bbv
MAKING VISIONS WORK.

BOOKLET

CONTINUOUS INTEGRATION FÜR EMBEDDED SYSTEME

Copyright © 2015
bbv Software Services AG

PROFITIEREN SIE VON UNSERER ERFAHRUNG!

Kontakt Schweiz
bbv Software Services AG
Blumenrain 10
6002 Luzern
Telefon: +41 41 429 01 11
E-Mail: info@bbv.ch

Kontakt Deutschland
bbv Software Services GmbH
Agnes-Pockels-Bogen 1
80992 München
Telefon: +49 89 452 438 30
E-Mail: info@bbv.eu

Der Inhalt dieses Booklets wurde mit Sorgfalt und nach bestem Gewissen erstellt. Eine Gewähr für die Aktualität, Vollständigkeit und Richtigkeit des Inhalts kann jedoch nicht übernommen werden. Eine Haftung (einschliesslich Fahrlässigkeit) für Schäden oder Folgeschäden, die sich aus der Anwendung des Inhalts dieses Booklets ergeben, wird nicht übernommen.

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

INHALT

1	Einleitung	4
2	Continuous Integration	6
2.1	Einführung	7
2.2	Workflow	7
	Check-in	8
	Build	9
	Test	10
	Feedback	12
2.3	Infrastruktur	13
3	Continuous Integration für Embedded Systeme	16
3.1	Entwicklungsumgebung	17
3.2	Cross-Compiling, mehrere Plattformen	18
3.3	Lizenzen	19
3.4	Anschlussmöglichkeiten	20
3.5	Hardware-Verfügbarkeit	20
3.6	Testbarkeit	21
4	Praxisbeispiele	22
4.1	Native Entwicklung auf Embedded Linux	23
4.2	Windows CE	24
4.3	FreeRTOS auf ARM7	25
5	Fazit	28
6	Anhang	30
6.1	Autor	31
6.2	Literatur	31

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

1 EINLEITUNG

Embedded Software ist nicht gleich wie andere Software. Diesen Satz wird wohl jeder Entwickler von Embedded Software unterschreiben. Diesem Umstand ist wohl auch zu verdanken, dass sich im Embedded Umfeld neue Technologien, Tools oder Methoden meistens eher langsam durchsetzen.

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

Dieses Booklet zeigt auf, wie man Continuous Integration für Embedded Softwareprojekte erfolgreich einsetzen kann. Nach einer kurzen Einführung[1] in das Grundkonzept von Continuous Integration werden die spezifischen Aspekte von Embedded Projekten beleuchtet. Abgerundet wird das Booklet durch verschiedene Beispiele, die erklären, wie Continuous Integration in Projekten mit unterschiedlichsten Hardware-/Software-Kombinationen eingesetzt werden kann.

[1] Für eine detailliertere Einführung zum Thema Continuous Integration empfiehlt sich die Lektüre des bbv-Booklets «Introduction to Continuous Integration» von Philipp Dolder.

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

2 CONTINUOUS INTEGRATION

Wahrscheinlich hat jeder, der schon einmal in einem grösseren Softwareprojekt mitgearbeitet hat, die «Integrations-Hölle» erlebt: Mehrere Entwickler (oder sogar mehrere Teams) haben über eine längere Zeit neue Features entwickelt oder Bugs gefixt und am Schluss muss alles zusammen integriert und ein lauffähiges Release erstellt werden. Plötzlich stellt sich heraus, dass es zu Inkompatibilitäten gekommen ist, Interfaces passen nicht mehr zusammen und Code, der auf einem Rechner lief, läuft auf einem anderen plötzlich nicht mehr. Dies alles passiert meistens erst unmittelbar vor einem Releasetermin, dementsprechend angespannt ist die Situation und gross der Druck.

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

2.1 EINFÜHRUNG

Genau hier kommt die Idee von Continuous Integration (CI) ins Spiel: Anstelle einer grossen «Big-Bang-Integration» am Schluss einer Entwicklungsphase wird jedes Feature oder jeder Bugfix kontinuierlich integriert. Kleinere Integrationsschritte verringern das Risiko und erhöhen die Transparenz des Projekts. Zusätzlich werden durch eine komplette Automatisierung der Integration die oftmals langwierigen und fehleranfälligen Integrations- und Releaseprozesse vereinfacht.

Obwohl der Begriff Continuous Integration schon vor langer Zeit verwendet wurde (Booch et al., 1991), hat er sich in der heutigen Form erst im Zusammenhang mit Extreme Programming (XP) und generell agilen Methoden etablieren können.

2.2 WORKFLOW

Der Workflow von Continuous Integration lässt sich in vier Schritte unterteilen: Check-in, Build, Test und Feedback.

