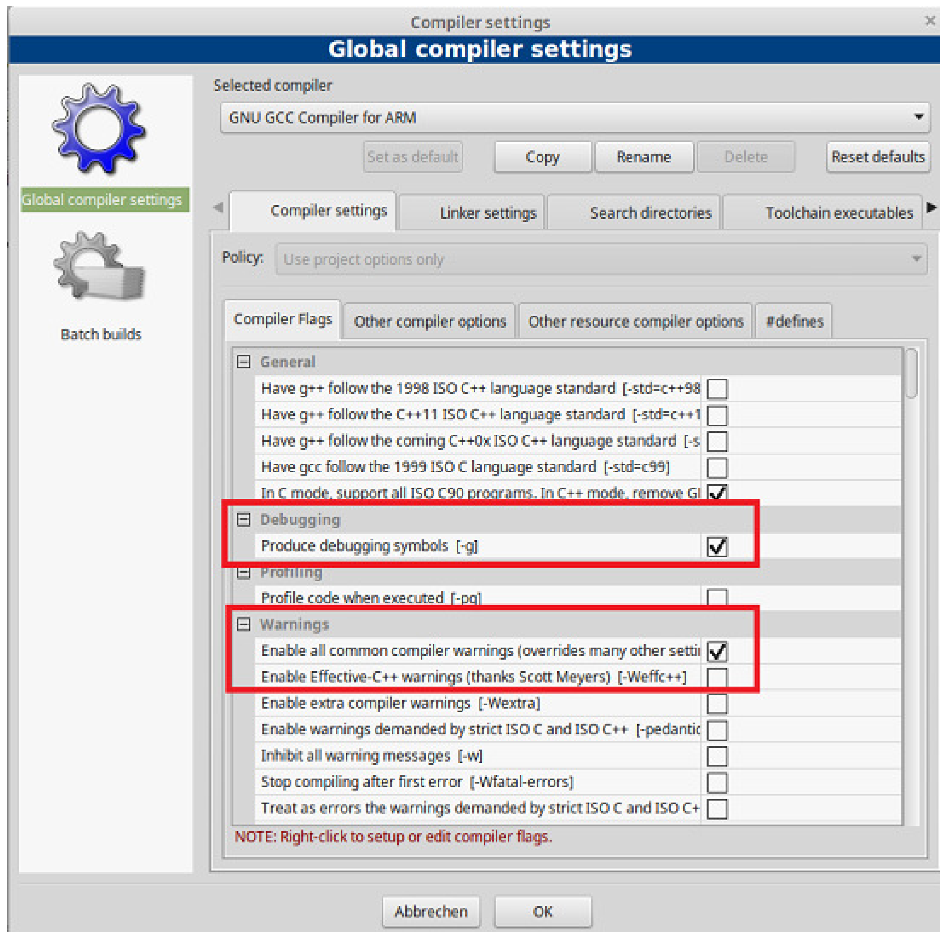

Abbildung 14.8: Compilereinstellungen

Hinweis

Die Compiler der GNU Compiler Collection für die verschiedenen Programmiersprachen bieten noch weitere Optionen, die den Warnlevel erhöhen. Hierzu zählen beispielsweise die Optionen `-Wpedantic`, `-Wextra` und noch viele weitere. Interessant bei `-Wextra` ist unter anderem, dass diese Option auch nichtinitialisierte Variablen anzeigt: Nichtinitialisierte Variablen sind eine häufig auftretende Fehlerursache. Verschaffen Sie sich einen Überblick über sämtliche Warnlevel mit einer Internetsuche unter Verwendung des Suchbegriffs »gcc warning levels«.

Nach dem erfolgreichen Kompilieren und Linken des Programms wird die ausführbare Datei mit dem Programm `scp` auf das Embedded System übertragen. Das Kopieren des Quelltexts auf das Target ist nicht erforderlich.

14.4.2 Was auf dem Target zu tun ist...

Starten Sie auf dem Target mit dem Kommando

```
gdbserver localhost:2000 power
```

den GDB-Server. Abbildung 14.9 zeigt, wie das Programm hierauf reagiert.

Hinweis

Abbildung 14.9 zeigt sogar noch mehr, da ich den Screenshot erst nach der vollständigen Untersuchung mit `gdb` erstellt habe.

