

EZStubs Entwicklungsumgebung

Einführung

Peter Ulbrich, Martin Hoffmann

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

25. Oktober 2010

Allgemein

Vorbereitung

SVN-Repository
Verzeichnisstruktur
Makefiles

Grundlagen

Startup
Scheduler
Unterbrechungssynchronisation

Testfälle

Implementierung
Ablaufkontrolle
Timer

Fehlersuche

Tutorial

Intention

- ▶ Familie von Echtzeitbetriebssystemen
- ▶ etwas komplexere Klassenhierarchie
 - ▶ trotzdem: Fokus eher auf Verständlichkeit denn auf Effizienz
- ▶ Entwickelt für diese Übungen
- ▶ Verfügbar für
 - ▶ Lego RCX (H8/3297)
 - ▶ Gameboy Advance (ARM7)
 - ▶ Nintendo DS Lite (ARM9)
- ▶ Implementierung hat noch die ein oder andere Schwäche

Hinweis

ergänzende Informationen findet man hier:

http://www4.informatik.uni-erlangen.de/Lehre/WS10/V_EZS/Uebung/

Initialisierung des SVN-Repositories

1. Das Repository auschecken:

```
faii48a:~> svn co https://www4.informatik.uni-erlangen.de:8088/i4ezs/gruppe0/ ezstubs
A ezstubs/branches
A ezstubs/tags
...
Checked out revision 1.
```

2. In das Verzeichnis ezstubs/trunk/aufgabe1 (Ü) bzw. ezstubs/branches/time_triggered (EÜ) wechseln

```
faii48a:~> cd ezstubs/trunk/aufgabe1/
faii48a:~/ezstubs/trunk/aufgabe1>
```

3. Die Datei gu_vorgabe_1.tar.gz (Ü) bzw. vorgabe_1.tar.gz (EÜ) entpacken:

```
faii48a:~/ezstubs/trunk/aufgabe1> tar xzf gu_vorgabe1.tar.gz
```

Initialisierung des SVN-Repositories (Forts.)

4. Dateien dem Repository hinzufügen:

```
faii48a:~/ezstubs/trunk/aufgabe1> svn status .
? debug/arch_testing.h
...
faii48a:~/ezstubs/trunk/aufgabe1> svn add ...
A debug/arch_testing.h
...
faii48a:~/ezstubs/trunk/aufgabe1> svn commit . -m 'Dateien aus vorgabe_1.tar.gz'
Adding debug/arch_testing.h
...
Transmitting file data .....
Committed revision 2.
faii48a:~/ezstubs/trunk/aufgabe1>
```

Hinweis
 Detaillierte Informationen zum Umgang mit SVN findet man unter:
<http://svnbook.red-bean.com/>

Das Verzeichnis `ezstubs` enthält folgende Unterverzeichnisse:

Name	Beschreibung
debug	Bibliothek zum Schreiben von Testfällen. Enthält z.B. Makros und Funktionen zum gezielten Auslösen von Unterbrechungen, zur Zeitmessung oder um eine bestimmte Zeitspanne zu warten.
devices	Gerätetreiber. Hier finden sich alle Gerätetreiber, sowohl abstrakte Geräte, die eine einheitliche Schnittstelle zur Anwendung ermöglichen, als auch die Abbildung von real existierender Peripherie auf Software-Konstrukte.
gen	In diesem Verzeichnis landet alles, was in durch einen Aufruf von <code>make</code> erzeugt wird - Objektdateien, Doxygen-Dokumentation ...
infra	Typdefinitionen für EZStubs und andere kleine Helfer, die in den Implementierungsdateien von EZStubs verwendet werden, sich aber nicht sinnvoll zu anderen Modulen zuordnen lassen.
interrupt	Unterbrechungsbehandlung und Unterbrechungssynchronisation.
make	Alles was nötig ist, um EZStubs zu bauen.
object	Eine kleine Hilfsbibliothek für EZStubs, enthält z.B. eine verkettete Liste.
shutdown	Alles was EZStubs herunterfährt.
startup	Alles was EZStubs zum starten benötigt.
tests	Testfälle
thread	Der Thread-Abstraction-Layer. Hier findet man die Fadenimplementierung, den Dispatcher, den Scheduler ...

