Automatisierung des Continuous Testing für Embedded Systeme

Testen. Wir wissen alle, dass wir es tun müssen. Wir alle behaupten aber, nicht genug Zeit dafür zu haben oder beschweren uns über mangelnde Unterstützung seitens des Managements oder fehlende Ressourcen. Dennoch haben wir alle bereits Stunden damit verbracht, Software-bezogene Fehler in der Endphase der Entwicklung eines Produkts zu beheben, während wir damit kämpfen, das vereinbarte Produkteinführungsdatum zu halten.

Mit steigender Komplexität der Embedded-Software und zunehmendem Vertrauen in Code von Drittanbietern, z.B. Echtzeit-Betriebssystem (RTOS), Protokoll-Stacks oder Peripherie-Treiber von Halbleiterherstellern, wird frühes und kontinuierliches Testen essentiell, um Zeit und Ressourcen während der Projektentwicklung zu schonen.

Es ist kein Geheimnis, dass frühes Testen und die daraus resultierende Möglichkeit, Fehler zu beheben, Einsparpotenziale bietet. Das Buch „Software Engineering Economics“ von Barry Boehm sowie „The Mythical Man Month“ von Frederick Brooks werden in diesem Zusammenhang oft zitiert. Vor einiger Zeit veröffentlichte die NASA einen Artikel [1], in dem die Kosten für die Behebung eines Fehlers in Bezug auf die Zeit, die für die Fehleranalyse während des Entwicklungsprozesses aufgewendet wurde, analysiert wurden. Unabhängig davon, ob die Kosten Bottom-Up, Top-Down oder sogar als Teil einer Gesamtkostenverteilung analysiert wurden, war das Ergebnis immer ein exponentieller Kostenanstieg in Relation zu der aufzuwendenden Zeit für die Fehlerbehebung.

Autor:

Stuart Cording, Technical Marketing Manager, iSYSTEM AG

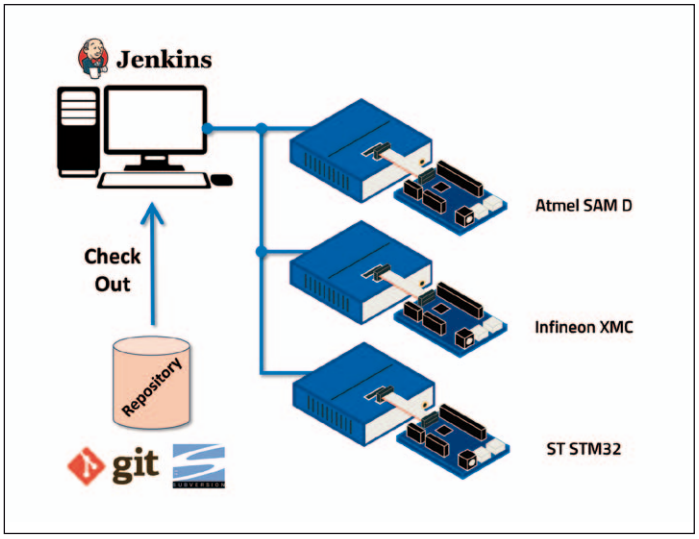

Bild 1: Jenkins checkt das Projekt aus dem Repository aus. Das gleiche Projekt kann mittels Jenkins auf mehreren Zielmikrokontrollern ausgeführt und getestet werden

Continuous Integration

Stellen Sie sich vor, dass sämtlicher Quellcode für Software-Stacks und Peripherie-Treiber so gut getestet wäre, dass Sie diesen als Ursprung eines Fehlers bei der eingebetteten Anwendung ausschließen könnten. Bei Kundengesprächen werden zunehmend die Themen Continuous Integration (CI) und Lösungen wie Jenkins, Hudson oder Bamboo angesprochen (Bild 3). Gründe hierfür sind unter anderem:

- **Automatisierung:** CI-Tools ermöglichen einem Ingenieur, Tests einmal zu konfigurieren und später automatisch auszuführen, so oft es erforderlich ist.
- **Integration:** Durch die nahtlose Verknüpfung mit gängigen Versionsverwaltungssystemen wie SVN und git können CI-Tools immer die aktuellste zu testende Codebasis auschecken. Neue Repositor-Check-ins können verwendet werden, um eine weitere Runde von Tests zu fahren.
- **Flexibilität:** Unabhängig davon, wie viele Ziel-MCUs, Compiler-Versionen oder Varianten Ihre Software unterstützt, kann eine einzelne CI-Task konfiguriert werden, um Tests für die zahlreichen Versionen desselben Quellcodes zu erstellen.
- **Erweiterbarkeit:** Wird eine Funktion benötigt, die nicht im Lieferumfang enthalten ist, steht eine Reihe von Plug-Ins zur Auswahl. Alternativ können eigene geschrieben werden.
- **Reporting:** Mit webbasierten Berichten und grafischen Darstellungen von Testergebnistrends und Code-Coverage, bis hin zu automatisierten E-Mail-Berichten und Ampeln (Grün – Test überstanden, Gelb – Fehler bei der Ausführung, Rot – Fehler bei der Übersetzung oder beim Testen) bietet ein CI-Tool dem Team ausreichend Möglichkeiten, über die Testergebnisse regelmäßig informiert zu werden.
- **Einfachheit:** Durch die webbasierten Schnittstellen lassen sich CI-Tools einfach und intuitiv konfigurieren.

