14 Einführung: Debugging

Selten funktioniert ein Programm auf Anhieb wie gewünscht. Während sogenannte *syntaktische Fehler* – hierunter versteht man Fehler, die beispielsweise auf einfachen Tippfehlern, unzulässigen oder falsch angewendeten Parametern, nicht-deklarierten bzw. undefinierten Variablen und Parametern, fehlerhafte Anwendung von Pointern oder anderen Verstößen gegen die Regeln der verwendeten Programmiersprache beruhen – bereits durch den Compiler erkannt werden und sich somit recht einfach beheben lassen, existiert mit den *semantischen Fehlern* eine andere Sorte von Fehlern, die häufig wesentlich schwerer zu entdecken ist. Semantische Fehler verstoßen nicht gegen die Regeln der Programmiersprache und werden daher auch nicht vom Compiler erkannt: Das Ergebnis eines Programms bzw. einer Funktion entspricht aber nicht dem erwarteten und erwünschten Ergebnis.

Die Suche nach semantischen Fehlern und deren Behebung wird als *Debugging* bezeichnet. Es gibt verschiedene Methoden, die sich hierfür einsetzen lassen:

- Die Methode des »scharfen Hinsehens« setzt voraus, dass der Quelltext des Programms verfügbar ist. Bei selbstentwickelter Software kann diese Methode daher im Allgemeinen ohne weiteres eingesetzt werden. Sie stellt die einfachste Möglichkeit des Debuggings dar. Häufig stellt sich aber heraus, dass die Fehlersuche und –korrektur sehr ineffizient ist und auf das Trial- und Error-Prinzip hinausläuft.
- Das Schreiben von Testprogrammen stellt eine andere Möglichkeit dar, Fehler von Beginn an zu vermeiden. Hierfür werden im Rahmen des sogenannten *Unit Testing* Programme geschrieben, die Teilbereiche eines Programms als eigenständige Einheit betrachten und ausführen. Die Testprogramme treffen begründete Annahmen darüber, wie das Ergebnis einer solchen Einheit auszusehen hat. Die Testprogramme sollten immer geschrieben werden, bevor die gewünschte Funktionalität implementiert wird. Diese Vorgehensweise hat den Vorteil, dass man sich gedanklich schon während der Planung einer Funktion/eines Programms damit beschäftigt, was die spätere Implementierung der Funktionen und Algorithmen erheblich erleichtert. Unit Tests prüfen immer mehrere Annahmen, z.B. durch die Verwendung verschiedener Argumente und Prüfwerte.
- Die Ausgabe der Ergebnisse von Zwischenschritten stellt eine weitere Möglichkeit zur Fehlersuche bzw. deren Eingrenzung dar. In den Kapiteln zur Treiberentwicklung haben Sie diese Möglichkeit – obwohl sie dort nicht ausdrücklich als Debugging-Methode bezeichnet wurde – bereits selber sehr oft eingesetzt: Die Ausgabe von Zwischenergebnissen erfolgte mit Funktionen wie `printk`, `pr_err` usw. Diese Vorgehensweise ist sehr einfach anwendbar, hat aber

mehrere gravierende Nachteile:

- Jeder Test, selbst wenn nur ein anderer Wert geprüft werden soll, setzt voraus, dass das Programm neu kompiliert, gelinkt und an das Zielsystem übertragen werden muss.
- Vergisst man, die Debugging-Ausgaben nach abgeschlossener Fehlerbehebung aus dem Quelltext zu entfernen, so werden die Ergebnisse auch bei der »fertigen« Software angezeigt. Dieser Nachteil kann aber leicht vermieden werden, wenn alle Ausgaben dieser Art in ein `#ifdef`-/`#endif`-Konstrukt gehüllt werden. Hierfür reicht es aus, an einer Stelle im Programm – vorzugsweise unmittelbar vor der `main`-Funktion – mit `#define DEBUG` ein Makro einzusetzen, das im letzten Schritt vor der Fertigstellung der Software auskommentiert wird. Natürlich setzt dies voraus, dass die Software noch einmal abschließend kompiliert wird.

