



Bedienung von JSIM-51

JSIM51 ist ein leistungsfähiges Werkzeug zur Softwareentwicklung für 8051-Controller und Derivate. Das Programm simuliert den Prozessorkern und ausgewählte Hardwarefunktionalitäten. Es ist entstanden aus da kommerziell verfügbaren Simulatoren alle in einer Preisklasse rangieren die für den Privatanwender jenseits von Gut und Böse rangieren. Natürlich wird wie bei allen Freewareprodukten keine Garantie für die Funktion übernommen. Allerdings werde ich versuchen Fehler die mir mitgeteilt werden so schnell wie mir möglich zu beseitigen und eine neue Version bereitzustellen. Im Folgenden das Wesentliche zur Bedienung des Programms.

- [Allgemeines](#)
- [Auswahl des zu simulierenden Prozessors](#) (neu)
- [Laden von Files](#)
 - [Details zum Laden von SDC51/ASxxxLink - Projekten](#)
- [Abspeichern der eingestellten Umgebung](#)
- [Browser-Fenster und Anzeige des geladenen Codes](#)
- [Den Code abarbeiten...](#)
- [Breakpoints](#)
- [Breakpoint-Fenster](#)
- [Register-Fenster](#)
- [Speicher-Fenster](#)
- [Speicher-Konfigurations Fenster](#)
- [Watch-Fenster](#)
- [Analysator-Fenster](#)
- [Stimulationsfile](#) (neu)
- [Interrupt-Fenster](#)
- [Stack-Fenster](#)
- [Terminal-Fenster](#)
- [Kommando-Zeile](#)

Allgemeines

Das Programm simuliert den Prozessorkern und einige Hardwarefunktionen des 8052. Es ist damit möglich Softwareteile unabhängig von der Zielplattform vorzutesten. Die Geschwindigkeit des Simulators erreicht auf einem Pentium 200 ungefähr die eines mit 10MHz getakteten 8051. Das sollte einigermaßen akzeptabel sein. Die Bedienphilosophie und Oberfläche des Simulators wurde soweit als möglich und sinnvoll, analog der von MS-VisualC++ 4.2 aufgebaut. **Es ist erforderlich, dass die Datei comctl32.dll die es ab dem IE3.0 gibt installiert ist.** Auf Grund der Komplexität des Programms sind Fehler keinesfalls auszuschließen, ja geradezu zu erwarten. Im Zuge der Fehlerbeseitigung und der Weiterentwicklung sind durchaus Änderungen in der hier dargestellten Bedien- und Funktionsweise möglich. Um dem Benutzer die Bedienung zu erleichtern besitzen die meisten Elemente der Oberfläche Tooltips. Alle Fenster können im Main-Window angedockt

werden wodurch die Oberfläche deutlich an Übersichtlichkeit gewinnt. Da es mich schon immer geärgert hat wenn ein Programm mehrere DLLs benötigt und in der Registry rumschmiert habe ich mich bemüht dieses zu vermeiden. Bis zur Version 2.09 war das auch so. Allerdings hat es sich gezeigt das viele den Wunsch haben unterschiedliche Derivate eines 8051 zu simulieren oder die gleiche Oberfläche auch für die Simulation anderer Controller zu nutzen. Also habe ich die Applikation geteilt. Es besteht jetzt aus einer EXE (jsim.exe) und mindestens einer DLL die die Prozessorspezifischen Dinge enthält. Ansonsten hat sich nichts geändert. **Es gbt keine Einträge in der Registry**, das Programm wird man genauso einfach wieder los indem man jsim.exe, jsim.ini und die Prozessor-DLLs im gleichen Verzeichnis löscht. Die Einstellungen eines Projektes werden bei dem Projekt mit der Dateierweiterung ".wsp" abgelegt.

Auswahl des zu simulierenden Prozessors

Der zu simulierende Prozessor muss ab Version 3.0 über eine DLL geladen werden. Das macht man über **Datei-Prozessor >Auswahl**. In dem sich öffnenden Dialog kann entweder gleich der Prozessor ausgewählt oder gesucht und geladen werden. Die vorhandenen DLLs werden in "jsim.ini" eingetragen und stehen dann in der Liste zur Verfügung. (Beim ersten Start muß also gesucht werden!). Der gewählte Prozessor wird im Workspacefile vermerkt, muss also nur einmal über die Liste selektiert werden. Wenn zum Zeitpunkt des Ladens eines Workspacefiles schon ein Prozessor ausgewählt ist wird dieser anstelle des im Workspacefiles eingetragenen genommen.

