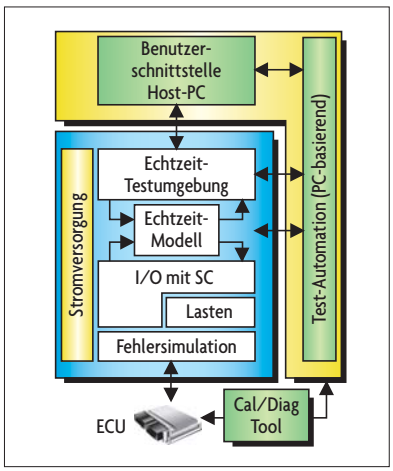
Python-Skripte testen Steuergeräte in Echtzeit

Echtzeit-Tests taktsynchron mit dem Hardware-in-the-Loop-Simulationsmodell ausführen

Die Ausführung von Steuergerätetests direkt durch den Echtzeit-Prozessor des HIL-Simulators eröffnet neue Testmöglichkeiten bezüglich Reaktivität, Genauigkeit und Reproduzierbarkeit. Der Einsatz der mächtigen objektorientierten Skriptsprache „Python" zur Programmierung von Echtzeit-Tests ermöglicht eine hohe Flexibilität bei der Testerstellung.

Von Holger Krisp

Im Automobilbereich stellen mehrere miteinander vernetzte Steuergeräte die elektronische Gesamtfunktion des Kraftfahrzeuges bereit. In der Entwicklungsphase müssen die Steuergeräte und ihr korrektes Zusammenspiel im Steuergeräteverbund möglichst frühzeitig im Labor getestet werden, um Fehler schon vor dem Aufbau realer Prototypen erkennen und beseitigen zu können. Durch die „Hardware-in-the-Loop"-Simulation (HIL-Simulation) kann die Einsatzumgebung im Fahrzeug realitätsgetreu schon im Labor vorgespiegelt werden.



I Bild 1. Testausführung synchron mit dem Simulationsmodell.

Hierbei werden ein oder mehrere Steuergeräte mit dem Simulationssystem verbunden, auf dessen Prozessorkarte(n) Echtzeit-Modelle zur Nachbildung gerechnet werden, z.B. von Motor-, Getriebe- und Fahrdynamikverhalten. Über die leistungsfähigen Hardwareschnittstellen des HIL-Simulators (z.B. für digitale und analoge I/O-Kanäle, Busschnittstellen, elektrische Fehlersimulation und Diagnoseschnittstellen) lassen sich die zu testenden Steuergeräte gezielt beeinflussen und in ihrer Reaktion beobachten.

Testautomatisierung auf der HIL-Prozessorkarte

Die HIL-Simulation im Zusammenspiel mit leistungsfähiger Testautomatisierung hat sich mittlerweile als zwingender Absicherungsschritt in der Elektronikentwicklung bei nahezu allen OEMs und Zulieferern durchgesetzt. Gleichzeitig sind die Anforderungen an die Steuergerätetests bezüglich Reaktivität, zeitlicher Auflösung und Reproduzierbarkeit deutlich gestiegen. Deshalb ist die heute oft anzutreffende PC-basierte Testausführung für anspruchsvolle Testszenarien nicht immer ausreichend, neuartige Ansätze zur Realisierung von Echtzeit-Tests sind erforderlich.

Das Testautomatisierungswerkzeug AutomationDesk 1.4 von dSpace [1] bietet hierfür als neue Möglichkeit an, Steuergerätetests taktsynchron zum Ablauf des HIL-Simulationsmodells auszuführen. Dafür wurde eine spezielle Ausführungsumgebung entwickelt (Real-Time Test Environment, **Bild 1**). Diese kann parallel zum Simulationsmodell auf der Prozessorkarte des HIL-Simulators Echtzeit-Tests abarbeiten. Die Kommunikation zwischen Tests und Simulationsmodell wird effizient über den gemeinsamen Speicher der Prozessorkarte abgewickelt und garantiert dadurch kurze Latenzzeiten bei Lese- und Schreibzugriffen auf Modellgrößen. Die enge Ausführungskopplung zwischen Echtzeit-Tests und Simulationsmodell garantiert, dass Steuergerätetests auf Modellgrößen jedes Simulationsschrittes lesend und schreibend zugreifen können. Ein „Übersehen" von Modellän-

derungen kann prinzipbedingt nicht auftreten.