- Der Einsatz spezieller Software, sogenannter *Software-Debugger*, stellt die meiner Meinung nach eleganteste Methode zur Fehlersuche in modernen Embedded Systemen dar. Hierbei handelt es sich um Programme, die es ermöglichen, das zu testende Programm durch das Setzen sogenannter *Breakpoints* an bestimmten Stellen anzuhalten und von diesem Punkt aus jedes einzelne Kommando und seine Auswirkung schrittweise im *Einzelschrittverfahren* (*Single-Step-Debugging*) fortzusetzen. Ruft ein Programmteil eine bereits getestete Unterfunktion auf, so bieten Software-Debugger die Möglichkeit, diese als Einzelschritt auszuführen und mit dem Ergebnis schrittweise weiterzuarbeiten. Software-Debugger ermöglichen es darüber hinaus auch, den Inhalt einzelner Speicherzellen, ganzer Speicherbereiche oder Register des Mikroprozessors/Mikrocontrollers zu untersuchen.
- Sehr komfortabel ist der Einsatz sogenannter *Hardware-Debugger* bzw. *In-Circuit-Emulatoren* (*ICE*). Große Beliebtheit haben hier die Debugger/Emulatoren der Firmen Segger und Lauterbach. Das Problem: Die Anschaffung ist im Regelfall mit Kosten von 1000 bis 2000 Euro verbunden. Zumindest von der Firma Segger ist mir bekannt, dass sie für ca. 50 Euro einen *JTAG*-Emulator für Schulungs- und Ausbildungszwecke anbietet. Beim Raspberry Pi ist hierfür aber die Konfektionierung eines Spezialkabels erforderlich, da die JTAG-Anschlüsse des Boards nicht zusammenhängend auf die GPIO-Leiste herausgeführt sind. JTAG ist übrigens die Abkürzung für *Joint Test Action Group*. Der Einsatz solcher In-Circuit-Emulatoren stellt sich bei komplexen modernen Mikrocontrollern, auf denen ein Betriebssystem (z.B. Embedded Linux) ausgeführt wird, allerdings als nicht einfach dar! *Virtuelle Speicheradressierung*, unterschiedlichste *Caching-Algorithmen* sowie das sogenannte *Pipelining* erfordern sehr spezielle Kenntnisse der verwendeten Prozessor-Architektur. Bei Embedded Systemen ohne Betriebssystem ist der Einsatz von ICE deutlich leichter, da diese vornehmlich nur eine einzige Aufgabe erfüllen sollen und auf denen vor allem keine anderen Programme (z.B. Office-Anwendungen) ausgeführt werden.

14.1 Software-Debugging

Der Einsatz von Software-Debuggern unter Embedded Linux ist besonders preiswert:

Die benötigte Software steht nämlich kostenlos zur Verfügung. Auf dem Zielsystem (also dem unter Embedded Linux arbeitenden Embedded System, dem Target) wird das mit `gdbserver` benannte Programm benötigt. Auf dem Entwicklungs-PC (Host) haben Sie bereits die Cross-Toolchain installiert: Bestandteil dieser Toolchain ist der Debugger `gdb`. Wenn Sie sich für die Toolchain von Linaro entschieden haben, dann lautet der vollständige Name des Debuggers dort `arm-linux-gnueabihf-gdb`.

14.1.1 Weitere Software

`gdb` und `gdbserver` sind Kommandozeilenwerkzeuge, die in einem Terminal ausgeführt werden. Dies funktioniert auch sehr gut, ist aber für den dauerhaften Einsatz wenig komfortabel. Man muss zwar nicht alle Funktionen verstehen bzw. anwenden, die `gdb` anbietet. Wenn man aber berücksichtigt, dass das offizielle Handbuch 788 Seiten stark ist, dann bevorzugt man sicher eine komfortablere Lösung – dies gilt zumindest für mich.

Komfortablere Lösungen existieren reichlich. Zwei Beispiele sind z.B. die Programme `ddd` (Data Display Debugger) und `Nemiver`. Während `ddd` etwas »altbacken« aussieht, wirkt `Nemiver` deutlich moderner. Damit Sie sich selber einen Eindruck von diesen beiden Debugger-GUIs machen können, sehen Sie in den Abbildungen 14.1 und 14.2 einen Screenshot dieser Programme.