Laden von Files

JSIM kann folgende Datei-Formate laden:

- Absolutfiles im Format OMF51 und OMF51-Ext. (seit V3.13 auch Objektfiles die mit dem MetaLink-ASM51 erzeugt wurden)
- Intel-Hex-Files
- Motorola Hex-Files (S-Records)
- MetaLink ASM-51Objektfiles mit Debuginformation

 OMF51-Ext wird nur durch den Keil-C51-Compiler unterstützt.

Dieses Format sollte auf jeden Fall bevorzugt werden da es erhebliche Vorteile beim Debuggen bietet ! (Unterscheidung zwischen Groß/Kleinschreibung von Symbolen, Typdefinitionen für komplexe Datentypen). Ich habe den Hochsprachendebugger mit Quelltexten getestet die mit dem Keil-C51 Compiler V3.40 übersetzt wurden. Mittlerweile gibt es schon die Version 5.1 und ich bin mir nicht sicher ob die Debuginformationen tatsächlich noch genauso dargestellt werden. Der Simulator unterstützt auch keine unterschiedlichen Speicherbänke wie die neueren Compiler und Linker.

Das einfache OMF51-Format kennt nur Großschreibung. Da keine Typbeschreibung existiert, ist von einem Symbol lediglich seine Adresse bekannt, nicht aber seine Größe. Beispielsweise Strukturen lassen sich deshalb im Watchfenster nicht darstellen. Ein Absolutfile wird mit **Datei-Öffnen->Auswahl** geladen. Die C-Sourcefiles müssen sich im gleichen Verzeichnis wie das Absolutfile befinden.

Seit Version 2.04 können Dateien auch mit Drag-Drop oder über die Kommandozeile geladen

werden. Wenn nicht muss in die Workspacedatei für das Projekt unter "**SourcePath=...**" per Hand dann für alle Quellen gültige Pfad eingetragen werden. Das zu ladende File wird sequentiell eingelesen. Für jedes enthaltene Modul (Sourcefile) wird ein Eintrag im Browserfenster mit dem Namen des Moduls angelegt. Ist ein Sourcefile jünger als das Absolutfile wird eine entsprechende Warnung ausgegeben. Wenn auf Grund solch eines Fehlers die Zuordnung zwischen Quellfile und Zeileninformation nicht richtig hergestellt werden kann, stürzt im ungünstigsten Fall das Programm in der MIX-Darstellung ab. Prozeduren die innerhalb eines Moduls definiert sind werden als Untereinträge unter dem Modul dargestellt. Libraryfunktionen und durch den Compiler generierter Code werden in einem eigenen Fenster mit dem Namen "LIB" angezeigt.

Laden von HEX-Files

Für manche Projekte ist es notwendig auch ein Hex-file zu laden. Ein Hexfile enthält allerdings keine Symbolinformationen. Das bedeutet beim Reassemblieren werden auch Konstanten in Programmcode übersetzt was natürlich Unsinn ist. Trotzdem kann u.u. auch ohne Symbolinformationen debuggen. Ein Hexfile kann auch vor (!) dem Absolutfile geladen werden. Das darauffolgende Laden eines Absolutfiles löscht den Speicher nicht. Ein Hexfile wird geladen mit: **Datei-Öffnen->Auswahl**.

JSIM51 merkt sich wenn ein Hexfile geladen wurde und versucht beim nächsten Laden des gleichen Projekts wieder das Hexfile vor dem Absolutfile zu laden. Wird das Hexfile nicht mehr gefunden wird es wieder aus dem Projekt ausgetragen. (Das kann natürlich auch von Hand erledigt werden indem man den Schlüssel "hexfile=" in der Workspacedatei löscht).

Abspeichern der eingestellten Umgebung

Alle eingestellten Fenster, Watches, Breakpoints usw. können unter **Datei-Speichern als->Auswahl** als ASCII-Datei abgespeichert werden. Die Datei erhält voreingestellt den gleichen Namen wie das geladene Absolutfile nur mit der Dateierweiterung ".wsp". Der Name kann aber auch freigewählt werden. Dieses File enthält alle Kommandos um über die aktuelle Oberfläche wieder aufzubauen. Das File könnte auch von Hand editiert werden.