Mehrere unabhängige Echtzeit-Tests können parallel ausgeführt werden, wobei das Herunterladen zusätzlicher Tests während der laufenden HIL-Simulation problemlos möglich ist. Echtzeit-Tests können Daten untereinander austauschen, was eine hohe Flexibilität bei der Testimplementierung gewährt. Auf die HIL-Hardwareschnittstellen greifen die Echtzeit-Tests über vom Simulationsmodell vorbereitete Modellgrößen zu (z.B. für das Schreiben eines digitalen Ausgangskanals).

Die Komponenten der Echtzeit-Testumgebung (Real-Time Test Environment) zeigt **Bild 2**. Den Kern bildet ein echtzeitfähiger Python-Interpreter, welcher für die HIL-Prozessorkarten DS1005 und DS1006 entwickelt wurde und Python-Skripte synchron zum Simulationsmodell abarbeiten kann. Ein Real-Time Test Scheduler (kurz: RTT-Scheduler) sorgt mit einer Zeitsteuerung dafür, dass mehrere Python-Skripte parallel ausgeführt werden können. Für das Herunterladen von Echtzeit-Tests auf die Prozessorkarte und die Steuerung ihres Ausführungszustandes ist eine spezielle Serviceschicht zuständig, das Echtzeit-Test-Management. Das PC-seitige Gegenstück hierzu bietet der Echtzeit-Test-Manager-Server, welcher über Python-Skripte auf dem PC durch AutomationDesk-Testsequenzen oder ein spezielles GUI (den Echtzeit-Test-Manager) angesprochen werden kann.

Zusätzlich sind auf dem PC spezielle RTT-Bibliotheken verfügbar, welche von Echtzeit-Tests verwendet werden können. Ein Bytecode-Generator sorgt für die Umsetzung von Python-Dateien in ablauffähige Testbeschreibungen auf der Prozessorkarte. Die laufenden Tests lassen sich vom PC aus beobachten (z.B. durch die Test- und Experimentiersoftware ControlDesk).

Vom Python-Skript zum Echtzeit-Test

Für die Ausführung von Echtzeit-Tests auf der Prozessorkarte ist ein spezieller Workflow (**Bild 3**) vorgesehen. Als erstes wird der „RTT Python Interpreter“ zur Echtzeit-Applikation hinzugefügt. Dieses geschieht über eine spezielle Compileroption beim Übersetzen des Simulationsmodells. Im zweiten Schritt wird die Applikation, welche nun den Python-Interpreter enthält, auf die Prozessorkarte des HIL-Simulators heruntergeladen. Nach Abschluss dieser beiden Schritte sind Echtzeit-Tests mit diesem so vorbereiteten Simulationsmodell möglich. Die Tests können hierbei auf alle enthaltenen Modellvariablen zugreifen, ohne dass das Modell noch einmal neu übersetzt werden muss.

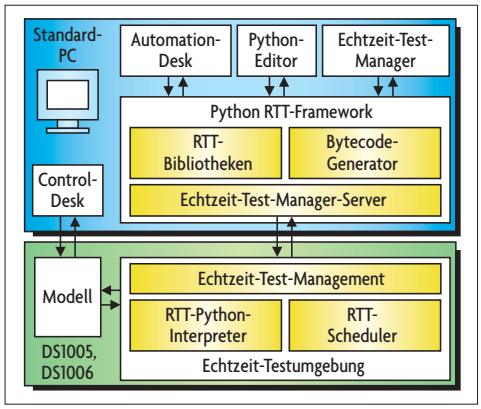


Bild 2. Die Echtzeit-Testumgebung besteht aus einem Standard-PC mit dem Python-RTT-Framework und den Prozessorkarten DS1005 und DS1006 mit der Echtzeit-Testumgebung.