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

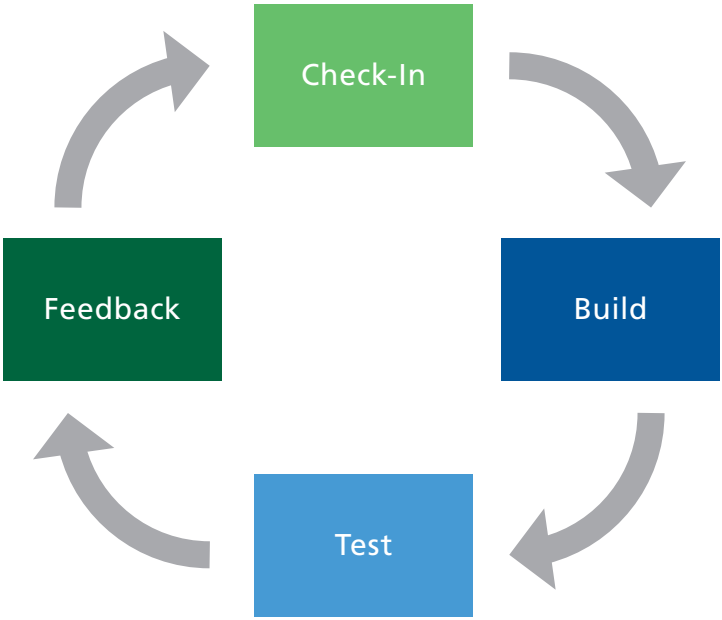


Abbildung 1: Grundlegender Workflow von Continuous Integration

Check-in

Der Einsatz von Software-Versionsverwaltungssystemen sollte in der heutigen Zeit auch für kleine oder Kleinstprojekte selbstverständlich sein. Die Handhabung dieser Systeme variiert je nach Projekt oder Firma sehr stark. Deshalb empfiehlt es sich, dass jeweils projekt- oder firmenspezifische Richtlinien erarbeitet werden, wie genau solche Systeme eingesetzt werden sollen. Diese Richtlinien definieren das gemeinsame Verständnis darüber, wie häufig beispielsweise Code eingecheckt werden soll, in wie grosse «Arbeitspakete» die Änderungen unterteilt werden, oder über die Formatierung und den Stil der Check-in-Kommentare. Letzteres mag auf den ersten

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

Blick etwas pingelig erscheinen, kann sich aber im Hinblick auf automatisch generierte Changelogs schnell bezahlt machen.

Generell hat sich der Grundsatz «commit early, commit often» bewährt. Der Check-in sollte möglichst in eine logische Einheit gebündelt werden, beispielsweise alles, was für die Implementation eines bestimmten Features notwendig ist, oder alles, was für die Behebung eines Fehlers nötig ist.

In einigen Systemen ist auch ein sogenannter «Pre-Flight Check» integriert. Mit diesem wird sichergestellt, dass nur kompilierbarer Code eingecheckt werden kann, bei Kompilierfehlern würde der Check-in verweigert werden.

Build

Nach dem Check-in des Sourcecodes wird der Build gestartet. Entscheidend dabei ist, dass dies automatisch geschieht und keine Aktion vom Entwickler benötigt. Zu diesem Zweck bieten die meisten Versionsverwaltungen sogenannte «Hooks» an. Damit können beliebige Programme oder Scripts nach einem Check-in ausgeführt werden. So ist es ein Leichtes, den Build-Server zu benachrichtigen, beispielsweise über die Weboberfläche[2] oder eine sonstige Schnittstelle. Alternativ kann auch der Build-Server entsprechend konfiguriert werden, dass er periodisch den Stand des Sourcecodes in der Versionsverwaltung prüft und dadurch Änderungen erkennt.

Neben dem reinen Kompilieren und Linken des Sourcecodes werden auch sämtliche nötigen Pre- und Post-Build-Prozesse durchgeführt,

[2] Dafür würde sich ein Command-Line-Tool wie wget oder curl anbieten.

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

um ein ausführbares Deployment zu erhalten. Darunter fallen z. B. Code-Generierung, Erstellung von verschiedenen Compile-Varianten, Generierung von Dokumentation, Konvertierung von Binaries sowie Verpacken und Signieren von Artefakten. Hier macht sich dann der erste Vorteil von CI bemerkbar. Denn hat man den Buildprozess erst einmal automatisiert – und ist er auch noch so komplex –, ist spätestens jetzt die Durchführung ein Kinderspiel.

Test

Eine CI-Infrastruktur, die den Code bei jedem Check-in kompiliert und so auf elementarste Fehler aufmerksam macht, ist erst die halbe Miete, um die ganzen Vorteile von CI nutzen zu können. Sobald man einmal die ersten Hürden zur Einführung von CI überwunden hat, sollte man sich überlegen, wie man einen Schritt weitergehen und die Qualität weiter erhöhen kann.

Als erster Schritt bietet sich an, eine statische Codeanalyse auf dem Build-Server durchzuführen. Dazu zählen einerseits simple Tests bezüglich Codestyle (je nach Umfeld z. B. MISRA-C) sowie ausführlichere Tests, die potenzielle Programmierfehler aufdecken. Für beide Kategorien gibt es für alle verbreiteten Programmiersprachen unzählige Tools, sowohl kostenpflichtige als auch freie Varianten. Der Klassiker für C ist sicherlich PC-Lint/FlexeLint[3] bzw. sein freies Pendant Splint[4].