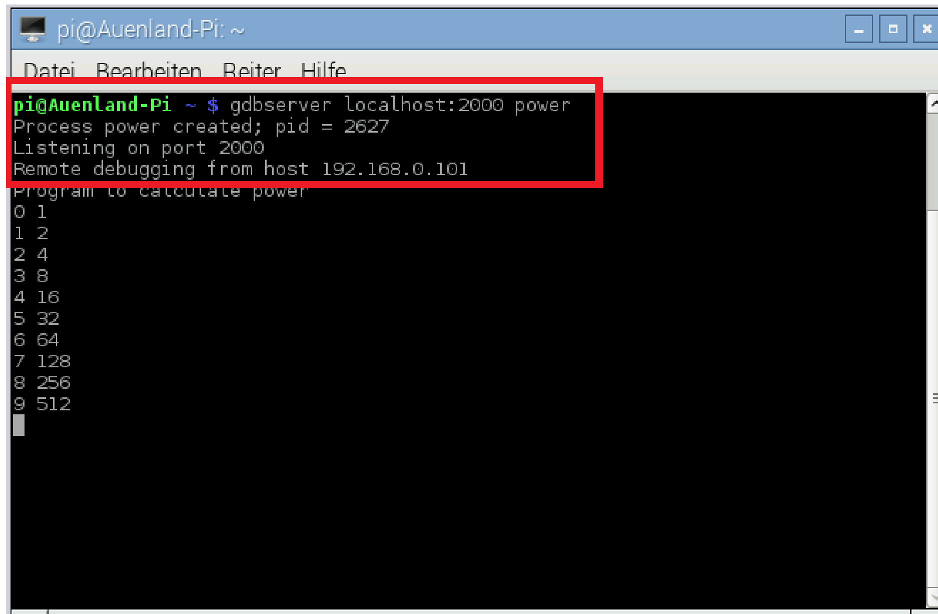

Abbildung 14.9: Meldungen von gdbserver

Der GDB-Server meldet sich zunächst nur mit den ersten beiden Zeilen: Er zeigt an, dass der Prozess `power` erzeugt wurde, dass seine Prozess-ID den Wert 2627 hat und dass er an Port 2000 auf `gdb`-Kommandos »lauscht«.

Hinweis

Ich hoffe, dass dieser Hinweis inzwischen überflüssig ist, möchte aber dennoch für die Leser, die sich noch nicht zu 100% mit der Materie vertraut fühlen, darauf hinweisen, dass die erzeugte Prozess-ID auf Ihrem Targetsystem mit sehr hoher Sicherheit anders lautet: Werte, wie die Prozess-ID, sind immer auch davon abhängig, wie viele andere Prozesse bereits gestartet wurden.

Die Meldung `Remote debugging from host 192.168.0.101` wird erst angezeigt, wenn der Debugger `arm-linux-gnueabihf-gdb` auf dem Host gestartet wurde.

14.4.3 Was auf dem Host zu tun ist...

Abbildung 14.10 zeigt Code::Blocks mit dem geladenen Projekt. Bisher ist noch nichts passiert, weshalb ich die Abbildung so stark verkleinert habe, dass die Inhalte praktisch nicht lesbar sind. Wichtig ist aber der hervorgehobene Bereich in der rechten oberen Ecke: Hier finden Sie die Schaltflächen, mit denen Sie einige der Funktionen des Debuggers steuern können. Diese »Shortcuts« werden weiter unten noch beschrieben.

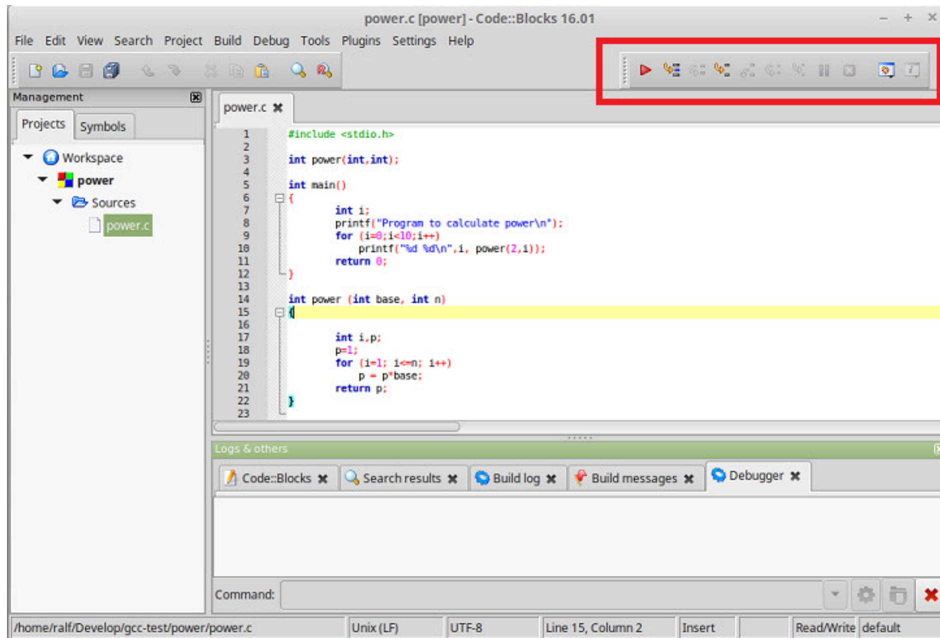

Abbildung 14.10: Code::Blocks nach dem Laden des Projekts power

Wenn Sie nun in der Menüleiste den Eintrag DEBUG anklicken, so öffnet sich die in Abbildung 14.11 gezeigte Auflistung aller Debugger-Steuerkommandos. Viele hiervon sind noch nicht anwählbar, da der Debugger noch nicht gestartet wurde. Dies ändert sich aber, sobald der Debugger gestartet ist.