Der Anwender implementiert seinen Echtzeit-Test in der Programmiersprache Python auf dem PC (z.B. in einem normalen Texteditor oder in einer speziellen Python-Entwicklungsumgebung). Hierbei stehen ihm alle aus Python bekannten Sprachkonstrukte zur Verfügung. Er kann in seinem Test zusätzlich selbstgeschriebene Python-Module importieren oder schon vorgefertigte Bibliotheken wiederverwenden (z.B. Python-Standardbibliotheken oder spezielle RTT-Bibliotheken von dSpace). Die entstandene Python-Datei wird im ASCII-Format dem Bytecode-Generator auf dem PC übergeben, welcher diese in eine BCG-Datei übersetzt. Hierbei werden auch die im Python-Test importierten Module in die BCG-Datei integriert, so dass diese Datei alle zum Testablauf notwendigen Informationen enthält.

Die BCG-Datei wird im nächsten Schritt auf die Prozessorkarte heruntergeladen. Das Herunterladen lässt sich auch bei schon laufenden Echtzeit-Tests durchführen, bereits laufende Tests müssen nicht in ihrer Ausführung unterbrochen werden. Beim Ladevorgang kann dem Test ein Python-Objekt als Übergabeparameter mitgegeben werden, welches der Test verwenden kann. Ein auf die Prozessorkarte heruntergeladener Test lässt sich im letzten Schritt des Workflow vom Host aus starten und in der Testausführung steuern.

Implementierung von Echtzeit-Tests

Die Akzeptanz einer Testlösung hängt neben dem reinen Funktionsumfang auch von der Art des Nutzerzugangs und der Testprogrammierung ab. Echtzeit-Tests werden bei der hier vorgestellten Lösung mit der objektorientierten Programmiersprache Python [2] beschrieben. Python bringt hierbei schon komplexe Beschreibungsmittel mit (wie z.B. Importmechanismen, Klassendefinitionen, aufwendige Datenstrukturen und umfangreiche Standardbibliotheken). Diese Python-Sprachkonstrukte können ohne Abstriche für die Implementierung von Echtzeit-Tests verwendet werden.

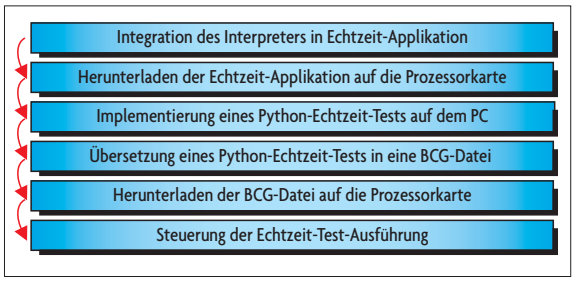


Bild 3. Vorgehensweise bei der Ausführung von Echtzeit-Tests.

Ergänzt werden sie durch Testbibliotheken, bereitgestellt von dSpace, welche z.B. für den symbolischen Modellvariablenzugriff, die Zeitsteuerung von Echtzeit-Tests und die Abbildung von Testparallelität zuständig sind.

Modellzugriff

Eine Hauptfunktion von Echtzeit-Tests ist die Beobachtung und Veränderung von Größen des Simulationsmodells

```
from rttlib import variable
Const1 = variable.Variable\
  (r'Model
Root/Const1/Value')
```

Listing 1. Mit der Python-Klasse *variable* kann ein Python-Variablenobjekt unter Bezugnahme auf den symbolischen Modellpfad angelegt werden

```
from rttlib import variable
Const1 = variable.Variable(r'Model Root/Const1/Value')

def MainGenerator():
        Const1.Value = 1.0  # Simulationsschritt n
        yield None
        Const1.Value = 2.0  # Simulationsschritt n+1
```

Listing 2. Zeitsteuerung mit dem *yield*-Befehl

```
from rttlib import variable
Time = variable.Variable(r'currentTime')

def wait (seconds):
        start_time = Time.Value
        while( (start_time + seconds) > Time.Value):
                yield None
```

Listing 3. Der *yield*-Befehl lässt sich auch verschachteln und kann damit eine Echtzeit-Wartefunktion erzeugen

```
Const1.Value = 1.0
yield wait(5.0)
Const1.Value = 2.0
```

Listing 4. Mit der *wait*-Funktion kann man Wartezeiten im Sekundenformat programmieren

während der Testausführung. Dieses ist nötig, um über die Hardwareschnittstellen des HIL-Simulators das zu testende Steuergerät gezielt stimulieren und dessen Reaktion beobachten zu können.

