" …>
…
<StringProperty Name="CustomValue" DisplayName="CustomValue"
                Description="Ein benutzerdefinierter Wert."
                F1Keyword="VC.Project.IVCTrace32DebugPageObject.Command">
    …
  </StringProperty>
</Rule>
```

Listening 1: Eine MSBuild Regel

4.2 Weiterreichen der Projekteigenschaften

Im Kapitel Anpassen der Projekteigenschaften (Seite 25) wird erklärt, wie man die Projekteigenschaften des Projektsystems anpassen kann. Ein Problem wird dabei allerdings nicht geklärt: Wie kommen die benötigten Daten jetzt über den Debugg-Launcher zu der Debug Engine. Die Projekteigenschaften selbst werden der Debugger Launch Extension als Parameter mitgegeben. Für das Übermitteln der Informationen an die Debug Engine existiert eine Optionseigenschaft. Diese dient dazu, zusätzliche Informationen an die Debug Engine zu übermitteln **[Mic1216]**. Über diese Optionseigenschaft werden Daten als in einer Zeichenkette verpackten Nachricht übergeben. Diese Nachricht muss dann in der *SuspendetLaunch-Methode* der Debug Engine wieder entpackt werden. Damit stehen die Projekteigenschaften in der Debug Engine zur Verfügung und können zum konfigurieren des Debuggers verwendet werden.

4.3 Eventsystem

Für die Initialisierung der Debug Engine müssen bestimmte Events zwangsläufig gesendet werden. Die erforderlichen Events, die in jeder Debug Engine gesendet werden müssen, sind in Abbildung 11 (siehe Seite 27) dargestellt. Die Events müssen innerhalb *der IDebugEngine-Launch2::ResumeProcess-Methode* oder der *IDebugEngine2::Attach-Methode* aufgerufen werden. Andernfalls wird mit dem Enden der *ResumeProcess-Methode* Visual Studio abstürzen. Für das Senden von Events wird in der *LaunchSuspendet-Methode* sowie in der *Attach-*

Methode ein *IDebugEventCallback2-Objekt* als Parameter übergeben. Dieses Objekt verfügt über eine *Event-Methode* mit welcher die Events an dem SDM geschickt werden. Um ein Ereignis zu senden, benötigt man diverse Parameter. Als ersten Parameter wird die Engine selbst verlangt. Als nächstes werden der Prozess, das Programm sowie der aktuell debuggte Thread benötigt. Danach folgt das Eventobjekt selbst. Abschließend wird noch eine Referenz auf ein „globally unique identifier“ (GUID) für den Typ des Events sowie ein Zahlenwert für zusätzliche Attribute übergeben **[Mic1230]**.

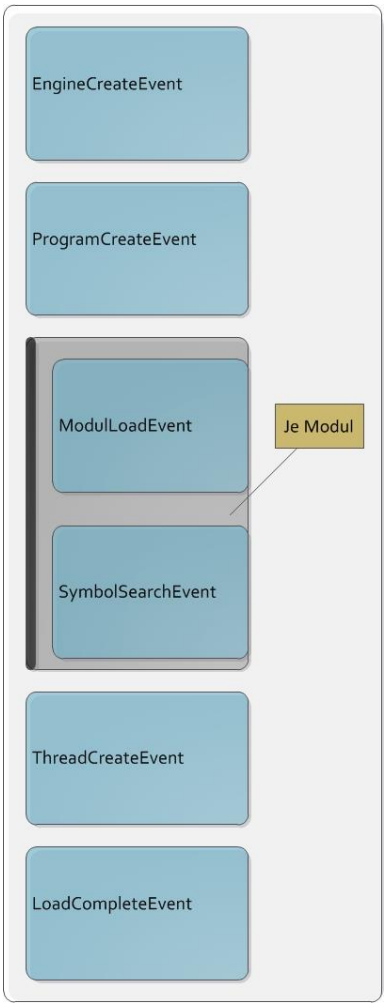

Abbildung 11: Events in Visual Studio

Die Eventobjekte muss man selbst implementieren. Dazu bietet Visual Studio das *IDebugEvent2-Interface*. Dieses Interface erwartet, dass über die *GetAttributes-Methode* die eventspezifischen Attribute zurückgegeben werden **[Mic1214]**. Eine vollständige Auflistung der möglichen Eventattribute finden Sie in dieser Quelle: **[Mic1213]**.

Das erste wichtige Ereignis, das für den Initialisierungsprozess gesendet werden muss, ist das „CreateEngineEvent“. Dieses Event liefert ein Objekt, welches das *IDebugEngineCreateEvent2-Interface* implementiert. Das „CreateEngineEvent“ wird vor dem eigentlichen Prozessstart des zu untersuchenden Programms ausgeführt. Deshalb kann beim Senden des Events noch kein Programm oder Thread angeben werden, der das Event ausgelöst hat.

Mit dem „ProgramCreateEvent“, welches das *IDebugProgramCreate-Interface* implementiert, ändert sich das. Es benötigt bereits zwingend ein DebugProgram-Objekt als Argument. Dieses muss das *IDebugProgram2-Interface* erfüllen. In der praktischen Ausführung erwies es sich als günstig, dass die Engine selbst das *IDebugProgram3* und damit indirekt auch das *IDebugProgram2-Interface* einbettet. Der Vorteil von dieser Vorgehensweise ist, dass man dadurch die bereits zum Großteil in der Engine vorhanden Variablen direkt benutzen kann.

Als nächstes werden die Module des Programms geladen. Je Modul muss im Visual Studio ein „ModulLoadEvent“ sowie ein „SymbolSearchEvent“ geschickt werden. Über das „ModulLoadEvent“ erhält Visual Studio die erforderlichen Informationen über das Modul. Mit dem „SymbolSearchEvent“ weiß Visual Studio, dass zusätzliche Symbolinformationen geladen sind. Jetzt kann Visual Studio bei der Debug Engine die notwendigen Symbolinformationen abfragen. Die Debug Engine kümmert sich um die Aufbereitung des aktuellen Standes des Moduls. Dazu gehört die Belegung der Variablen oder das Berechnen der aktuellen Codezeile. Nach diesen Events beginnt Visual Studio damit bereits gesetzte Breakpoints in den Modulen aufzulösen.

Nachdem die Module geladen sind, wird Visual Studio das „ThreadCreateEvent“ geschickt. Dieses muss mindestens einmal gesendet werden. Selbst wenn die Plattform, welche gerade debuggt werden soll, kein Threading unterstützt. In diesem Fall ist das gesamte zu debuggente Programm als ein Thread zu betrachten.

Abschließend muss noch das „LoadCompleteEvent“ gesendet werden. Damit teilt man dem Visual Studio mit, dass die Initialisierungsschritte abgeschlossen sind und das Programm starten kann.

4.4 Registrieren der Debug Engine

Ein großes Problem ist das Registrieren einer eigenen Debug Engine. Das Thema ist nicht trivial, da man die Debug Engine über Windows Registry Keys mit Visual Studio verbindet. Dafür benötigt man zum einen eine Klassen-Bibliothek in welcher die Debug Engine implementiert wird und man benötigt zum anderen ein Setup Projekt, mit dem man diverse benötigte „Windows Registry Keys“ schreiben kann. Hier sollte man zunächst noch ein wenig auf den Aufbau von Visual Studio eingehen. Die gesamte Debugger Schnittstelle arbeitet mit COM **[Mic128]**. Für die Debug Engine werden zwei Klassen als COM-Objekte regis-

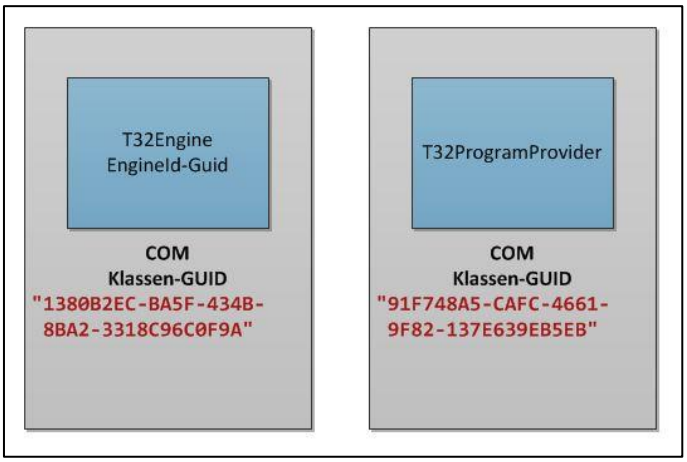

Abbildung 12: Die zentralen COM Objekte

triert. Das sind die Engine-Klasse und eine „Program Provider“-Klasse (siehe Abbildung 12). Jede der beiden Klassen bekommt eine GUID zugeordnet. Zusätzlich bekommt die Engine noch einen internen GUID. Genau dieser Engine-GUID wird für die Debugger Launch Extension benötigt. Der Ablauf nach dem der Debug-Launcher ausgeführt wird, ist in Abbildung 13 dargestellt. Zunächst wird in der Debugger Launch Extension über dem GUID die Debug Engine ausgewählt. Visual Studio sucht dann über diesen GUID den entsprechenden Registry Schlüssel. Anschließend werden die benötigten Informationen aus den Unterschlüsseln und Daten, die sich hinter der Debug Engine befinden, ausgelesen. Visual Studio erhält so die notwendigen Informationen, wo sich unsere Implementierung der Debug Engine befindet. Im Anhang (Seite 55) befindet sich eine vollständige Liste der notwendigen „Windows Registry Keys“.