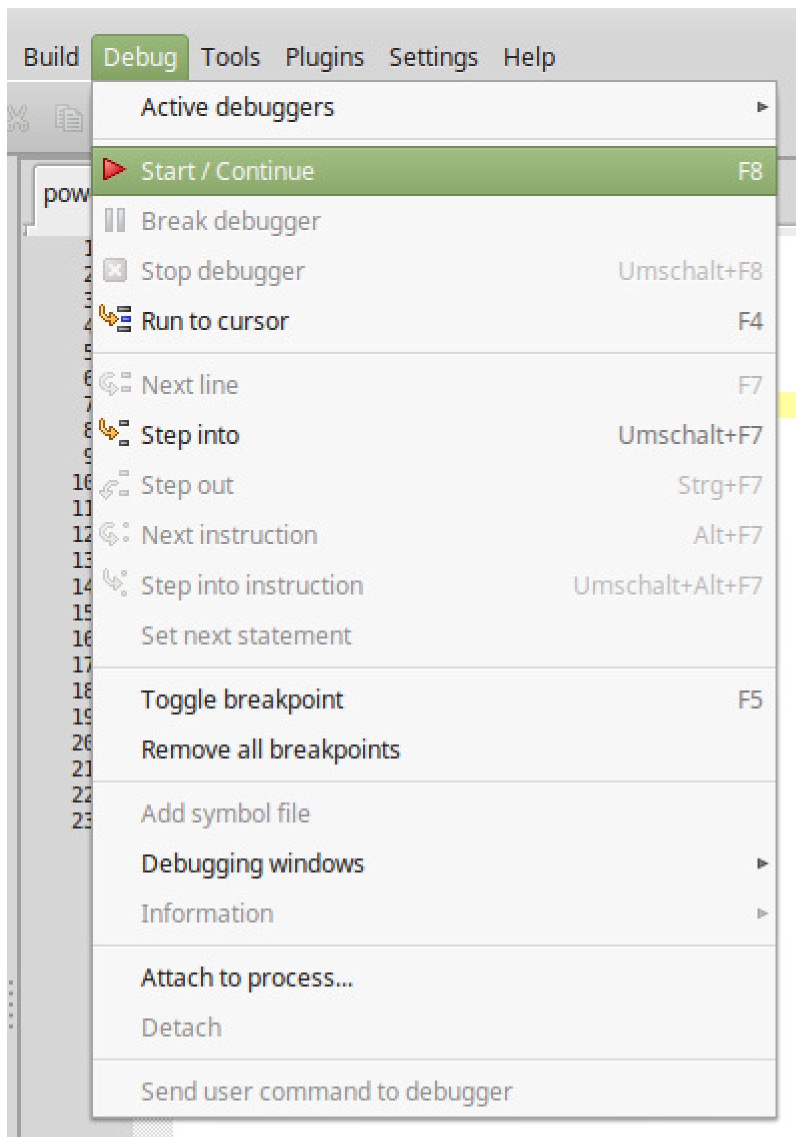

Abbildung 14.11: Inhalt des Menüs Debug von Code::Blocks

Ein Streifzug durch das Debug-Menü

Bevor das Beispiel praktisch mit dem Debugger untersucht wird, sollen zunächst die zur Verfügung stehenden Steuerkommandos beschrieben werden.

Active debuggers

Standardmäßig ist der Eintrag START/CONTINUE aktiv. Wir wollen aber zunächst die Funktion des Eintrags ACTIVE DEBUGGERS prüfen. Abbildung 14.12 zeigt dann

sämtliche Debugger-Konfigurationen an, die über die in Abschnitt 14.3 beschriebene Prozedur CREATE CONFIG erzeugt wurden. Der aktive Debugger ist durch einen weißen Punkt markiert.

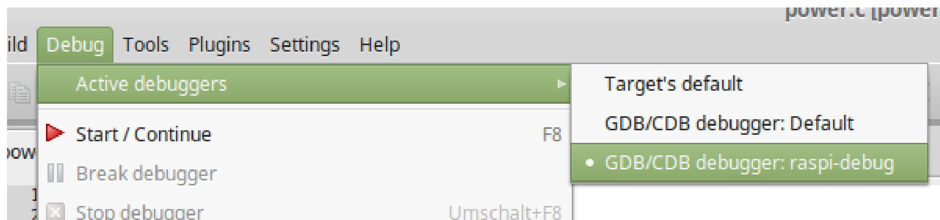

Abbildung 14.12: Liste der verfügbaren Debugger-Konfigurationen

Wenn Sie den bisher beschriebenen Schritten gefolgt sind, so wird auch bei Ihnen die Konfiguration RASPI-DEBUG aktiviert sein.

Hinweis

Zudem erkennen Sie in der Debugger-Funktionsliste auch die in Abbildung 14.10 markierten Symbole wieder. Um später wirklich komfortabel auf die Fehlersuche gehen, empfehle ich Ihnen, sich die Tastenkürzel zu merken, mit denen die verschiedenen Funktionen ebenfalls ausgelöst werden können.

Start/Continue ((F8))

Mit dem Menüeintrag START/CONTINUE wird der Debugging-Prozess gestartet bzw. fortgesetzt, wenn Sie ihn vorher mit der Funktion BREAK DEBUGGER angehalten haben.