Für diesen Zweck steht in den RTT-Bibliotheken eine spezielle Python-Klasse *variable* zur Verfügung, die ein Python-Variablenobjekt unter Bezugnahme auf den symbolischen Modellpfad anlegt (**Listing 1**).

Schreib- und Lesezugriffe auf diesen Modellparameter können über die *Value*-Eigenschaft dieses Variablenobjektes durchgeführt werden. Um den Inhalt der Modellvariable zu inkrementieren, reicht der Befehl:

```
Const1. Value +=1.0
```

Die Adressauflösung wird durch die Variablenklasse geleistet. Hierfür wird in gepackter Form die Zuordnung zwischen symbolischem Pfad der Modellelemente und dazugehörigen Speicheradressen der Echtzeit-Applikation auf die Prozessorkarte geladen. Damit können Echtzeit-Skripte ohne Rückgriff auf den PC auf alle Modellvariablen zugreifen. Ändert sich diese Zuordnung (z.B. durch Neuübersetzen des Modells nach Modifikation), werden diese Informationen automatisch aktualisiert und sind für Echtzeit-Tests verfügbar. Dies führt zu einer prinzipiellen Unabhängigkeit zwischen Simulationsmodell und Testbeschreibung.

Zeitsteuerung mit *yield*

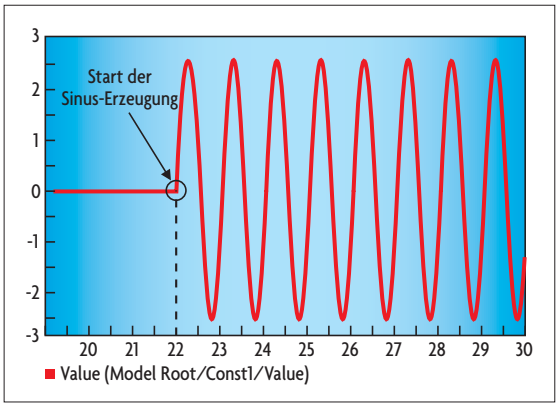
Eine wesentliche Anforderung an Echtzeit-Tests ist die Möglichkeit zur Spezifikation des genauen Zeitverhaltens aller Testaktionen, welches während der Testausführung exakt reproduziert werden soll. Bei der hier vorgestellten Produktlösung wird der Python-Standardbefehl *yield* (offiziell verfügbar ab Python 2.3) zur Zeitsteuerung verwendet.

Wird in einer Python-Funktion der Befehl *yield* verwendet, so wird diese Funktion automatisch zu einer so genannten Generatorfunktion. Diese speziellen Funktionen haben die Eigenschaft, dass sie schrittweise ausgeführt werden können. Erkennt der Python-Interpreter einen *yield*-Befehl im Laufe der Skriptabarbeitung, wird die Kontrolle an die Funktion zurückgegeben, welche die Generatorfunktion ursprünglich aufgerufen hat. Die Generatorfunktion kann danach erneut aktiviert werden, wobei sie die Ausführung genau an der Stelle ihrer Unterbrechung durch den *yield*-Befehl fortsetzt. Die lokalen Variableninhalte werden hierbei wieder hergestellt, weshalb man Generatorfunktionen auch als so genannte unterbrechbare Funktionen bezeichnet.

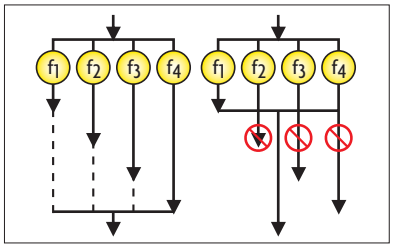
Alle Echtzeit-Testskripte sind in eine übergeordnete Generatorfunktion *MainGenerator* eingebunden. Diese wird zu Beginn des Echtzeit-Tests vom RTT-Scheduler aufgerufen und startet die Testausführung. Beim Auftreten des ersten *yield*-Befehls wird die Ausführung des Skriptes unterbrochen. Der RTT-Scheduler aktiviert das Skript dann erst wieder im nächsten Simulationsschritt. **Listing 2** zeigt, wie der *yield*-Befehl zur Zeitsteuerung eingesetzt werden kann. Hierbei wird über ein Variablenobjekt eine Modellgröße im ersten Testschritt mit dem Wert 1.0 beschrieben. Danach folgt ein *yield None*-Befehl, welcher die Ausführungskontrolle an den RTT-Scheduler zurückgibt. Im genau darauf

```
from math    import sin, pi
from rttlib  import variable

Time = variable.Variable(r'currentTime')
Const1 = variable.Variable(r'Model Root/Const1/Value')

def generate_sin(var,amp,freq):
        start_time = Time.Value
        omega = 2.0 * pi * freq

while(1):
        var.Value = amp * sin(omega*(Time.Value-start_time))
        yield None

def MainGenerator():
        yield generate_sin(Const1, 2.5, 1.0)
```