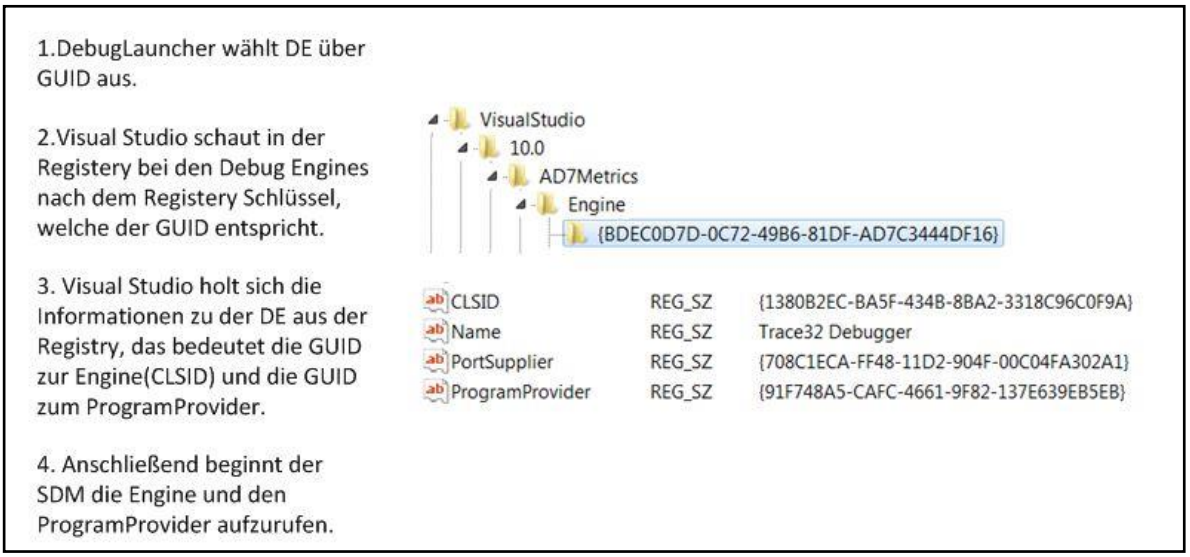

Abbildung 13: Ablaufdiagramm starten der Debugger-Engine

4.5 Starten und Initialisieren des Trace32

Die Debug Engine ist gestartet und die benötigten Konfigurationen stehen dieser zur Verfügung. Nun erfolgt der Start des Trace32 durch die Debug Engine. Dafür wird die *Suspendet-Launch-Methode* eingesetzt. Diese erfordert den Prozess des zu debuggenten Programms als Rückgabe-Parameter. Um die Informationen für das Starten und Beenden des Trace32 Prozesses zu entkoppeln, ist in dieser Arbeit die T32Process-Klasse implementiert.

Nach dem Start des Trace32 kann die API des Trace32 mit dem Verbindungsaufbau beginnen. Die Schwierigkeit bei dem Aufbau der Verbindung besteht darin, dass diese nicht unbe-

dingt zustande kommen muss. Wenn zum Beispiel die API sich nicht richtig abmeldet, kann es beim nächsten Verbindungsversuch zu Problemen kommen. Ein anderer Fall ist, dass der Port unter Umständen nicht verwendet werden kann, weil er bereits von einem anderen Programm verwendet wird. Um diese Fälle ausschließen zu können, muss der Verbindungsversuch überwacht werden. Diese Überwachung übernimmt ein Thread, welcher nach einem bestimmten Zeitintervall den erfolglosen Verbindungsversuch abbricht. Wenn das passiert, wird eine Ausnahme (Exception) geworfen und die Debug Engine beendet den Debugg-Vorgang.

Nachdem die Trace32-API bereit ist, wird in der *Attach-Methode* der Debug Engine der Mikrocontroller ausgewählt und das entsprechende Programm in den Trace32 geladen.

4.6 Befehlsadresse zu Modul/Sourceline

In der Arbeit stellte sich die Frage, wie viele und welche Module unser Programm hat. Hier ergibt sich das Problem der Zuordnung einer Assembler Zeile zur entsprechenden C-Zeile mit Modul (siehe Abbildung 14). In diesem Kapitel wird es um die Lösung zu diesem Auflösungsproblem gehen.

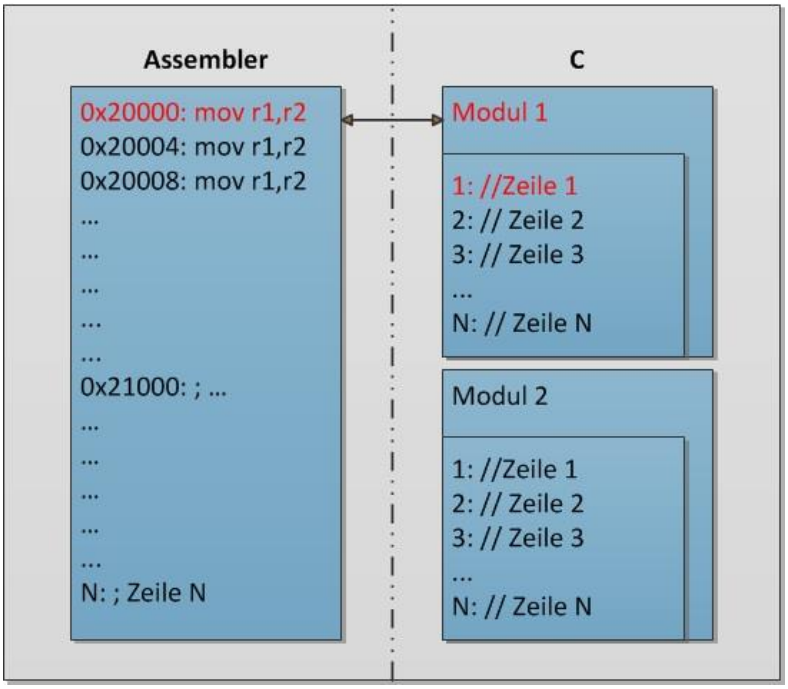

Abbildung 14: Zuordnung zwischen einer Assembler-Zeile zu einer C-Zeile

Die geladenen Module kann der Trace32 in einem Fenster darstellen. Über dem Menüpunkt View->Symbols->Browse Modules kann man zu diesem Unterfenster navigieren. Dort werden alle Module in einer Tabelle aufgelistet (Abbildung 15).

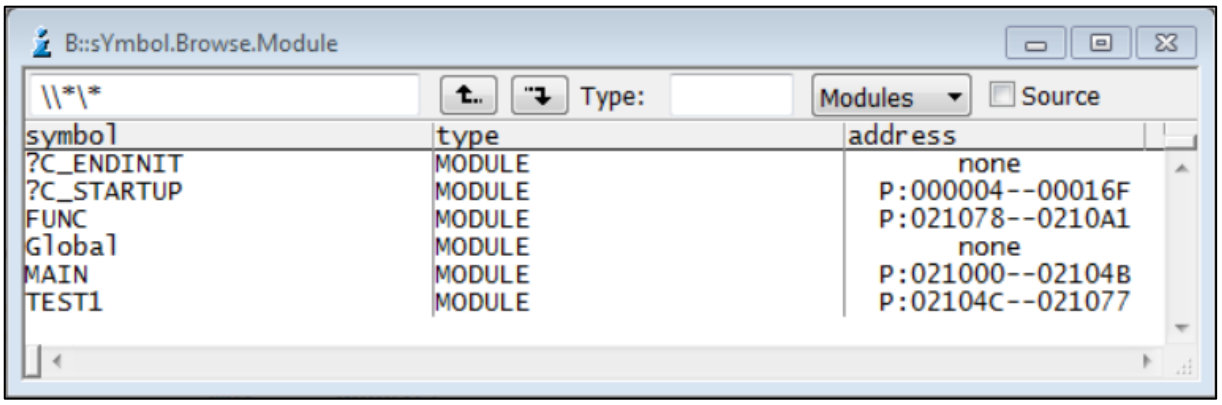

Abbildung 15: Ausschnitt Trace32 Browse.Module Fenster (Screenshot: Trace32®)

Als Name sieht man die Trace32-Konvention. Ebenfalls sieht man die Adressräume, in welchem das entsprechende Modul verlinkt wurde. Setzt man ein Haken bei der Option „Source“, werden die richtigen Dateinamen mit Dateityp angezeigt (Abbildung 16).