[3] http://www.gimpel.com
[4] http://www.splint.org

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

Hilfreich für die Testplanung/-strategie ist ebenfalls die Ermittlung der wichtigsten Kennzahlen/Metriken des Codes. Darunter fallen die reine Codegrösse (z. B. «effective Lines of Code») oder die Codekomplexität, z. B. «Cyclomatic Complexity» (McCabe, 1976).

Im Gegensatz zur statischen Codeanalyse, die mit geringem Initialaufwand und immer kleiner werdendem kontinuierlichem Aufwand betrieben werden kann, ist der Aufwand für eine dynamische Analyse schon deutlich grösser. Wer aber schon an einem grösseren Projekt gearbeitet hat, das über eine gute Abdeckung mit automatisierten Tests verfügt, möchte die Vorteile nie mehr missen.

Neben den eigentlichen Testresultaten können dank automatisierter Tests auch weitere Informationen gewonnen werden. So sagt die Anzahl an durchgeführten Tests noch nichts über die Qualität einer Software aus. Dafür muss auch die Code-Abdeckung der Tests in Betracht gezogen werden (am besten unter Berücksichtigung der Codekomplexität). Zudem bietet es sich bei dynamischen Tests an, zusätzlich statistische Daten über den Speicherverbrauch aufzuzeichnen («profiling»). Damit können auch Fehler wie «Memory Leaks» gefunden werden, die von Hand nur sehr mühsam zu finden sind.

Wenn man den initialen Aufwand für die Implementation von Tests erst einmal gemeistert hat, kann man die Transparenz eines Projekts deutlich vergrössern und dadurch das Risiko minimieren. Quasi als willkommener Seiteneffekt können sich die Entwickler dank einer durchgehenden Automatisierung wieder vermehrt auf ihre Kernaufgaben konzentrieren, anstelle sich mit den unbeliebten, mühsamen manuellen Tests abzumühen (die deshalb auch gerne «wegoptimiert» werden).

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

Feedback

Die oftmals unterschätzte Komponente in einem CI-System ist das Feedback. Die besten Tests nützen nichts, wenn der Entwickler nicht mitbekommt, welche Resultate herausgekommen sind. Denn stellt man erst vor dem grossen Release fest, dass der Build seit Tagen oder gar Wochen Fehler aufgewiesen hat, ist ein CI-System ziemlich nutzlos.

Wichtig dabei ist, dass das Feedback dazu eingesetzt wird, um die Entwickler zu unterstützen und das Team zu motivieren. Ein «Missbrauch» als Managementinstrument ist kontraproduktiv und sollte tunlichst vermieden werden. Bei der richtigen Anwendung entwickelt sich innerhalb des Teams oftmals ein beachtlicher Ehrgeiz, defekte Builds möglichst schnell zu reparieren oder die Fehlerrate von Tests tiefzuhalten.

Grundsätzlich können die Testresultate jederzeit auf dem Build-Server abgerufen werden, typischerweise über eine Weboberfläche. Gerade aber bei umfangreichen Projekten mit langen Build- oder Testzeiten ist eine aktive Benachrichtigung des Entwicklers unabdingbar. Dies geschieht typischerweise durch E-Mail, kann aber heutzutage auch via SMS, Instant-Messaging oder Twitter erfolgen.

Ein sehr wichtiger Aspekt beim Feedback ist die zentrale Verfügbarkeit der Daten. Sämtliche Resultate müssen auf einfache Weise zentral abgerufen werden können. Die gängigen CI-Lösungen bieten dafür oftmals umfangreiche Plugins an, mit denen Compiler-Fehler, Testresultate, Testabdeckung und weitere Messgrössen ohne Aufwand abgerufen werden können. Wenn dies nicht der Fall ist, läuft man Gefahr, dass die Resultate auf die Dauer nicht mehr beachtet werden. So wird beispielsweise ein Entwickler die Coverage-

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

Daten nicht regelmässig kontrollieren, wenn er diese zuerst von einem separaten Server herunterladen und noch aus einem ZIP-Archiv extrahieren muss.

Obwohl von vielen als Spielerei belächelt, von der Wirkung auf das Team aber nicht zu unterschätzen, können auch sogenannte «eXtreme Feedback Devices» (XFD) eingesetzt werden. Darunter versteht man ein Gerät, das den Status des Projekts auf einen Blick anzeigt. Der spielerischen Ader der Entwickler sind keine Grenzen gesetzt, wie man sich durch eine Suche im Internet selbst überzeugen kann. Das Spektrum geht von einfachen Lava-Lampen über Ampeln bis hin zu USB-gesteuerten Spielzeugraketenwerfern.

Umfangreiche CI-Lösungen bieten auch statistische Auswertungen über die vergangenen Builds an. Damit kann man sehr einfach die wichtigen Trends von Testabdeckung, Komplexität, Fehlerrate etc. erkennen und frühzeitig die nötigen Gegenmassnahmen ergreifen.

