

Remote-Debugging mit Eclipse

Durch Eclipse wird Debugging von Linux-Systemen so komfortabel wie ein Tool vom Hersteller

Das Debugging von Code, der nicht auf dem Entwicklungs-PC, sondern auf einem Board mit einem anderen Prozessor läuft, war unter Linux ursprünglich nicht vorgesehen. Doch inzwischen ist die Entwicklungsumgebung Eclipse so ausgereift, dass eine TCP/IP-Verbindung zum Zielsystem reicht, um alle Debug-Funktionen zu nutzen – und mit zusätzlichen Open-Source-Tools lassen sich sogar Speicherfehler aufspüren.

Von Thomas Brinker

In der Welt der betriebssystemlosen Entwicklung mit hersteller-spezifischen Tools ist das Remote-Debugging Standard. Ein Hardware-Debugger wird beispielsweise per JTAG an die Ziel-CPU angeschlossen und per USB an den Entwicklungs-PC. Dieser hat eine CPU-Hersteller-spezifische Entwicklungsumgebung installiert, die das Embedded-System mittels des Hardware-Debuggers „fernsteuert“. Damit sind beliebige Zugriffe auf den Speicher und die Prozessorregister möglich.

Mittels Schreibzugriffen auf Speicher und Register kann nun die Entwicklungsumgebung beliebige Programme auf der Ziel-CPU starten. Zusätzlich lassen sich eventuell vorhandene Debugging-Register nutzen, um den Programmablauf zu steuern. Aber auch ohne solche Register ist das Setzen von Breakpoints möglich. Es werden unterbrechende Prozessorbefehle eingefügt und bei Erreichen durch die Ziel-CPU von Hardware-

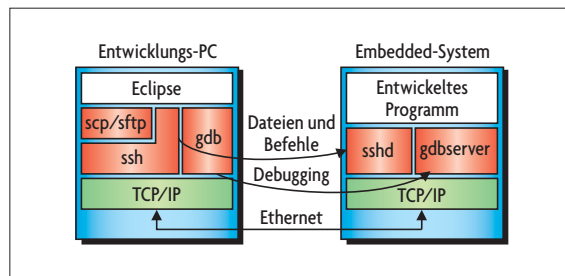


Bild 1. Für das Remote-Debugging unter Linux ist nur eine TCP/IP-Verbindung erforderlich. Der Entwickler arbeitet wie gewohnt mit dem Debugger gdb. Im Zielsystem ist der gdbserver installiert, der die Debugbefehle vom Entwicklungs-PC entgegennimmt und verarbeitet.

Debuggern erkannt und entsprechend behandelt.

Durch die nahtlose Integration der Debugging-Funktion in den Editor und ergänzt um den Compiler und andere Tools entsteht dann eine sehr komfortable Entwicklungsumgebung. Brüche durch den Wechsel zu einem anderen Tool oder durch unnötige Standardaufgaben wie etwa das Kopieren von Dateien oder Beschreiben von Flash-Spei-

chern werden vermieden. Beispiele für solche hochintegrierten Entwicklungsumgebungen (IDE) sind Freescales Codewarrior oder VisualDSP von ADI.

Entwicklungsland Linux

In der Entwicklung von Applikations-Software für Embedded-Linux-Systeme sah es lange Zeit ganz anders aus. Compiler, Editor und Debugger waren nur lose oder gar nicht miteinander gekoppelt. Zusätzlich waren oft einige Schritte Handarbeit erforderlich, um ein Programm auf der Ziel-CPU

starten zu können. Durch Netzlaufwerke, die vom Entwicklungsrechner auf das Embedded-System eingebunden werden können, kann zwar eine sehr kurze Turn-around-Zeit in der Test-Compile-Phase erreicht werden, dann jedoch steht noch kein „echter“ Debugger zur Verfügung und der Ent-

wickler ist auf Trace-Meldungen (siehe **Kasten** „Trace-Meldungen oder printf()-Debugging“) aus dem Programm angewiesen.