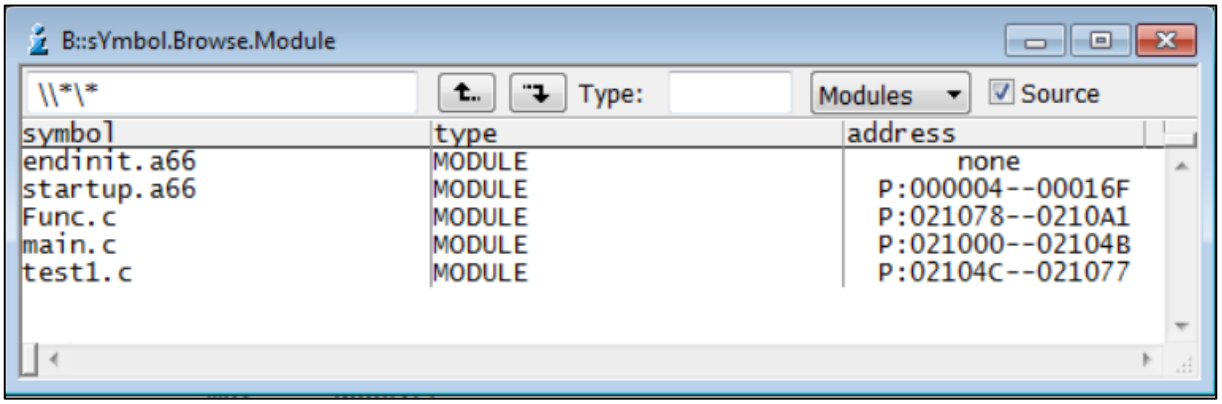

Abbildung 16: Ausschnitt Trace32 Browse.Modul Fenster mit Source (Screenshot: Trace32®)

Zur Identifizierung der Projektdatei von Visual Studio wird der korrekte Dateiname benötigt. Für das Setzen eines Breakpoints an einer C-Zeile wird die Trace32-Konvention benötigt. Darum ist es wichtig, beide Informationen in die Debug Engine zu laden und zu speichern. Dafür musste das Framework um den neuen Datentyp „EmbeddedModul“ erweitert werden. Diese Daten-Struktur speichert den Dateiname, den Modulname sowie den Adressraum des Moduls.

Das nächste Problem ist das Einlesen der Informationen aus dem Fenster von Trace32 für die Debug Engine. Über eine Ausgabe-Umleitung im Trace32 kann man die Daten des Fensters

auch in einer Datei speichern. Diese kann dann von der Debug Engine wieder eingelesen werden, um die benötigten Informationen aus der Datei zu extrahieren. Bei der Implementierung entstand das Problem, in welcher Datei speichert man diese temporäre Information. Das .NET Framework bietet dazu eine Funktion *Path.GetTempFileName*. Die Funktion liefert einen vollständigen Dateipfad zu einer temporären Datei, in welcher Daten geschrieben sowie gelesen werden können. Dadurch muss sich die Debug Engine nicht um das Anlegen einer Datei kümmern. Es entfällt das suchen nach gültigen Speicheradresse mit LeseundSchreibrechten.

Ähnlich kommt man an die Informationen für die Zuordnung zwischen den C-Zeilen und den Assembler-Zeilen. Hier verwendet man nur das Kommando „y.l.line“, darüber öffnet sich wieder ein Fenster ähnlich des Modul-Fensters (Abbildung 17).

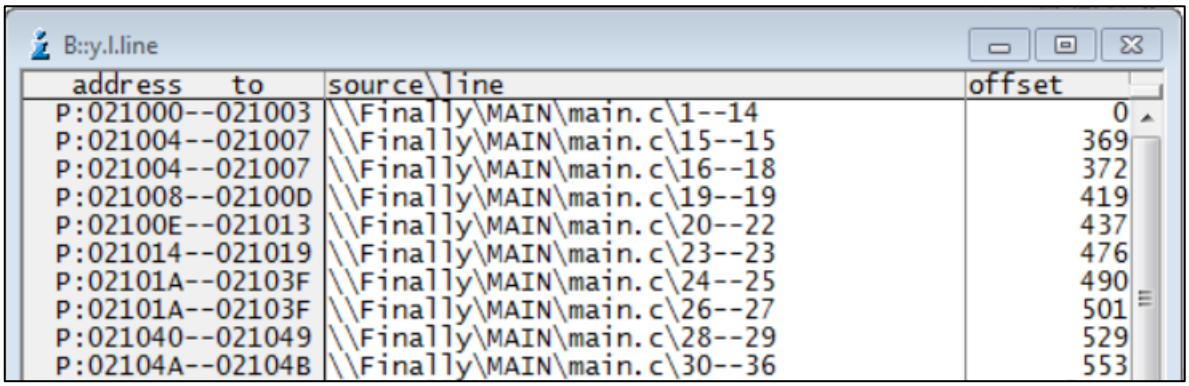

Abbildung 17: Ausschnitt Trace32 Line Fenster (Screenshot: Trace32®)

In diesem Fenster findet man die Informationen für die Zuordnung eines Assembleradressraums zu einer Quellcode-Zeile eines bestimmten Moduls. Die in der „EmbeddedLine“ gespeicherten Informationen sind: die Anfangs- und Endadresse der Assemblerzeilen, das Quellcodemodul, in welchem sich die entsprechende Quellcodezeile befindet sowie die Zeilennummer der Quellcodezeile.

Das Ermitteln der Module und Zeilenzuordnung erfolgt nach dem Attach-Vorgang in der *ResumeProcess-Methode*. Dabei werden auch das ModulLoadEvent sowie das SymbolSearchEvent für jedes gefundene Modul an den SDM von Visual Studio geschickt.

4.7 Herausfinden der aktuellen Position

Nach dem Starten des Programms hat man das Problem, dass man bei einem Pausieren des Ablaufs nicht mehr weiß, wo man sich im Programm befindet. Um dieses problem zu lösen gibt es verschiedene Ansätze. Zum einen kann man diese Information direkt vom Debugger in Erfahrung bringen. Zum anderen kann man den aktuellen Befehlszeiger auch selbst berechnen. Hierzu muss man je nach Mikrocontroller der entsprechenden Spezifikation folgen.

Berechnet man zum Beispiel für den C166-Mikrocontroller den Befehlszeiger selbst, benötigt man die Informationen aus zwei Registern. Das sind einmal das Register des Befehlszeiger(IP) und zum anderen das Register des Befehlsegementzeigers **[Sie12]**. Insgesamt berechnet sich die aktuelle Adresse dann wie folgt:

$$\boxed{currentAddr = csp * 10000h + ip}$$

Jeder Mikrocontroller besitzt individuelle Register, die je nach Hersteller unterschiedlich verwendet werden. Dieses Vorgehen ist dadurch sehr hardwareabhängig. Die Berechnung mit dem Trace32 ist darum die bessere Variante, um die Kopplung zur Hardware möglichst gering zu halten. Der Trace32 bietet zur Berechnung der aktuellen Position die *readPC-Methode* an.