2.3 INFRASTRUKTUR

Die zentralen Infrastruktur-Komponenten einer CI-Lösung sind die Versionsverwaltung, der eigentliche Build-Server sowie mindestens ein Build-Agent. Je nach Grösse können alle auf dem gleichen physikalischen Server laufen oder auf separater Hardware. Gerade bei Umgebungen für grosse Projekte mit vielen Entwicklern macht es meistens Sinn, dass auch die eigentlichen Build-Agents vom Build-Server getrennt sind. Durch die so gewonnene Lastteilung kann die Zeit vom Check-in bis zum Resultat der Tests verkürzt werden. Bei extrem zeitaufwendigen Tests kann es zudem hilfreich sein, diese auf mehrere Test-Agents aufzuteilen.

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

Auch die Kompilation für verschiedene Betriebssysteme oder Prozessorarchitekturen oder die Tests auf verschiedenen Zielsystemen können so sehr einfach realisiert werden: Man ergänzt die CI-Infrastruktur einfach um ein entsprechendes Gerät, bindet es an den Build-Server an, und dieser kümmert sich dann um den Rest.

Dass die besten Entwickler nicht produktiv arbeiten können, wenn das Netzwerk instabil oder die Server nicht richtig gewartet werden, ist offensichtlich. Dass die Motivation und dadurch auch die Produktivität der Entwickler jedoch durch den Einsatz geeigneter Tools massiv gesteigert werden kann, ist vielleicht nicht jedem auf den ersten Blick klar. Wer sich aber schon einmal mit einer altertümlichen, umständlichen Entwicklungsumgebung herumschlagen musste, nur weil bei der Auswahl des Controllers diesem Umstand keine Beachtung geschenkt wurde, der wird dem nur zustimmen können.

Auch der Einsatz von leistungsfähigen Entwicklungsrechnern (und natürlich auch Servern) zahlt sich meistens innerhalb kürzester Zeit aus.

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

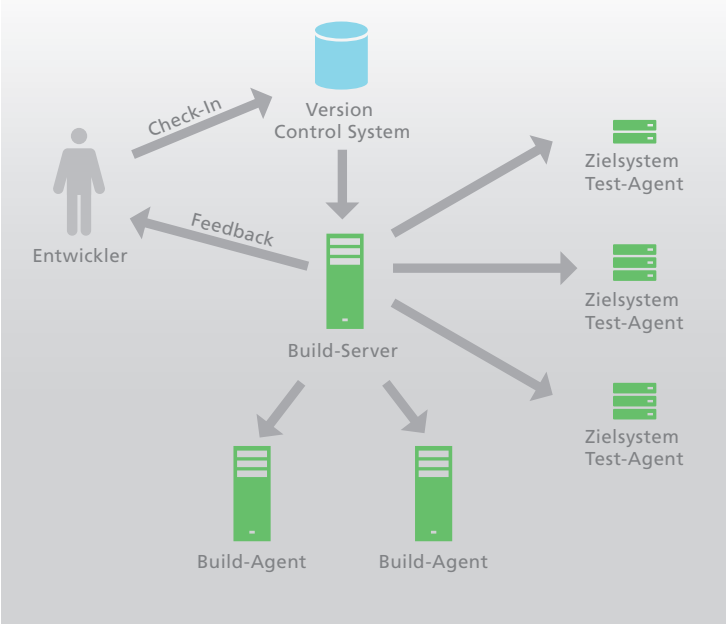


Abbildung 2:
Die Infrastruktur für
Continuous Integration

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

3 CONTINUOUS INTEGRATION FÜR EMBEDDED SYSTEME

Continuous Integration hat sich in breiten Teilen der Software-Entwicklungs-Community durchgesetzt. Trotzdem sieht man im Umfeld von Embedded Software nur wenige Firmen, die CI konsequent und durchgängig einsetzen. Erfahrungsgemäss ist das Embedded Umfeld aber seit jeher eher konservativ, was den Einsatz von neuen Technologien oder Methoden angeht.

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

Den Satz «… das mag ja für PCs funktionieren, aber wir haben ganz andere Anforderungen» hat wohl jeder schon etliche Male gehört. Es ist auch nicht von der Hand zu weisen, dass ein Softwareentwickler für Embedded Software ganz andere Herausforderungen zu meistern hat als einer für PC-Software. Einer der offensichtlichsten Punkte ist sicherlich die Heterogenität: Ein .NET-Entwickler hat eine sehr homogene Umgebung: Es gibt nur eine unterstützte Prozessorarchitektur, nur eine Entwicklungsumgebung, meistens nur eine Programmiersprache, und in der heutigen Zeit muss man sich auch nur noch mit einem Betriebssystem herumschlagen.

In der Welt von Embedded Projekten sieht das dann meistens ganz anders aus: Es gibt verschiedenste Prozessorarchitekturen, eine Vielzahl von Entwicklungsumgebungen, mehrere Programmiersprachen und diverse Betriebssysteme (und diese sind häufig noch «handgestrickt»).

Von diesem Aspekt her ist es ein Stück weit verständlich, dass viele Entwickler eher skeptisch gegenüber neuen Konzepten sind. Die Erfahrung hat aber gezeigt, dass man speziell im Embedded Umfeld sehr stark von den Vorteilen von CI profitieren kann.