Debugging von Programmen in einem Embedded-Linux-System unter Einsatz eines Hardware-Debuggers ist wegen der strikten Trennung zwischen virtuellen und physikalischen Adressen von erheblicher Komplexität und wird daher praktisch nicht eingesetzt.

Applikations-Programme für Linux werden mit dem GNU-Debugger, kurz gdb, debuggt. gdb ist der Debugger in der Linux- und Unix-Welt schlechthin. Seine Entwicklung ist eng an die des Standard-Compilers gcc gekoppelt. Der gdb wird typischerweise eingesetzt, wenn es um das Debugging von PC-Software geht. Dabei wird fast immer auf ein grafisches Front-End gesetzt, das die Bedienung des gdb vereinfacht.

Trace-Meldungen oder printf()-Debugging?

Das so genannte printf()-Debugging genießt keinen guten Ruf. Eigentlich zu Unrecht, denn alleine die Überlegung, wo welche Meldung mit welchen Daten am sinnvollsten ist, hat schon so manchen Entwickler das Problem erkennen lassen. Dennoch liegen die Nachteile klar auf der Hand: Jede Modifikation an den Trace-Meldungen erfordert ein Recompile und

einen Neustart des Programms. Außerdem haben die Meldungen massiven Einfluss auf die Laufzeit des Programms. Traces können oft eine unbeherrschbare Größe von tausenden von Zeilen erreichen. Die Bezeichnung printf()-Debugging leitet sich aus der Standard-C-Funktion printf() her, die oft für die Ausgabe der Trace-Meldungen genutzt wird.

■ Remote-Debugging mit dem gdbserver

Der gdb ist aber genauso flexibel für Embedded-Systeme einsetzbar. Dazu enthält er den *gdbserver*, welcher als Stub das Debugging eines Programms auf einem anderen Rechner ermöglicht. Der *gdbserver* übernimmt alle Aufgaben des Debuggers auf der Embedded-Hardware, während der Entwickler mit dem gdb auf dem Entwicklungsrechner das eigentliche Debugging durchführt. Dazu wird zunächst der *gdbserver* auf dem Embedded-System gestartet. Als Parameter werden das zu debuggende Programm und ein TCP-Port angegeben:

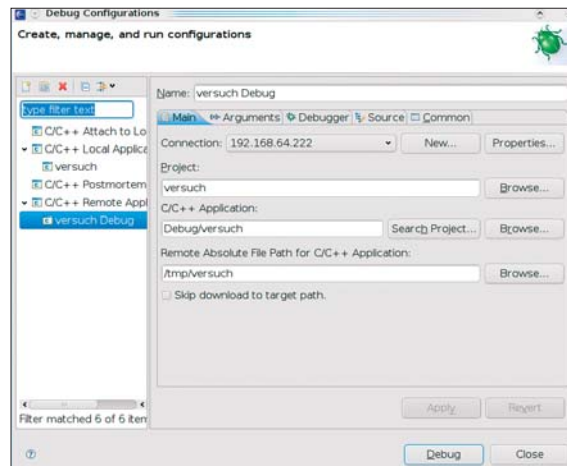
```
gdbserver :7777 testapp
```

Unter der IP-Adresse der Ziel-CPU und dem konfigurierten TCP-Port kann dann der eigentliche gdb vom Entwicklungsrechner Kontakt mit dem *gdbserver* aufnehmen und diesem Befehle zum Debuggen des Programms geben (Bild 1). Der *gdbserver* fügt dann beispielsweise Breakpoints ein und meldet deren Erreichen per TCP/IP an den Entwicklungsrechner. Das Debugging der Applikations-Software auf der Ziel-CPU ist damit genauso möglich, als würde nur auf dem Host-PC gearbeitet.

Das Protokoll, welches zwischen gdb und dem *gdbserver* genutzt wird, definiert alle erforderlichen Befehle und Datenformate für das Debugging. Es definiert keine Befehle für das Starten des Programms, die Parametrierung oder Ausgabe von Textmeldungen. Ebenso sieht es keine Möglichkeit vor, den eigentlichen Programmcode zu übertragen. Diese Aufgaben bleiben einer höheren Protokollschicht überlassen. Oft wurden sie jedoch manuell vom Entwickler für jeden Test-Compile-Zyklus durchgeführt. Dieses Vorgehen hat sich als fehleranfällig und damit als sehr frustrierend herausgestellt.