4.8 Setzen des Breakpoints

In diesem Kapitel wird beschrieben, welche Probleme auftreten sobald der Nutzer in Visual Studio einen Breakpoint setzt. Denn es muss auch automatisch im Trace32 ein Breakpoint gesetzt werden. Dazu bietet Visual Studio eine Methode an, die vom *IDebugEngine2-Interface* bereitgestellte *CreatePendingBreakpoint-Methode*. Diese Methode muss von der Debug Engine implementiert werden. Sie ist dafür zuständig aus einem *IDebug-BreakpointRequest2-Objekt* ein *IDebugPendingBreakpoint2-Objekt* für Visual Studio bereitzustellen. Der „IDebugBreakpointRequest“ stellt den Zugriff auf die Informationen zu dem anzulegenden Breakpoint her **[Mic1217]**. Die Informationen sind zum Beispiel die Position des Breakpoint oder bestimmte Bedingungen an den ein Breakpoint gebunden sein kann. Aus diesen Daten kann man dann einen „PendingBreakpoint“ erzeugen. Unter diesem Begriff „PendingBreakpoint“ wird in Visual Studio allgemein jeder Breakpoint verstanden, der vom Nutzer angefordert wurde, aber dessen Überprüfung noch aussteht **[Mic1217]**.

Die Implementierung eines „PendingBreakpoints“ muss von der Debug Engine geschehen. Die wichtigste Methode stellt dabei die *Bind-Methode* dar. Diese Funktion wird nach der *CreatePendingBreakpoint-Methode* für jedes erzeugte PendingBreakpoint-Objekt aufgerufen. In dieser wird aus den BreakpointRequest-Informationen ein *IDebugBoundBreakpoint-Objekt* erstellt. Unter einem „BoundBreakpoint“ versteht man einen Breakpoint, der an einen bestimmten Code-Kontext gebunden ist, das heißt an ein bestimmtes Modul und einer bestimmte Quellcodezeile. Der „PendingBreakpoint“ muss sich um das Speichern und Aufräumen seiner gebundenen Breakpoints selber kümmern **[Mic1218]**.

Ein „BoundBreakpoint“ muss ebenfalls von der Debug Engine erstellt werden. Diese Klasse speichert den Code-Kontext sowie seinen Zustand, wie aktiviert oder deaktiviert, an dem der Breakpoint gebunden wurde **[Mic1219]**.

Problematisch ist das Auslesen der Quellcodezeile sowie das Erkennen des Modulnamen aus dem *IDebugBreakpointRequest-Objekt*. Dieses Objekt stellt zum Auslesen eine *GetRequestInfo-Methode* bereit(siehe Listening 2).

```
int GetRequestInfo(
   eunm_BPREQI_FIELDS  dwFields,
   BP_REQUEST_INFO[]   pBPRequestInfo
);
```

Listening 2: C# Definition der GetRequestInfo-Methode [Mic1221]

Diese benötigt als erstes Argument eine Kombination an Flags. Anhand der Flags wird die im zweiten Parameter zurückgegebene Struktur mit nur den angeforderten Informationen gefüllt **[Mic1221]**. Die Liste der verfügbaren Flags kann man aus der Referenz entnehmen: **[Mic1220]**. Für die Position des Breakpoints benötigt man das „BPREQI_BPLOCATION“-Flag. An dem Code-Beispiel aus Listening 3 erkennt man, dass im C++-Code der zweite Parameter ein Zeiger ist.

```
HRESULT GetRequestInfo(
   BPREQI_FIELDS    dwFields,
   BP_REQUEST_INFO* pBPRequestInfo
);
```

Listening 3: C++ Definition der GetRequestInfo-Methode [Mic1231]

In C# gibt es aber keine Zeiger, daher wird ein Array verwendet, um das Objekt in verwalteten Code abbilden zu können. Dort ist der eigentliche Inhalt im ersten Array-Element enthalten. Das ist der Grund, warum man auch von der „BP_REQUEST_INFO“-Struktur nur das erste Element benötigt.

Von der „BP_REQUEST_INFO“-Struktur ist für die Position nur die „bpLocation“ von Bedeutung. In C# muss der Strukturinhalt erst umgewandelt werden. Dabei hilft die entsprechende Referenz in der Mikrosoft-Dokumentation: **[Mic1222]**. Von „bpLocation“ benötigt man das *IDebugDocumentPosition2-Objekt*. Mit diesem Objekt kann man sich dann den Dateiname über die *GetFileName-Methode* holen. Mit der *GetRange-Methode* des Objektes bekommt man den Anfang und das Ende der Zeile. In Listening 4 wird die implementierte Lösung der Arbeit dargestellt.

```
var ptrCodeContext = m_bpRequestInfo.bpLocation.unionmember2;
var docPosition = (IDebugDocumentPosition2)
                        (Marshal.GetObjectForIUnknown(ptrCodeContext));

// Get the document name.
string documentName;
docPosition.GetFileName(out documentName);

// Get the location in the document.
TEXT_POSITION[] startPosition = new TEXT_POSITION[1];
TEXT_POSITION[] endPosition = new TEXT_POSITION[1];
docPosition.GetRange(startPosition, endPosition);
```

Listening 4: Ausszug der Implementierten Lösung

Mit diesen Informationen von Visual Studio kann man auch im Trace32 einen Breakpoint erstellen. Dazu muss aus dem Dateiname und der Quellcodeadresse die korrespondierende Trace32-Adresse generiert werden. Dafür verwendete man die gespeicherten Informationen zwischen Quelladresse und Assembleradresse. Gibt es zu der Quellcodeadresse keine passende Assembleradresse, wird der Breakpoint nicht gebunden und es wird ein Fehlercode zurückgegeben.

In der praktischen Arbeit existiert ein Breakpoint Manager, der alle erstellten Breakpoints verwaltet. Dadurch weiß die Debug Engine, welche Breakpoints im Einsatz sind und kann diese zwischen Visual Studio und Trace32 synchron halten.

4.9 Erreichen eines Breakpoints

Die Breakpoints sind in der Debug Engine bekannt und im Trace32 erstellt. Nun stellt sich ein zusätzliches Problem beim Erreichen eines Breakpoints im Trace32. Beim Erreichen eines Breakpoints muss in Visual Studio angezeigt werden, dass das Programm gestoppt wurde und in welcher Zeile.

```
      14 int main()
P:021000 06F0F8FF  main:     add     r0,#0xFFF8
      15 {
                   //TODO: Configuration HERE
      18           int led = 0x01;
P:021004 E014                mov     r4,#0x1
P:021006 B840                mov     [r0],r4
      19           int led2 = 0x02;
P:021008 E024                mov     r4,#0x2
P:02100A C4400200            mov     [r0+#0x2],r4
```

Abbildung 18: Ausschnitt Trace32 beim Erreichen eines Breakpoints

Dazu bekommt man vom Trace32 ein detectedProgramBreak-Event, das signalisiert, dass das Programm pausiert. Das detectedProgramBreak-Event wird vom Trace32 geworfen sobald ein Breakpoint erreicht wird oder der Nutzer den Programablauf pausiert. In Abbildung 18 wird ein typisches Bild vom Trace32 dargestellt, wenn das Programm einen Breakpoint erreicht hat.

Aus dieser Abbildung wird außerdem ersichtlich, dass man jetzt im Assembler-Code ein Problem hat. Die Assemblerzeilen, hier die 21004 und die 21006, repräsentieren in dem Fall die C-Quellcodezeilen 15 bis 18 (siehe Abbildung 19).

```
14  int main()
15  {
16
17      //TODO: Configuration HERE
18      int led = 0x01;
19      int led2 = 0x02;
20
```

Abbildung 19: Ausschnitt Visual Studio beim Erreichen eines Breakpoints

Das heißt, man kann aus der aktuellen Assembler-Zeile nur die ungefähre Position im C-Quellcode angeben. Aus diesem Grund werden in dieser Arbeit alle C-Quellcodezeilen markiert, die mit der Assembler-Zeile in Verbindung stehen. In Abbildung 19 sieht man, wie der Zustand von meiner Lösung in Visual Studio dargestellt wird. Durch diese Vorgehensweise kann man recht gut Quellcodeteile erkennen, die durch Optimierung entfernt wurden sind. Zum Beispiel bei einer Schleife mit einer falschen Aussage. Setzt man hier innerhalb der Schleife ein Breakpoint und debuggt das Programm, würde die gesamte Schleife plus die erste Anweisung nach der Schleife markiert werden. Dies geschieht, da der Trace32 nur mit dem Original-Quellcode die Assembler-Zeilen zuordnet.

4.10 Überwachen des Trace32

Die Trace32-API bietet für das Empfangen von C#-Events ein Interface für einen Event-Listener an. Implementierungen von diesem können nach der Registrierung bei der Trace32-API folgende Events abhören: Breakpoint wurde erstellt, Breakpoint wurde gelöscht, Breakpoint wurde verändert, Quellcode wurde verändert sowie das Programm wurde pausiert.