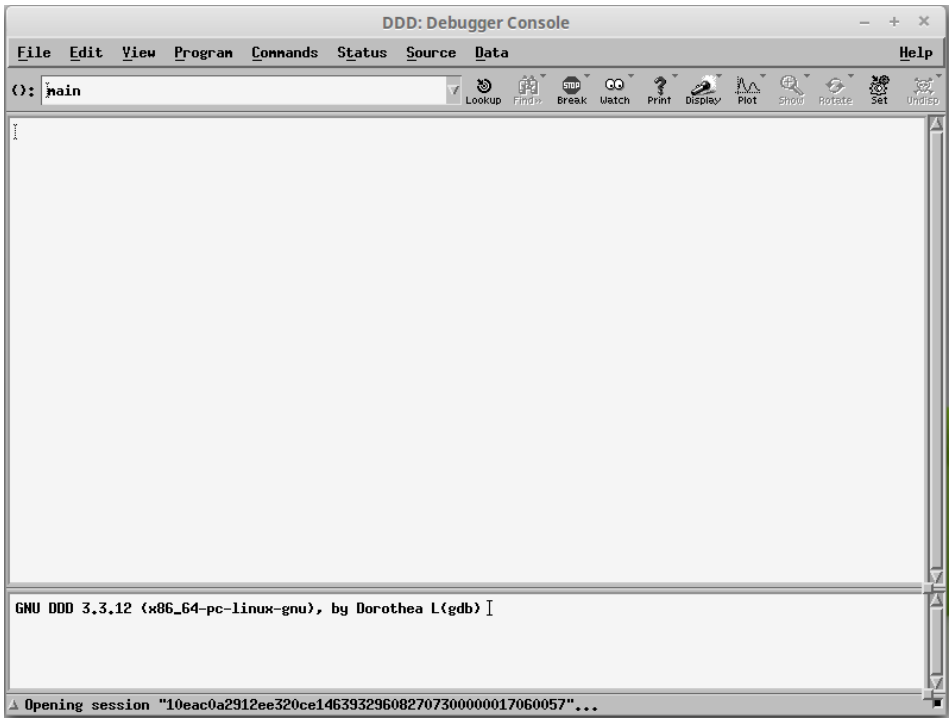

Abbildung 14.1: Startbildschirm des Data Display Debuggers ddd

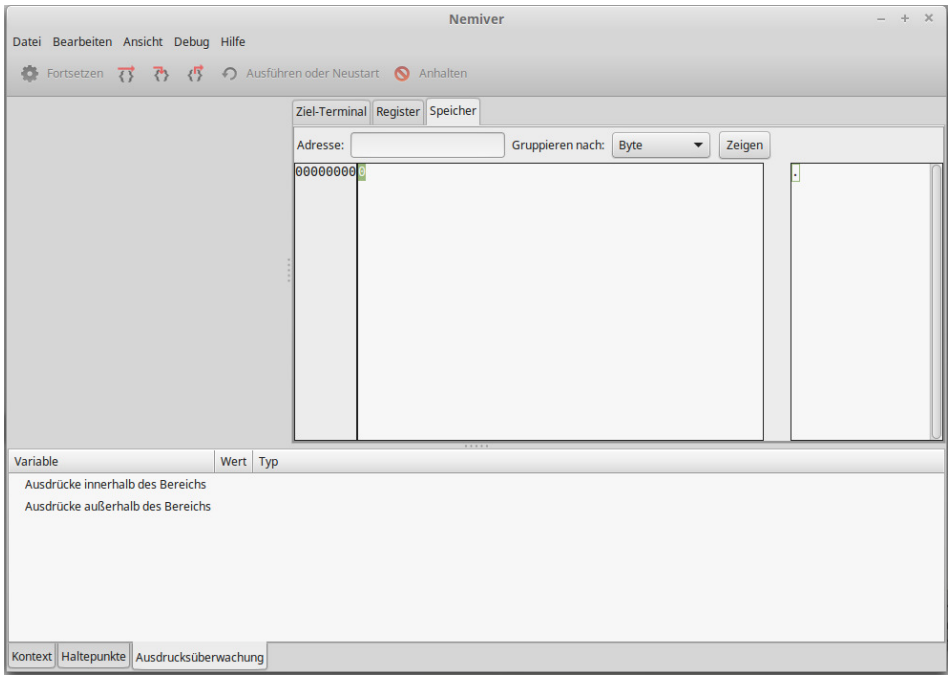

Abbildung 14.2: Startbildschirm von Nemiver

Drei Möglichkeiten, den Debugger einzusetzen, wurden bisher angesprochen: die reine Konsolenanwendung, der Data Display Debugger `ddd` und das Programm `Nemiver`.

Hier wird aber keine der drei Möglichkeiten verwendet: Da die Beispiele in diesem Buch unter Einsatz der Entwicklungsumgebung Code::Blocks entwickelt wurden, wird weiter unten gezeigt, wie `arm-linux-gnueabihf-gdb` unter Code::Blocks »ins Schwitzen« gebracht werden kann.

14.2 Verbindungsaufbau

Wenn der Debugging-Prozess vom Entwicklungs-PC (Host) ferngesteuert erfolgen soll – man nennt dies auch *Remote Debugging* – muss natürlich eine Verbindung zwischen dem Host und dem Target hergestellt werden. In den bisherigen Kapiteln war diese Verbindung bereits vorhanden, denn sonst hätten Sie niemals Programme oder Dateien per `scp` zwischen den beiden Systemen austauschen können. Abbildung 14.3 zeigt die zwei Möglichkeiten, die in der Praxis am häufigsten anzutreffen sind: eine Verbindung über Ethernet und eine Verbindung durch Einsatz von einer oder (optional) zwei seriellen Schnittstellen.