■ Eclipse bringt Komfort für Linux-Entwickler

Seit der Version Ganymede von Eclipse, einer Entwicklungsumgebung von IBM unter anderem für Java, C- und C++-Programme, wird das komplette Handling rund um den *gdbserver* von Eclipse übernommen und damit automatisiert. Tatsächlich übernehmen der



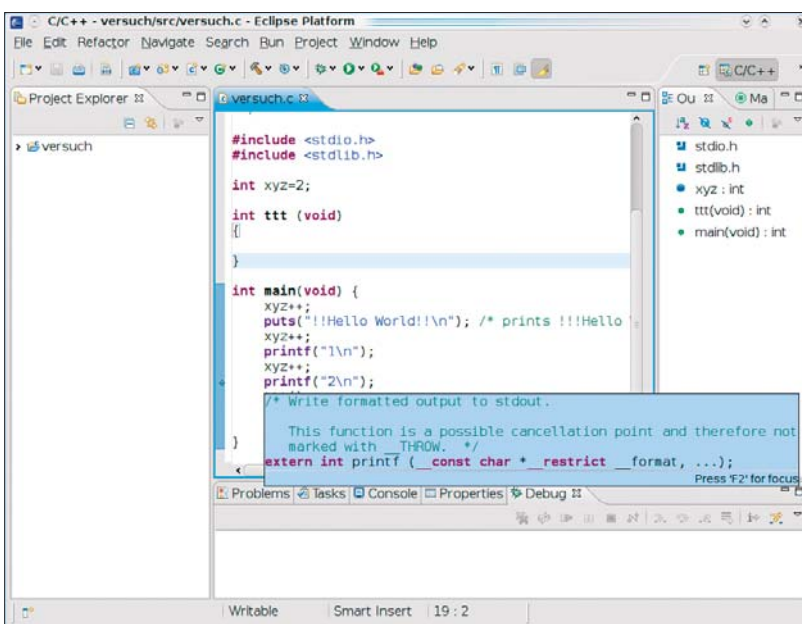
! Bild 3. Eine Debugging-Sitzung wird gestartet. Die Ziel-Hardware wird durch die IP-Adresse des Boards ausgewählt.

Remote-System-Explorer und das Debugging-Plugin von Eclipse die eigentliche Arbeit beim Remote-Debugging. Es muss also sichergestellt werden, dass diese Plugins erfolgreich installiert wurden.

Eclipse ist eine sehr vielfältige Entwicklungsumgebung, die weit über die Funktionen eines Editors mit Syntax-Highlighting hinausgeht. Eclipse wurde ursprünglich von IBM zur Entwicklung von Software auf Basis von Suns Java entwickelt. Das erklärt zum einen den Namen (Sun = Sonne; Eclipse = Sonnenfinsternis) und zum anderen, warum Eclipse selbst in Java geschrieben wurde. Eclipse kann durch Plugins erweitert und modifiziert werden. Tatsächlich ist es sogar so, dass Plugins erforderlich sind, um überhaupt C/C++-Programme entwickeln zu können.

Mit Hilfe der Plugins kann Eclipse automatisch Makefiles erzeugen, die die erforderliche Cross-Compilation für Embedded-Systeme sicher beherrschen. Komfortabel lässt sich die Entwicklung mit den Cross-Referenzierungs-Funktionen beschleunigen. Der Entwickler kann direkt vom Funktionsaufruf zur Deklaration der Funktion springen oder auch umgekehrt. Eclipse hält dazu eine Datenbank aller Funktionen und Variablen vor (Bild 2).

Bei Neuanlage einer Source-Code-Datei berücksichtigt Eclipse diese im Bauprozess, integriert sie in das Makefile und sorgt dafür, dass der Compiler den Code übersetzt. Der nachfolgende Linker integriert dann



! Bild 2. Eclipse gibt Informationen zu einem Funktionsinterface aus. Eclipse hat eine Datenbank aller Funktionen und Variablen.

den Code in eine Library oder ein Executable. Der gesamte Bauprozess kann komfortabel per Mausklick von Eclipse durchgeführt werden. Treten Fehler auf, so zeigt Eclipse diese direkt im Code-Editor mit Korrekturzeichen an.