Allerdings werden diese Events nur dann an den Listener weitergeleitet, wenn die API einen Status update anfordert. Aus diesem Grund wird in der Debug Engine eine T32UpdateStatusThread-Klasse erstellt. Dessen Aufgabe besteht darin im Sekundentakt die *Update-Methode* der Trace32-API aufzurufen.

Da damit die Versorgung des Frameworks mit Events gesichert ist, müssen diese nur noch an die Debug Engine weitergereicht werden. Dazu muss die Debug Engine einen eigenen Listener bei der Trace32-API anmelden. Damit dies entkoppelt geschieht oder geschehen kann, verfügt das in der Arbeit erstellte Framework über ein eigenes Listener-Interface. Dieses verwendet dann je nach Debugger den richtigen Listener. Über das Interface werden die Events der Debug Engine gesendet. (Abbildung 20).

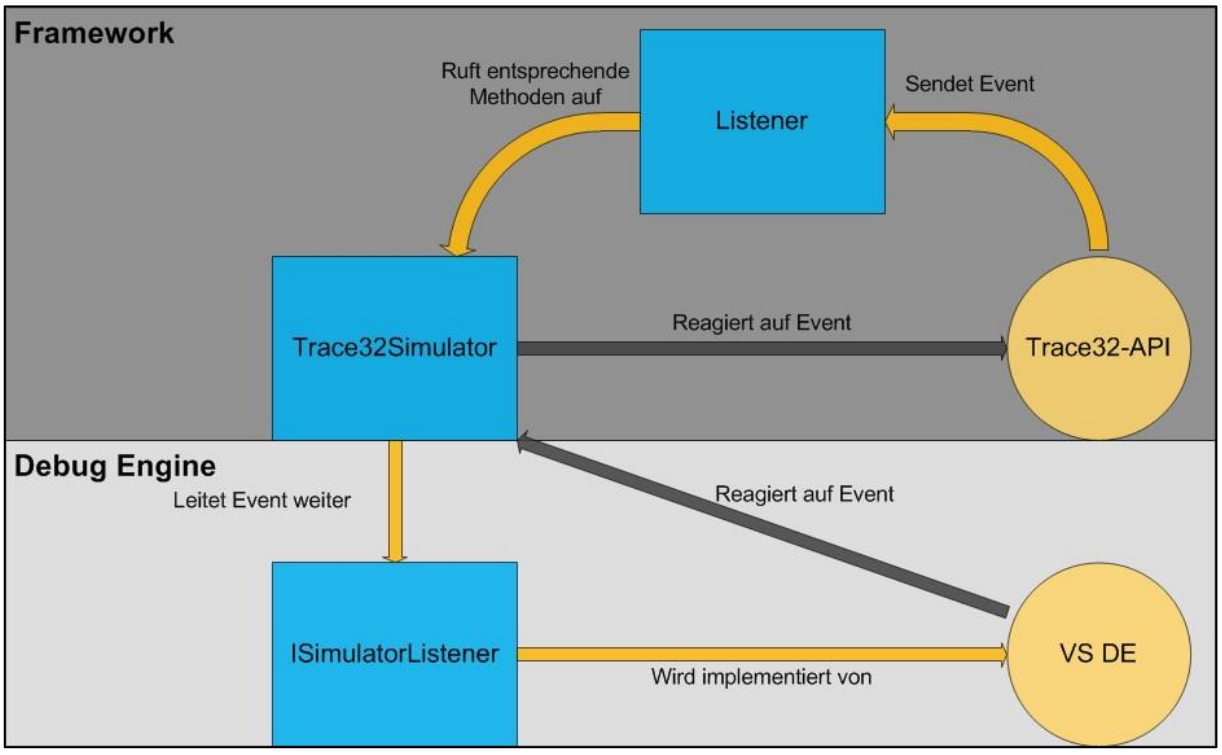

Abbildung 20: Kommunikation zwischen Trace32 API und Visual Studio Debug Engine

In dieser Arbeit ist für das Lauschen auf Events vom Trace32 die T32EngineCallback-Klasse zuständig. Diese Klasse erfüllt auch das Speichern des Callback-Handlers zum SDM und dient dem Senden von Events.

5 Ergebnisse

In dieser Arbeit wird Visual Studio erweitert, um auch im Embedded-Umfeld gebräuchliche Mikrocontroller debuggen zu können. Mit der in Kapitel 3.3 (siehe Seite 18) dargestellten Ausgangslage ist das um Trace32 erweiterte Visual Studio als „All-In-One“-Tool für den Embedded-Bereich verwendbar. Der Prototyp deckt das Erstellen eines Programms für den C166, das Debuggen des Mikrocontrollers sowie das automatische Testen ab.

Der praktische Teil der Arbeit ist auf Kompatibilität zu Visual Studio 2010 SP1, Visual Studio 2010 SP1 mit „TFS 2012 Compatibility GDR“-Patch sowie Visual Studio 2012 getestet.

Untersucht wird dabei, dass die Aktionen: Erstellen des C166-Programms, Debuggen des C166 sowie das Testen des C166 funktionieren. Das Starten des debuggens über den Visual Studio „Debuggen starten“-Button sowie das Starten über ein eigenes Add-In werden ebenfalls überprüft. Ein wichtiger Analysepunkt ist auch die Möglichkeit, ob man die Debug Engine selbst mit der Visual Studio Version debuggen kann. In Tabelle 2 ist das Ergebnis dieser Analyse dargestellt.

	Visual Studio 2010 SP1	Visual Studio SP1 TFS 2012 Compatibility GDR	Visual Studio 2012
Funktionalität	✓	✓	✓
Erstellen	✓	✓	✓
Debuggen	✓	✓	✓
Testen	✓	✓	✓
Start per Visual Studio Button möglich?	✓	Nein	✓
Start per Add-In möglich?	✓	✓	✓
Debug Engine kann debuggt werden?	✓	✓	Nein

Tabelle 2: Kompatibilitätsvergleich

Im Visual Studio SP1 mit „TFS 2012 Compatibility GDR“-Patch kann man den Debugger nicht mehr über den Visual Studio Button starten. Um den C166 debuggen zu können, muss man über ein Add-in den Debugg-Vorgang starten. In Visual Studio 2012 stellt man bei der normalen Verwendung keine Unterschiede zum Visual Studio 2010 fest. Allerdings kann man mit Visual Studio 2012 die Debug Engine nicht debuggen, da diese dann an sich veränderteren Positionen MemoryAccessViolationExceptions wirft. Das erschwert ein Weiterentwickeln in Visual Studio 2012. Das Analysieren sowie das Beheben der Fehler ist im Zeit-Rahmen der Arbeit nicht möglich gewesen.

6 Zusammenfassung und Ausblick

In dieser Arbeit wird das Wissen aufgebaut, um einen eigenen oder externen Debugger in Visual Studio einzubinden. Dazu werden die angebotenen Schnittstellen von Visual Studio aufgezeigt. Es wird eine eigene Debug Engine entwickelt. Diese wird in Visual Studio registriert und kann über eine Debugger Launch Extension gestartet werden.

Das Projektsystem von Visual Studio wird um eine angepasste Eigenschaftsseite erweitert, um den Debugger steuern zu können. Dieses Wissen lässt sich nutzen, um beliebige Konfigurationseigenschaften im MSBuild-System vorzunehmen.

Weiterhin wird am Trace32 die Integration eines externen Debuggers gezeigt. Probleme, die während der Implementierung aufgetreten sind, werden in den jeweiligen Kapiteln beleuchtet.

Ein möglicher Untersuchungspunkt ist, dass der Trace32 momentan immer ein eigenes Fenster öffnet. Später sollte man einstellen können, ob dieses Fenster sichtbar oder unsichtbar erstellt werden soll.

Für den produktiven Einsatz reicht die in dieser Arbeit erarbeite Debugg-Lösung nicht aus. Bevor man diese Software einsetzen kann, müssen weitere Debugg-Features implementiert werden. Es fehlt zum Beispiel die Anzeige von Register- oder Variableninhalten. Auch bei den Breakpoints können noch viele Features integriert werden. Hier sollen nur mal die einfachsten Funktionen, wie das Entfernen und das Ein- sowie Ausschalten von Breakpoints genannt werden.

7 Begriffsverzeichnis

Application Programming Interface oder kurz API ist eine Programmierschnittstelle, welche Methoden, Makros und Objekte zur Verfügung stellt, um auf interne Prozesse einer Anwendung zugreifen zu können [ITW12].