Break debugger (kein Tastatur-Shortcut)

Über diesen Eintrag kann der Debugging-Prozess vorübergehend angehalten werden. Dies ist beispielsweise dann sinnvoll, wenn Sie einen Haltepunkt (Breakpoint) hinzufügen oder entfernen möchten. Wenn Sie hiermit fertig sind, kann der Debugging-Prozess an der unterbrochenen Stelle mit der Funktion START/CONTINUE fortgesetzt werden.

Stop debugger ((Shift)+(F8))

Hierüber wird der Debugger `gdb` beendet und gleichzeitig die Verbindung mit dem GDB-Server heruntergefahren. Die Wiederaufnahme eines Debug-Vorgangs ist dann nur möglich, indem die Verbindung zwischen Host und Target komplett neu aufgebaut wird. Hierfür ist es erforderlich, `gdbserver` auf dem Target erneut zu starten.

Run to cursor ((F4))

Dieser Menüpunkt sorgt dafür, dass der Debugger das Programm bis zu dem Punkt ausführt, an dem der Cursor im Quelltext zu sehen ist: Die Funktion entspricht also der eines Breakpoints, ohne dass er explizit gesetzt werden muss. Aufgrund der vielen Einstellmöglichkeiten bei der Farbgebung besteht aber die Gefahr, dass der Cursor nicht gut zu erkennen ist. Diese Funktion kann aber auch vorteilhaft sein, da immer die

Möglichkeit besteht, dass die Funktion an einem bereits getesteten und korrigierten Bereich erneut unterbrochen wird, nur weil vergessen wurde, einen Breakpoint an dieser Stelle wieder zu entfernen.

Next line ((F7))

Die Funktion NEXT LINE wird bei anderen Debuggern häufig auch als *Single Step* (*Einzelschritt*) bezeichnet. Wenn die Ausführung eines Breakpoints unterbrochen wird, besteht der Wunsch, ab diesem Punkt die weitere Funktion eines Programmabschnitts schrittweise auszuführen.

Step into ((Shift)+(F7))

Je komplexer Ihre Programme werden, umso häufiger werden Sie bestimmte Funktionalitäten in separate Funktionsblöcke auslagern und diese dann an den benötigten Stellen immer wieder aufs Neue zu verwenden. Nach einer gewissen Zeit werden Sie wissen, welche Funktionen korrekt funktionieren und welche Funktionen noch einer näheren Untersuchung bedürfen. Bereits getestete Funktionen würden Sie normalerweise mit dem Kommando NEXT LINE überspringen und nur deren Ergebnisse weiterverwenden. Mit STEP INTO (Tastenkombination können Sie in diese noch nicht ausgiebig untersuchten Funktionen hineinspringen und sie dort mit dem Kommando NEXT LINE gezielt genauer untersuchen.

Step out ((Strg)+(F7))

Springen Sie mit STEP INTO versehentlich in eine bereits getestete Funktion, können Sie diese mit STEP OUT sofort wieder verlassen. Anstatt in solchen Fällen mit NEXT LINE mühsam durch den Rest der Funktion zu wandern, können Sie die Funktion sofort wieder verlassen. Das Ergebnis der Funktion steht für die folgenden Tests dennoch, wie im Einzelschrittverfahren auch, sofort zur Verfügung.

Next instruction ((Alt)+(F7)) / Step into instruction ((Shift)+(Alt)+(F7)) / Set next statement (kein Tastatur-Shortcut)

Für die drei genannten Funktionen habe ich bisher keine vernünftige Anwendung gefunden. Ich habe allerdings eine Vermutung darüber, was hier beabsichtigt ist: Sie geht in die Richtung, dass hier Funktionen implementiert wurden, die Nutzern von Microsofts Visual Studio oder KDevelop bekannt sind. Vermutlich besteht hier die Möglichkeit, beliebige Register des Mikrocontrollers im laufenden Programm zu verändern. So könnte SET NEXT STATEMENT dazu dienen, den sogenannten *Instruction pointer* direkt zu manipulieren. Ich selber habe solche Funktionen noch nie angewendet.

Hinweis

Wenn Sie wissen, wie diese Funktionen praktisch genutzt warden, so bitte ich um eine kurze Rückmeldung an embedded@ralf-jesse.de, damit ich Ihre Erkenntnisse auch anderen Lesern zur Verfügung stellen kann.

Toggle breakpoint ((F5)), Remove all breakpoints (kein Tastatur-Shortcut)

Mit der ersten Funktion können Sie einen Breakpoint umschalten. Die zweite Funktion dient zum Entfernen aller Breakpoints. Dies ist sinnvoll, wenn alle Tests erfolgreich abgeschlossen sind und stellt sicher, dass kein Breakpoint versehentlich im fertigen Programm verbleibt.

Add symbol file (kein Tastatur-Shortcut)

Obwohl der Menüeintrag ADD SYMBOL FILE auch bei Verwendung von `gdb` auswählbar ist, bin ich der Ansicht, dass diese Funktion aus der Microsoft-Welt stammt und daher nur in Verbindung mit `cdb` nutzbar ist. Soweit mir bekannt ist, handelt es sich bei Symboldateien um ein Microsoft-spezifisches Format. `gdb` unterstützt Symboldateien nicht.