Interne Entwicklungsprozesse anpassen

Continuous Integration ist eine Einstellungssache, ebenso ein Werkzeug, das erfordert, dass einige interne Entwicklungsprozesse angepasst werden müssen, um die Vorteile und den Nutzen zu sehen. Ein Teammitglied muss die Verantwortung für die CI-Umgebung übernehmen, die Testaufgaben implementieren und dafür sorgen, dass die notwendige Software (z.B. Com-

algorithm.c

```
int newPwmTime(int actual_temp, int ref_temp)
{
  int delta_temp;
  int newPwmValue;

  delta_temp = actual_temp - ref_temp;

  newPwmValue = NOMINAL_TEMP + (GAIN * delta_temp);

  if (newPwmValue > 100)
  newPwmValue = 100;

  if (newPwmValue < 0)
  newPwmValue = 0;

  return newPwmValue;
}
```

Oven Temp ADC	Reference Temp ADC	Corresponding Temperature °C	Calculated PWM Value
111	70	-16	100
87	70	24	62
71	70	51	19
60	70	69	0
52	70	82	0
46	70	92	0

Bild 2: Um den Algorithmus gründlich zu testen, werden Eingabeparameter und die zu erwartenden Ergebnisse festgelegt und als Testspezifikation im Python-Skript hinterlegt

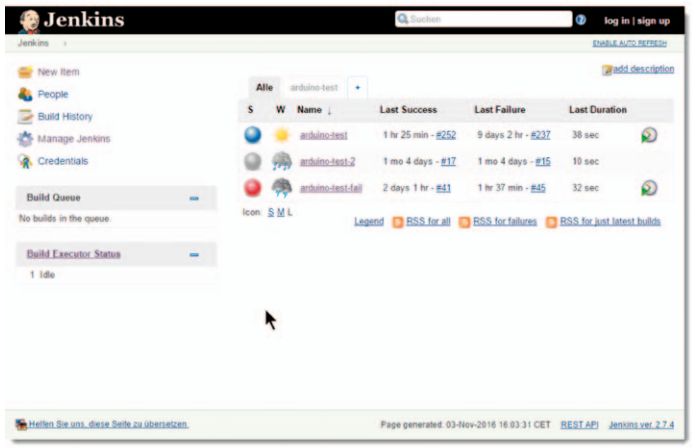

Bild 3: Über die CI-Tool Oberfläche (hier Jenkins) ist der Zustand anhand ein Ampelausgabe oder Wetterzeichen aller Tasks zu sehen

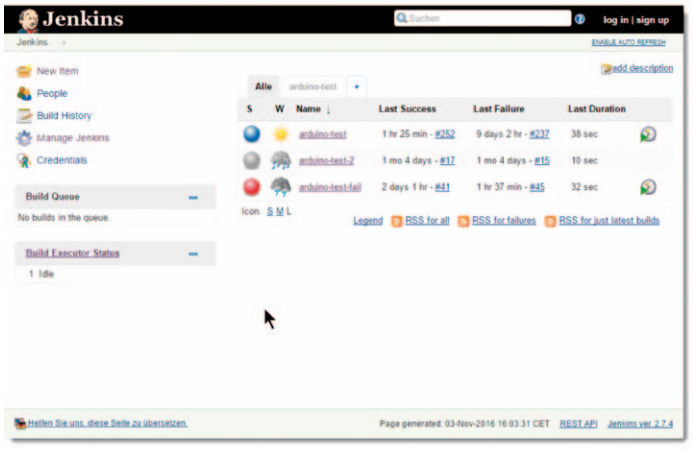

Bild 4: Die Anzahl der fehlgeschlagenen Tests wird graphisch für jeden einzelnen Task dargestellt

piler, Software Libraries) der aktuellen Version entspricht. Fehler, die während eines Tests auftreten, müssen unverzüglich dem verantwortlichen Software-Entwickler mitgeteilt werden, woraufhin eine Problemanalyse und -behebung erfolgen muss. Ebenso erfordert CI zwingend ein zentrales Repository. Gleichzeitig empfiehlt es sich, ein Bug- oder Issue-Tracking-System zu nutzen. Viele der Open-Source-Repository-Lösungen bieten diese Art von Funktionalität an.

Bedürfnisse der Embedded-Entwickler

Die meisten Dokumentationen rund um CI-Tools konzentrieren sich auf den Bau von komplexen Software-Projekten für den PC, webbasierten Diensten oder Smartphone-Anwendungen. Häufig ist ein Schwerpunkt auf Continuous Deployment oder Continuous Delivery erkennbar, welche dafür sorgen, dass die neueste Buildversion eines Programms oder einer App mit einem einzigen Mausklick erzeugt wird. Doch Embedded-Entwickler haben andere Bedürfnisse und Ziele. Ein CI-Tool nimmt einen festen Platz im Produktentwicklungsprozess ein, damit der Code für ein Embedded System während der Integrations- und Systemtests erstellt und getestet werden kann. Zusätzlich, eignet es sich für regelmäßige Tests des Codes der z.B. Peripherietreiber und Software-Stacks, die die Grundlage für eine komplette Embedded Applikation darstellen. Mit den richtigen Tools kann auch die als Binaries gelieferte Software getestet werden.