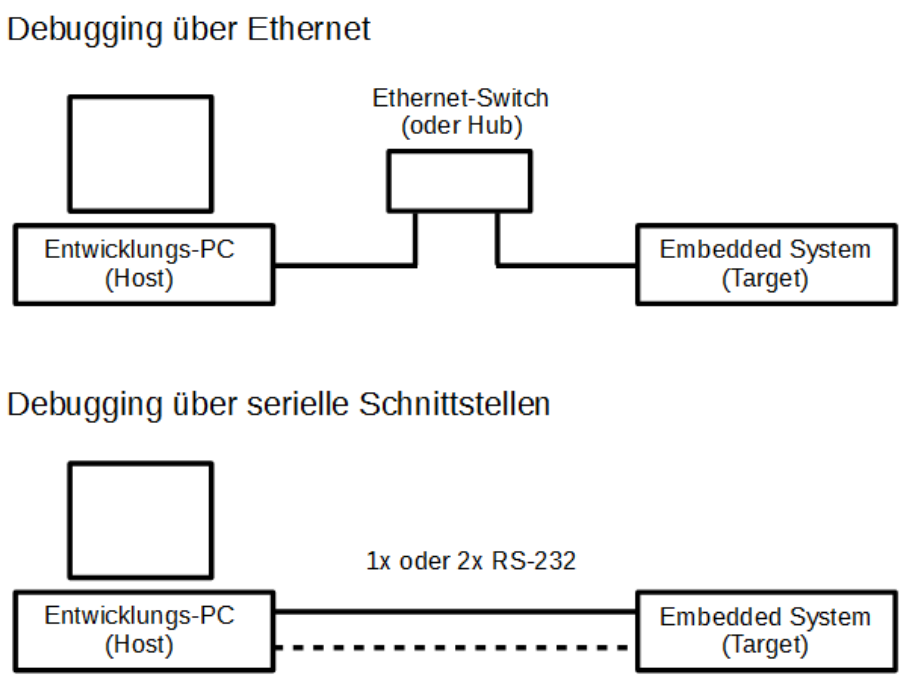

Abbildung 14.3: Zwei Möglichkeiten, eine Debuggerverbindung herzustellen

Hinweis

Ich habe bei der Entwicklung der Beispiele die Verbindung über Ethernet eingesetzt und aus diesem Grund eine serielle Verbindung selber nicht getestet.

Hinweis

Im unteren Teil von Abbildung 14.3 sehen Sie eine zweite gestrichelte Verbindung zwischen Host und Target. Diese Verbindung ist optional. Sie ermöglicht die Ausgabe der Antworten des Debug-Servers (`gdbserver`) an ein zusätzliches Terminalfenster auf dem Host. Dies sorgt für Übersichtlichkeit, da das `gdb`-Terminal dann nur zur Eingabe der Debugger-Kommandos verwendet wird, während das zweite Terminalfenster ausschließlich die Ergebnisse dieser Kommandos anzeigt.

Die Hardwarevoraussetzungen sind somit geschaffen. Der nächste Schritt besteht darin, `gdbserver` auf dem Target zu starten. Die Kommunikation zwischen Host und Target erfolgt über einen (nahezu) beliebigen *TCP-Socket*. Ein TCP-Socket setzt sich immer aus einer *IP-Adresse* und einem *Port* zusammen und setzt hierbei auf das *TCP-Protokoll*. Die Form ist immer `<IP-Adresse>:<Portnummer>`. Die Portnummer des Sockets kann weitestgehend frei gewählt werden. Theoretisch kann jeder Client 64k (also von Port 0 bis Port 65535) Verbindungen zur Außenwelt haben. Die ersten 1023 sind allerdings für bestimmte Aufgaben reserviert und sollten nicht anders verwendet werden, als dies spezifiziert ist. Zu den bekanntesten Ports zählen die Ports 21 (FTP, File Transfer Protocol), 22 (ssh, Secure Shell), 25 (SMTP, Simple Mail Transfer Protocol), 80 (http, Hypertext Transfer Protocol), 110 (POP3, Email-Protokoll, bei dem

Emails vom Mailserver heruntergeladen werden) oder 143 (IMAP, Email-Protokoll, bei dem die Nachrichten auf dem Server verbleiben).

