Ingenieurbüro für Echtzeitprogrammierung

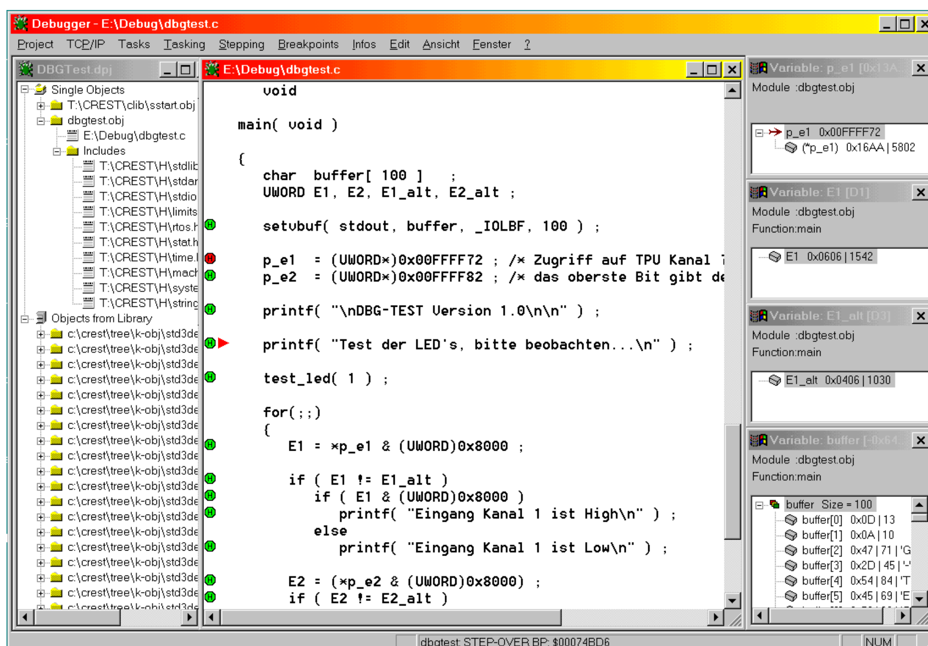
IEP

Debugger

für CREST-C und PEARL90

Dok-Rev. 1.8 vom 19.02.2010

Software-Rev. 2.667 vom 18.02.2010



Inhaltsverzeichnis

1 Urheberrecht und Haftung 4

1.1 Handhabung 4

1.2 Erklärung 4

2 Der Debugger 5

2.1 Realisierung des Debuggers 5

2.2 Lieferumfang 5

2.3 Voraussetzungen für Debugging über Ethernet 6

2.4 Voraussetzungen für Debugging über CSLIP 6

2.5 Netzwerksoftware unter RTOS-UH 6

3 Inbetriebnahme des Debuggers-Servers 7

3.1 Vorbereitung einer CREST-C-Debugsession 8

3.2 Vorbereitung einer UH-PEARL-Debugsession 9

4 Arbeiten mit dem Debugger 10

4.1 Start einer Debugsession 10

4.2 Debugsession für PEARL-Shellmodule 15

4.3 Verbinden mit dem Zielsystem 16

4.4 Setzen und Löschen von Breakpoints 18

4.5 Betrachten von Variablen 19

4.6 Setzen von Variablenwerten 21

4.7 Watchpoints 22

4.8 Post Mortem Debugging 23

5 Beschreibung der Menüpunkte 24

5.1 Menü Project 24

5.2 Menü TCP/IP 25

5.3 Menü Tasks 25

5.4 Menü Tasking 26

5.5 Menü Stepping 27

5.6 Menü Breakpoints 28

5.7 Menü Infos 28

5.8 Menü Edit 29

5.9 Menü Ansicht 29

5.10 Menü Fenster 29

5.11 Menü ? 30

6 Ausblick 31

Revisionsliste:

Rev.	Datum	Na.	Änderung
1.0	05.12.2001	Ko	Erstellung
1.1	05.06.2003	Kr	Kapitel WIBU-Key und DBGTRPxQ.SR
1.2	12.06.2003	ha	Beschreibung der erforderlichen Dateien ergänzt
1.3	04.02.2004	Ha	Konfiguration Wibu-Key (Systemsteuerung) ergänzt
1.4	27.09.2004	Kr	Neues Menue Projekt->Settings
1.5	01.03.2005	Ko	Versionsnummer angepasst
1.6	28.06.2007	Kr	PEARL-Shellmodule debuggen
1.7	30.10.2008	Ha	Installation Wibu-Key an Runtime 5.20b angepaßt
1.8	19.02.2010	Kr	Kein Wibu-Key mehr nötig

1 Urheberrecht und Haftung

Alle Rechte an diesen Unterlagen liegen bei der IEP GmbH, Langenhagen.

Die Vervielfältigung, auch auszugsweise, ist nur mit unserer ausdrücklichen schriftlichen Genehmigung zulässig.

In Verbindung mit dem Kauf von Software erwirbt der Käufer einfaches, nicht übertragbares Nutzungsrecht. Dieses Recht zur Nutzung bezieht sich ausschließlich darauf, daß dieses Produkt auf oder in Zusammenhang mit jeweils **einem** Computer zu benutzen ist. Das Erstellen einer Kopie ist ausschließlich zu Archivierungszwecken unter Aufsicht des Käufers oder seines Beauftragten zulässig. Der Käufer haftet für Schäden, die sich aus der Verletzung seiner Sorgfaltspflicht ergeben, z.B. bei unautorisiertem Kopieren, unberechtigter Weitergabe der Software usw.. Der Käufer gibt mit dem Erwerb der Software seine Zustimmung zu den genannten Bedingungen. Bei unlizensiertem Kopieren muß vorbehaltlich einer endgültigen juristischen Klärung von Diebstahl ausgegangen werden. Dies gilt ebenso für Dokumentation und Software, die durch Modifikation aus Unterlagen und Programmen von IEP hervorgegangen ist, gleichgültig, ob die Änderungen als geringfügig oder erheblich anzusehen sind.

Eine Haftung seitens IEP für Schäden, die auf den Gebrauch von Software, Hardware oder Benutzung dieses Manuskriptes zurückzuführen sind, wird ausdrücklich ausgeschlossen, auch für den Fall fehlerhafter Software oder irrtümlicher Angaben.

Das Einverständnis des Käufers oder Nutzers für den Haftungsausschluß gilt mit dem Kauf und der Nutzung der Software und dieser Unterlagen als erteilt.

1.1 Handhabung

Lesen Sie bitte zuerst sorgfältig diese Dokumentation bevor Sie anfangen zu programmieren. Sie sparen Zeit und vermeiden Probleme.

1.2 Erklärung

Wir behalten uns das Recht vor, Änderungen, die einer Verbesserung der Schaltung oder des Produktes dienen, ohne besondere Hinweise vorzunehmen. Trotz sorgfältiger Kontrolle kann für die Richtigkeit der hier gegebenen Daten, Schaltpläne, Programme und Beschreibungen keine Haftung übernommen werden. Die Eignung des Produktes für einen bestimmten Einsatzzweck wird nicht zugesichert.

Alle Rechte vorbehalten. © IEP GmbH 1996-2010

2 Der Debugger

Diese Seiten geben einen groben Überblick über den aktuelle Implementierungsstatus, aktuelle Funktionalität und geplante Erweiterungen des Debuggers RT-Debug.

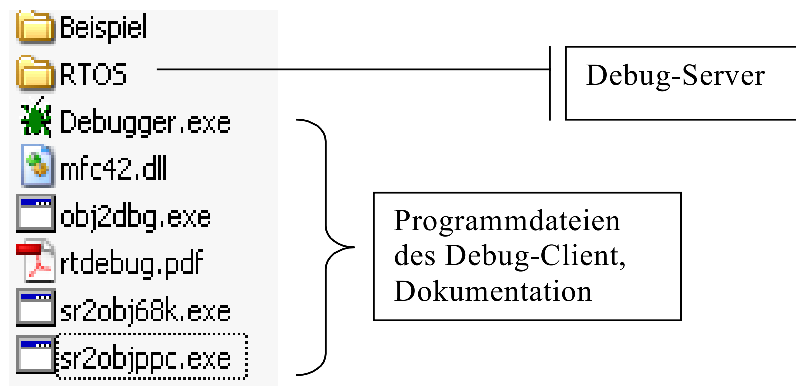
2.1 Realisierung des Debuggers

Bei dem Debugger handelt es sich um einen Remote-Debugger, der auf dem Client/Server-Prinzip basiert. Auf der Zielmaschine unter RTOS-UH wird ein Debugger-Server (DK) gestartet. Die Visualisierung und Steuerung des „Debuggers" erfolgt von einem Debugger-Client auf einem Windows-Rechner. Die Kommunikation zwischen Server und Client verwendet als Transportprotokoll TCP/IP. Als physikalisches Medium sind sowohl Ethernet als auch serielle Verbindungen über SLIP möglich. Der Debugger ist für MC68K- und für PowerPC-basierte RTOS-UH-Systeme verfügbar.