! Größte Vorsicht ist allerdings geboten da viele Angaben zur Fensterposition, ID und Dockzustände sehr kryptisch und auch noch voneinander abhängig sind. Sollte das Programm nach dem Editieren beim Laden abstürzen so empfiehlt es sich zuallererst die Workspacedatei zu löschen oder umzubenennen.

Das Programm läuft dann mit seinen Defaulteinstellungen hoch. Einstellungen zu Watchausdrücken oder Breakpoints lassen sich dann problemlos und ohne Risiko aus der korrupten WSP-Datei übernehmen. Beim Laden eines Absolutfiles (ohne .wsp) wird zunächst nach der entsprechenden Workspace-Datei gesucht und wenn gefunden, die Oberfläche wieder hergestellt.

Browser-Fenster und Anzeige des geladenen Codes

Als Startfenster wird das Modaul das die Marke "main" enthält gesucht und dargestellt. Gibt es diese Marke nicht wird der Code ab der Resetadresse (0x0000) angezeigt. In der Regel (d.h C-Programme) wird das das "LIB"-fenster sein und Assemblercode enthalten.

Man kann nun durch Doppelklick auf ein Modul oder eine Prozedur im Browserfenster sofort zum

gewünschten Code schalten. Jedes Modul besitzt ein eigenes Fenster das je nach Bedarf geladen oder in den Vordergrund geschaltet wird. Ist der Programmteil in "C" geschrieben und der Quelltext ist im eingestellten Pfad bzw. im Verzeichnis des Absolutfiles verfügbar, erfolgt die Darstellung als C-Quelltext. Alternativ können über die Toolbar-Buttons auch die Modi: Mixed(MIX) und Assembler(ASM) gewählt werden. Enthält der Code keine HLL-Informationen oder ist der Quelltext nicht verfügbar sind die Buttons "MIX" und "HLL" grau und können nicht betätigt werden.

Codebereiche die nicht über das Absolutfile geladen wurden, können auch auf Assemblerlevel betrachtet werden. Die Eingabe der Adresse erfolgt dann über die Kommandozeile. ("d.p. adresse"). Ab dieser Adresse wird der Inhalt des Codespeichers reassembliert und im Fenster "UnKnown" angezeigt. Zur Unterscheidung von geladenem Code ist das Fenster farblich hinterlegt. Es kann nur vorwärts gescrollt werden.

Wird das Fenster geschlossen und erneut geöffnet wird jedesmal neu reassembliert. (Für den Fall jemand baut selbstmodifizierenden Code.) Die Anzeige kann alternativ auch über die Kommandozeile durch Angabe der Adresse oder eines Labels umgeschaltet werden.

! Der Zustand der angezeigten Module (Größe,Position,Sichtmode) wird NICHT im Workspacefile gespeichert. **Beispiele:**

d.p.0x380 -zeigt den Code ab Adresse 0x380

d.p.main -zeigt den Code ab dem Label main

Beginnt an der Adresse kein Opcode wird der je nach Displaymodus entweder die nächste folgende HLL-Zeile oder der nächste Assemblerbefehl "angefahren".

Den Code abarbeiten...

Die Bedienung kann sowohl durch die Tastatur als auch über Toolbar-Buttons erfolgen. Die aktuelle Position des Programcounters wird immer durch einen gelben Pfeil markiert. (Der Cursor wird durch einen grauen Pfeil dargestellt. Die Wirkung der Befehle ist abhängig vom aktuell

eingestellten Darstellungsmodus. ! Ab Version V2.00 ist im Codefenster auch ein Kontextmenü über die rechte Maustaste verfügbar was oft benötigte Befehle enthält.

Step-In (F11):

Assembler- und Mixmode Es wird genau ein Assemblerbefehl abgearbeitet. Ist der Befehl ein "call" wird in die Prozedur gesprungen.

HLL Es wird eine HLL-Zeile abgearbeitet. Das bedeutet, es werden solange Assemblerbefehle ausgeführt bis der Programcounter wieder auf eine Adresse zeigt der eine HLL-Zeile zugeordnet ist. Das kann die nächst folgende HLL-Zeile sein oder aber auch eine Zeile in einer anderen Prozedur/Modul.