Zum Debugging kann Eclipse auf Knopfdruck ein entsprechendes Binärprogramm erstellen lassen und hiermit auf dem Target, also auf der Embedded-Hardware eine Debug-

ging-Sitzung starten (Bild 3), dass auch hierin Programme zum Debugging abgelegt werden können.

■ Verbindung zum Zielsystem über ssh

Mit dem Hauptprogramm aus der ssh-Suite, ssh, baut Eclipse nach dem Kopieren der Datei erneut eine Terminal-Verbindung auf, über die Befehle per Textzeile übermittelt werden. Innerhalb dieser Session wird dann der gdb-

sind JTAG-Adapter oder ähnliches verzichtbar. Gelegentlich haben Embedded-Systeme kein Ethernet. In diesem Fall kann auf das Netzwerk via USB zurückgegriffen werden.

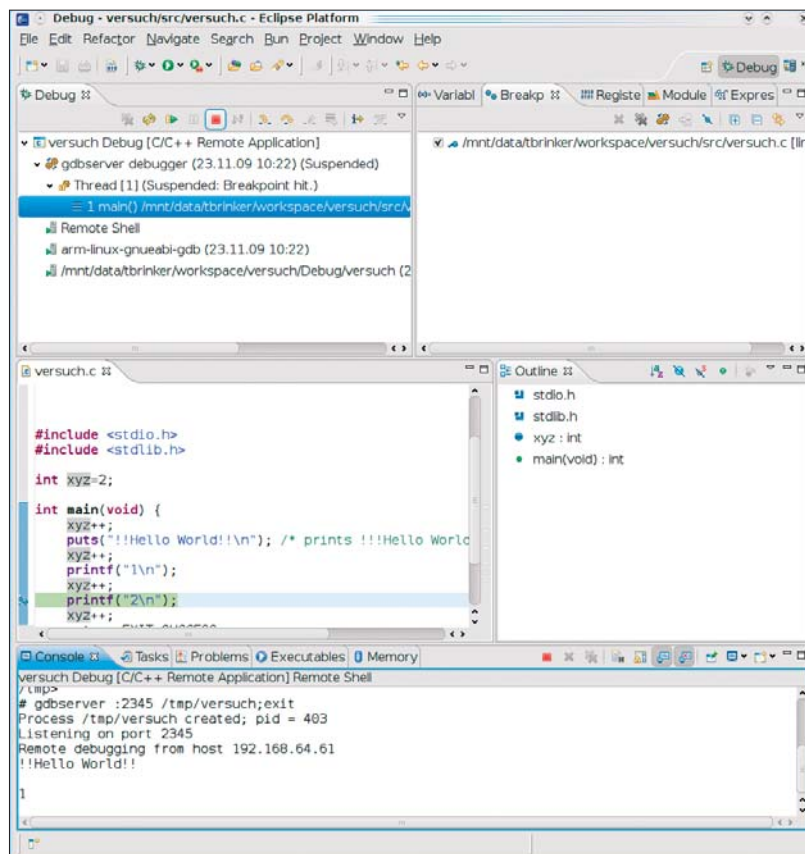
Damit scp und ssh mit dem Embedded-Board funktionieren, ist ein entsprechender Dienst auf dem Embedded-Linux-System erforderlich. Der Standard-SSH-Server *OpenSSH* kann auf vielen Embedded-Linux-Systemen problemlos integriert werden, sofern er nicht sowieso schon Bestandteil des Standardumfangs ist. Für kleinere, etwa MMU-lose Systeme gibt es mit dem ssh-Server *Dropbear* auch eine leichtgewichtige Alternative.