Hinweis

Man liest häufig, dass die Zahl der nutzbaren Ports auf maximal 64k beschränkt ist. Diese Aussage wird davon abgeleitet, dass eine Portnummer durch einen 16-Bit-Wert dargestellt wird. Diese Aussage ist allerdings nicht ganz korrekt! Ohne auf Details eingehen zu wollen: Jedem Port wird ein sogenannter *File Descriptor* (eine bessere deutsche Übersetzung als Datei-Deskriptor ist mir nicht bekannt) zugeordnet. Für jede Datei, die geöffnet wird, stellt das Betriebssystem einen File Descriptor zur Verfügung, der ein einfacher numerischer Wert ist. Die Zahl der möglichen Datei-Deskriptoren ist aber weitaus größer als 64k. Entsprechend viel größer ist auch die Zahl der möglichen Ports. In der Praxis liegt die Zahl der für den Datenaustausch über ein Netzwerk genutzten Ports zwischen 80000 und 300000. In diesem Buch verwende ich für die Debugger-Verbindung den Port 2000.

Beim Cross-Debugging wird `gdbserver` immer auf dem Target ausgeführt, während der Cross-Debugger `gdb` immer auf dem Host ausgeführt wird.

`gdbserver` benötigt außerdem noch den Namen des Programms, dessen Funktion getestet werden soll. Wollen Sie beispielsweise die Funktion des Programms `power` (siehe weiter unten) auf dem Embedded System mit dem Remote-Debugger überprüfen, so geben Sie einfach Folgendes ein:

```
$ gdbserver localhost:2000 power
```

Hinweis

Eigentlich müsste ich an dieser Stelle beschreiben, auf welche Weise nun die Verbindung zwischen `gdb` und `gdbserver` hergestellt wird. Hierfür ist das `gdb`-Kommando `Target remote` zu verwenden. Ich verzichte aber auf die Beschreibung, weil ich – wie bereits weiter oben gesagt – aus Gründen der Bequemlichkeit das Debuggen unter Code::Blocks durchführen werde.

14.3 Der GNU Debugger unter Code::Blocks

In den folgenden Abschnitten wird zunächst die Konfiguration von Code::Blocks beschrieben, die allerdings sehr leicht vonstattengeht.

Debugger-Einstellungen durchführen

Nach dem Start von Code::Blocks klicken Sie, wie in Abbildung 14.4 gezeigt, in der Menüleiste auf SETTINGS und anschließend auf DEBUGGER.

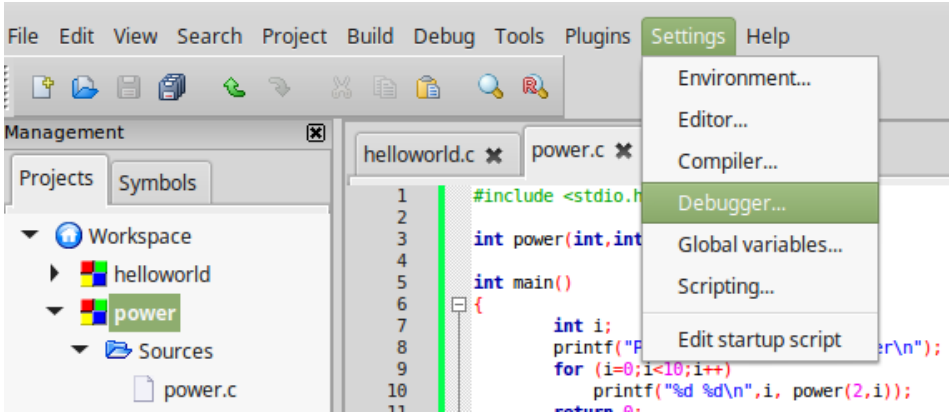

Abbildung 14.4: Konfiguration des Debuggers unter Code::Blocks

Hierauf öffnet sich der in Abbildung 14.5 gezeigte Dialog. Hier können Sie Einstellungen vornehmen, die gleichermaßen für alle Targets/Ziele gültig sind. Die Standardeinstellungen sind bereits gut gewählt. Dies soll Sie aber nicht davon abhalten, ein wenig mit den sehr überschaubaren Einstelloptionen »zu spielen«. Die Standardeinstellungen sind in Abbildung 14.5 zu erkennen.