C166 ist ein 16Bit-Microcontroller auf Basis der Von-Neumann-Architektur, welcher von Siemens/Infineon in den 80iger Jahren entwickelt wurde [web12].

Component Object Model ist eine von Microsoft entwickelte Technologie, um Softwarekomponenten zu erzeugen, die unabhängig von der Programmiersprachen verwendet werden können [Wik121] [Mic1224] [Loo00].

Event-Listener ist ein objektorientierter Programmieransatz, um ereignisgesteuerte Anwendungen zu erstellen. Der Event-Listener ist dabei ein Objekt, welches ein vorgegebenes Interface definieren muss [MSR02].

Flag ist eine binäre Variable oder Struktur um einen bestimmten Zustand zu kennzeichnen [Wik122].

Globally unique identifier ist eine eindeutige Referenz Nummer, welche 128 Bit groß ist. Sie wird verwendet um Software-Objekte, wie Interfaces, eindeutig zu identifizieren [Mic1225].

Hexadezimaldatei ist eine Datei die im Format des hexadezimalen Zahlensystems aufgebaut ist.

Log-Datei ist eine bestimmte Sorte von Datei, es handelt sich dabei meist um einen Speicherpunkt, um Informationen eines Programms mit zu schreiben, quasi zu loggen.

OH166 Object-HEX Converter ist ein Tool von ARM, welches Intel-HEX Dateien aus dem vom Linker generierten OMF166-Dateien erzeugt [ARM12].

Onboard oder auch Chip-On-Board ist eine Technologie, bei der der Chip ohne Gehäuse auf der Hardware integriert wird [DAT12].

Speicher-Dump ist eine Darstellung des Inhaltes aus dem Hauptspeicher [atm12].

Stack ist ein Speicherbereich der Daten nach dem LIFO-Prinzip speichert [Mue10]**.**

Stack-Frame ist ein Ausschnitt eines Stacks der Informationen zu einer Funktion speichert, wie Parameter, gespeicherte Registerwerte sowie die Rücksprungadresse. [Mic1223]

Windows Registry ist ein hierarchisch aufgebautes Datenbanksystem, das Informationen zum System und Anwendungen speichert [Mic1227].

Windows Registry Key ist ein Schlüssel der Windows Registry. Ein Schlüssel besitzt einen Namen, der ihn in seiner Hierarchie-Ebene einordnet. Jeder Schlüssel kann Daten (engl. „values“) oder Unterschlüssel (engl. „subkeys“) besitzen [Mic1226].

8 Quellenverzeichnis

[Bar99] Michael Barr: Programming Embedded Systems in C and C++. O'Reilly Media, Inc., 1999.

[Loo00] Peter Loos: Go to COM. Addison-Wesley, 1. Aufl., Bonn, 2000.

[MSR02] Stefan Middendorf, Reiner Singer, Jörn Heid: JavaTM Programmierhandbuch und Referenz für die JavaTM-2-Plattform. dpunkt.Verlag, Heidelberg, 2002.

[Mue09] Thomas Müller: Add-In-Entwicklung für Visual Studio. Software & Support Verlag GmbH, Frankfurt am Main, 2009.

[Zel06] Andreas Zeller: Why Programs Fail: A Guide to Systematic Debugging. Elsevier Inc. and dpunkt.verlag, Heidelberg, 2006.

[Amb10] Michael Amberg, Michael Reinhardt: *Application Lifecycle Management – Trends, Herausforderungen und IT-Unterstützung.* Online im Internet: http://www.wi3.uni-erlangen.de/fileadmin/Bilder/Forschung/ALM2010.pdf, 2010. Stand: 24.9.2012.

[ARM12] ARM Ltd and ARM Germany GmbH: *OH166 Object-HEX Converter.* Online im Internet: http://www.keil.com/c166/oh166.asp, 2012. Stand: 9.11.2012.

[atm12] at-mix.de: *Dump - Fachbegriffe Online Lexikon.* Online im Internet: http://www.at-mix.de/dump.htm, 2012. Stand: 9.11.2012.

[Bau12] Uwe Baumann: *Visual Studio News-Blog.* Online im Internet: http://blogs.msdn.com/b/vsnewsde/archive/2012/08/01/visual-studio-2012-ist-fertig-verf-252-gbar-am-15-august.aspx, 2012. Stand: 19.9.2012.

[Cha12] David Chappell: *What is Application Lifecycle Management?.* Online im Internet: http://davidchappell.com/writing/white_papers/What_is_ALM_v2.0--Chappell.pdf, 2012. Stand: 15.9.2012.

[DAT12] DATACOM Buchverlag GmbH: *CoB (chip on board).* Online im Internet: http://www.itwissen.info/definition/lexikon/chip-on-board-COB.html, 2012. Stand: 9.11.2012.

[Ele06] Elektronik Praxis - Redakteur: (hh): *Mikrocontroller Allgegenwärtig - MCUs in der Embedded-Welt - Größe, Struktur und Trends.* Online im Internet: http://www.elektronikpraxis.vogel.de/themen/hardwareentwicklung/mikrocontrollerprozessoren/articles/33590/, 2006. Stand: 25.9.2012.

[Fir12] Andy Firth: *Extending the Watching Window in Visual Studio via a Debugger Addin.* Online im Internet: http://www.altdevblogaday.com/2011/06/22/extending-the-watching-window-in-visual-studio-via-a-debugger-addin/, 2012. Stand: 26.10.2012.

[Gee12] Geekadelphia: *Arduino Bild.* Online im Internet: http://geekadelphia.com/wp-content/uploads/2008/01/ardino_microcontroller.jpg, 2012. Stand: 22.11.2012.

[ITW12] DATACOM Buchverlag GmbH: *API (application programming interface).* Online im Internet: http://www.itwissen.info/definition/lexikon/application-programming-interface-API-Programmierschnittstelle.html, 2012. Stand: 8.11.2012.

[Kos04] Lasse Koskela: *Introduction to Code Coverage.* Online im Internet: http://www.javaranch.com/journal/2004/01/IntroToCodeCoverage.html, 2004. Stand: 2012.9.25.

[Lau12] Lauterbach GmbH: *Company Brochure.* Online im Internet: http://www.lauterbach.com/company_brochure.pdf, 2012. Stand: 13.9.2012.

[Mic08] Microsoft Corporation: *Debug Engine Sample Version 2.0.* Online im Internet: http://archive.msdn.microsoft.com/debugenginesample/Release/ProjectReleases.aspx?ReleaseId=4149, 2008. Stand: 28.9.2012.

[Mic12] Microsoft Corporation: *Visual Studio Ultimate 2012-System Requirements.* Online im Internet:

http://www.microsoft.com/visualstudio/eng/products/visual-studio-ultimate-2012#product-edition-ultimate-Details, 2012. Stand: 24.9.2012.

[Mic121] Microsoft: *Erstellen von automatisierten Tests.* Online im Internet: http://msdn.microsoft.com/de-de/library/vstudio/dd380755.aspx, 2012. Stand: 25.9.2012.

[Mic1210] Microsoft: *IDebugEngine2.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb145310(v=vs.80).aspx, 2012. Stand: 29.9.2012.

[Mic1211] Microsoft: *Sending Events.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb144993(v=vs.80).aspx, 2012. Stand: 13.10.2012.

[Mic1212] Microsoft: *Supported Event Types.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb161274(v=vs.80).aspx, 2012. Stand: 13.10.2012.

[Mic1213] Microsoft: *EVENTATTRIBUTES.* Online im Internet: http://msdn.microsoft.com/de-de/library/bb144974(v=vs.90).aspx, 2012. Stand: 15.10.2012.

[Mic1214] Microsoft: *IDebugEvent2.* Online im Internet: http://msdn.microsoft.com/de-de/library/bb161977(v=vs.90).aspx, 2012. Stand: 15.10.2012.

[Mic1215] Microsoft: *StackFrame Interface.* Online im Internet: http://msdn.microsoft.com/en-gb/library/d091z44x.aspx, 2012. Stand: 26.10.2012.

[Mic1216] Microsoft: *IDebugEngineLaunch2::LaunchSuspended.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb146223.aspx, 2012. Stand: 2.11.2012.

[Mic1217] Microsoft: *IDebugEngine2::CreatePendingBreakpoint.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb147033.aspx, 2012. Stand: 4.11.2012.