3.1 ENTWICKLUNGSUMGEBUNG

Viele Embedded Softwareprojekte haben keine automatisierten Builds. Oft werden spezielle oder gar proprietäre Entwicklungsumgebungen eingesetzt, die sich auf den ersten Blick nicht so einfach automatisieren lassen. Doch dieser erste Blick trügt meistens. Denn mittlerweile haben etliche Hersteller entsprechende Funktionalitäten eingebaut, die aber vielen Entwicklern nicht bekannt sind. So bietet beispielsweise IAR für ihre Embedded Workbench ein Tool an, mit dem man bestehende Projekte direkt von der Kommandozeile

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

heraus kompilieren kann[5]. Diese Möglichkeit ist natürlich relativ eingeschränkt, bietet aber einen einfachen Start.

Wenn man eine flexiblere Lösung braucht, wird man normalerweise den Ansatz verfolgen, dass das Projekt nicht auf den proprietären IDE-Tools des Herstellers aufbaut, sondern ein generisches «Make-System» (GNU make etc.) eingesetzt wird, das dann schlussendlich den Compiler und Linker aufruft. Ein weiterer Vorteil, der durch die Verwendung eines nichtproprietären Build-Systems entsteht, ist die Unabhängigkeit zur eigentlichen IDE. So kann ein solches Projekt komfortabel mit einer generischen IDE wie z. B. Eclipse verwendet werden. Das hat den grossen Vorteil, dass man für sämtliche Projekte, Architekturen, Chips etc. die gleiche Entwicklungsumgebung einsetzen kann. Auch ist ein Wechsel beispielsweise auf einen anderen Compiler dadurch viel einfacher, weil der Build-Prozess weniger stark von den proprietären Tools abhängig ist.

3.2 CROSS-COMPILING, MEHRERE PLATTFORMEN

Die wenigsten Embedded Systeme laufen auf der gleichen CPU-Architektur und dem gleichen Betriebssystem wie der Entwicklungsrechner. Das hat zur Folge, dass mit speziellen Cross-Compile-Toolchains gearbeitet werden muss, die für die konkrete Kombination Host/Target ausgelegt sind. Oft wird man auch mit der Problematik konfrontiert sein, dass der Code auf mehreren Plattformen lauffähig sein muss. Was aufgrund der Komplexität auf den ersten Blick als Nachteil erscheinen mag (hoher Initialaufwand), stellt sich schnell als enormer Vorteil heraus: Denn wenn der Initialaufwand einmal erbracht wurde und das CI-System automatisch prozessor- und plattformübergreifend kompilieren kann, wird einem schnell klar, wie

[5] Siehe «BUILDING FROM THE COMMAND LINE» im IAR Embedded Workbench® IDE User Guide

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

viel Arbeit einem ein CI-System abnehmen kann. Denn welcher Entwickler hat schon Zeit, jede Änderung immer auf sämtlichen Plattformen zu testen? Vielleicht lassen sich auch nicht einmal mehrere Entwicklungsumgebungen parallel auf dem gleichen Rechner installieren, oder es steht keine genügend grosse Anzahl von Zielsystemen zur Verfügung, damit jeder Entwickler seine Änderungen plattformübergreifend testen könnte.

3.3 LIZENZEN

In vielen Embedded Softwareprojekten werden kostenpflichtige Toolchains eingesetzt. Wenn jetzt ein Build-Server bzw. Build-Agent aufgesetzt wird, werden natürlich zusätzliche Lizenzen fällig. Neben dem rein finanziellen Aspekt können auch zusätzliche Ärgernisse zum Tragen kommen. So sind beispielsweise Lizenzen, die über USB-Dongles implementiert sind, für Server-Systeme meistens relativ mühsam, da diese oft nur eine sehr begrenzte Anzahl USB-Ports zur Verfügung stellen und in einem Server-Rack herausstehende USB-Sticks sehr unpraktisch sind. Wesentlich flexibler sind natürlich sogenannte «Floating Licenses», die auf einem zentralen Lizenzserver abgespeichert werden und die der jeweilige Client (Entwicklungsrechner oder Build-Agent) kurzzeitig reservieren kann.

Es gibt aber auch noch eine andere, interessante Alternative: der Einsatz von verteilter Kompilierung. So können die normalen Entwicklungsrechner über das Netzwerk als verteilte Build-Agents genutzt werden. Dadurch braucht man nicht nur keine zusätzlichen Lizenzen, sondern erhält auch eine viel grössere Rechenkapazität inklusive Redundanz. Ob ein solches Szenario sinnvoll ist, muss natürlich im Einzelfall abgeklärt werden. Je nach Anzahl und Auslastung der Clients ist dies natürlich nicht zwingend gegeben.

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

3.4 ANSCHLUSSMÖGLICHKEITEN

Bei vielen Embedded Systemen sind die «herkömmlichen» Anschlüsse wie Ethernet nicht immer vorhanden. Deshalb kann es durchaus eine gewisse Herausforderung darstellen, ein Embedded System als Build- oder Test-Agent an den Build-Server anzuschliessen. Vor allem auf «richtigen» Server-Systemen ist die Anzahl und Vielfalt der vorhandenen Anschlüsse meistens sehr beschränkt.