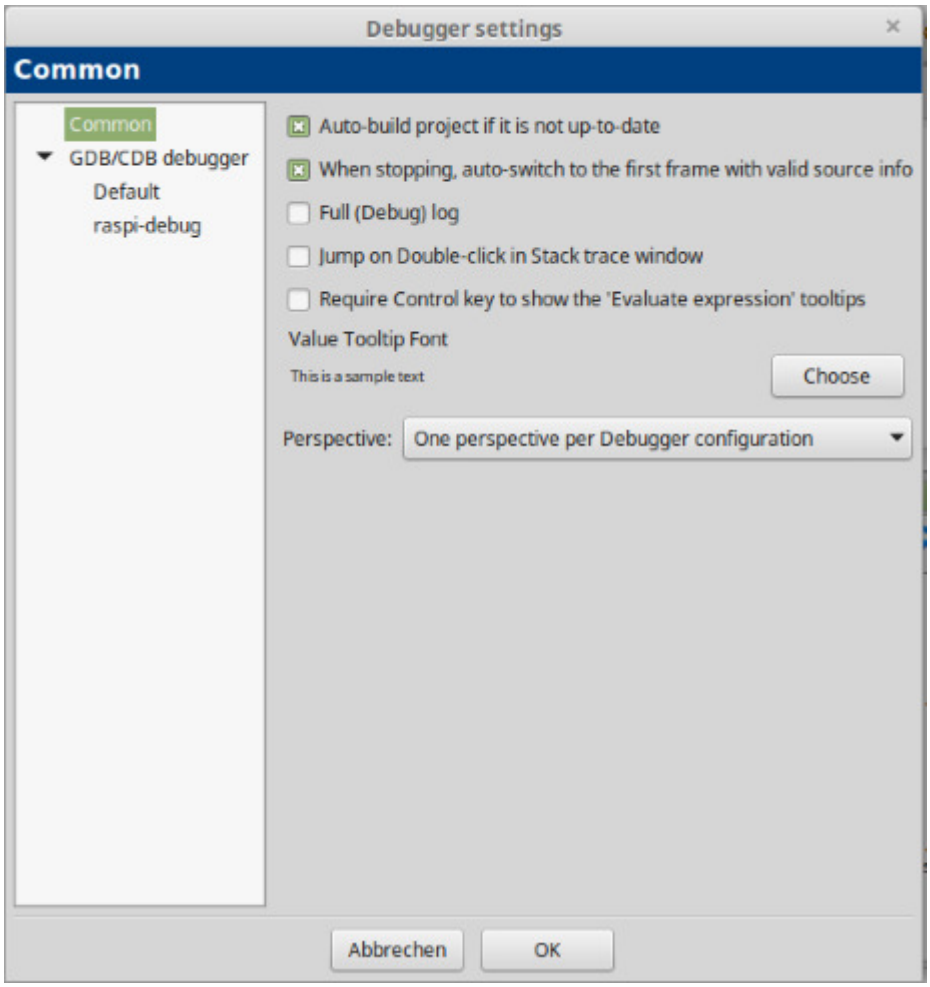

Abbildung 14.5: Standardeinstellungen, die für alle Konfigurationen gültig sind

In der linken Hälfte des Dialogs sehen Sie auch den Eintrag GDB/CDB DEBUGGER. Die in der Abbildung angezeigte Konfiguration `raspi-debug` existiert zu diesem Zeitpunkt noch nicht: Diese muss erst von Ihnen erzeugt werden. Wenn Sie GDB/CDB DEBUGGER angeklickt haben, sehen Sie Schaltflächen CREATE CONFIG, DELETE CONFIG und RESET DEFAULTS.

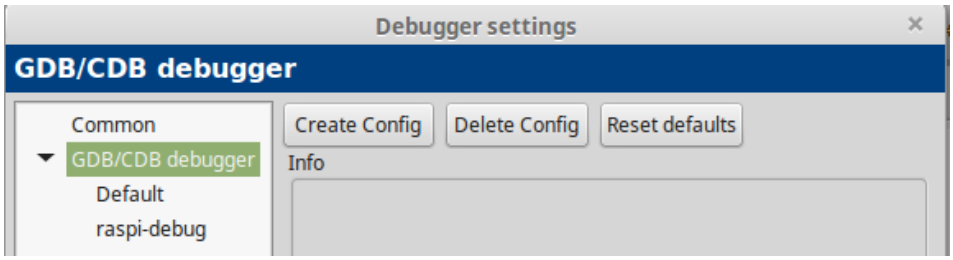

Abbildung 14.6: Erzeugen einer neuen Debugger-Konfiguration

Durch das Anklicken der Schaltfläche CREATE CONFIG können Sie eine neue Konfiguration anlegen, mit DELETE CONFIG eine bestehende Konfiguration löschen und mit RESET DEFAULTS die Standardeinstellungen der ausgewählten Konfiguration wieder herstellen.

Wenn Sie nun CREATE CONFIG anklicken, öffnet sich ein – hier nicht gezeigter – Dialog, der Sie auffordert, einen Namen für die neue Konfiguration einzugeben.