Listing 5. Implementierung eines Sinusgenerators mit Hilfe des Python *math*-Moduls



I Bild 4. Erzeugung einer einzelnen Sinusfunktion.



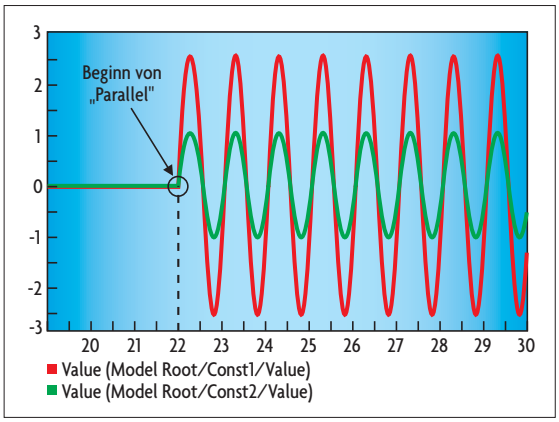
I Bild 5. „Parallel“ (links) und „ParallelRace“ (rechts) im Vergleich.

folgenden Simulationsschritt wird die Ausführung nach dem *yield*-Befehl fortgesetzt und die gleiche Modellgröße mit dem Wert 2.0 beschrieben. Dadurch beschreibt das Skript die Modellvariable in zwei aufeinanderfolgenden Simulationsschritten mit den Werten 1.0 und 2.0.

Echtzeit-Wartefunktion

Der *yield*-Befehl kann auch verschachtelt verwendet werden. Hierdurch kann einfach eine Echtzeit-Wartefunktion implementiert werden (**Listing 3**).

Die dort implementierte *wait*-Funktion speichert die Simulationszeit ihres Aufrufes ab und verlässt diese



I Bild 6. Sinuserzeugung mit dem Parallel-Befehl.

```
from rttlib   import scheduler, variable
from userlib  import generate_sin

Const1 = variable.Variable(r'Model Root/Const1/Value')
Const2 = variable.Variable(r'Model Root/Const2/Value')

def MainGenerator():
    yield scheduler.Parallel(generate_sin(Const1, 2.5, 1.0),\
          generate_sin(Const2, 1.0, 1.0))
```

I Listing 6. Stimulation von zwei Modellparametern mit Hilfe des Parallel-Befehls

```
from rttlib   import scheduler, variable
from userlib  import generate_sin, wait

Const1 = variable.Variable(r'Model Root/Const1/Value')
Const2 = variable.Variable(r'Model Root/Const2/Value')

def MainGenerator():
   yield scheduler.ParallelRace \
         (generate_sin(Const1, 2.5, 1.0),\
         generate_sin(Const2, 1.0, 1.0),\
         wait(5.0))
```

I Listing 7. Zeitlich begrenzte Sinusstimulation mit Hilfe des „ParallelRace“-Befehls und der „wait“-Funktion

Funktion erst wieder, wenn die als Parameter angegebene Anzahl von Sekunden verstrichen ist. Verwendet man diese Funktion in dem Skriptfragment in **Listing 4**, so wird dem Modellparameter zuerst der Wert 1.0 und nach genau fünf Sekunden der Wert 2.0 zugewiesen. Zu beachten ist hierbei, dass Generatorfunktionen (hier: *wait*) per Konvention mit einem führenden *yield*-Befehl aufgerufen werden müssen.

Import von Bibliotheksmodulen

Mit dem *import*-Befehl von Python ist es möglich, Echtzeit-Tests zu modularisieren. Hierbei können häufig verwendete Funktionen (z.B. die vorgestellte *wait*-Funktion) in separate Python-Dateien ausgelagert und wiederverwendet werden.