Pfade in den Makefiles überprüfen/anpassen

In den Makefiles müssen drei Pfade korrekt gesetzt sein. Die passenden Werte für den CIP-Pool sind:

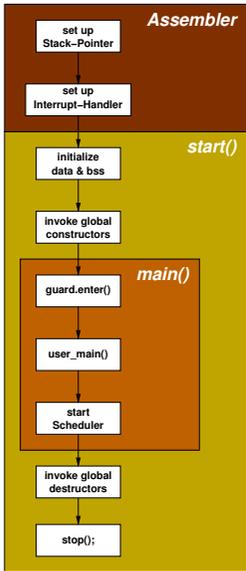
Dateiname	Variable	Wert
make/variables_arm.mk	TOOL_PATH	/proj/i4ezs/tools/gnuarm
make/variables_arm_nds.mk	GBA_DIR	/proj/i4ezs/tools/GBA
make/variables_arm_nds.mk	DESMUME_DIR	/proj/i4ezs/tools/desmume

EZStubs administrative Targets

Das EZStubs-Makefile implementiert folgende administrative Targets:

Target	Beschreibung
help	Gibt die verfügbaren Make-Targets zusammen mit einer Erläuterung auf der Konsole aus (entspricht in etwa dieser Tabelle).
all	Übersetzt den Quelltext von EZStubs und erzeugt die EZStubs-Bibliothek.
abgabe	Eine Aufgabe abgeben - die Aufgabe wird durch die Variable <code>EXERCISE = aufgabex</code> , <code>x = 1,2,3,4,5</code> bestimmt.
doxygen	Erzeugt die HTML-Quelltext-Dokumentation.
showtests	Zeigt die verfügbaren Testfälle an.
buildtests	Übersetzt und bindet alle verfügbaren Testfälle.

Startup



1. SP Register initialisieren:
 - ▶ Interrupt Stack & System Stack
 - ▶ Startup: nutzt den Interrupt Stack
2. bss & data initialisieren
 - ▶ bss mit 0 beschreiben
 - ▶ initialisierte Variablen kopieren
3. globale Konstruktoren ausführen
4. kritischen Abschnitt betreten
5. benutzerdefinierte main-Funktion
6. Scheduler starten
7. globale Destruktoren ausführen
8. Ende (z.B. warten oder reset)

Allgemein

Der Scheduler ...

- ▶ verwaltet das Betriebsmittel CPU
- ▶ kann verschiedene Strategien bzw. Verfahren verwenden
 - ▶ zeit-/ereignisgesteuert
 - ▶ deterministisch/probabilistisch
 - ▶ ...
- ▶ muss vom System aktiviert werden (sog. **Points of Rescheduling**)
 - ▶ diese entscheiden über die Verdrängbarkeit von Fäden
- ▶ hat (in EZStubs) zwei verschiedene Schnittstellen
 - ▶ Benutzerschnittstelle
 - ▶ Systemschnittstelle

Benutzerschnittstelle

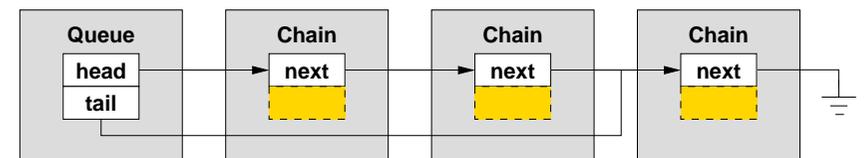
- ▶ hängt von der konkreten Implementierung des Schedulers ab
- ▶ z.B. `Schedule_Table_Scheduler`
 - ▶ Ablauftabelle setzen
 - ▶ Ablauftabelle wechseln
- ▶ z.B. `Multilevel_Queue_Scheduler`
 - ▶ Faden aktivieren
 - ▶ Faden beenden
 - ▶ Kontrolle über den Prozessor abgeben
 - ▶ ...