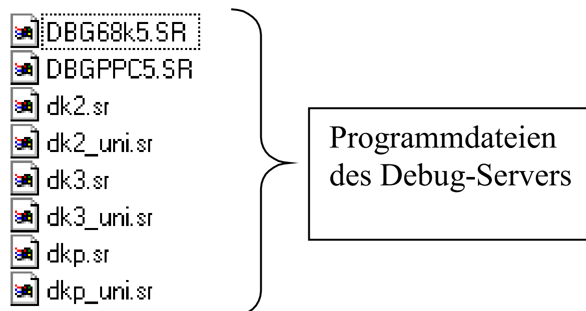
Der Debugger setzt geladene Programme im RAM voraus. Das Debuggen von Programmen im EPROM wird z.Z. nicht unterstützt!

2.2 Lieferumfang

RT-Debug wird auf CD geliefert. Je nach Lieferumfang enthält die CD:



Im Verzeichnis RTOS liegen:



2.3 Voraussetzungen für Debugging über Ethernet

Auf der PC-Seite kann eine beliebige von WINDOWS unterstützte Ethernetkarte Verwendung finden. Für diese Karte muß als Protokoll TCP/IP hinzugefügt werden. Auf dem Zielrechner muß ebenfalls eine Ethernetkarte und der TCP/IP-Protokollstack (netio.sr), alternativ der Protokollstack des IRTm zur Verfügung stehen. Bei der Vergabe der IP-Adressen ist darauf zu achten, daß RTOS-Rechner und Windows-PC sich im gleichen Subnetz befinden.

2.4 Voraussetzungen für Debugging über CSLIP

Auf beiden Seiten wird jeweils eine freie serielle Leitung erwartet. Der Port auf dem RTOS-UH-Rechner benötigt zwingend eine Implementierung des seriellen Treibers mit `CX`-Port, um das CSLIP-Protokoll einsetzen zu können. Auf der Windows-Seite ist die serielle Schnittstelle über das „Dial Up Networking“ passend zu konfigurieren. Es wird angeraten, die Verbindung als „Serial Cable between 2 PC’s“ zu betreiben. Auf der RTOS-UH-Seite wird neben dem TCP/IP-Protokollstack (`netio.sr`) noch ein Treiber für das CSLIP-Protoll (`slip.sr`) benötigt. Weiterhin muß ein Programm gestartet werden, das dem Windows-Rechner die „Einwahl“ auf dem RTOS-Rechner ermöglicht (`modem.sr`).

2.5 Netzwerksoftware unter RTOS-UH

Die benötigte Netzwerksoftware für den TCP/IP-Stack (`netio.sr`) bzw. für das CSLIP-Protokoll (`cslip.sr`) ist nicht Bestandteil des Debugger-Paketes und muß gesondert erworben werden. Die Installationshinweise entnehmen Sie bitte dem zugehörigen RT-LAN-Manual. Das Programm `modem.sr` ist im Lieferumfang des Debugger-Paketes enthalten.

Optional empfiehlt sich die Installation eines `telnet`-, `ftp`- und/oder `samba`-Servers auf der RTOS-UH-Zielmaschine, um die Bedienung und den Datenaustausch zwischen Windows-Client und RTOS-Rechner zu vereinfachen.

3 Inbetriebnahme des Debuggers-Servers

Der Debugger-Server liegt im Verzeichnis RTOS der RT-Debug-CD in Form von S-Records für unterschiedliche Zielsysteme vor.

- `dk2.sr`: der Server für MC68020-Targets, für den TCP-Stack von IEP
- `dk3.sr`: der Server für CPU32-Targets, für den TCP-Stack von IEP
- `dkp.sr`: der Server für PowerPC-Targets, für den TCP-Stack von IEP
- `dk2_uni.sr`: der Server für MC68020-Targets, für den TCP-Stack vom IRT
- `dk3_uni.sr`: der Server für CPU32-Targets, für den TCP-Stack vom IRT
- `dkp_uni.sr`: der Server für PowerPC-Targets, für den TCP-Stack vom IRT

Der Debugger-Server `dk`*`x`*`.sr` und das zu debuggende Programm sind z.Z. noch manuell auf die Zielmaschine zu übertragen. Ist auf dem Zielsystem eine Festplatte oder resetfeste RAM-Disk verfügbar, so empfiehlt es sich, `dk`*`x`*`.sr` dort abzulegen und nach jedem SYSTEM_RESET automatisch zu laden. Entsprechend ist mit den zu debuggenden Modulen zu verfahren.

Zusätzlich wird auf dem RTOS-UH System der sogenannte DBGTRP benötigt. Dieser liegt auf der RT-Debug-CD ebenfalls im Verzeichnis RTOS als S-Record vor.

- `DBG68k5.SR`: Debug-Trap für 68k-basierte Zielsysteme
- `DBGPPC5.SR`: Debug-Trap für PPC-basierte Zielsysteme

Der DBGTRP kann sowohl nachgeladen als auch in einem überscannten Bereich abgelegt werden. Falls er nachgeladen wird muss der Bedienbefehl *Install_DBG_Trap* einmal ausgeführt werden, damit der DBGTRP wirksam ist.

Der Debugger-Server wird mittels des Bedienbefehls

```
DK [-n=netzwerk-ldn] [-l=dk-ldn]
```

netzwerk-ldn:	LDN des TCP-Protokollstacks default bei `dk?_uni`: 102, bei `dk?`: 16
dk-ldn:	intern im DK genutzt, default: 60

gestartet und wartet dann unter Verwendung von Port 4711 auf eine Verbindungseröffnung des Debugger-Clients. Die LDN-Angaben können in C-Notation wahlweise dezimal, oktal oder hexadezimal erfolgen.

3.1 Vorbereitung einer CREST-C-Debugsession

Zunächst sind alle Module des Programmes komplett mit entsprechenden Debug-Optionen zu übersetzen und zu linken. Als Compilerparameter ist `-z` anzugeben. Beim Linken ist die Option `-z` vorzugeben. Weiterhin ist eine Stack-Variante der Standard-Bibliothek zu verwenden, da sonst einige Features des Debuggers nicht nutzbar sind. Diese besitzt die Endung `debug` im Namen (z.B. `std3stack.lib`). Die Stack-Bibliotheken entsprechen weitestgehend den korrespondierenden Bibliotheken mit der Endung `long` und enthalten sämtliche Debuginformationen, die der Compiler z.Z. exportieren kann.

Beim Linken wird zusätzlich zum S-Record ein Debugfile mit der Endung `.dbg` generiert – ein Binärfile, das den Windows-Rechner nicht zu verlassen hat! Der erzeugte S-Record ist bei der Generierung von Debug-Code nur unwesentlich länger als ohne Verwendung der Debug-Optionen `-z`. Im Debugmodus wird das Adreßregister `A6` für die normale Codegenerierung gesperrt und dient dazu, eine Rückverfolgungskette der aufgerufenen Funktionen zu realisieren. Weiterhin ist so – unabhängig von der Kenntnis der aktuellen Stackbelastung – innerhalb jeder derartig kodierten Funktion der Zugriff auf Variablen möglich, die auf dem Stack allokiert wurden. Der eigentliche Code des Programmes entspricht – abgesehen von der Blockierung des einen Adressregisters – dem der Nicht-Debug-Version und sollte vom Laufzeitverhalten her betrachtet fast identisch sein.

Grundsätzlich gilt, dass die Debuginformationen die absoluten Pfad- und Namensangaben der darin verarbeiteten Quelltexte enthalten müssen. Für die Debuggersitzung ist es also notwendig, daß die im Debugfile gespeicherten Pfadangaben auf dem Client-PC gültig sind. Es empfiehlt sich deshalb, auf dem Rechner zu compilieren, der zum Debuggen verwendet werden soll. Alternativ ist darauf zu achten, daß der Host, auf dem die Quelltexte abgelegt ist, sowohl auf dem Compilations-PC als auch auf dem Debugger-PC korrekt als „Network Drive" mit dem gleichen Laufwerksbuchstaben gemapt ist. Da die Quelltexte der C-Bibliotheken nicht im Lieferumfang des Debuggers beinhaltet sind, werden die einzelnen Module des Projektes vom Debugger als nicht zugreifbar angezeigt.