[Mic1218] Microsoft: *IDebugPendingBreakpoint2.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb161807.aspx, 2012. Stand: 4.11.2012.

[Mic1219] Microsoft: *IDebugBoundBreakpoint2.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb161979.aspx, 2012. Stand: 4.11.2012.

[Mic122] Microsoft: *Test Types.* Online im Internet: http://msdn.microsoft.com/en-us/library/ms182514(v=vs.90).aspx, 2012. Stand: 25.9.2012.

[Mic1220] Microsoft: *BPREQI_FIELDS.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb146318.aspx, 2012. Stand: 4.11.2012.

[Mic1221] Microsoft: *IDebugBreakpointRequest2::GetRequestInfo.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb146225.aspx, 2012. Stand: 4.11.2012.

[Mic1222] Microsoft: *BP_LOCATION.* Online im Internet: http://msdn.microsoft.com/de-de/library/bb162191(v=vs.90).aspx, 2012. Stand: 4.11.2012.

[Mic1223] Microsoft: *Stack Frames.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb161394.aspx, 2012. Stand: 8.11.2012.

[Mic1224] Microsoft: *Component Object Model.* Online im Internet: http://msdn.microsoft.com/de-de/library/aa286559.aspx, 2012. Stand: 8.11.2012.

[Mic1225] Microsoft: *GUID structure.* Online im Internet: http://msdn.microsoft.com/en-us/library/aa373931(VS.85).aspx, 2012. Stand: 8.11.2012.

[Mic1226] Microsoft: *Structure of the Registry.* Online im Internet: http://msdn.microsoft.com/en-us/library/ms724946.aspx, 2012. Stand: 8.11.2012.

[Mic1227] Microsoft: *Registry.* Online im Internet: http://msdn.microsoft.com/en-us/library/ms724871.aspx, 2012. Stand: 8.11.2012.

[Mic1228] Microsoft: *Diagramm "Automationsobjektmodell.* Online im Internet: http://msdn.microsoft.com/de-de/library/za2b25t3.aspx, 2012. Stand: 9.11.2012.

[Mic1229] Microsoft: *Debugging Components.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb146991(v=vs.80).aspx, 2012. Stand: 28.9.1012.

[Mic123] Microsoft: *Team Foundation Server Reporting.* Online im Internet: http://msdn.microsoft.com/en-us/library/ms194922(v=vs.80).aspx, 2012. Stand: 26.9.2012.

[Mic1230] Microsoft: *IDebugEventCallback2::Event .* Online im Internet: http://msdn.microsoft.com/en-us/library/bb162146(v=vs.80).aspx, 2012. Stand: 22.11.2012.

[Mic1231] Microsoft: *IDebugBreakpointRequest2::GetRequestInfo.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb146225.aspx?cs-save-lang=1&cs-lang=cpp#code-snippet-1, 2012. Stand: 9.12.2012.

[Mic124] Microsoft: *Scrum Process Template for Visual Studio ALM.* Online im Internet: http://msdn.microsoft.com/en-us/library/ff731587.aspx, 2012. Stand: 26.9.2012.

[Mic125] Microsoft: *MSBuild (Visual C++).* Online im Internet: http://msdn.microsoft.com/de-de/library/ms171452(v=vs.90).aspx, 2012. Stand: 27.9.2012.

[Mic126] Microsoft: *Buildsystemänderungen.* Online im Internet: http://msdn.microsoft.com/de-de/library/vstudio/ee862524(v=vs.100).aspx, 2012. Stand: 27.9.2012.

[Mic127] Microsoft: *Gewusst wie: Integrieren von benutzerdefinierte Tools in die Projekteigenschaften.* Online im Internet: http://msdn.microsoft.com/de-de/library/ff770593, 2012. Stand: 27.9.2012.

[Mic128] Microsoft: *Registering a Custom Debug Engine.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb147122(v=vs.80).aspx, 2012. Stand: 28.9.2012.

[Mic129] Microsoft: *IDebugEngineLaunch2.* Online im Internet: http://msdn.microsoft.com/en-us/library/bb146230(v=vs.80).aspx, 2012. Stand: 29.9.2012.

[Mue10] Alexander Müller: *Der Stack Frame.* Online im Internet: http://www.a-m-i.de/tips/stack/stack.php, 2010. Stand: 8.11.2012.

[Sch12] Hansjörg Scherer: *ALM schafft mit Automatisierung und Standardisierung Wettbewerbsvorteile.* Online im Internet: http://www.netzwoche.ch/de-CH/News/2012/06/20/ALM-schafft-mit-Automatisierung-und-Standardisierung-Wettbewerbsvorteile.aspx?pa=5, 2012. Stand: 24.9.2012.

[Sie12] Siemens: *SAB 80C166/83C166.* Online im Internet: http://www.keil.com/dd/docs/datashts/infineon/sab166_ds.pdf. Stand: 3.11.2012.

[The12] The Ganssle Group: *Beginner's Corner - In-Circuit-Emulators.* Online im Internet: http://www.ganssle.com/articles/BegincornerICE.htm, 2012. Stand: 26.9.2012.

[Tri10] Tritec Benelux BV: *Trace32 Bild.* Online im Internet: http://www.tritec.nl/oldsite/images/bdmppcaltivec__000.jpg, 2010. Stand: 22.11.2012.

[web12] webmaster@mikrocontroller.net: *C166.* Online im Internet: http://www.mikrocontroller.net/articles/C166, 2012. Stand: 8.11.2012.

[Who12] Whole Tomato Software, Inc: *Refactoring Features of Visual Assist X.* Online im

Internet: http://www.wholetomato.com/products/featureRefactoring.asp, 2012. Stand: 26.9.2012.

[Wik121] Wikipedia: *Component Object Model.* Online im Internet: http://de.wikipedia.org/wiki/Component_Object_Model, 2012. Stand: 8.11.2012.

[Wik122] Wikipedia: *Flag (Informatik).* Online im Internet: http://de.wikipedia.org/wiki/Flag_(Informatik), 2012. Stand: 8.11.2012.

[Wil02] Arnold Willemer: *Debugger.* Online im Internet: http://www.willemer.de/informatik/unix/debugger.htm, 2002. Stand: 24.9.2012.

[Zum1230] ZoomZum: *Visual Studio Logo.* Online im Internet: http://zoomzum.com/wp-content/uploads/2012/05/Visual_Studio.jpg, 2011. Stand: 29.9.2012.