Hinweis

Die Systemschnittstelle ist Thema in Aufgabe 1

Warum Unterbrechungssynchronisation?

- ▶ Unterbrechungen (nicht Traps!)
 - ▶ treten asynchron auf
 - ▶ sind nicht vorhersagbar in Häufigkeit und Zeitpunkt
- ▶ **Problem:** Wie soll der Zugriff auf gemeinsame Daten stattfinden?
- ▶ Beispiel: verkettete Liste:



Verkettete Liste

Klasse Chain, Klasse Queue

```

class Chain {
public:
    Chain* next;
};

class Queue {
protected:
    Chain* head;
    Chain** tail;
public:
    Queue() { head = 0;
              tail = &head; }
    void enq(Chain* item);
    Chain* deq();
};
    
```

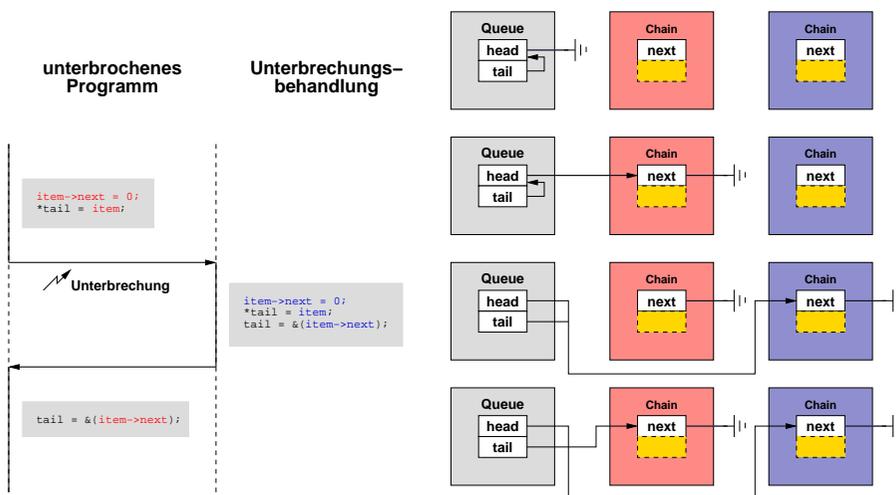
Implementierung

```

void Queue::enq(Chain* item) {
    item->next = 0;
    *tail = item;
    tail = &(item->next);
}

Chain* Queue::deq() {
    Chain* item;
    item = head;
    if (item) {
        head = item->next;
        if (!head) tail = &head;
        else item->next = 0;
    }
    return item;
}
    
```

Verkettete Liste - enqueue () überlappt enqueue ()



ISR-/DSR-Synchronisation

- ▶ Lösung: Aufteilung der Unterbrechungsbehandlung
- ▶ ISR (Interrupt Service Routine)
 - ▶ werden *asynchron* zum Programm ausgeführt
 - ▶ dürfen keine gemeinsamen Datenstrukturen modifizieren
 - ▶ können die Ausführung einer DSR anfordern
- ▶ DSR (Deferred Service Routine)
 - ▶ werden *synchron* zum Programm ausgeführt
 - ▶ dürfen gemeinsame Datenstrukturen modifizieren
- ▶ Ausführung der DSRs kann verhindert/verzögert werden

Unterbrechungsbehandlung

- ▶ Klasse Gate: Grundlage für die Implementierung

Klasse Gate

```

class Gate {
public:
    // ISR
    // Rueckgabewert true: fuehre DSR aus
    // Rueckgabewert false: fuehre DSR nicht aus
    virtual bool isr() = 0;
    // DSR
    virtual void dsr() = 0;
    // Setze Interrupt-Quelle zurueck
    virtual void acknowledge() = 0;
};
    
```

Synchronisation

- Klasse `Guard` sichert kritische Abschnitte

Klasse `Guard`

```
class Guard {
public:
    // Aufrufe von enter() und leave() muessen
    // immer paarweise erfolgen

    // Betrete einen kritischen Abschnitt
    // unterbinde Ausfuehrung von DSRs
    void enter();
    // Verlasse einen kritischen Abschnitt
    // Fuehre anstehende DSRs aus
    void leave();
}
```