Hinweis

Ich empfehle Ihnen, mitgelieferte Beispielkonfigurationen eines Programmes nicht zu ändern, und stattdessen eine neue Konfiguration zu erstellen. Dies gilt für die meisten Programme, die professionell einsetzbar sind. Obwohl Code::Blocks mit der Schaltfläche RESET DEFAULTS einen »Airbag« mitliefert, die eine fehlerhafte Konfiguration wieder auf die Standardeinstellungen zurückzusetzt, so sollten Sie möglichst immer eine eigene Konfiguration anlegen. Wenn Sie zu einem späteren Zeitpunkt mit einem anderen Embedded System arbeiten wollen, so gehen früher erstellte Konfigurationen so zumindest nicht verloren.

Klicken Sie nun Ihre neue Konfiguration an, und es öffnet sich der in Abbildung 14.7 gezeigte Dialog.

Hinweis

Natürlich enthält der Dialog zu diesem Zeitpunkt noch keine Daten. Hier wurde der Dialog bereits von mir so ausgefüllt, dass einfache Anwendungen bereits mit dem Debugger untersucht werden können.

Da die Inhalte der Abbildung aufgrund technischer Vorgaben beim Satz möglicherweise nicht gut lesbar sind, habe ich nachfolgend für Sie die Eingaben, wie sie für meine Konfiguration funktionieren, noch zusätzlich aufgeführt:

- Unter EXECUTABLE PATH tragen Sie den vollständigen Pfad zum Installationsort des Cross-Debuggers ein. Auf meinem Host lautet er `/opt/toolchain/bin/arm-linux-gnueabihf-gdb`.
- Im Feld ARGUMENTS tragen Sie die IP-Adresse des Targets (z.B. des Raspberry Pi) mitsamt dem Port ein, über den der Debugger mit `gdbserver` kommuniziert. Auf meinem System lautet die IP-Adresse 192.168.0.15, der Port hat, wie bereits weiter oben erwähnt, die Nummer 2000.

Hinweis

Natürlich können Sie hier auch statt einer IP-Adresse den konkreten Namen des Targets eingeben (z.B. auenland-pi). Voraussetzung hierfür ist, dass Sie den Namen des Targets in der Datei `/etc/hosts` eingetragen und mit einer IP-Adresse verknüpft haben.

- Unter DEBUGGER TYPE wählen Sie GDB als Debugger.
- Im Feld DEBUGGER INITIALIZATION COMMANDS tragen Sie zusätzlich noch das `gdb`-Kommando `Target remote auenland-pi:2000` ein.

Hinweis

Ich hatte nicht erwartet, dass dieses Feld ausgefüllt werden muss: Die Dokumentation von Code::Blocks ist für den Abschnitt »Debugging« noch nicht ganz vollständig. Gefühlsmäßig hatte ich erwartet, dass das Ausfüllen des ARGUMENTS-Feldes ausreicht. Hier hatte ich mich aber »geschnitten«: Ohne die gezeigte Eingabe funktioniert es nicht, und Code::Blocks zeigt eine Fehlermeldung an.

Das Feld muss aber auf jeden Fall ausgefüllt werden, wenn Sie einen Hardware-Debugger (ICE) einsetzen, da hier bestimmte Eigenschaften eingestellt werden müssen. Ein Beispiel hierzu finden Sie in meinem Buch »ARM Cortex-M3 Mikrocontroller«, das unter der ISBN-Nummer 9-783826-694752 ebenfalls im mitp-Verlag erschienen ist.