Der generierte S-Record ist auf das Zielsystem zu transportieren und zu laden.

3.2 Vorbereitung einer UH-PEARL-Debugsession

Zunächst sind alle Module des Projektes so umzustellen, daß die erzeugten S-Records zusätzliche T-Records mit Debuginformationen enthalten. Zu diesem Zwecke ist `MODE=DEBUG;` am Anfang des PEARL-Programms einzugeben. Am Dateiende muß `MODEND DEBUG;` stehen. Weiterhin müssen Linemarker mit `/*+M*/` generiert werden. Die Erzeugung einer Ausgabeliste beim Compilieren ist zwingend, da diese Dateien später im Debugger zur Quelltextanzeige verwendet werden.

```
MODE=DEBUG;
SC=5000;
/*+M*/
MODULE TEST;
...
MODEND DEBUG ;
```

Für jedes UH-PEARL-Modul erfolgt nun einzeln die Umwandlung der erzeugten S-Records in ein binäres Zwischenformat. Das Programm `sr2obj` generiert aus den S-Records Objektdateien, die die gesammelten Debuginformationen eines Moduls enthalten. Dem Programm sind vier Parameter zu übergeben:

- Name des S-Records
- Name der Listingdatei
- Name der zu erzeugenden Objektdatei
- Name des zu erzeugenden, von T-Records befreiten und somit ladbaren S-Records

```
sr2obj d:\test.sr d:\test.lst d:\test.obj d:\test.srl
```

Die erzeugten Objektdateien sind danach mittels des Programmes `obj2dbg` zu einer für den Debugger verständlichen Debugdatei zu linken. Zu diesem Zweck ist eine Linkdatei zu erstellen, die eine Liste sämtlicher zuvor generierter Objektdateien beinhaltet. Dem Programm sind zwei Parameter zu übergeben:

- Name der Linkdatei
- Name der zu erzeugenden Debugdatei

```
obj2dbg d:\test.lnk d:\test.dbg
```

Um diesen Vorgang zu automatisieren, empfiehlt sich die Verwendung des „make"-Utilities:

```
test.dbg: test1.obj test2.obj
   obj2dbg test.lnk test.dbg

*.obj : *.p
   P90 *
   sr2obj68k *.sr D:\output\*.lst *.obj
```

Im Beispiel stellt `P90` den Aufruf einer Batchdatei dar. Endresultat des Linkerlaufes ist eine Debug-Datei, die sämtlichen zur Zeit exportierten Informationen aller PEARL-Module enthält.

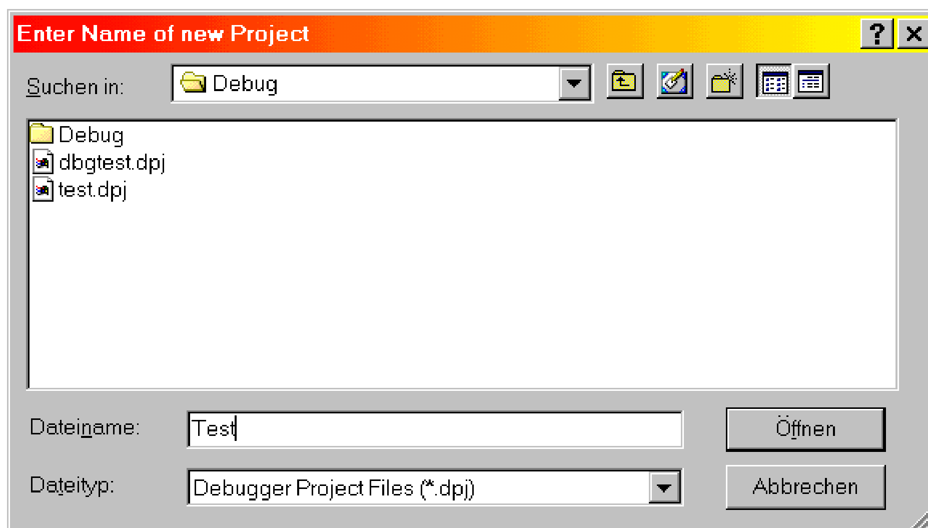
4 Arbeiten mit dem Debugger

4.1 Start einer Debugsession

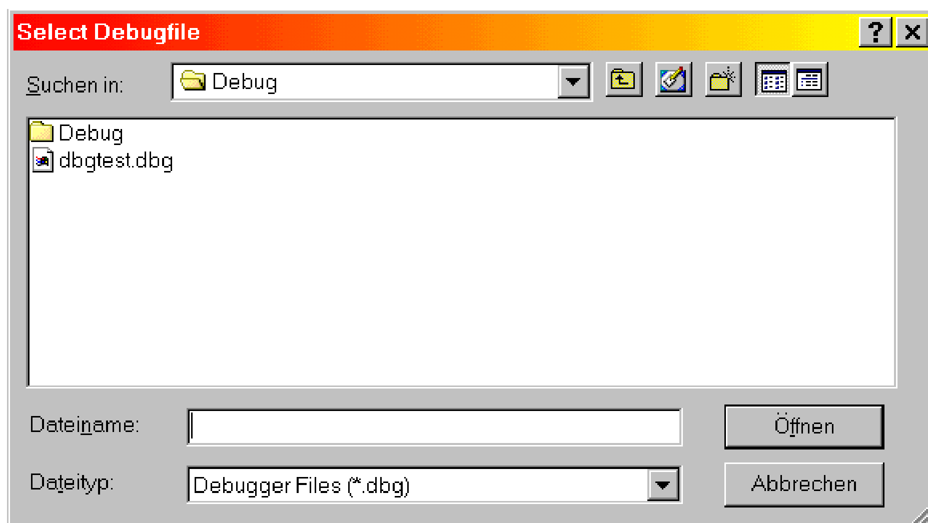
Nach dem Start erwacht der Debugger-Client auf dem PC mit einer spartanischen Bedienerführung, die lediglich folgende Möglichkeiten bietet:

- `Project/New`: Anlegen eines neuen Projektes
- `Project/Open`: Öffnen eines Projektes
- `Project/Exit`: Verlassen des Debuggers

Nach Auswahl von `Project/New` öffnet sich eine Fileselectorbox *Enter Name of new Project*:



Der anzugebende Name kann frei gewählt werden. Es wird eine Datei mit der Endung `.dpj` generiert, in der projektbezogene Informationen abgelegt werden. Wenn Sie eine bereits existierende Datei angeben, so wird diese überschrieben! Danach öffnet sich eine Fileselectorbox *Select Debugfile*, in der eine existierende Debug-Datei ausgewählt werden muß.



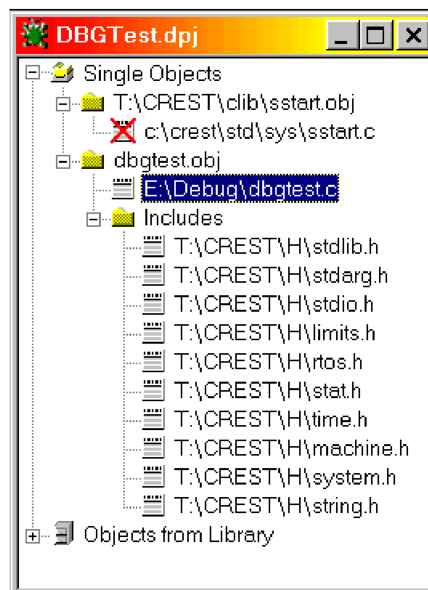
Abschließend öffnet sich noch eine kleine Auswahlbox, in der vorgegeben werden muß, ob es sich bei dem Projekt um `CREST-C` oder `PEARL 90` handelt:

Sind diese Abfragen beendet, so liest der Debugger automatisch die Debug-Datei ein. Je nach Geschwindigkeit des PC's kann der Ladevorgang der Debugdatei durchaus einige Sekunden dauern, da es sich in der Regel um Dateigrößen im (hohen) Megabytebereich handelt.

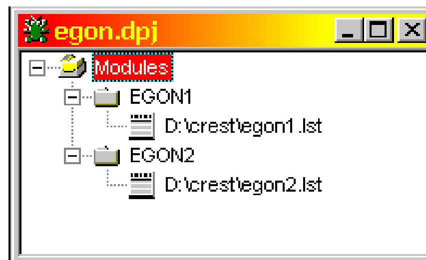
Es baut sich ein Fenster mit einer zusammengefalteten Baumdarstellung auf. Wird dieses Fenster geschlossen, so beendet sich die Debuggersitzung. In der Darstellung sind folgende Punkte im CREST-C-Modus zu erkennen:

- Single Objects
- Objects from Library
- Objects from Reference

Die Einträge *Objects from Library* und *Objects from Reference* sind optional und nur enthalten, wenn beim Linken Objekt-Dateien aus der `.lib`- oder einer `.ref`-Datei ins Programm eingebunden wurden.