Synchronisation - Scoped Locking

Klasse `Secure`

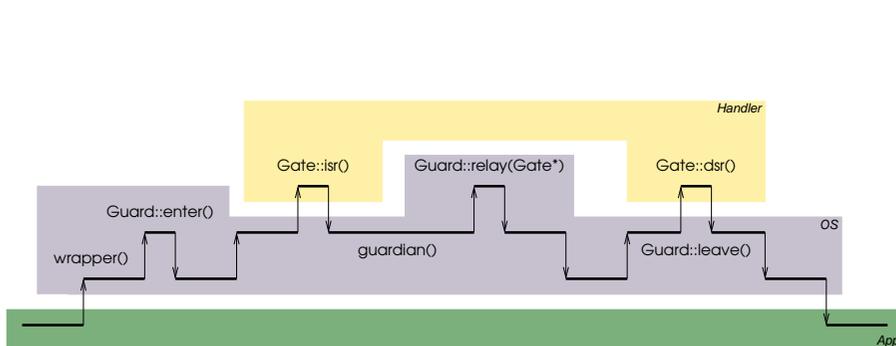
```
class Secure {
public:
    Secure() { guard.enter(); }
    ~Secure() { guard.leave(); }
};
```

Verwendung

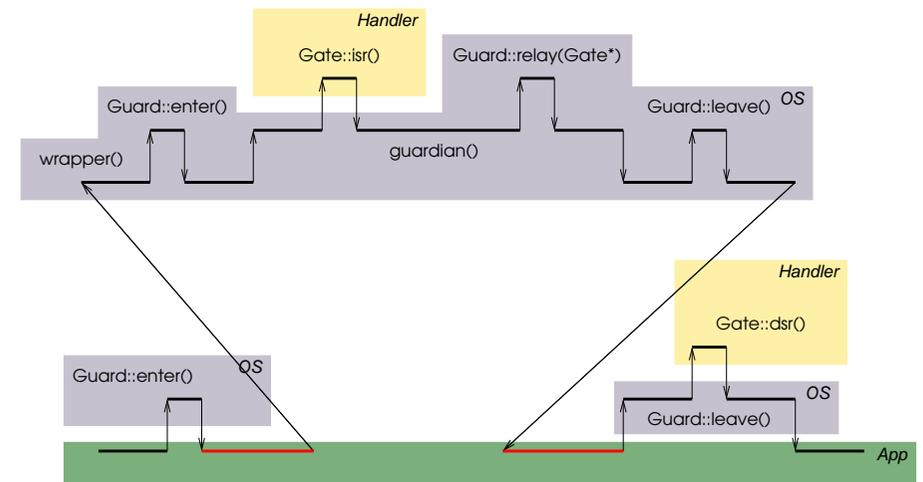
```
void do_something_critical() {
    Secure secure;

    // do critical stuff ...
}
```

Unterbrechung - Ablauf



Unterbrechung - Ablauf (secured)



Warum? Wie? Wann? Wo?

- ▶ Funktioniert das Programm?
 - ▶ Testen ...
 - ▶ andere Verifikationsmethoden (formal, ...)
- ▶ Testfälle sind **einfach!**
 - ▶ feingranular
 - ▶ testen nur **eine** Funktion
- ▶ eure Testfälle unterstützen den Entwickler
 - ▶ stehen **bald** zur Verfügung
 - ▶ sind hierarchisch strukturiert
- ▶ unsere Testfälle sind *Abnahmetests*
 - ▶ Testen das *fertige Produkt*
 - ▶ sind u.U. relativ komplex

Was ist ein Testfall?

Ein Testfall ...