Debugging windows (kein Tastatur-Shortcut)

Dieses Menü ist besonders nützlich, da Sie hier verschiedene für wichtige Ansichten aktivieren können. Bei den unterstützten Ansichten handelt es sich um

- die Liste aller Breakpoints,
- die Liste der CPU-Register,
- den sogenannten Call-Stack (hiermit können Sie die Reihenfolge von Funktionsaufrufen prüfen),
- der Anzeige des Programms im Assemblercode (so, wie er vom Compiler erzeugt wird),
- Speicherauszüge,
- laufende Threads
- sowie um Watches (das sind Variablen, deren Wert Sie immer im Blick behalten wollen).

Abbildung 14.13 zeigt die verfügbaren Ansichten:

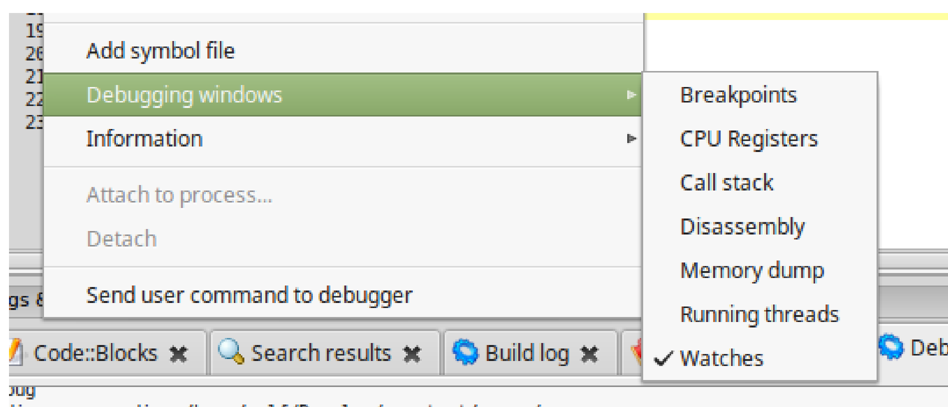

Abbildung 14.13: Liste der verfügbaren Ansichten

Üblicherweise werden die Ansichten in eigenständigen Fenstern dargestellt. Steht Ihnen aber ein ausreichend großer Monitor zur Verfügung (mindestens 24 Zoll), so können sie aber auch in die Gesamtansicht von Code::Blocks integriert werden (siehe Abbildung 14.14).

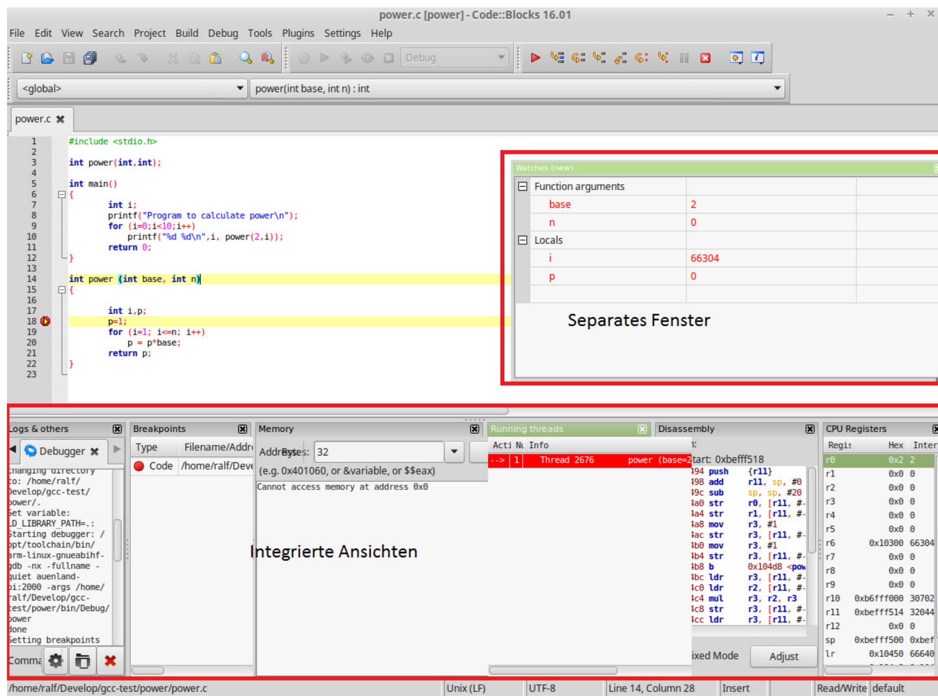

Abbildung 14.14: Ansichten des Menüs Debug/Debugging windows

Information (kein Tastatur-Shortcut)

Dieser Menüeintrag stellt eine Komfortfunktion zur Verfügung, mit der Sie sich einen Überblick über verschiedene Aspekte des laufenden Debugging-Prozesses verschaffen können. Abbildung 14.15 zeigt die Optionen:

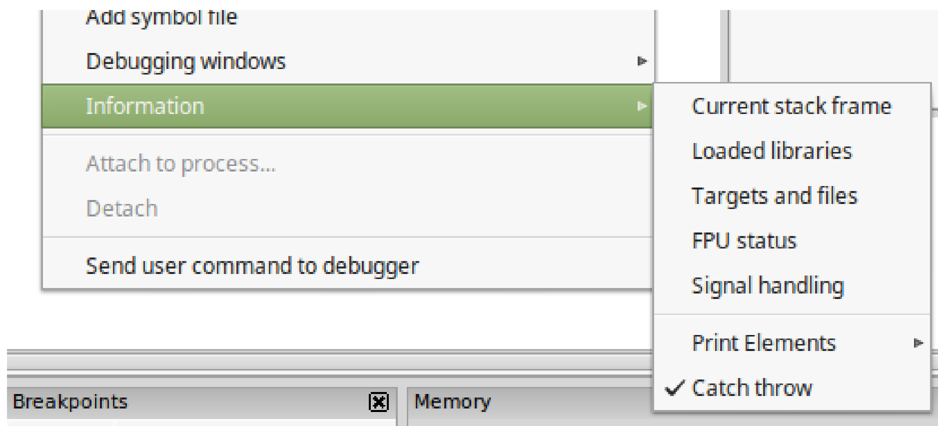

Abbildung 14.15: Informationen zum laufenden Prozess

Die verfügbaren Informationen werden immer in einem eigenständigen Fenster angezeigt und lassen sich nicht in die Code::Blocks-Gesamtansicht integrieren. Dies mag daran liegen, dass sie immer nur eine Momentaufnahme (Snapshot) des laufenden Prozesses darstellen. Sie erhalten mit diesem Menü Informationen zum aktuellen Zustand des Stacks, zu geladenen Bibliotheken, Targets, Register der Fließpunkteinheit (FPU) usw. Die Ansicht erfolgt in einem modalen Fenster, d.h. das Fenster muss erst geschlossen werden, bevor der Debugging-Prozess fortgesetzt werden kann.

Attach to process und Detach (kein Tastatur-Shortcut)

Über diese Menüeinträge können Sie den Debugger an einen bereits laufenden Prozess anhängen bzw. wieder davon abkoppeln. Diese Funktion verlangt aber sehr fortgeschrittene Kenntnisse in den Bereichen Debugging und Prozessmanagement.

Send user command to debugger (kein Tastatur-Shortcut)

Diese Funktion bietet Ihnen die Möglichkeit, Kommandos, die nicht von der Entwicklungsumgebung unterstützt werden, manuell an den Debugger zu senden. Ich habe diese Funktion bisher nie benötigt – allerdings habe ich auch nicht alle der bereits erwähnten ca. 800 Seiten Dokumentation von `gdb` gelesen ;-).

14.4.4 Jetzt geht's los...

Prüfen Sie vorab noch einmal ob der GDB-Server auf dem Target bereits gestartet ist: Sie erkennen dies an der Meldung `Listening on port 2000`. Wechseln Sie nun zum Host und klicken Sie im DEBUG-Menü den Eintrag START/CONTINUE an. Alternativ können Sie natürlich auch das rote Dreieck anklicken oder die Taste (F8) drücken. Setzen Sie nun den Cursor an den Beginn der Zeile, in der `int main()` steht und drücken Sie dann die Taste (F4) (RUN TO CURSOR). Die Zeile, die der Debugger als nächstes untersuchen wird, ist durch ein gelbes nach rechts gerichtetes Dreieck erkennbar (siehe Abbildung 14.16). Setzen Sie zusätzlich innerhalb der Funktion `power()` einen Breakpoint in die Zeile `p=1;`.