Im PEARL90-Modus sieht die Darstellung nur geringfügig anders aus, da hier die Unterteilung in Bibliotheken und Projektdateien keinen Sinn machen würde.



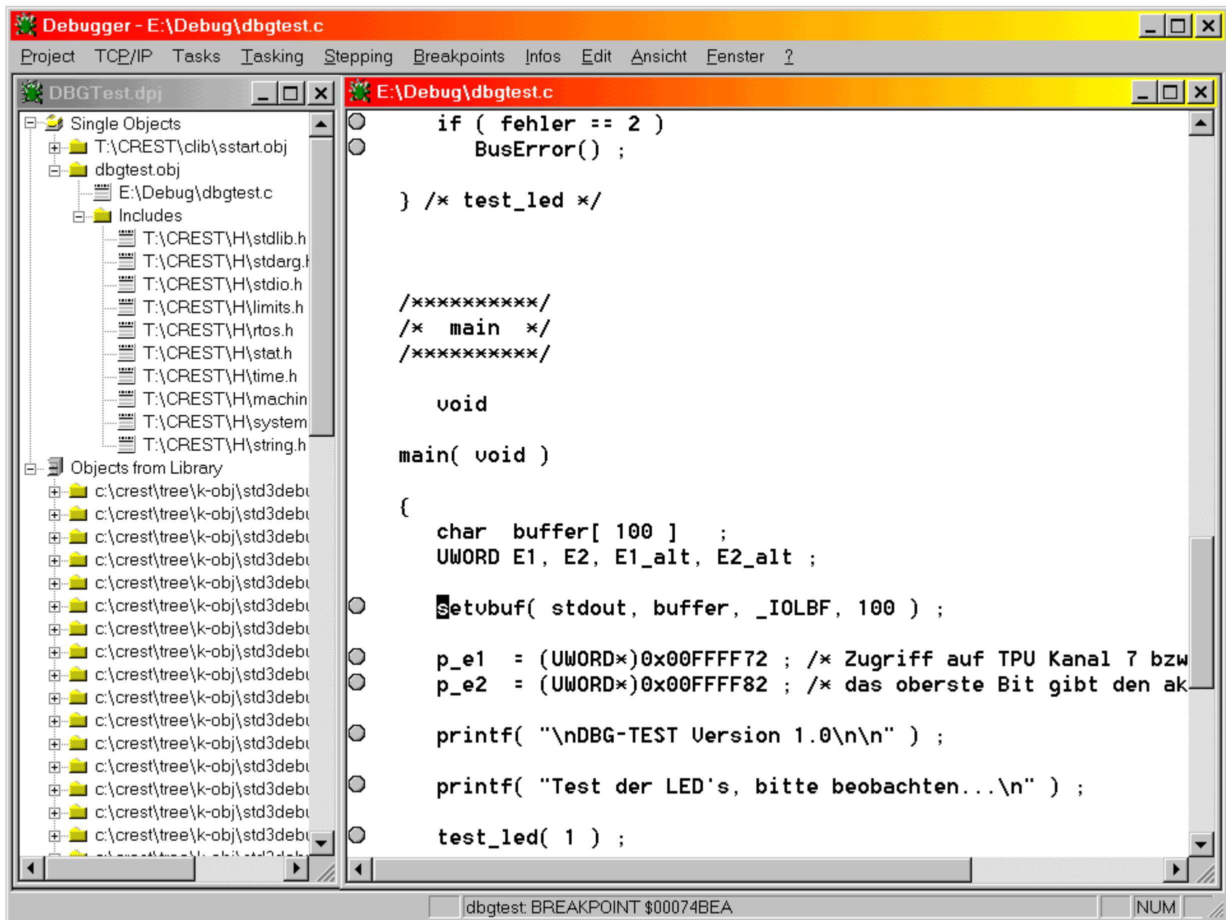
Die Baumdarstellung enthält grundsätzlich die zusammengebundenen Objektdateien in der Reihenfolge, in der sie beim Linken im Linkfile angegeben wurden. Ist das Symbol des Ordners rot durchkreuzt, so handelt es sich um eine Objektdatei, zu der keine Debuginformationen vorliegen und der Ordner läßt sich nicht öffnen – was z.B. der Fall ist, wenn beim Linken eine Nicht-Debug-Bibliothek verwendet wurde oder ohne Compileroption `-z` übersetzte Module eingebunden wurden.

Der Debugger ist zu diesem Zeitpunkt noch Offline, ermöglicht es dem Anwender aber bereits, sich durch die Quelltexte des Projektes zu bewegen.

Aufgeklappt enthalten diese Icons jeweils ein oder zwei weitere Icons. Das Icon mit der Darstellung eines beschriebenen Blattes symbolisiert die Hauptdatei eines Modules – also die jeweilige C- oder PEARL-Datei, die vom Compiler verarbeitet wurde. Ist das betreffende Symbol mit einem roten Kreuz durchgestrichen, so ist der Debugger nicht in der Lage gewesen, den dort angegebenen Quelltext unter dem gespeicherten Pfadnamen zu lokalisieren – was z.B. bei den Bibliotheksroutinen der Fall ist.

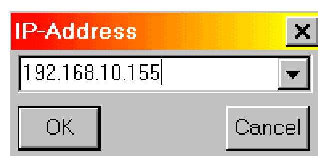
Ein Doppelklick auf dieses Icon öffnet die Datei in einem Read-Only-Editorfenster. Unterhalb befindet sich im CREST-C-Modus noch ein kleiner Ordner, der die Icons sämtlicher includierter Dateien enthält. Auch diese Dateien lassen sich bei Bedarf öffnen und betrachten.

Im nachfolgenden Beispiel wurde die Datei `E:\Debug\dbgtest.c` geöffnet. Deutlich ersichtlich sind die grauen Kugeln in der ersten Spalte, die die möglichen Breakpoints innerhalb des Quelltextes symbolisieren.

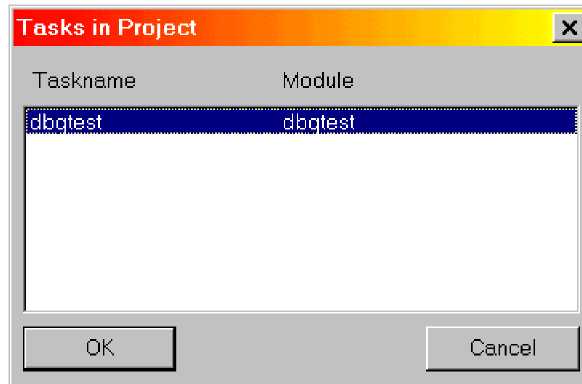


Die Möglichkeiten innerhalb dieses Betriebszustandes sind recht eingeschränkt. Über den TreeView lassen sich weitere Dateien öffnen und über den Menüpunkt *Edit* kann mittels *Search* und *Goto Line* innerhalb einer Datei navigiert werden. Ansonsten verhält sich der „read only“-Editor in Bezug auf seine Bedienung weitestgehend Windows-konform.

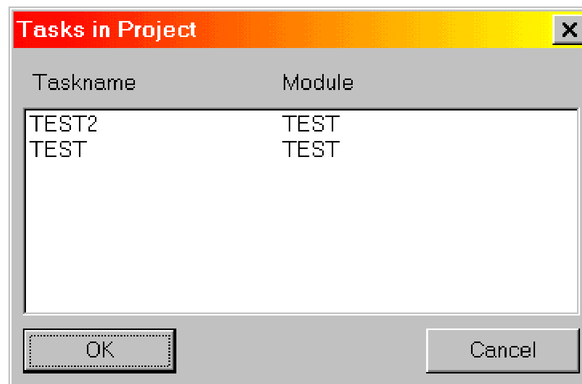
Vor dem physikalischen Verbindungsaufbau ist die IP-Adresse des Zielsystems anzugeben. Dies erfolgt über den Menüpunkt *TCP/IP / Set IP-Address*. Die Angabe kann wahlweise mittels symbolischer Namen (z.B. TESTRECHNER) oder der IP-Schreibweise (z.B. 192.168.10.131) vorgenommen werden.