Das erforderliche Passwort bei der Authentifizierung ohne Public/Private Key kann komfortabel, aber unsicher in Eclipse gespeichert werden. Das nimmt das lästige Eintippen des Passwortes ab, öffnet aber ein Einfallstor für mögliche Angreifer. Es empfiehlt sich also, vor der Auslieferung des Embedded-Linux-Systems die Passwörter wenigstens zu ändern oder besser die Authentifizierung per Benutzername/Passwort zu deaktivieren oder – wenn möglich – den ssh-Server ganz zu entfernen.

■ Schritt für Schritt oder „Step-over“

Während der Debugging-Sitzung mit Eclipse kann der Entwickler im Programmcode zeilengenau Breakpoints setzen oder von Zeile zu Zeile springen (Bild 4). Der Programmverlauf kann so „live“ mitverfolgt werden. Der Wert jeder Variablen lässt sich betrachten und analysieren. Funktionsaufrufe können ebenso analysiert oder die Funktionen schnell ohne Debugging durchschritten werden. Dieses sogenannte Step-over empfiehlt sich immer dann, wenn es sich um Funktionsaufrufe in Libraries handelt, die nicht debuggt werden sollen oder können. Oftmals stehen für diese Libraries auch keine Debugging-Symbole zur Verfügung, sodass sinnhaftes Debugging hier nicht ohne weiteres möglich ist.

Bei populären Libraries etwa aus der Open-Source-Welt ist die Wahrscheinlichkeit für einen Bug in der Library eher gering, aber natürlich nicht ausgeschlossen. Die Erfahrung hat ge-



! Bild 4. Ein Breakpoint wurde erreicht. Die Programmausgaben stehen im untersten Fenster.

server gestartet, der seinerseits in der Debugging-Umgebung das Programm ausführt. Über die ssh-Shell kann Eclipse das zu debuggende Programm mit Aufrufparametern und Umgebungsvariablen versorgen. Die Ausgaben auf dieser Shell können in einem Eclipse-Fenster eingesehen werden, sodass beispielsweise Trace-Meldungen aus dem Programm nicht verlorengehen.

Alle erforderlichen und oben beschriebenen Protokolle setzen auf TCP/IP auf. Für das Debugging ist also lediglich eine Netzwerkverbindung erforderlich. Auf die serielle Schnittstelle kann verzichtet werden. Ebenso

server gestartet, der seinerseits in der Debugging-Umgebung das Programm ausführt. Über die ssh-Shell kann Eclipse das zu debuggende Programm mit Aufrufparametern und Umgebungsvariablen versorgen. Die Ausgaben auf dieser Shell können in einem Eclipse-Fenster eingesehen werden, sodass beispielsweise Trace-Meldungen aus dem Programm nicht verlorengehen.

Alle erforderlichen und oben beschriebenen Protokolle setzen auf TCP/IP auf. Für das Debugging ist also lediglich eine Netzwerkverbindung erforderlich. Auf die serielle Schnittstelle kann verzichtet werden. Ebenso

```

versuch.c
Copyright : Your copyright notice
Description : Hello World in C, Ansi-style
*/

#include <stdio.h>
#include <stdlib.h>

int xyz=2;
int main(void) {
    xyz++;
    puts("!!!Hello World!!!\n"); /* prints !!!Hello World!!!
    xyz++;
    printf("\n");
    xyz++;
    printf("2\n");
    xyz++;
}

```

Bild 5. Bei der Analyse von Library-Funktionen ist es sinnvoll, die Korrektheit der Aufrufparameter zu überprüfen. Eclipse gibt dazu den aktuellen Variableninhalt aus.

zeigt, dass bei Fehlern, die innerhalb von Libraries auftreten, oftmals falsch initialisierte Datenstrukturen oder unbeabsichtigte Modifikationen am Stack oder Code die Ursache sind. Zur Analyse übergebener Strukturen ist die Analyse der Aufrufparameter mit Eclipse komfortabel möglich (Bild 5). Die eigentliche Library-Funktion sollte dann per Step-over ohne Unterbrechung durchschritten werden.

Der Entwickler kann zu jedem Zeitpunkt die Debugging-Sitzung abbrechen und den Quellcode editieren. Nur sehr wenige Klicks oder Tastenkürzel sind erforderlich, um das modifizierte Programm neu zu übersetzen, auf das Zielsystem zu kopieren und dort eine neue Debugging-Sitzung zu beginnen.