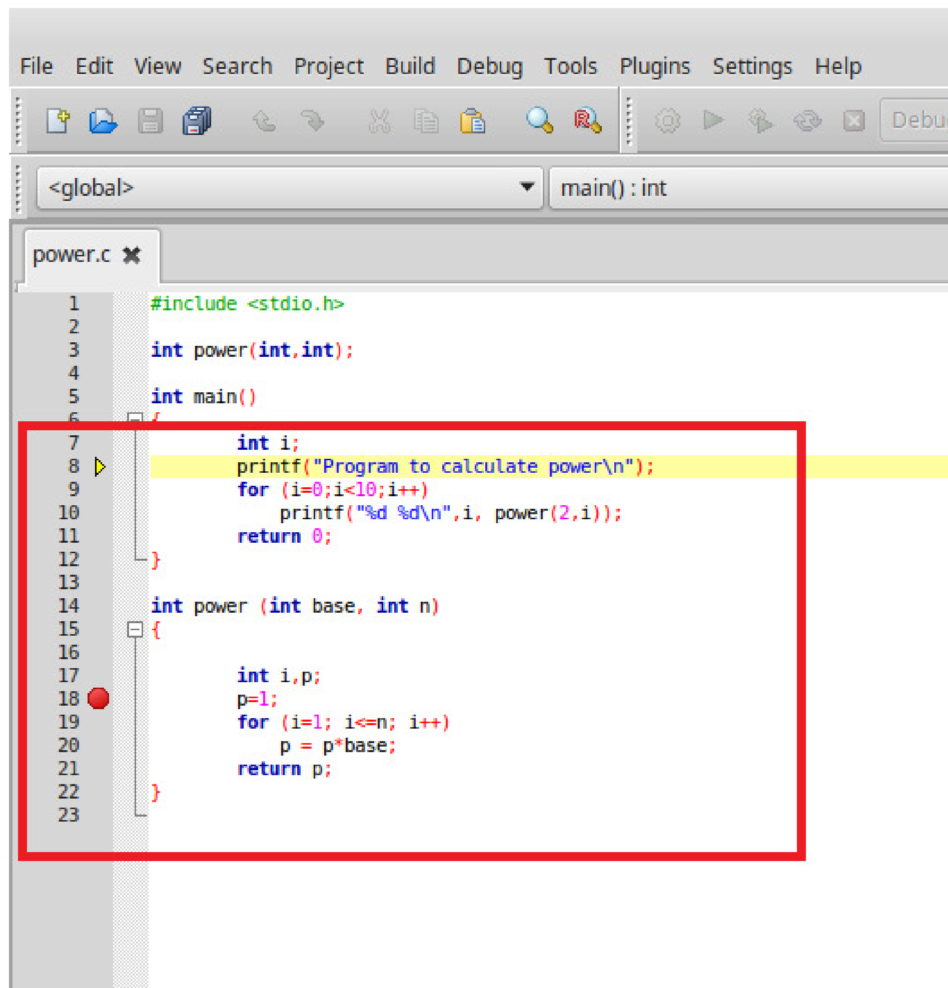

Abbildung 14.16: Anzeige der aktuelle Zeile und eines Breakpoints

Hinweis

Einen Breakpoint können Sie auf verschiedene Weisen setzen bzw. löschen: Klicken Sie mit der rechten Maustaste an die Stelle, an der der Breakpoint gesetzt werden soll oder setzen Sie den Cursor an die entsprechende Stelle und drücken die Taste (F5) (Toggle Breakpoint). Die Aktivierung bzw. Deaktivierung eines Breakpoints über das Menü ist meiner Meinung nach am wenigsten effizient.

Nachdem der Debugger gestartet wurde, erhält das Terminalfenster auf dem Target die Zeile `Remote debugging from host <IP-Adresse>`: Der GDB-Server signalisiert hiermit, dass er »scharf geschaltet« ist und auf Debugger-Kommandos vom Host wartet. Überprüfen Sie an dieser Stelle noch, dass im Menü DEBUG/DEBUGGING WINDOWS mindestens der Eintrag WATCHES aktiviert ist. Dieser sorgt dafür, dass Funktionsargumente, die an die aufgerufene Funktion übergeben werden sowie lokale Variablen innerhalb der Funktion `power()` automatisch angezeigt werden: Dies ist

besonders komfortabel, da Sie die Variablen nicht manuell auswählen müssen und somit immer im Blick haben.

Drücken Sie nun die Taste (F7) (NEXT LINE). Je nachdem, welche Funktion bzw. welches Kommando in der aktuellen Zeile steht, kann es erforderlich sein, diese Taste mehrmals zu betätigen. Lassen Sie sich deshalb Zeit zwischen einzelnen Betätigungen dieser Taste und beobachten Sie immer das Terminalfenster des Targets.

Die erste interessante Zeile, die ausgeführt wird, enthält den Text `printf(„Program to calculate power\n");`. `printf` ist eine komplexe Bibliotheksfunktion und erforderte bei mir dreimaliges Betätigen der Taste (F7), bevor der angegebene Text im Terminal des Targets angezeigt wurde. Ab jetzt wird es richtig interessant: Drücken Sie nun erneut die Taste F7. Je öfter Sie F7 betätigen, umso mehr verändert sich die Anzeige im Terminalfenster des Targets bis schließlich das Programm beendet wird. Auf dem Host verändert sich entsprechend auch die Ansicht der WATCHES: Dies wird in Abbildung 14.17 exemplarisch gezeigt.

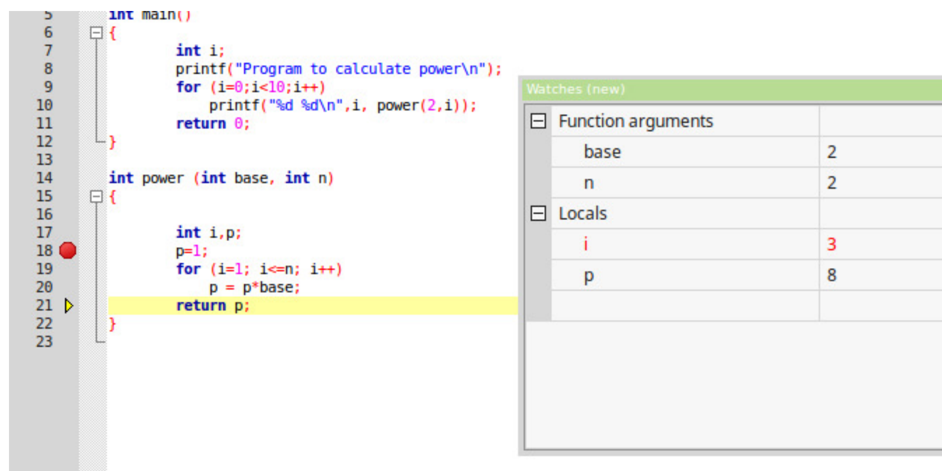

Abbildung 14.17: ... Nach Ausführung einiger »Next line«-Kommandos

Hiermit soll die Einführung in das Debugging von Embedded Systemen abgeschlossen werden.