Abbildungsverzeichnis

Tabellenverzeichnis

Listeningverzeichnis

Anhang

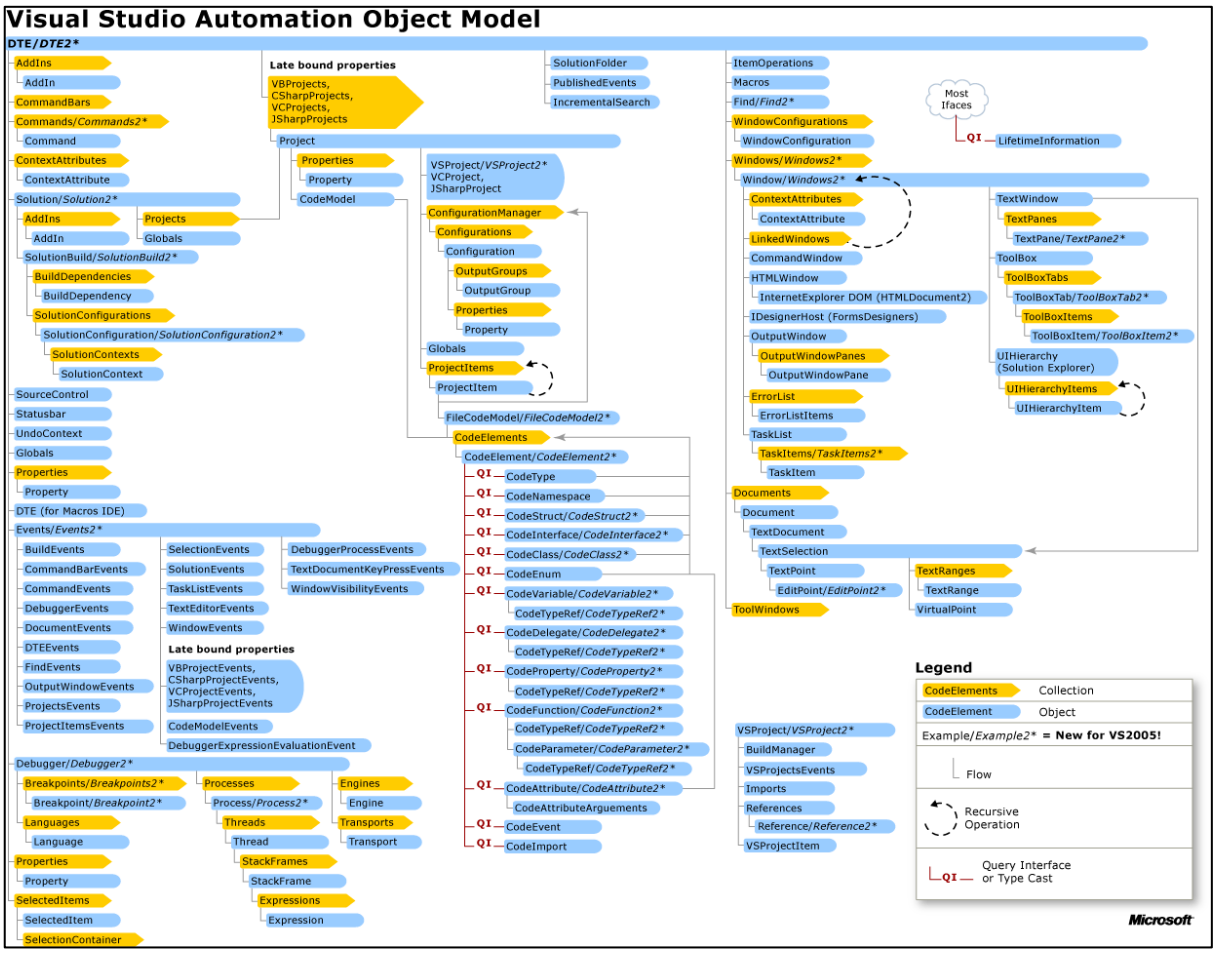

Abbildung 21: Visual Studio Automation Object Model ([Mic1228]**)**

Windows Registry Keys

Debbuger:

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\VisualStudio\10.0\AD7Metrics\Engine\{BDEC0D7D-0C72-49B6-81DF-AD7C3444DF16}

Name	Typ	Daten
(Standard)	REG_SZ	guidT32Debugger
AddressBP	REG_DW...	0x00000000 (0)
Attach	REG_DW...	0x00000001 (1)
AutoSelectPriority	REG_DW...	0x00000004 (4)
CallstackBP	REG_DW...	0x00000001 (1)
CLSID	REG_SZ	{1380B2EC-BA5F-434B-8BA2-3318C96C0F9A}
Name	REG_SZ	Trace32 Debugger
PortSupplier	REG_SZ	{708C1ECA-FF48-11D2-904F-00C04FA302A1}
ProgramProvider	REG_SZ	{91F748A5-CAFC-4661-9F82-137E639EB5EB}

Engine:

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\VisualStudio\10.0\CLSID\{1380B2EC-BA5F-434B-8BA2-3318C96C0F9A}

Name	Typ	Daten
(Standard)	REG_SZ	(Wert nicht festgelegt)
Assembly	REG_SZ	Microsoft.VisualStudio.Debugger.T32ConnectionEngine
Class	REG_SZ	Microsoft.VisualStudio.Debugger.T32ConnectionEngine.T32Engine
CodeBase	REG_SZ	C:\Dev\Conplement_intern_TFS2012\Praktika\Alexander_Alm4Es\CPP_UnitTest Framework\Trace32AddIn\T32Connec
InprocServer32	REG_SZ	c:\windows\system32\mscoree.dll

ProgramProvider:

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\VisualStudio\10.0\CLSID\{91F748A5-CAFC-4661-9F82-137E639EB5EB}

Name	Typ	Daten
(Standard)	REG_SZ	(Wert nicht festgelegt)
Assembly	REG_SZ	Microsoft.VisualStudio.Debugger.T32ConnectionEngine
Class	REG_SZ	Microsoft.VisualStudio.Debugger.T32ConnectionEngine.T32ProgramProvider
CodeBase	REG_SZ	C:\Dev\Conplement_intern_TFS2012\Praktika\Alexander_Alm4Es\CPP_UnitTest Framework\Trace32AddIn\T32Connec
InprocServer32	REG_SZ	c:\windows\system32\mscoree.dll

Aufbau einer Debugger Launcher Extension

Eine Debugger Launcher Extension kann man sich mit der Projekt Vorlage von Microsoft erstellen lassen. Man erhält dadurch ein Projekt, welches bereits eine Debugger Launcher Extension enthält. In dieser befindet sich die „VendorNameCoolDebugger“-Klasse. Im Listening 5 ist ein Ausschnitt von dieser Klasse. Das erste wichtige Element ist der DebuggerScope, dieses Klassenargument erhält als Parameter den Namen des Debugger Launchers. Es muss der gleiche Name sein, wie vorher in der XML Datei der Eigenschaftsseite des Projektes festgelegt. Die Klasse implementiert das *IDebugLaunchProvider-Interface*, welche die Implementation von zwei Methoden verlangt. Das ist zum einen die *CanLaunch-Methode*, welche überprüft, ob die Projekteigenschaften mit diesem Debugger kompatibel sind. Die zweite Methode ist die *PrepareLaunch-Methode*. In dieser werden die Einstellungen für das Starten der Debug Engine vorgenommen. Wichtig ist, dass die *IDebugLaunchSettings.setting.LaunchDebugEngineGuid* mit dem GUID der Debug Engine übereinstimmt.

```
[Export(typeof(IDebugLaunchProvider))]
[ProjectScope(ProjectScopeRequired.ConfiguredProject)]
[DebuggerScope("Trace32 Debugger")]
public class VendorNameCoolDebugger : IDebugLaunchProvider
{
    //...

    public bool CanLaunch(DebugLaunchOptions launchOptions,
                          IDictionary<string, string> projectProperties)
    {
        //...
    }

    public IEnumerable<IDebugLaunchSettings> PrepareLaunch(
                                DebugLaunchOptions options,
                                IDictionary<string, string> projectProperties)
    {
        //...
    }
}
```

Listening 5: IDebugLaunchProvider Interface

Konzepte der Debug Engine

Die Engine muss die *IDebugEngine2* und die *IDebugEngineLaunch2-Interfaces* bedienen. Das IDebugEngineLaunch Interface definiert Methoden, um Prozesse und Programme zu starten und zu beenden, welche von der Debug Engine benötigt werden. Dieses Interface muss nur von der Debug Engine implementiert werden, wenn spezielle Start-Bedingungen vorhanden sind **[Mic129]**. Das ist in dieser Arbeit der Fall, da der externe Debugger gestartet und initialisiert werden muss. Das IDebugEngine2 Interface repräsentiert die Debug Engine und muss von der Engine implementiert werden **[Mic1210]**. Bevor die Schnittstelle mit dem Debuggen beginnt, müssen zuerst einige Initialisierungsschritte durch die Engine erfolgen. Zuerst wird die *IDebugEngineLaunch2::LaunchSuspended-Methode* der Debug Engine aufgerufen. Diese erhält als Parameter alle wichtigen Informationen, die in der erstellten Debugger Launcher Extension übergeben können. Diese Methode bekommt zusätzlich ein Objekt vom Typ *IDebugEventCallback2* als Parameter. Mit diesem Objekt kann man Events an Visual Studio senden. Als Rückgabeparameter ist noch ein Objekt vom Typ *IDebugProcess2* zu instanziieren. Visual Studio lässt sich nach der *LaunchSuspended-Methode* von dem *IDebugProcess2 Objekt*, die Systemprozess-ID geben und ruft anschließend die *IDebugEngine-Launch2::ResumeProcess-Methode* auf. Diese Methode bekommt das *IDebugProcess2-Objekt* als Parameter übergeben. Ihre Aufgabe liegt darin, den Debugger dem zu debuggenden Prozess anzuhängen. Dafür wird die *IDebugEngine2::Attach-Methode* aufgerufen, welche den Debugger für den zu debuggenden Prozess konfiguriert. Anschließend lädt sie entsprechende Symbol-Informationen und arbeitet eine Reihe von Events ab. Diese müssen Visual Studio gesendet werden, um die Initialisierung abzuschließen. Wer sich mit dem Thema ebenfalls beschäftigen möchte dem wird empfohlen sich die Beispiel Debugg-Engine herunterzuladen und zu analysieren: **[Mic08]**.