Glücklicherweise gibt es heutzutage fast für alle im Embedded Bereich gängigen Schnittstellen wie CAN, RS-485 oder SPI etc. auch entsprechende Adapter auf eine «PC-kompatible» Schnittstelle in Form eines Ethernet- oder USB-Anschlusses.

3.5 HARDWARE-VERFÜGBARKEIT

Eine typische Einschränkung in einem Embedded Projekt ist die späte oder spärliche Verfügbarkeit von Hardware. Oftmals ist die definitive Hardware erst sehr spät im Projektverlauf verfügbar. Bis dahin muss die Software entweder auf einem möglichst ähnlichen System (z. B. Entwicklungsboard) getestet werden, oder man setzt eine Simulation oder Emulation ein.

Auch hier kann man durchaus von Continuous Integration profitieren. Beispielsweise, wenn nicht für jeden Entwickler eine Hardware zur Verfügung steht. Dank Continuous Integration und am zentralen Build-Server angeschlossener Hardware können alle Entwickler ihre Tests automatisch auf dem Zielsystem ausführen. Die Entwickler können lokal an einem Teil des Projekts arbeiten, den sie möglicherweise auf einem Entwicklungsboard oder mittels Simulation lokal testen können. Auf dem Build-Server wird dann die komplette Software integriert und auf dem Zielsystem getestet.

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

Hier kommt einem natürlich eine Systemarchitektur entgegen, die auf Portierbarkeit oder Simulation ausgelegt ist. Ein Beispiel dafür ist das von bbv entwickelte Apparatus Framework[6], das auch als Opensource-Variante frei erhältlich ist. Dank diesem Framework kann man seine Applikation einfach auf einer anderen Hardware laufen lassen oder einzelne Funktionalitäten oder Schnittstellen gezielt durch simulierte Teile ersetzen.

3.6 TESTBARKEIT

Die Testbarkeit von Embedded Software ist meistens deutlich schlechter als diejenige von PC-Software. Oder zumindest erhöht sich der Aufwand merklich, um eine gute Testabdeckung zu erreichen.

Neben der Tatsache, dass Testing bei vielen Entwicklern von Embedded Software nach wie vor verpönt zu sein scheint, gibt es auch handfeste technische Gründe. So sind klassische Unit-Tests für Software in C zwar grundsätzlich möglich, aber nur mit deutlich grösserem Aufwand als mit objektorientierten Sprachen wie C++ oder gar C#. Zudem ist es natürlich viel aufwendiger, Software zu testen, die auf einer anderen Architektur als der des Entwicklungsrechners läuft. Da sich aber glücklicherweise selbst im Umfeld von Embedded Systemen zunehmend die Erkenntnis durchsetzt, dass Tests unverzichtbar sind, gibt es auch immer bessere Tools und Frameworks, die den Entwickler unterstützen. Beispiele dafür sind das Mocking-Framework CMock[7] und das Testing-Framework Unity[8].

[6] http://apfw.sourceforge.net
[7] http://cmock.sourceforge.net
[8] http://unity.sourceforge.net

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

4 PRAXISBEISPIELE

Anhand einiger Beispiele aus der Praxis wollen wir aufzeigen, wie Continuous Integration für verschiedenste Embedded Anwendungen eingesetzt werden kann.

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

4.1 NATIVE ENTWICKLUNG AUF EMBEDDED LINUX

In der Zeit von immer leistungsstärkeren Embedded Systemen kann es manchmal sinnvoll sein, Software direkt auf dem Targetsystem zu kompilieren («nativ»). Da gerade auf Linux-Systemen die Toolchain (GCC) praktisch «frei Haus» dabei ist, kann damit die komplexe Thematik des Crosscompilierens einfach umgangen werden.

Ein typisches Setup für ein entsprechendes Linux-System ist Folgendes: Der Build-Server verbindet sich über das Netzwerk via Secure Socket Shell (SSH) auf das Zielsystem und kompiliert dort mit einer nativen Toolchain die Software. Der Sourcecode dafür wird entweder direkt auf dem Zielsystem ausgecheckt (dafür ist natürlich eine entsprechende Client-Software auf dem Zielsystem nötig) oder der Sourcecode wird zuerst auf dem Build-Server heruntergeladen, gepackt und dann via Secure Copy (SCP) oder File Transfer Protocol (FTP) auf das Zielsystem kopiert.