Zudem kann der Anwender zahlreiche Funktionen verwenden, welche in Standard-Bibliotheksmodulen mit Python ausgeliefert werden. Damit kann z.B. ein Sinusgenerator unter Rückgriff auf das Python *math*-Modul implementiert werden (**Listing 5**). Wird dieses Echtzeit-Skript gestartet, so ergibt sich der in **Bild 4** dargestellte zeitliche Verlauf der stimulierten Variablen. Hierbei ist zu beachten, dass die Variable vom Skript endlos stimuliert wird, da die *generate_sin*-Funktion als Endlosschleife mit Hilfe eines *while(1)*-Konstruktes implementiert ist. Soll die Sinuserzeugung unterbrochen werden, so muss die Skriptausführung unterbrochen oder gestoppt werden.

Beschreibung von Parallelität

In Tests ist es häufig erforderlich, parallele Abläufe beschreiben zu können. Damit ist es möglich, Testaktionen innerhalb eines Testskripts nebenläufig ausführen zu lassen. Zu diesem Zweck gibt es zwei spezielle Sprachelemente, *Parallel* und *ParallelRace*, welche sich in ihrer Abbruchbedingung unterscheiden (**Bild 5**).

Bei beiden Konstrukten können mehrere Python-Generatorfunktionen parallel gestartet werden (in diesem Beispiel die Funktionen f_1 bis f_4). Beim *Parallel*-Befehl ist die parallele Ausführung erst dann beendet, wenn alle Teilzweige vollständig abgearbeitet sind. Beim *ParallelRace*-Befehl hingegen ist die parallele Ausführung schon dann beendet, wenn der erste Teilzweig vollständig abgearbeitet worden ist. Damit „gewinnt“ praktisch der schnellste Zweig das Ausführungsrennen (im Beispiel f_1) und zwingt die übrigen drei Zweige zum vorzeitigen Abbruch.

Ein Beispiel zum praktischen Einsatz des Parallel-Befehls zeigt das **Listing 6**, bei welchem zwei Sinusgenera-

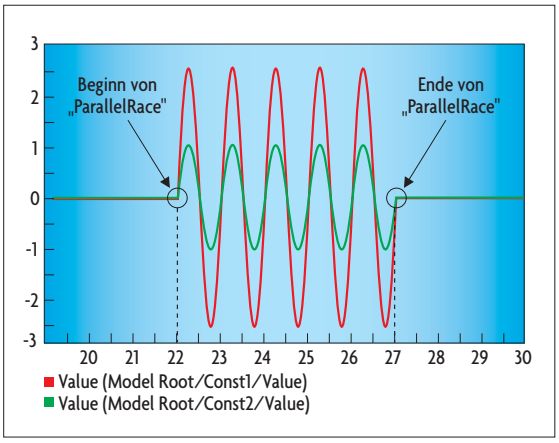


Bild 7. Sinuserzeugung mit dem *ParallelRace*-Befehl.

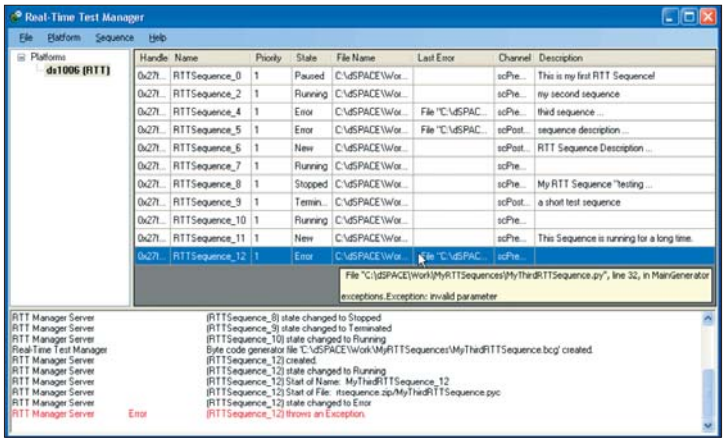


Bild 8. Der Real-Time Test Manager als GUI für die Verwaltung von Echtzeit-Tests.

toren kombiniert werden. Hierfür wird die *scheduler*-Klasse importiert, welche die *Parallel*- und *ParallelRace*-Funktionen zur Verfügung stellt.