- ▶ ... ist eine kleine Anwendung für das EZStubs-Betriebssystem
- ▶ ... ist eine Ansammlung von Funktionen, die vom Betriebssystem aufgerufen werden:
 - ▶ die Methode `void action()` eines Fadens
 - ▶ die Methoden `bool isr()` und `void dsr()` einer Unterbrechungsbehandlung
 - ▶ Callback-Funktionen von Alarmen
- ▶ ... beginnt immer in der Funktion `user_main()`

Beispiel

```
// evtl. benoetigte Header werden eingebunden
#include "thread/guarded_scheduler.h"
#include "thread/thread.h"
#include "debug/testing.h"
#include "thread/schedule_table.h"

// Aufzeichnung des Verlaufs wird initialisiert
char test_sequence[4];
const char* right_sequence = "aba";
volatile int counter = 0;
...
```

Beispiel (Forts.)

```
...
// Fadenimplementierung fuer den Testfall, enthaelt
// die Anwendungslogik des Testfalls
class MyThread : public Thread
{
private:
    char thread_char;
public:
    MyThread(char c) : thread_char(c) {};
    void action()
    {
        test_sequence[counter++] = thread_char;
        scheduler.exit();
    };
};
...
```

Beispiel (Forts.)

```

...
// der Checker_Thread ueberprueft abschliessend die Sequenz
class CheckerThread : public Thread
{
public:
    CheckerThread() {};
    void action()
    {
        if (!check_sequence(right_sequence, test_sequence))
        {
            Panic("Test failed!");
        }
        else
        {
            TestOK("Test successful!");
        }
        scheduler.exit();
    }
};
...

```

Beispiel (Forts.)

```

...
// Stack allokieren, Ablauftabelle und Faeden anlegen
static unsigned int stack[256];
Schedule_Table<4> table(&stack[256], 100000);
MyThread T1('a');
MyThread T2('b');
CheckerThread Tchecker;

// Faeden hinzufuegen, Tabelle uebergeben und Scheduler starten
void user_main()
{
    init_sequence(right_sequence, test_sequence);

    table.add(0, &T1);
    table.add(1000, &T2);
    table.add(2000, &T1);
    table.add(3000, &Tchecker);

    scheduler.set_schedule_table(&table);
}

```

Wann ist ein Test erfolgreich?

- ▶ Testfall erfolgreich: `TestOK("Message");`
 - ▶ alle API-Aufrufe haben sich wie erwartet Verhalten (Seiteneffekte, Rückgabewerte, ...).
 - ▶ bei mehrfädigen Testfällen entspricht der tatsächliche dem vorgesehenen Ablauf.

- ▶ Testfall fehlgeschlagen: `Panic("Message");`
 - ▶ falscher Rückgabewert
 - ▶ fehlender oder nicht-erwarteter Seiteneffekt
 - ▶ falscher Verlauf des Testfalls

Verlauf protokollieren

- ▶ Idee: Vergleiche Sollsequenz gegen eine aufgezeichnete Sequenz
- ▶ Aufzeichnen der Sequenz:

```

const char* right_sequence = "abcd"; // Sollsequenz
char sequence[4]; // aufgezeichnete Sequenz
volatile int counter = 0; // wohin kommt der Buchstabe // in der Sequenz

...
sequence[counter++] = 'a'; // Sequenz aufzeichnen
...
if (!check_sequence(right_sequence, sequence)) { // Sequenz
    Panic("Wrong sequence!"); // ueberpruefen
}

TestOK("Test successful finished!");

```

Verlauf protokollieren (Forts.)

- ▶ **Achtung:** Jede Abweichung vom vorgesehenen Ablauf muss sich in der Sequenz bemerkbar machen:

```
const char* right_sequence = "abcd"; // Sollsequenz
char sequence[4]; // aufgezeichnete Sequenz
void MyThread1::action() {
    sequence[counter++] = 'a'; // Sequenz aufzeichnen
    ...
    scheduler.exit();
    // dieses Statement sollte nie erreicht werden
    sequence[counter++] = 'x';
}
```

- ▶ ... oder anderweitig erkannt werden:

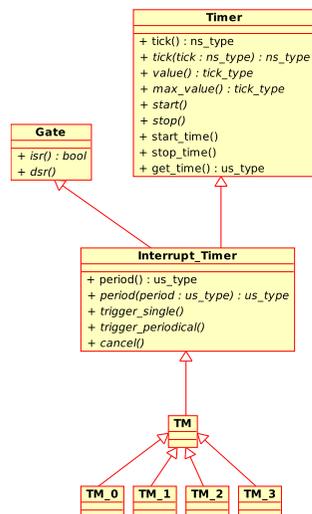
```
void MyThread1::action() {
    ...
    scheduler.exit();
    Panic("Should not come here!");
}
```

Testfall übersetzen und ausführen

Das EZStubs-Makefile implementiert folgende Targets:

Target	Beschreibung
<test>.elf	Übersetzt und bindet den Testfall <test>.
<test>.clean	Räumt den Testfall <test> auf.
<test>.sim_exec	Führt den Testfall <test> auf dem DeSmuME aus.
<test>.sim_gdb	Startet den DeSmuME, startet den GDB-Debugger und lädt den Testfall <test> in den DeSmuME.
<test>.sim_ddd	Startet den DeSmuME, startet den DDD-Debugger und lädt den Testfall <test> in den DeSmuME.
<test>.target_exec	Lädt den Testfall <test> auf den Nintendo DS Lite und führt in dort aus (diese Option steht für den Nintendo DS Lite leider nicht zur Verfügung).
<test>.target_debug	Startet den DDD-Debugger und lädt den Testfall <test> in den Nintendo DS Lite (diese Option steht für den Nintendo DS Lite leider nicht zur Verfügung).
checkalltests_sim	Übersetzt und bindet alle Testfälle und führt sie anschließend auf dem Simulator aus. Die Testfallergebnisse werden protokolliert und angezeigt.

Überblick



- ▶ Timer auf dem NDS
 - ▶ 4 Timer TM_{0,1,2,3} für den ARM7
 - ▶ 4 Timer TM_{0,1,2,3} für den ARM9
 - ▶ alle Timer sind identisch
 - ▶ 16 bit Zählerregister
 - ▶ kaskadierbar zu 32, 48 und 64 bit
 - ▶ Unterbrechung beim Überlauf
- ▶ Klasse Timer
 - ▶ Zeitmessung
- ▶ Klasse Interrupt_Timer
 - ▶ Eieruhr
 - ▶ Erzeugung von Unterbrechungen
 - ▶ *single-shot* oder zyklisch

Klasse Timer

ns_type tick() und **ns_type tick(ns_type)**

- ▶ Dauer eines Zählschritts
- ▶ wie lange dauert es, bis das Zählerregister inkrementiert wird
- ▶ nicht alle Zeitspannen darstellbar → Rückgabewert
- ▶ invalidiert us_type period(us_type)

tick_type value() und **tick_type max_value()**

- ▶ aktuellen/maximal möglichen Wert des Zählerregisters abfragen

void start() und **void stop()**

- ▶ Zählvorgang starten und stoppen

void start_time() und **void stop_time()**

- ▶ Zeitmessung starten und stoppen

us_type get_time()

- ▶ Ergebnis der letzten Zeitmessung abfragen

Klasse Interrupt_Timer

us_type period() und us_type period(us_type)

- ▶ Dauer bis zur nächsten Unterbrechung
- ▶ ab dem nächsten Aufruf von trigger_{single, periodical}()
- ▶ nicht alle Zeitspannen darstellbar \mapsto Rückgabewert
- ▶ es darf keine Unterbrechung erzeugt werden \mapsto vorher cancel()
- ▶ invalidiert ns_type tick(ns_type)

void trigger_single()

- ▶ Erzeugen **einer** Unterbrechung

void trigger_periodical()

- ▶ **Periodisches** Erzeugen von Unterbrechungen

void cancel()

- ▶ das Erzeugen der Unterbrechungen wird gestoppt
- ▶ der Timer wird angehalten

Beispiel: Verwendung des Interrupt Timers

```
class MyTimer : public TM_2 {
public:
    bool isr() {
        acknowledge();
        doSomething = true;
        return false; // Keine DSR
    };
    void dsr() {}; // Bleibt leer
};

MyTimer test_timer; // Timer anlegen

void user_main() {
    // Zufaelliche Periode generieren und Timer konfigurieren
    random_init();
    us_type rand = random() % 100000;
    testtimer.period(rand);
    testtimer.trigger_single(); // Einmalige Ausloesung nach rand
    ...
}
```