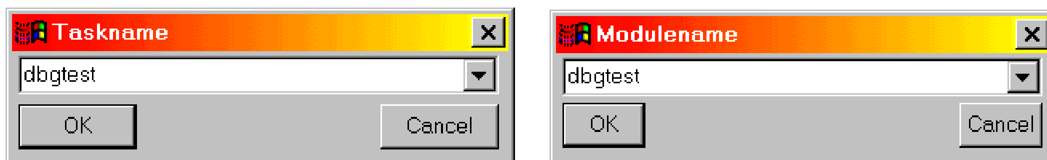
Im nächsten Schritt ist die Angabe der zu überwachenden Task und des Modules, das die Task enthält, notwendig. Hierzu steht unter *Tasks / Tasklist...* eine entsprechende Auswahlbox zur Verfügung, die bei CREST-C-Projekten stets den Namen der „main()“-Task des C-Programmes enthält.



Bei PEARL90-Projekten sind hier alle dem Debugger bekannten Tasks aufgelistet.

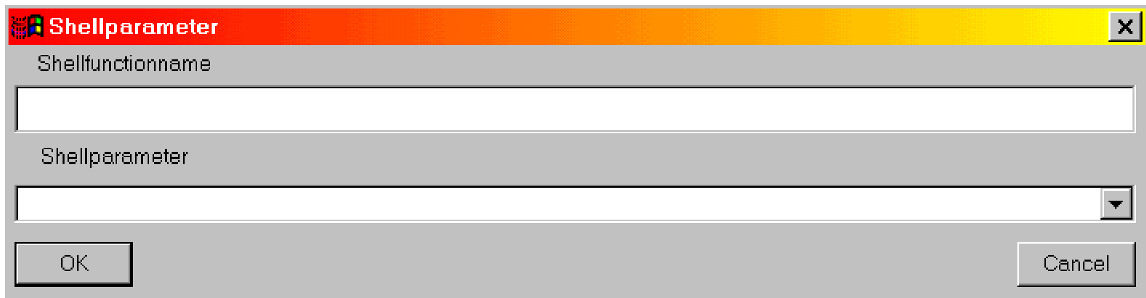


Sollen Tasks entwanzt werden, die in dieser Liste nicht aufgeführt sind (z.B. dynamisch generierte C-Tasks oder Shellmodule), so ist die Eingabe von Task- und Modulname über die Menüpunkte *Tasks / Set Taskname* und *Tasks / Set Modulename* durchzuführen.



Es kann z.Z. nur **exakt eine einzige Task** mit einem Client überwacht werden. Der Debugger-Kernel ist bereits in der Lage, 32 Tasks gleichzeitig zu kontrollieren, aber der Windows-Client nicht!

Soll ein Shell-Modul mit Parameter gestartet werden, kann dies über den Menüpunkt *Tasks / Set Parameter* erreicht werden.



4.2 Debugsession für PEARL-Shellmodule

Bei PEARL-Shellmodulen müssen folgende Einstellungen vorgenommen werden:

1. Taskname setzen auf den PEARL-Bedienbefehlname
2. Modulename setzen auf den ShellModulename
3. Shellfuntionname setzen auf die PEARL-Procedure, die mit dem Bedienbefehl verknüpft ist

z.B: Wir haben ein PEARL Shellmodule folgend kodiert:

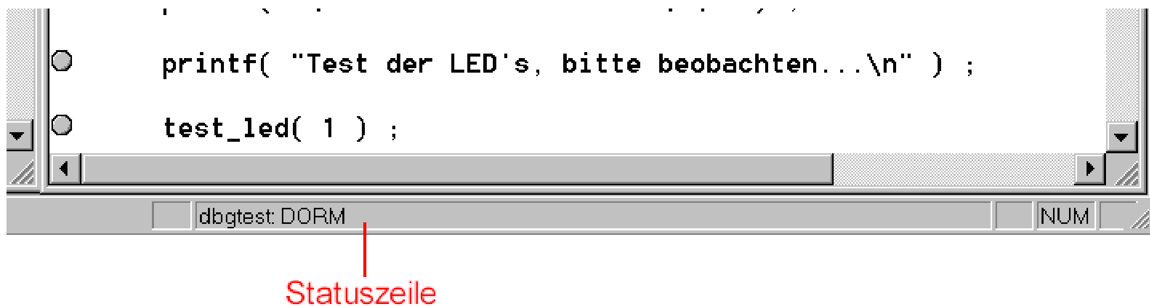
```
SHELLMODULE modem;    set_timeout: 'MODEM_TIMEOUT';
PROBLEM;
...
set_timeout: PROC(( (in,out,err) DATION INOUT ALPHIC IDENT ,
                    textlen FIXED(15) , text  CHAR(255) ) RETURNS( BIT(1) ) ;
```

dann muss in der BOX Taskname *MODEM_TIMEOUT* eingetragen werden,

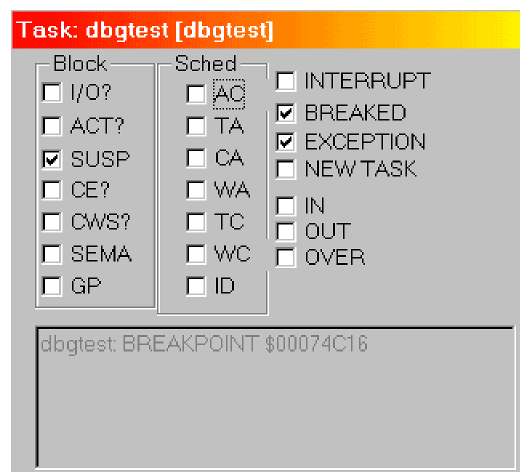
in der BOX Modulename *modem* und bei dem Shellfunctioname *set_timeout*, bei den Shellparametern können die Kommandozeilen-parameter angegeben werden.

4.3 Verbinden mit dem Zielsystem

Als nächster Schritt ist das Einloggen auf dem Zielsystem notwendig, da die meisten weiteren Aktionen nur Online erfolgen können. Das Einloggen erfolgt mittels *TCP/IP / Connect* oder automatisch, wenn Sie z.B. einen Breakpoint setzen wollen. Wird die Task nicht gefunden, so wird auch keine TCP/IP-Verbindung aufgebaut. Wurde die Verbindung erfolgreich hergestellt, so erscheint der Status der Task in der Statuszeile des Debuggers, die Sie am unteren Fensterrand finden:



Unter *Task / Show Taskstate* können Sie einen detaillierten Überblick über die Task bekommen:



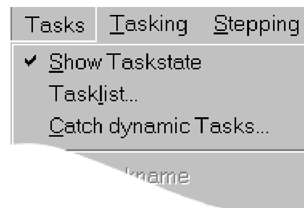
Dargestellt wird das Block- und das Schedule-Byte der Task sowie drei Einträge, in denen vermerkt ist, in welchem Zustand die Task sich aktuell befindet.

➢ Belegung des Block-Bytes

- I/O?: Task wartet auf Beendigung einer I/O-Operation
- ACT?: Task wartet auf Aktivierung
- SUSP: Task ist suspendiert
- CE?: Task wartet auf Zuteilung eines CE‘s
- CWS?: Task wartet auf Workspace
- SEMA: Task ist durch Semaphore blockiert
- GP: Task ist durch den Debugger blockiert

- Belegungen des Schedule-Bytes:
    - AC: auf Aktivierung eingeplant
    - TA: auf zeitliche Aktivierung eingeplant
    - CA: auf zyklische Aktivierung eingeplant
    - WA: Aktivierung bei Event-Eintritt
    - TC: auf zeitliche Fortsetzung eingeplant
    - WC: Fortsetzung bei Event-Eintritt
    - ID: Task ist #IDLE
- Sonstige Anzeigen
    - INTERRUPT: Task wurde vom Debugger blockiert.
    - BREAKED: Task ist auf einen Breakpoint gelaufen.
    - EXCEPTION: Task wurde durch eine Ausnahmebehandlung der CPU gestoppt.
    - IN: Debugger ist im Step-In-Modus
    - OUT: Debugger ist im Step-Out-Modus
    - OVER**:** Debugger ist im Step-Over-Modus

Der Zustand dieser Dialogbox wird etwa alle 500 Millisekunden durch eine Anfrage des Clients beim Debug-Kernel aktualisiert. Um diese Box wieder zu verstecken, ist unter *Tasks / Show Taskstate* das Häkchen zu entfernen.