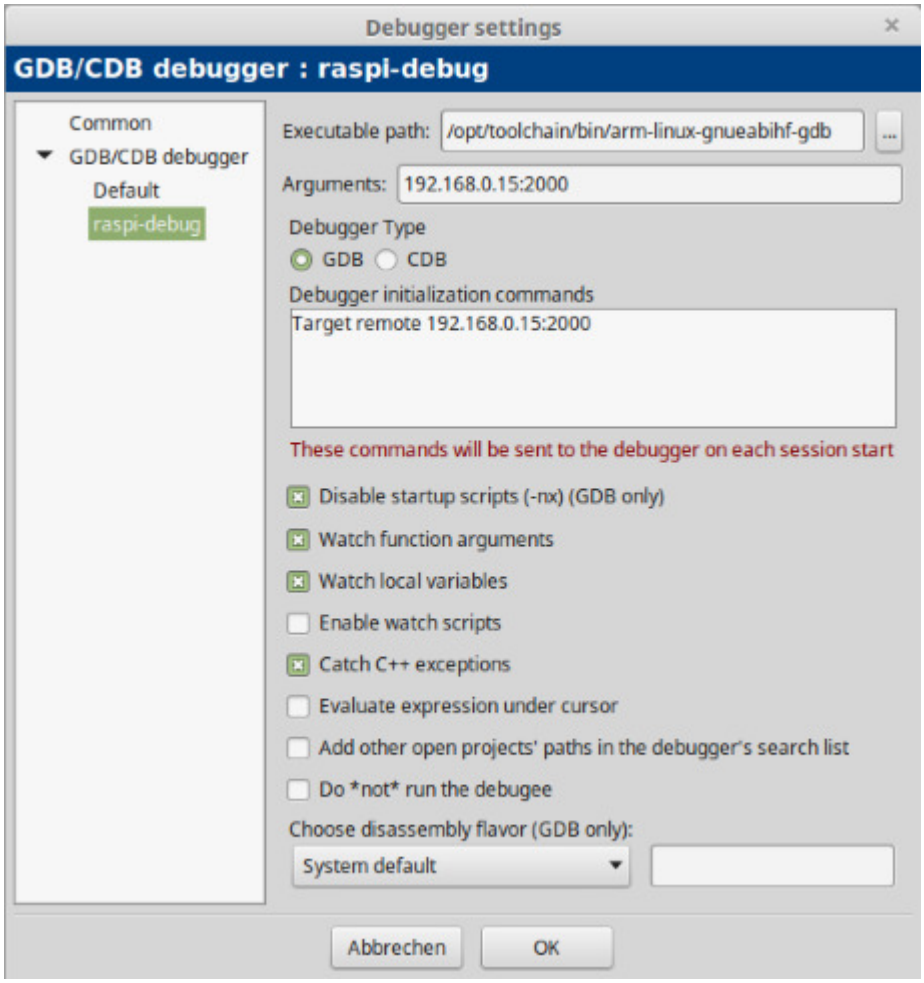

Abbildung 14.7: Debugger-Einstellungen für den Raspberry Pi

Die weiteren Einstellungen können Sie so anpassen, wie Sie Ihren Bedürfnissen entsprechen. Sehr interessant sind die Optionen WATCH FUNCTION ARGUMENTS und WATCH LOCAL VARIABLES: Die entsprechenden Werte werden, wie Sie später sehen werden, automatisch von Code::Blocks angezeigt.

14.4 Ein Beispiel

Anhand eines einfachen Beispiels wird nun gezeigt, wie der Cross-Debugger genutzt werden kann.

14.4.1 Ein einfaches C-Programm

Das folgende Programm berechnet die Potenzen von 2 mit den Werten 0 bis 9 als

Exponent. Die Ausgabe des Ergebnisses erfolgt mit der Funktion `printf` innerhalb der `main`-Funktion, während die Berechnung innerhalb der Funktion `power` durchgeführt wird.

```
#include <stdio.h>

int power(int,int);
int main()
{
    int i;
    printf("Program to calculate power\n");
    for (i=0;i<10;i++)
        printf("%d %d\n",i, power(2,i));
    return 0;
}

int power (int base, int n)
{

    int i,p;
    p=1;
    for (i=1; i<=n; i++)
        p = p*base;
    return p;
}
```

Listing 14.1: Listing des Beispielprogramms

Erzeugen Sie mit Code::Blocks (oder der Entwicklungsumgebung Ihrer Wahl) ein neues Projekt mit dem Namen `power`. Berücksichtigen Sie, dass Sie beim Kompiliervorgang die Option `-g` angeben: Diese Option sorgt dafür, dass das ausführbare Programm Debug-Symbole enthält.

Hinweis

Sie können zwar auch auf die Option `-g` verzichten, allerdings erschwert dies – besonders bei komplexen Programmen – später die Fehlersuche.

Abbildung 14.8 zeigt empfohlene Compilereinstellungen. Die erforderlichen bzw. die empfohlenen Einstellungen sind durch die Kästchen in Abbildung 14.8 hervorgehoben.

Hinweis

Das Aktivieren von ENABLE ALL COMMON COMPILER WARNINGS ... entspricht der Option `-Wall`, was die Ausgabe der üblichsten Compilerwarnungen erzwingt. Bei eigenen Projekten strebe ich grundsätzlich immer an, alle Compilerwarnungen zu entfernen. Warnhinweise des Compilers sind keine Fehler, weisen aber darauf hin, dass der Programmierer hier möglicherweise etwas anderes erreichen wollte, als der Compiler schließlich produziert. Persönlich empfinde ich es ästhetischer, wenn mein Monitor beim Kompilieren nicht von Compilerwarnungen »überflutet« wird. Darüber hinaus bin ich dann sicher, dass der Compiler wirklich den erwarteten Code erzeugt.