Während oder direkt nach der Code-Entwicklung testen

Eine weitere Änderung im Entwicklungsprozess besteht darin, dass die Tests während oder direkt nach der Entwicklung des Codes erfolgen. Nach bestandenen Tests, die nachweisen, dass der ausgeführte Code fehlerfrei ist, können sie, zusammen mit dem Quellcode, in das Repository hochgeladen werden. Dies ermöglicht ein späteres Testen für alle Teammitglieder im Falle von Code-Änderungen. Wird der Original Binary Code (OBC) Ansatz zur Codeprüfung verwendet, bei welchem die Tests auf dem Ziel MCU als Binärdateien nach dem Kompilieren ausgeführt werden, ist es sogar möglich, innerhalb kurzer Zeit Codefunktionalität auf unterschiedlichen Prozessorarchitekturen zu testen. Dadurch wird sichergestellt, dass sich das Kompilieren von C/C++ Datentypen nicht nachteilig auf die Funktionalität des Codes auswirkt.

Beispiel

Nehmen wir als Beispiel einen einfachen Algorithmus, der das erforderliche pulsbreitenmodulierte Signal (PWM) berechnet, um das Heizelement eines Ofens zu regeln (Bild 2). Die Dokumentation im Quelltext oder, noch besser, die Spezifikation gibt an, wie die ankommende Temperaturmessung ausgewertet werden soll und wie der erforderliche neue PWM-Wert berechnet wird. Dadurch ist es möglich, die erwarteten Rückgabewerte aus dem Code für bestimmte Eingabeparameter zu bestimmen. Um den Code mit Hilfe des OBC-Ansatzes zu testen, ist es auch notwendig, eine einfache Applikation (eine einfache main()-Funktion reicht, bei der die Funktion einmal genannt wird) zu haben, die zur Programmierung in den Zielmikrocontroller kompiliert werden kann. Zur Generierung eines Python-Skriptes zum Testen des Codes On-Target kann ein Test-Tool wie testIDEA von iSYSTEM verwendet werden. Dieses Python-Skript wird zusammen mit dem Quellcode in das Repository eingecheckt.

Der nächste Schritt besteht aus einer neuen Aufgabenerstellung innerhalb eines CI-Tools wie Jenkins, die diesen Software-Algorithmus auscheckt, erstellt und testet. Nachdem in Jenkins hinterlegt wurde, wo sich das zentrale Repository befindet, wird im nächsten Schritt die Binärdatei kompiliert. Hier erweist es sich als sinnvoll, ein einfaches Makefile zu erstellen, das die Binärdateien für die verwendeten Mikrocontroller der Embedded-Anwendung erstellen kann. Für den Fall, dass verschiedene Compiler/Mikrocontroller-Kombinationen den C-Quellcode-Algorithmus unterschiedlich interpretieren, schlagen die Tests auf diesem speziellen Ziel-Mikrocontroller fehl. Ist der C-Quellcode tatsächlich fehlerhaft werden die Tests auf allen Targets fehlschlagen (Bild 1).

Die Einstellmöglichkeiten in Jenkins erlauben zudem, das erstellte Python-Testskript als Teil der neuen Aufgabe auszuführen. Um sicherzustellen, dass Testerfolg und -ausfall dokumentiert werden, muss das Python-Testskript vorab so konfiguriert werden, dass die Testergebnisse in einem XML-Format, der so genannten „junit“ -Datei, gespeichert werden. Mit dem entsprechenden Plug-In in Jenkins kann zusätzlich die Pass-/Fail-Rate für die erstellten Tests grafisch angezeigt werden, wodurch das Team einen Überblick über Verbesserungen in der Testabdeckung erhält.

Code-Coverage

Code-Abdeckung ist ein zusätzliches Maß, das in die Tests eingebaut werden kann. Die derzeit verfügbaren Plug-ins erfordern aber einige Änderungen, um die Metriken vollständig zu unterstützen, die OBC-Tests generieren. Der Mehrwert der Code-Coverage Ergebnisdarstellung liegt in den Einblicken des Teams in Bezug auf die Gründlichkeit ihrer Tests, ebenso, ob alle Zeilen des Codes untersucht wurden (Bild 4).

Fazit

Eins ist klar: der Weg zur Implementierung von CI-Tools ist steinig. Prozesseinführung, -änderung oder -modifikation, wird häufig auf Widerstände stoßen. Sobald CI aber im gesamten Entwicklungsprozess integriert ist, werden alle Anstrengungen der Realisierung schnell vergessen sein.

■ *iSYSTEM AG*
www.isystem.com

Links: [1] - https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20100036670.pdf