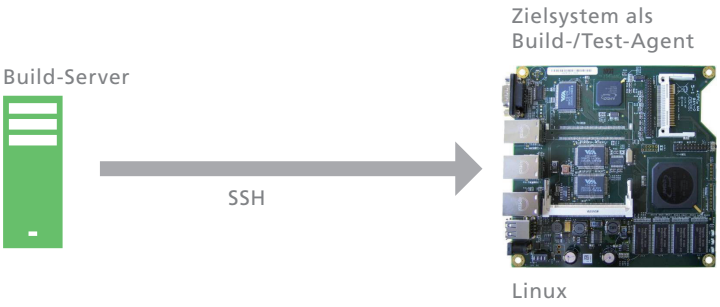


Abbildung 3:
CI mit Linux als Build- und Test-Agent

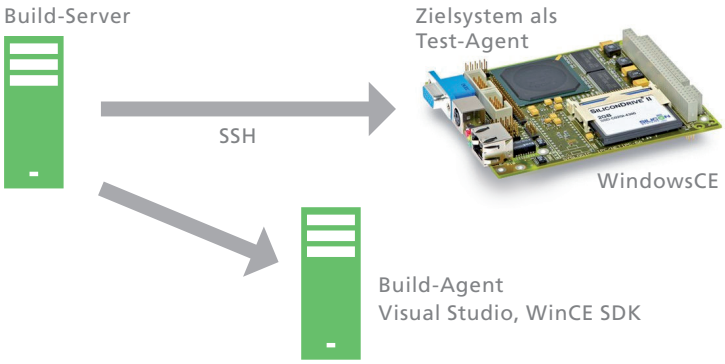
Die fertig erstellte Software wird direkt auf dem Zielsystem getestet, und die Testresultate werden anschliessend auf den Build-Server zurückkopiert.

Dadurch, dass das Zielsystem sowohl als Build- wie auch als Test-Agent verwendet wird, können die üblichen Probleme bei Crosscompilation weitgehend vermieden werden. Dazu gehören falsche oder inkompatible Bibliotheken, eine Vermischung der Host- und Zielsystem-Umgebung oder generell Probleme beim Einsatz von Tools, die nicht für den Crosscompile-Einsatz vorgesehen sind.

4.2 WINDOWS CE[9]

Da Microsoft keine native Toolchain für Windows CE anbietet, ist man hier zwingend auf das Visual Studio und zusätzliche Add-ins angewiesen. Daher braucht man auch zwingend einen Build-Agent mit Windows-Betriebssystem. Je nach Infrastruktur wird dieser Build-Agent auf einer dedizierten Hardware oder gerade auf dem Build-Server selber ausgeführt. Das Setup könnte dann folgendermassen aussehen:

Abbildung 4:
CI mit Build-Agent für Windows CE



[9] Bzw. Windows Embedded Compact, wie das Produkt neu heisst

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

Die Software wird auf dem Windows-Build-Agent kompiliert. Das Deployment auf das Zielsystem geschieht z. B. mit dem Microsoft-eigenen ActiveSync®-Protokoll über Ethernet, USB oder serieller Schnittstelle. Alternativ kann das Deployment auch über andere Protokolle wie SSH erfolgen, falls ein entsprechender Server auf dem Zielsystem installiert wird[10].

Wenn nicht nur eine einzelne Applikation, sondern ein komplettes System inkl. aller Treiber getestet werden soll, kann auch ein Speicherabbild mit einem JTAG-Adapter direkt ins FLASH geschrieben werden. Auch ein Download mit Hilfe eines Bootloaders via Trivial File System Protocol (TFTP) oder serieller Schnittstelle (z. B. XModem) ist eine praktische Methode.

4.3 FREERTOS AUF ARM7

Entgegen der weit verbreiteten Meinung ist Continuous Integration – selbst in Kombination mit Tests auf dem Zielsystem – auch auf leistungsschwachen Systemen ohne «High-level»-Netzwerkprotokolle etc. einsetzbar. Grundsätzlich sind dafür nicht einmal ein Betriebssystem oder ein Bootloader nötig, wobei natürlich diese zumindest das Deployment erheblich vereinfachen. Folgendes Szenario veranschaulicht quasi ein Minimalbeispiel für CI auf Embedded Systemen:

[10] OpenSSH for Windows CE: http://cessh.codeplex.com

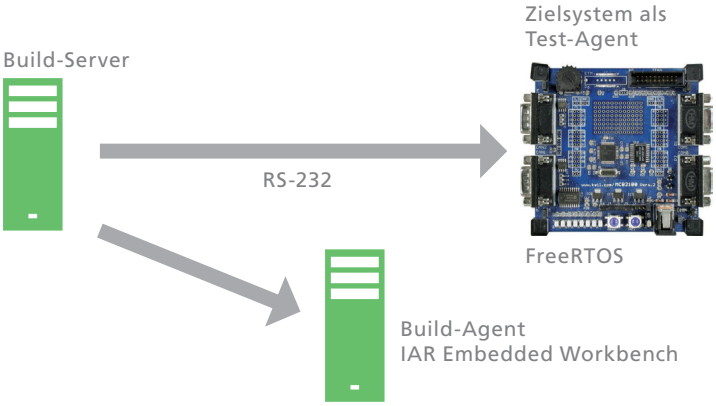


Abbildung 5:
CI mit Build-Agent für ein minimales Embedded System

Die Software wird auf einem Build-Agent mit der passenden Crosscompile-Toolchain kompiliert. Anschliessend wird das erzeugte HEX-File via serielle Schnittstelle auf den im Microcontroller integrierten Bootloader geladen. Ebenfalls über die serielle Schnittstelle können dann auf dem Target die Tests gestartet und am Schluss die Resultate ausgelesen werden.