4.4 Setzen und Löschen von Breakpoints

Die Bearbeitung von Breakpoints erfolgt stets von den Editorfenstern aus. In der linken Spalte sind kleine graue Kugeln an den Textpositionen eingeblendet, zu denen der Debugger Breakpoint-Informationen besitzt. Zeilen ohne Markierung enthalten aus Sicht des Debuggers keinen Code, an dem die Task angehalten werden kann. Mittels der linken Maustaste lassen sich die inaktiven grauen Kugeln in aktive rote Kugeln umwandeln. Alternativ kann dieser Vorgang für die Zeile, in der der Textcursor blinkt, mit der Taste F9 oder über den Menüpunkt *Breakpoints / Toggle Breakpoint* durchgeführt werden. Befindet sich der Debugger noch – oder schon wieder – Offline, so wird zu diesem Zweck eine Verbindung zum Zielrechner aufgebaut.

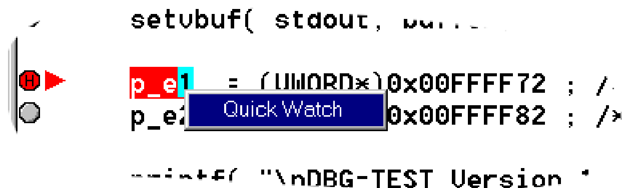
Eine rote Markierung impliziert, daß an der betreffenden Speicherposition auf dem Zielsystem nunmehr ein ILLEGAL-Befehl eingepatcht wurde, der die Task, die diese Position überläuft, zu einer Exception veranlaßt. Laufen nichtüberwachte Tasks über einen derart veränderten Codebereich, so laufen sie einfach weiter!

Sie sollten immer im Hinterkopf behalten, daß der Debugger den Code auf der Zielmaschine beim Setzen von Breakpoints gewollt zerstört! Gerät der Debugger in einen Betriebszustand, in dem die Verbindung zu Zielsystem abbricht und nicht wieder aufgebaut werden kann, so hinterläßt er ein vernichtetes Programm auf der Zielmaschine. Auf derart korruptem Code mit einem frischgestarteten Debugger-Client aufzusetzen, führt allenfalls zu unerwünschten Resultaten. Ebenso seltsam fallen die Resultate aus, wenn bei laufender Debuggersitzung auf dem Zielsystem der gepatchte Programmcode gegen eine unverbrauchte Variante ausgetauscht wird. Der Debugger reagiert hochgradig verwirrt, wenn an den Stellen, an denen er Breakpoints gesetzt zu haben glaubt, auf dem Zielsystem keine ILLEGAL-Befehle mehr zu finden sind.

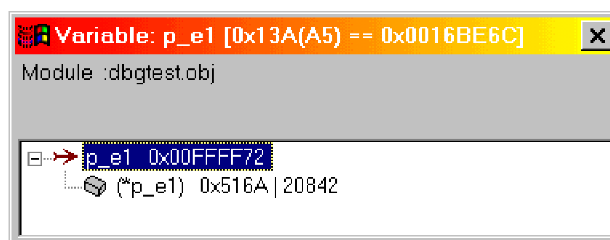
4.5 Betrachten von Variablen

Der Debugger ermöglicht das Betrachten von globalen und funktionslokalen Variablen. Dazu ist die überwachte Task zu stoppen, indem in der zu untersuchenden Funktion ein Breakpoint gesetzt wird. Wird der Breakpoint angelaufen, so erscheint an der entsprechenden Textposition ein kleines rotes Pfeilchen rechts neben dem Breakpoint. In der Statuszeile erscheint die Meldung `BREAKPOINT` *`adresse`*.

Nun kann mittels der Maus die gewünschte Variable markiert werden. Dazu genügt es, den Textcursor auf den Variablennamen zu setzen und mittels der rechten Maustaste aus dem sich öffnenden Pulldown-Menü den *Quick-Watch*-Dialog zu aktivieren:

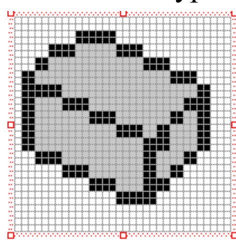


Es erscheint eine Dialogbox, in der die Variable angezeigt wird. Bei Nicht-Basisdatentypen wird dazu eine zusammengeklappte Baumdarstellung aufgebaut.

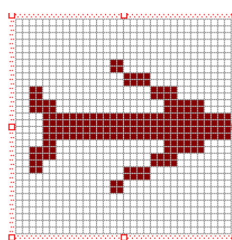


In der Kopfzeile wird die Ablageadresse der Variablen (hier mit Offset `0x13A` zu Register `A5`) sowie die absolute Speicheradresse (hier `0x0016BE6C`) bzw. das Register angegeben.

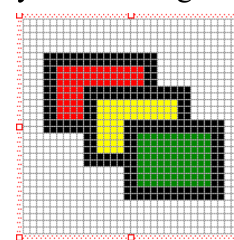
Die verschiedenen Datentypen werden hierzu mit unterschiedlichen Symbolen dargestellt:



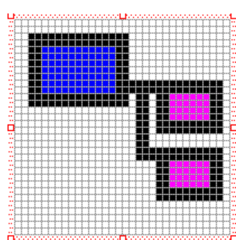
Basisdatentyp



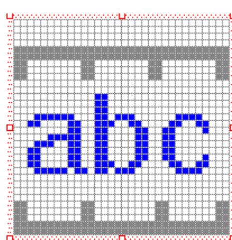
Pointer



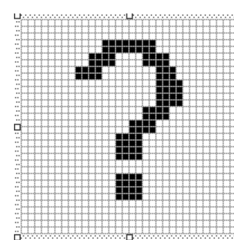
Array



Strukturen und Unions



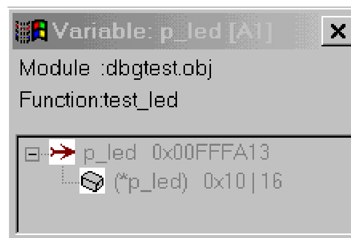
PEARL90-CHAR's



Unbekannter Datentyp

Durch Öffnen eines Zweiges erfolgt z.B. bei Pointern eine Dereferenzierung und eine typenkorrekte Darstellung des Wertes, auf den der Pointer verweist. Bei Strukturen und Unions werden die Strukturmitglieder dargestellt. Handelt es sich dabei nicht um Basisdatentypen, so ist dieser Vorgang entsprechend zu wiederholen. Bei mehrdimensionalen Feldern ist der Baum ebenfalls solange an den gewünschten Positionen zu öffnen, bis Basisdatentypen dargestellt werden können.

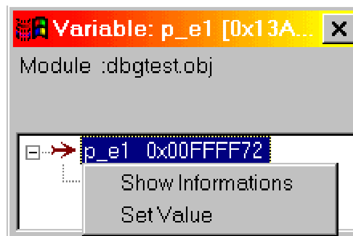
Globale Variablen lassen sich im gesamten Quelltext auf diese Art betrachten. Funktionslokale Variablen können nur in der Funktion angezeigt werden, in der sich die gestoppte Task aktuell befindet, da der Speicherbereich, in dem sich die Variablen befinden, gar nicht zur Verfügung steht. Wird eine lokale Variable ungültig, so wird das Fenster grau dargestellt:



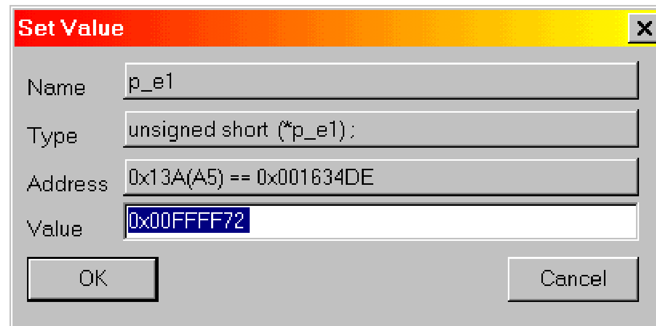
Wird die Variable wieder gültig, weil z.B. die Prozedur ein zweites Mal aufgerufen wird, wird auch die Variable wieder mit ihrem dann gültigen Wert angezeigt.

4.6 Setzen von Variablenwerten

Das Setzen von Variablenwerten erfolgt ebenfalls aus dem „*Quick-Watch*“-Dialog heraus. Hierzu ist das betreffende Objekt mit der rechten Maustaste zu selektieren.