Hinweis

Ich empfehle Ihnen, so viele Programme wie möglich auch mit dem Debugger zu untersuchen. Das Debugging ist ein komplexer Vorgang und verlangt ein tiefes Verständnis des Prüflings, was bei diesem einfachen Beispiel bei weitem nicht erreicht wird.

14.5 Tipps zur Fehlersuche mit Debuggern

Wie am Ende des letzten Abschnitts gesagt: Debuggen ist nicht einfach und kann sehr aufwendig werden! Mit den bisher vorgestellten Techniken können Sie es beispielsweise »vergessen«, den Kernel oder ladbare Module zu debuggen.

Der Kernel lässt sich deshalb nicht so einfach debuggen, weil er hierfür auf eine besondere Weise kompiliert werden muss, was nur aufgrund der Verwendung der Compileroption `-g` nicht möglich ist: Der Kernel muss auf eine andere Weise kompiliert werden, die ich in einem weiteren Kapitel sowie auf meiner Website http://www.ralf-jesse.de nachreichen werde. Entsprechend gilt dies natürlich für »echte« Treiber, da diese fest in den Kernel integriert werden. Aber auch ladbare Module lassen sich auf diese Weise nicht debuggen: Da Module erst später (und bei Bedarf manuell oder automatisch durch Hot-Plugging) geladen werden, können Symboltabellen gar nicht aktuell sein. Die Symboltabellen müssen zunächst aktualisiert werden, bevor sie zuverlässig verwendet werden können.

Die nachfolgenden Tipps sind teilweise selbstverständlich; ich halte es aber für sinnvoll, dass Sie diese verinnerlichen.

14.5.1 Variablen immer initialisieren

Erst kürzlich hat ein junger und wenig erfahrener Kollege von mir ein komplexes Programm geschrieben, dass nach jedem Start nur genau einmal funktionierte; weitere Versuche führten immer zu Fehlermeldungen. Die Lösung war, dass der Kollege vergessen hatte, Variablen zu initialisieren und sich darauf verlassen hat, dass dies vom Compiler gemacht wird. Die Initialisierung der Variablen und anschließendes Neukompilieren löste das Problem.

14.5.2 Was wollen Sie erreichen?

So trivial diese Frage anmutet: Sie ist dennoch sehr berechtigt! Versuchen Sie immer zu **verstehen**, was Sie tun! Dass ich das Wort »verstehen« durch Fettdruck hervorgehoben habe, soll diesen Anspruch unterstreichen. Machen Sie sich immer klar, welche Startbedingungen Ihnen zur Verfügung stehen und wie das gewünschte Resultat aussehen soll. Wenn Ihnen die Funktionsweise nicht aus dem Quelltext klar wird, erstellen Sie eine Zeichnung: »Malen« Sie einen Ablaufplan oder ein Struktogramm oder erstellen Sie ein UML-Diagramm. Notieren Sie zudem die einzelnen Schritte, die erforderlich sind, um ein bestimmtes Ergebnis zu erzielen.

14.5.3 Reihenfolge beim Debuggen

Prüfen Sie bei Funktionsaufrufen zunächst immer, ob Funktionsargumente korrekt sind. Gehen Sie anschließend in umgekehrter Reihenfolge vor, also von hinten nach vorne: Setzen Sie einen Breakpoint, bevor die Funktion zu der Stelle zurückkehrt, von der sie aufgerufen wurde. Auf diese Weise werden Sie relativ sicher die Stelle erkennen, bis zu der noch alles korrekt funktioniert: Der Fehler muss dann hinter dieser Stelle sein.

14.5.4 Prüfen Sie Speicherinhalte

Fehler durch falsche bzw. unerwartete Speicherinhalte treten häufig bei C- oder C++-Programmen auf, in denen Pointer verwendet werden. Machen Sie sich immer klar, dass Pointer nichts anderes als Adressen sind, die auf andere Stellen im Speicher zeigen. Handelt es sich um Pointer auf andere Pointer, so machen Sie sich klar, dass es sich

hierbei um eine Adresse handelt, die auf eine weitere Adresse im Arbeitsspeicher zeigt. Erst an der hier genannten Adresse finden Sie dann den gesuchten Wert. Prüfen Sie dann, ob der Pointer wirklich auf die erwartete Speicherstelle oder möglicherweise auf einen ganz anderen Speicherbereich zeigt. Prüfen Sie in solchen Fällen mit dem Menüeintrag DEBUG/DEBUGGING WINDOWS/MEMORY DUMP, welche Speicherbereiche genutzt werden und welche Werte dort stehen.