Unter Umständen kann sich der Aufwand bei spezielleren Zielsystemen etwas erhöhen. Beispielsweise, wenn vor dem Download die Spannungsversorgung getrennt werden muss oder wenn der Bootloader nur per Tastendruck aktiviert werden kann. Meistens lassen sich solche Tasks aber einfach mit einer preisgünstigen Relais-Karte oder mit etwas teurerer Data-Acquisition-Hardware erledigen.

Der Einsatz von CI mit Tests auf dem Zielsystem ist bei noch leistungsschwächeren Systemen wie z. B. 8-Bit-Microcontroller meistens nicht praktikabel. Eine Testautomatisierung zum Beispiel in Form von kompletten Systemtests ist aber auch für solche Systeme äusserst hilfreich. Gerade bei kleinen Microcontrollern

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

mit einer gut überschaubaren Funktionalität kann oftmals mit vernünftigem Aufwand die komplette Umgebung simuliert oder emuliert werden. Vom Testsystem können dann sämtliche Ein- und Ausgänge stimuliert bzw. überwacht werden.

Als Minimallösung lohnt es sich fast immer, wenigstens den Sourcecode kontinuierlich zu integrieren. Anstelle von umfangreichen Tests mit dem Zielsystem selber können wenigstens die generischen, Hardware unabhängigen Codeteile wie Algorithmen oder Datenverarbeitung mit Unit-Tests auf dem Build-Server getestet werden.

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

5 FAZIT

Continuous Integration für Embedded Softwareprojekte ist nicht ganz einfach – aber es lohnt sich!

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

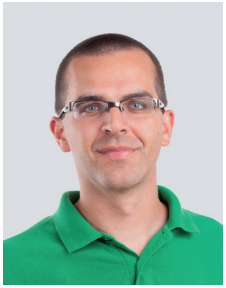
Wer damit beginnt, eine CI-Infrastruktur aufzubauen und Embedded Softwareprojekte darauf zu integrieren, der wird auf etliche kleinere und grössere Stolpersteine stossen. Man sollte sich aber nicht entmutigen lassen und versuchen, in kleinen Schritten zum Ziel zu kommen. Denn gerade in komplizierten und grossen Projekten wird sich die anfangs investierte Zeit umso mehr auszahlen.

bbv Software Services AG und bbv ICT Solutions AG bieten das ganze Spektrum von Dienstleistungen im Bereich von Continuous Integration an, von der Beratung über die Implementation bis hin zu Hosting-Services für CI-Umgebungen. Kontaktieren Sie uns, wir freuen uns, Ihnen zu helfen!

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

6 ANHANG

 COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG



6.1 AUTOR

David Büchi ist Dipl. Inf. Ing. FH und arbeitet seit 2007 bei der bbv Software Services AG als Senior-Softwareingenieur. Als Spezialist für Embedded Systeme hat er schon an etlichen Projekten mit Continuous Integration in den unterschiedlichsten Ausprägungen mitgearbeitet.

6.2 LITERATUR

Booch et al., G. (1991). Object Oriented Analysis and Design with Applications. Addison-Wesley.

Chiappori, F., Marbach, D., & Minder, R. (2011). Agile Entwicklung von Embedded Software.

Dolder, P. (2012). Introduction to Continuous Integration.

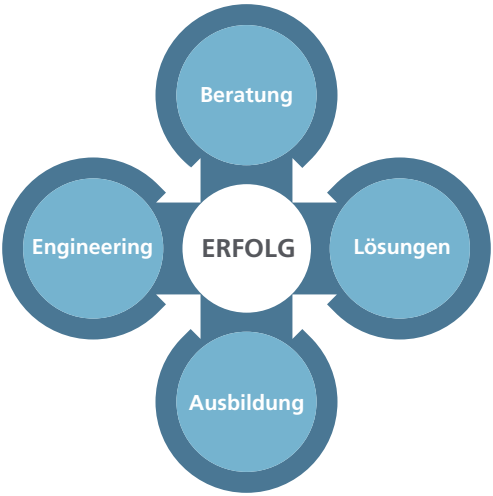
Duvall, P. M. (2007). Continuous Integration. Addison-Wesley.

McCabe, T. J. (1976). A Complexity Measure. IEEE Transactions on Software Engineering, 308-320.

COPYRIGHT © 2015, BBV SOFTWARE SERVICES AG

bbv
MAKING VISIONS WORK.

bbv Software Services AG ist ein Schweizer Software- und Beratungsunternehmen, das Kunden bei der Realisierung ihrer Visionen und Projekte unterstützt. Wir entwickeln individuelle Softwarelösungen und begleiten Kunden mit fundierter Beratung, erstklassigem Software Engineering und langjähriger Branchenerfahrung auf dem Weg zur erfolgreichen Lösung.



Unsere Booklets und vieles mehr finden Sie unter
www.bbv.ch/publikationen

MAKING VISIONS WORK.

www.bbv.ch · info@bbv.ch