Nach Auswahl von *Set Value* aus dem Popup öffnet sich ein kleiner Dialog, in dem Name, Typ, physikalische Speicherposition und der aktuelle Dateninhalt angezeigt wird:

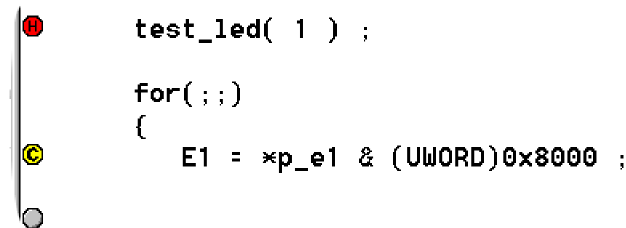


Wird die Box mittels *OK* verlassen, so wird der eingegebene Wert zunächst auf dem Zielsystem geschrieben und anschließend zur Kontrolle zurückgelesen. Bei Auswahl des Menüpunktes *Show Informations* öffnet sich ein identischer Dialog, der jedoch keinerlei Eingabemöglichkeiten besitzt.

4.7 Watchpoints

Watchpoints dienen der Anzeige von Variablen, ohne die Task länger anhalten zu müssen. Läuft die Task über einen Watchpoint, werden alle Fenster zur Anzeige von Variablen upgedated. Sie erhalten also einen konsistenten Überblick über Ihre Variablen, ohne einen Breakpoint setzen zu müssen! Die Task wird nur solange angehalten, bis alle benötigten Werte vom Zielsystem übertragen sind. Nach dem Überlaufen wird der Watchpoint automatisch gelöscht. Damit können Sie den Debugger auch an der Maschine einsetzen, wo ein Unterbrechen einer Task oft nicht möglich ist.

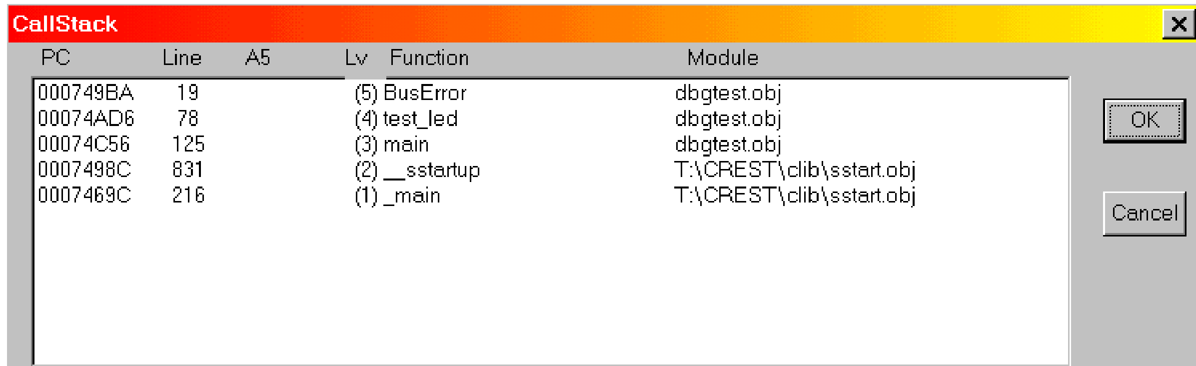
Um den Watchpoint zu setzen, klicken Sie mit der rechten Maustaste auf einen Breakpoint. Der Watchpoint wird gelb dargestellt:



Um den Überblick zu behalten, kann es nur genau einen Watchpoint geben. Versuchen Sie einen weiteren anzulegen, erhalten Sie eine Fehlermeldung. Zum Löschen eines Watchpoints klicken Sie einfach wieder mit der rechten Maustaste auf den gelben Watchpoint.

4.8 Post Mortem Debugging

Wenn Ihnen eine Task mit einer Fehlermeldung – z.B. Bus Error – liegen bleibt, können Sie den Debugger noch einsetzen, um den Fehler zu analysieren. Über den Call Stack sehen Sie, von wo die Prozedur aufgerufen wurde:



Die oberste Zeile gibt die aktuelle Prozedur der Task aus, d.h. an dieser Stelle steht die Task. Wenn Sie eine Zeile im Call Stack mit der Maus markieren und dann OK drücken, wird automatisch die zugehörige Stelle im Quelltext geöffnet.

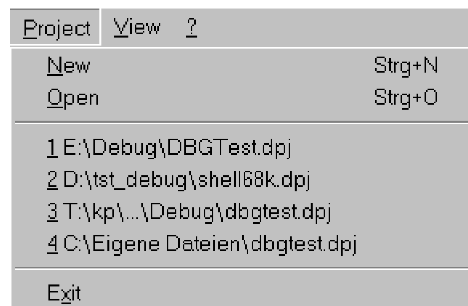
Sie können dem Call Stack also entnehmen, dass Ihre Task in der Prozedur `BusError` steht. Diese Prozedur wurde von `test_led` aufgerufen. `test_led` wiederum wurde aus `main` heraus aufgerufen. `__sstartup` und `_main` stammen aus dem Start-Up für C-Programme und können von Ihnen ignoriert werden.

Weiterhin können Sie sich jetzt noch Variablen ansehen, so dass es ein leichtes sein sollte den Fehler zu finden.

5 Beschreibung der Menüpunkte

Aufbau der Menüzeile:

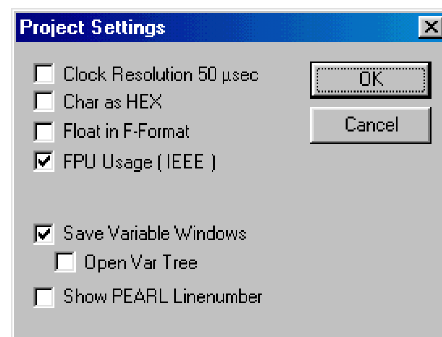
5.1 Menü Project



Dies ist das Menü *Projekt*, bevor eine Debug-Sitzung gestartet wird.

- *New*: Es wird ein neues Debug-Projekt angelegt.
- *Open*: Es wird ein bestehendes Projekt geöffnet
- *1-4*: Liste der letzten 4 Debug-Sitzungen
- *Exit*: Verlassen des Debuggers

Nachdem ein Projekt geöffnet wurde, bleibt vom obigen Menü noch der Punkt *Exit* erhalten und es erscheint ein neuer Menüpunkt *Settings*. In dieser Box können Parameter des Debuggers angepasst werden. So kann bei einem 20 KHz System die Darstellung der CLOCK und Duration Variablen angepasst werden. Oder auch die Darstellung der Float Variablen presettet werden.



Weiterhin gibt es hier Knöpfe zur Einstellung, ob die in einer Sitzung geöffneten Variablenwindows beim Erneuten Start wieder rekonstruiert werden sollen und auch Unterselektionen berücksichtigt werden sollen.

Es können die PEARL-Zeilennummern eingeblendet werden.

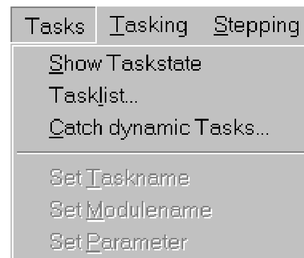
Dieses Menü ist nur zugänglich, wenn das Projektbaumfenster aktiv ist. Alle Einstellungen werden in diesem Projekt gespeichert.

5.2 Menü TCP/IP

Hier wird die Verbindung zum Zielsystem ein- und hergestellt:

- *Set IP-Adress*: Setzt die IP-Adresse des Zielsystems.
- *Connect*: Verbindung zum Zielsystem herstellen.
- *Disconnect*: Verbindung zum Zielsystem trennen.

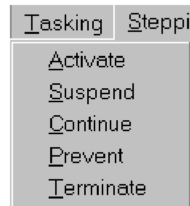
5.3 Menü Tasks



Das Menü Tasks zeigt sich zweigeteilt. Wenn keine Verbindung zum Zielsystem besteht, kann der untere Teil des Menüs genutzt werden. Besteht Verbindung zum Zielsystem ist der obere Teil des Menüs aktiv.

- *Show Taskstate*: Toggelt die Anzeige des Taskstatus-Fensters.
- *Tasklist...*: Zeigt alle in diesem Projekt verfügbaren Tasks an.
- *Catch dynamic Task...*: z.Z. nicht zu benutzen!
- *Set Taskname*: Setzt den Namen der zu überwachenden Task.
- *Set Modulname*: Setzt das Modul der zu überwachenden Task.
- *Set Parameter*: Ermöglicht den Start einer Shelltask mit Parametern.

5.4 Menü Tasking



Über diesen Menüpunkt ist es möglich, das Tasking der überwachten Task zu beeinflussen.

- *Activate*: Aktiviert die zu debuggende Task.
- *Suspend*: Suspendiert die zu debuggende Task. Sollte nur verwendet werden, wenn die zu debuggende Task in den Zustand SUSP überführt werden soll.
- *Continue*: Setzt die zu debuggende Task fort.
- *Prevent*: Plant die zu debuggende Task aus. Es existieren innerhalb des Debuggers allerdings z.Z. keine Benutzerschnittstellen um neue Einplanungen vorzunehmen.
- *Terminate*: Terminiert die zu überwachende Task.