Beispiel: Zeitmessung

```
MyTimer test_timer; // Timer anlegen

class MyThread : public Thread {
public:
    bool action() {
        test_timer.stop_time(); // Zeitmessung stoppen
        kout << "Zeitmessung: " << test_timer.get_time();
        scheduler.exit();
    };
};

MyThread T1;

void user_main() {
    test_timer.tick(5000); // Zaehlerintervall festlegen
    test_timer.start(); // Zaehler starten
    ...
    schedule_table.add(500, &T1);
    test_timer.start_time(); // Zeitmessung starten
}
```

Den Testfall im Debugger (DDD) ausführen

- ▶ Debugger starten und den Testfall laden:

```
make tests/thread/Scheduler/RoundRobin/test1/test1.sim_ddd
```

- ▶ die Ausführung des Testfalls starten:

- ▶ **Cont**-Button klicken
- ▶ auf der GDB-Konsole folgendes eingeben:

```
(gdb) cont
```

Achtung

Der **Run**-Button bzw.

```
(gdb) run
```

wird **nicht** funktionieren!

Den Testfall neu starten ohne den DDD zu beenden

- zunächst muss man die Verbindung zum GDB-Stub trennen

```
(gdb) detach
```

- erneut eine Verbindung aufbauen (<PORT> steht in der Ausgabe von make)

```
(gdb) target remote localhost:<PORT>
```

- das Testfallprogramm laden

```
load
```

- das hat einige Vorteile:

- Breakpoints und Watches bleiben erhalten

Debugger neu starten

Manchmal kann es vorkommen, dass der GDB-Stub bzw. der deSmuME sich komisch verhält. Man hat dann drei Möglichkeiten:

1. DDD beenden und den Testfall erneut laden

- Breakpoints und Watches gehen verloren

2. nur den GDB beenden

- den `arm-elf-gdb` beenden

```
killall arm-elf-gdb
```

- DDD wird das bemerken und folgender Dialog erscheint

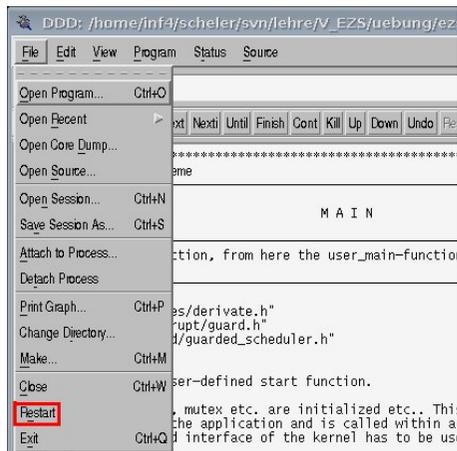


- auf den "Restart GDB"-Button klicken

Debugger neu starten

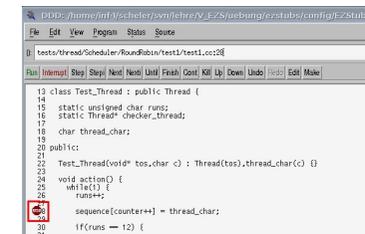
3. den kompletten DDD neu starten

- aus dem Menü *File* die Option *Restart* auswählen



Breakpoints setzen ...

- ... durch einen Doppelklick ins Quelltextfenster



- ... auf der GDB-Konsole für eine bestimmte Funktion

```
(gdb) break <function_name>
(gdb) break <file_name>:<line>
```

Achtung

Auf eingebettete Funktionen kann man keinen Breakpoint setzen!

Bedingungen für Breakpoints

Bevor das Programm an einem Breakpoint tatsächlich angehalten wird, kann man ...

- (a) ... den Breakpoint n-mal ignorieren
- (b) ... auf die Erfüllung einer Bedingung warten.