Beim Start des Skripts werden die beiden Modellparameter Const1 und Const2 parallel mit einer Sinusfunktion mit jeweils unterschiedlichen Parametern stimuliert (**Bild 6**). Da die Stimulation jeweils über die endlos laufende *generate_sin*-Funktion durchgeführt wird, terminiert der *Parallel*-Befehl nie. Durch den *ParallelRace*-Befehl kann die Sinuserzeugung in ihrer Zeit begrenzt werden (**Listing 7**). Hierbei ist der *Parallel*-Befehl gegenüber Listing 6 durch den *ParallelRace*-Befehl ersetzt worden, wobei dort noch eine zusätzliche Wartefunktion eingebaut worden ist.

Dadurch, dass der *wait*-Befehl eine begrenzte Ausführungsdauer aufweist, wird nach dieser Zeit auch die Ausführung der parallel gestarteten, endlosen Sinusgeneratoren beendet. Damit erhält der Anwender eine Sinuserzeugung auf den beiden Modellparametern für exakt 5 Sekunden (**Bild 7**).

Testverwaltung mit dem Real-Time Test Manager

Neben der Integration mehrerer paralleler Abläufe in einem Testablauf können auch mehrere Testskripte parallel abgearbeitet werden. Der Übersichtlichkeit wegen ist mit dem Real-Time Test Manager eine grafische Benutzeroberfläche verfügbar (**Bild 8**), welche einen Überblick über die auf der Prozessorkarte vorhandenen Tests gibt (mit Testnamen, zugehörigem Dateipfad, Ausführungsstatus, Fehlerrückgaben usw.). Zusätzlich erlaubt das GUI auch das Herunterladen und die Ausführungssteuerung der Echtzeit-Tests.

Ausgeführter Echtzeit-Test	Ausführungszeit
Echtzeit-Wartefunktion (Listing 3)	0,41 µs
Einzelner Sinusgenerator (Listing 5)	1,30 µs
Zwei parallel abgearbeitete Sinusgeneratoren (Listing 6)	2,35 µs
Zwei für 5 Sekunden parallel abgearbeitete Sinusgeneratoren (Listing 7)	2,75 µs

Typische Ausführungszeiten für Echtzeit-Tests auf der HIL-Prozessorkarte DS1006 (Taktfrequenz 2,6 GHz).

Die hier angebotene Funktion kann auch durch reine Python-Programmierung auf dem PC oder in Testsequenzen von AutomationDesk genutzt werden. Der Real-Time Test Manager zeigt hierbei immer den aktuellen Status aller auf der Prozessorkarte vorhandenen Echtzeit-Tests an.

Benchmarkergebnisse

Die Echtzeit-Tests werden auf der Prozessorkarte des HIL-Simulators gerechnet; sie benötigen eine bestimmte Rechenzeit pro Rechenschritt. Typische Testskripte sind auf der dSpace-Prozessorkarte DS1006 mit AMD-Athlon-Prozessor bei einer Taktfrequenz von 2,6 GHz vermessen worden. Das verwendete Simulationsmodell verfügte hierbei über eine Rechenschrittweite von 1 ms. Das Einbetten des Python-Interpreters verursacht einen zusätzlichen Overhead von ca. 0,25 µs. Die benötigte Rechenzeit für die Ausführung typischer Testskripte sind in der **Tabelle** aufgelistet. Reserviert man sich ca. 10 % der Ausführungszeit für Echtzeit-Tests (in diesem Beispiel wären dies 100 µs), so sind parallel zur Modellrechnung umfangreiche Testszenarien durch das Echtzeit-Testen abdeckbar.

Als Ausführungsplattformen für Echtzeit-Tests werden die HiL-Prozessorkarten DS1005 (IBM-PowerPC) und DS1006 (AMD-Athlon-Prozessor) von dSpace unterstützt. Darüber hinaus erweitert dSpace kontinuierlich sein Produktsortiment, z.B. durch neue RTT-Bibliotheken. *gs*

Links

[1] www.dspace.com
[2] www.python.org

Dipl.-Ing. Holger Krisp
studierte Technische Informatik an der Universität Hannover und arbeitet als Produktmanager für Test- und Experimentiersoftware bei der dSpace GmbH in Paderborn.
HKrisp@dspace.de