5.5 Menü Stepping

Stepping Breakpoints Infos Ed

Run	F5
Step Over	F8
Step Into	F11
Step Out	Shift+F11
Break Execution	
Continue Execution	

Hier sind Befehle realisiert, um die überwachte Task unter Debuggerkontrolle zu stoppen und kontrolliert fortzusetzen. Die angegebenen Funktionstasten können ebenfalls genutzt werden.

- *Run*: Eine Task, die mittels „Break Execution“ gestoppt wurde oder auf einen Breakpoint gelaufen ist, kann mit diesem Befehl fortgesetzt werden. Befindet sich die Task nicht in einem vom Debugger kontrollierten Betriebszustand, so wird der Befehl schlicht ignoriert.
- *Step Over*: Die unter Debuggerkontrolle blockierte Task wird angewiesen, bis zum nächsten möglichen Breakpoint innerhalb der aktuellen Funktion zu laufen. Der Befehl dient dazu, Unterprogrammaufrufe mit voller Prozessorgeschwindigkeit auszuführen. Ein Breakpoint muß nicht extra im Editorfenster markiert werden. Der Befehl bewirkt vielmehr, daß der Debugger alle möglichen Breakpoints selbstständig aktiviert – die grauen Breakpointmarkierungen im Editorfenster färben sich dann grün. Die Task läuft mit voller Geschwindigkeit bis zur nächsten Position, an der ein aktiver Breakpoint (grün oder rot) gefunden wird. Dieser Betriebszustand wird automatisch beendet, wenn ein roter Breakpoint angelaufen oder die aktuelle Funktion verlassen wird.
- *Step Into*: Die Task wird angewiesen, den nächsten bekannten Breakpoint anzulaufen. Der Befehl dient dazu, in Funktionen einzutauchen zu können. Es werden in der lokalen Funktion alle Breakpoints und bei jeder anderen Funktionen des Projektes der Einstiegs-Breakpoint gesetzt. Dieser Betriebszustand wird automatisch beendet, wenn ein roter Breakpoint angelaufen oder ein dem Debugger bekannter Breakpoint erreicht wurde.
- *Step Out*: Der Debugger setzt einen Breakpoint an der Rückkehradresse der aktuellen Funktion der zu überwachenden Task. Anschließend wird die Blockierung der Task aufgehoben.
- *Break Execution*: Stoppt die zu überwachende Task. Realisiert wird dies durch das Setzen des General Purpose-Bits im Block-Byte der Task. Da dieses Bit durch reguläres RTOS-UH-Tasking nicht beeinflußt werden kann, bleibt die überwachte Task unter Debugger-Kontrolle blockiert.
- *Continue Execution*: Löscht das General Purpose-Bit und stellt somit wieder den normalen Betriebszustand der überwachten Task her – was nicht zwangsweise bedeutet, daß die Task weiterläuft. Sind weitere Bits im Blockbyte der Task gesetzt, so bleibt die Task eben weiterhin blockiert.

5.6 Menü Breakpoints

Mit Hilfe dieser Menüpunkte können Breakpoints beeinflußt werden. Direkt können Breakpoints auch mit der Maustaste ge- bzw. zurückgesetzt werden.

- *Toggle Breakpoint*: Bewirkt, daß eine graue oder grüne Breakpointmarkierung an der aktuellen Cursorposition innerhalb des Editors in eine rote Markierung umgewandelt wird bzw. rote Markierungen wieder auf inaktive graue zurückgesetzt werden.
- *Clear Breakpoints*: Der Debugger versetzt den gepatchten Code wieder in den Originalzustand. Es werden alle Breakpoints des gerade aktiven Editorfensters zurückgenommen. Das Schließen eines Editorfensters bewirkt eine identische Aktion.
- *Clear All Breakpoints*: Der Debugger versetzt den gepatchten Code wieder in den Originalzustand. Es werden alle Breakpoints aller Editorfenster zurückgenommen. Das Beenden der Debugsitzung bewirkt eine identische Aktion.

5.7 Menü Infos

Hier können Informationen über die Task – zum Teil automatisch – angefordert werden.

- *Read Register*: Liest die Register der überwachten Task aus und zeigt die Inhalte in einer Dialogbox an.
- *Callstack*: Dieses Kommando bewirkt, daß für eine unter Debuggerkontrolle blockierte Task der Aufrufstack ermittelt wird. Aus dem aktuellen PC der Task wird ermittelt, in welcher Funktion sich die Task befindet und von welcher Funktion diese wiederum aufgerufen wurde. Es öffnet sich eine Dialogbox, in der selektiert werden kann, welche Position im Aufrufstack näher betrachtet werden soll.
- *Auto Update ModVar*: Diese Funktion ist z.Z. nur für PEARL90 Programme verfügbar. Fenster, in denen Modulvariablen angezeigt sind, werden automatisch ca. alle 2 sec aktualisiert. (Dieses Einstellung wird im Projekt gespeichert).
- *Update Var-Wind.*: Alle Variablen-Fenster werden upgedated. Vorsicht, Sie dürfen diese Funktion nur aufrufen, wenn die Task steht! Sonst kann Ihr Debugger abstürzen, da er versuchen könnte Speicher zu betrachten, der nicht mehr der Task zugeordnet ist.

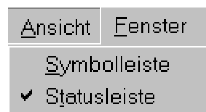
  Wenn die Task auf einen Breakpoint gelaufen ist, werden die Variablen-Fenster automatisch upgedated. Laufen aber auf dem Zielsystem noch andere Task, die z.B. globale Variablen verändern, können Sie sich den aktuellen Zustand dieser Variablen holen lassen.

5.8 Menü Edit

Das Menü Edit ist sehr kurz gehalten, da es sich nicht um einen Editor handelt, sondern der Debugger nur Dateien anzeigen soll.

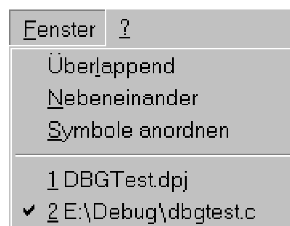
- *Search*: Sucht nach einer Zeichenkette innerhalb des aktiven Editorfensters.
- *Goto Line*: Positioniert das Editorfenster auf eine bestimme Zeile.
- *Task Proc List* : zeigt alle im Projekt verfügbarem Tasks und Prozeduren an. Durch einen Doppelklick auf den jeweiligen Eintrag wird direkt in das Editorfenster an der jeweiligen Position positioniert.

5.9 Menü Ansicht



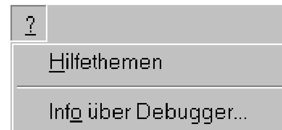
Hier kann z.Z. nur die Statuszeile ein-/ausgeblendet werden.

5.10 Menü Fenster

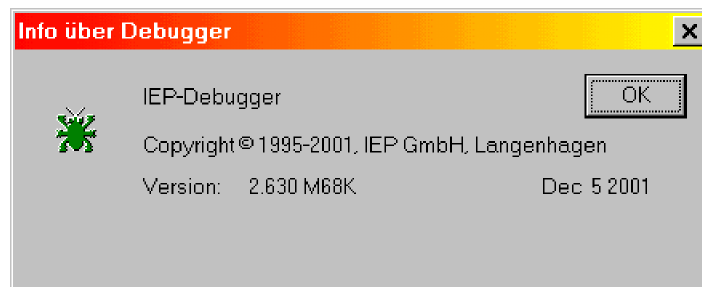


Das Menü Fenster enthält die unter Windows üblichen Einträge.

5.11 Menü ?



- *Hilfethemen*: Eine Online-Hilfe ist z.Z. leider nicht verfügbar.
- *Info über Debugger...*: Sie erhalten eine Info Box, die das Erstellungsdatum, die Versionsnummer des Debuggers angibt:



6 Ausblick

Der Debugger befindet sich inzwischen in einem Zustand, wo man mit einigen RTOS-Kenntnissen sinnvolle Debuggersitzungen durchführen kann. Er ist weder narrensicher, perfekt, vollständig noch fehlerfrei, wird jedoch in Absprache mit den Anwendern ständig weiterentwickelt.

Folgende Punkte stehen aktuell auf der Liste der noch zu implementierenden Features:

- Einführung eines WATCH-Fensters
- Positionieren auf Variablendefinitionen
- Automatischer Download des zu debuggenden Programmes