- ▶ aus dem Quelltextfenster heraus



- ▶ auf der GDB-Konsole

```
(gdb) ignore <breakpoint_nr> <n>           # (a)
(gdb) condition <breakpoint_nr> runs == 7 # (b)
```

Bedingungen für Breakpoints (Forts.)

Achtung

bedingte Breakpoints können **Ein-/Ausgabe-Operationen** zur Folge haben, die das **Laufzeitverhalten** des Programms beeinflussen können.

Sinnvolle Stellen für Breakpoints

- ▶ Warum wird mein DSR nicht ausgeführt? → Breakpoint auf ...
 1. ...wrapper_body → Wurde ein IRQ ausgelöst?
 2. ...void guardian(unsigned int) → Welches Gerät?
 3. ...den isr() → Wurde ein DSR aktiviert?
 4. ...void Guard::leave() → Welchen Wert hat lock?
- ▶ Warum wird der Scheduler nicht aktiviert? → Breakpoint auf ...
 1. ...void Guard::leave() → Welchen Wert hat lock?
 2. ...reschedule() → Wurde der Scheduler aktiviert?
 3. ...Thread* schedule() → Welcher Faden kommt als nächstes?

Achtung

1. Ein Debugger ersetzt nicht das Verständnis des Programms!
2. Ohne sinnvolle Annahmen kann man keine Breakpoints setzen!

Weitere Möglichkeiten, das Programm zu kontrollieren

- ▶ Break
 - ▶ Anhalten des Programms "von außen".
 - ▶ normalerweise per `Ctrl-C` in der GDB-Konsole
- ▶ Watchpoints

```
(gdb) watch xyz
```

- ▶ Programm anhalten wenn Variable `xyz` ihren Wert ändert

Werte von Variablen beobachten

► globale Variablen und Objekte

```

3 class Test_Thread : public Thread {
4
5   static unsigned char runs;
6   static Thread* checker;
7   char thread_char;
8 public:
9   Test_Thread(void* tos,
10 void action() {
11   while(1) {
12     runs++;
13     sequence[counter++];
14     if(runs == 12) {
15     scheduler.add(checker, thread);

```

(gdb) # Achtung: voll qualifizierter Name!
(gdb) graph display Test_Thread::runs

► lokale Variablen und Argumente

► Menü *Data* →

- *Display Local Variables*
- *Display Arguments*

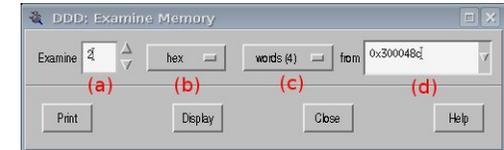
► Beobachtung problematisch:

- Speicherstellen/Register werden vom GCC wiederverwendet
- passende Debug-Information wird (noch) nicht erzeugt
- [Var Tracking Assignments \[VTA\]](#) ab GCC 4.5

Speicherstellen beobachten

► Menü *Data* →

► *Memory*



► Parameter

- (a) Wie viele Bytes, Half Words, Words, ...
- (b) Hexadezimal, Oktal, ...
- (c) Bytes, Half Words (2 Bytes), Words (4 bytes), ...
- (d) Startadresse

► Hilfreich zum Kontrollieren von I/O-Registern

Tutorial

1. die Datei `tutorial.tar.gz` entpacken
2. Makefiles anpassen
3. EZStubs-Bibliothek und Testfälle übersetzen und binden
4. Doxygen-Dokumentation erzeugen
5. Testfälle auf dem Simulator ausführen
6. Testfälle mit Hilfe des DDD debuggen
 - Breakpoints setzen (siehe Folie 45)
 - dort das Programm schrittweise ausführen
 - und die Abläufe auf den Folien 13,21, 21 und ?? nachvollziehen
7. Selbst ein, zwei Testfälle schreiben
 - einen Interrupt-Handler implementieren (die Klasse `Test_Interrupt_0` steht zur Verfügung).
 - im Interrupt-Handler einen Faden aktivieren