Der Durchlauf der Compile-Test-Zyklen ist damit optimal kurz, und der Entwickler kann sich voll auf den Code konzentrieren und diesen systematisch prüfen oder nach Programmfehlern (Bugs) suchen. Bei massiv falschen Speicherzugriffen unterbricht der Debugger sofort, nicht jedoch bei nur geringfügigem Abweichen vom legalen Speicherbereich. Diese unzulässigen Speichermodifikationen sind eine Klasse von Fehlern, die nicht selten Grund für schwere Programmfehler sind.

■ Speicherfehler entdecken mit DUMA und valgrind

Die unzulässige Modifikation (Schreibbefehl) bleibt dabei zunächst unentdeckt, da sie noch kein unmittelbares Fehlverhalten bedeutet. Wird jedoch später wieder von der betroffenen Speicherstelle gelesen, so führt der dann

vorgefundene und unerwartete Wert typischerweise zu einem schweren Fehler. Es ist nun relativ leicht, das Lesen des falschen Wertes zu analysieren. Der eigentliche Fehler aber, also das Schreiben an die falsche Stelle, kann jedoch nicht immer mit den Mitteln von Eclipse und gdb analysiert werden.

Unterstützung bei Problemen dieser Klasse erfährt der Entwickler durch Libraries wie DUMA (Detect Unintended Memory Access) oder durch die Nutzung von Hostside-Debuggern wie *valgrind*.

DUMA umklammert jeden angeforderten Speicherbereich so, dass jeder Zugriff außerhalb des angeforderten Bereiches zwangsläufig und sofort zu einem Abbruch führt. Diese Abbrüche lassen sich dann gut mit Eclipse und gdb analysieren und damit schnell auflösen. Zur Nutzung von

DUMA muss der Entwickler lediglich mittels Linker-Befehl die Library einbauen.

```
arm-linux-gnueabi-gcc -g -o memerror memerror.c -lduma
```

Der Speicher-Debugger „valgrind“ steht typischerweise nicht auf Embedded-Linux-Systemen zur Verfügung. Ausnahmen bilden hier nur die Power-Architektur und x86-Systeme. Dennoch kann Software, die etwa auf einem ARM eingesetzt werden soll, auch mit Hostside-Debuggern analysiert werden. Es muss die Software aber mit dem Host-Compiler des Entwicklungsrechners übersetzt werden. Danach kann das zu prüfende Programm mit valgrind automatisch instrumentiert und gestartet werden. Die Instrumentierung der Software erfolgt nach der Übersetzung, unmittelbar vor dem Start. Spezielle Compiler-Schalter sind also nicht erforderlich. Dennoch ist es hilfreich, mit dem Compiler-Schalter „-g“ zu übersetzen, sodass valgrind in der Lage ist, die Quelldatei und die Code-Zeile auszugeben. Diese Angaben vereinfachen das Auffinden von

Debugging mit valgrind

Im Folgenden sind ein fehlerhaftes Programm und die Resultate von „valgrind“ zu sehen:

```
#include <stdlib.h>
int main (int argc, char** argv)
{
    char * ptr;
    ptr = (char*) malloc(10);
    ptr[10] = 0;
}
```

Aus diesem Quellcode generiert valgrind die folgende Ausgabe:

```
==29824== Invalid write of size 1
==29824== at 0x80483EA: main (bug.c:28)
==29824== Address 0x41a5032 is 0 bytes after a block
of size 10 alloc'd
==29824== at 0x4026FDE: malloc (vg_replace_malloc.c:207)
==29824== by 0x80483E0: main (bug.c:26)
...
==29825== 10 bytes in 1 blocks are definitely lost
in loss record 1 of 1
==29825== at 0x4026FDE: malloc (vg_replace_malloc.c:207)
==29825== by 0x80483E0: main (bug.c:26)
```

„valgrind“ erkennt nicht nur das Überschreiten des Speichers um ein Byte. Es gibt auch detaillierte Informationen über das Speicherleck.

Problemen im Quellcode. Zusätzlich sollte auf Optimierungen mit „-O1“, „-O2“ oder „-O3“ verzichtet werden.

Nach Erzeugung des binären Programms wird valgrind wie folgt gestartet:

```
valgrind -leak-check=full testapp
```

Durch die Instrumentierung analysiert valgrind jede Operation des Programms und zeigt Fehler und potentielle Probleme an, wie etwa die Nutzung uninitialisierten Speichers, Fehlzugriffe in ungültige Speicherbereiche und Speicherlecks (siehe **Kasten** „Debugging mit valgrind“). Grundsätzlich empfiehlt es sich, jede Software mit valgrind zu überprüfen, um die Qualität zu erhöhen. Die Instrumentierung durch valgrind verlangsamt die Ausführung in ganz erheblichem Maße, sodass das Debugging mit valgrind durchaus eine Geduldsprobe darstellen kann. Eine Investition, die sich jedoch fast immer lohnt.

▣ Studieren geht über Probieren

Die Werkzeuge gdb, Eclipse, DUMA und valgrind sind hilfreiche und

mächtige Debugging-Tools, die die Entwicklung von Software für Embedded-Linux-Systeme erheblich vereinfachen. Die intensive Nutzung von Debuggern führt schnell zu funktionierender Software, ist aber nicht ganz unkritisch. Es besteht die Gefahr, dass der Entwickler sich nur auf seinen Debugger und seine eigenen Tests verlässt und den Blick und das Verständnis für das Umfeld, in dem seine Ergebnisse später Verwendung finden, vernachlässigt. Insbesondere bei komplizierter Algorithmik kann ein Debugger dazu verleiten, eine Lösung für ein Problem nur durch Herumprobieren zu finden. Ein Durchdringen des Algorithmus ist hierfür nicht erforderlich. Entsprechend gering ist die Qualität des dadurch entstehenden Codes.

Ein Debugger kann die Lösung eines Problems in einem Programm beschleunigen, weil die fehlerhafte Stelle schneller lokalisiert wird. Bei der eigentlichen Ausarbeitung der Lösung ist ein Debugger jedoch wenig hilfreich. Hierzu muss der Entwickler den Code sowie den Grund des Fehlers genau verstehen und dann eine saubere Lösung erarbeiten. Mit einem Schnellschuss, der im Debugger keine Fehler mehr zeigt, ist das Problem oftmals

nur verschoben und tritt später möglicherweise verschärft wieder auf. Vollständige Testszenarien und ein Code-Review des Fixes mit Kollegen oder bei Open-Source-Software mit der Community steigern die Qualität des Fixes erheblich und sollten daher obligatorisch sein.

Die Nutzung von Debuggern direkt aus den Source-Code heraus ist mit den Software-Paketen Eclipse, ssh und gdb für Embedded-Linux-Entwickler erheblich vereinfacht worden. Der Arbeitsablauf ist damit der gleiche wie innerhalb einer herstellerspezifischen Entwicklungsumgebung mit Hardware-Debugger. Einsparungen sind jedoch auf der Seite des Linux-Entwicklers möglich, da dieser keinen teuren Hardware-Debugger und keine kostenpflichtigen Software-Lizenzen braucht. Der Setup von Eclipse und die erforderliche Konfiguration von Toolchain, Debugger und ssh-Verbindung sind von nicht geringer Komplexität, brauchen jedoch nur einmalig durchgeführt zu werden. Embedded Linux Board Support Packages von emlix in der Professional Edition enthalten immer ein konfiguriertes Eclipse und beschreiben die erforderlichen Schritte für das Remote-Debugging detailliert. *jk*



**Dipl.-Ing.
Thomas Brinker**

studierte an der Technischen Universität Berlin Technische Informatik. Seit 2005 ist er bei der emlix GmbH in Göttingen als Embedded Linux Systemengineer beschäftigt und dort an der hardwarenahen Entwicklung von Produkten für die Medizintechnik, Automatisierungstechnik und Datentechnik beteiligt. Er arbeitet im Bereich der Kernel- und Treiberentwicklung und leitet seit 2006 die Emlix-Niederlassung in Berlin.

tb@emlix.com