Step-Over (F10):

Assembler- und Mixmode Ist der Assemblerbefehl KEIN "call" wirkt Step-Over wie Step-In, der Befehl wird ausgeführt und der PC zeigt danach auf die Adresse des nächsten Assemblerbefehls. Wenn der Befehl ein "call" ist, wird auf die, dem call folgende Adresse ein temporärer Breakpoint gesetzt und das Programm im freien Lauf gestartet. Bei Erreichen des Breakpoints wird angehalten und der Breakpoint gelöscht. Dadurch können innerhalb des "calls" weitere Unterprogramme verschachtelt sein bei denen nicht angehalten wird. Wichtig zu beachten ist, daß das Programm nicht mehr anhält wenn der auf den "call" folgende Befehl nie erreicht wird. Das passiert zum Beispiel dann wenn innerhalb des Unterprogramms der Stack verändert wird, so daß der Returnbefehl zu einer anderen Adresse führt.

HLL Enthält die HLL-Zeile keinen Prozeduraufruf wirkt Step-Over wie Step-In auf HLL-Level. Wird innerhalb der Zeile eine Prozedur aufgerufen so werden solange Assembleranweisungen ausgeführt bis der Programcounter wieder auf eine HLL- Zeile in der aktuellen Prozedur zeigt. Da diese Abbruchbedingung nach jedem Assemblerbefehl geprüft werden muss ist ein solcher Step-over recht langsam. Der Grund, nicht wie beim Assemblmode die nächste HLL-Zeile zu suchen und dort einen temporärer Breakpoint zu setzen, liegt z.B. an der goto-Anweisung deren Ziel nicht bekannt ist aber in aller Regel nicht die nächst folgende HLL-Zeile sein wird.

Run-To-Cursor (F8)

**Assembler- ,
Mix- und HLLmode** Auf die Adresse des Befehls der in der Zeile steht auf die der graue Cursorpfeil zeigt wird ein temporärer Breakpoint gesetzt und das Programm im freien Lauf gestartet. Es werden solange Assembleranweisungen abgebeitet bis der nächste Breakpoint erreicht ist. (Das kann auch durchaus ein anderer permanenter Breakpoint sein). Wird der temporärer Breakpoint erreicht wird das Programm angehalten und der Breakpoint gelöscht.

Run (F5)

**Assembler- ,
Mix- und HLLmode** Das Programm wird ohne Abbruchbedingung gestartet. Es werden solange Assembleranweisungen ausgeführt bis entweder ein Breakpoint erreicht wird oder der Benutzer die Programmausführung selbst mit Stop abbricht.

Stop (F6)

**Assembler- ,
Mix- und HLLmode** Die Programmausführung wird sofort abgebrochen. Es wird zum aktuellen Stand des Programcounters gescrollt.

🟢 Breakpoints

Neben den intern verwalteten temporären Breakpoints hat der Nutzer die Möglichkeit an beliebige Stellen im Programm Breakpoints zu setzen. Breakpoints werden durch einen roten Punkt dargestellt. Sie können direkt von der Oberfläche mit **F9** gesetzt und gelöscht werden. Das ist in allen Modi möglich. Im Mixed-Mode werden Breakpoints auf Assembler-Anweisungen gesetzt. Die HLL-Anweisungen dienen nur zur Kommentierung des Codes (**SHIFT-F9** setzt/toggelt einen Breakpoint deaktiviert).

Alternativ können Breakpoints auch über die Kommandozeile eingegeben werden. Die Angabe der Adresse kann aber nur als Label erfolgen. Der Grund besteht darin, dass ein Breakpoint durch das Einfügen eines verbotenen OP-codes an der entsprechenden Stelle realisiert wird. Könnte man einen Breakpoint durch Angabe einer beliebigen Adresse setzen, würde möglicherweise nur ein Datenbyte und nicht ein OP-code geändert. Die Folge wäre fatal: Der Breakpoint würde nie erreicht aber das Programm macht Unvorhersehbares!

! Ab Version 1.10 werden die Breakpoints durch ein zusätzliches Byte für jeden Speicherplatz im Codespeicher verwaltet, und nicht mehr über illegal Opcode realisiert. Am oben dargestellten Verhalten ändert sich dadurch aber nichts.

! Ab Version 2.00 können Breakpoints auch auf Speicherzugriffe, wahlweise auf Lese/Schreib oder Beides, gesetzt werden. Das kann man auf verschiedene Weise tun:

- Im geöffneten [Speicherfenster](#) (nur in der **Byte- oder ASCII-Ansicht möglich!!**).
 1. Speicherzelle mit Click auf linke Maustaste die zu überwachende Zelle markieren.
 2. Mit rechter Maustaste das Kontextmenü öffnen und den Typ der Überwachung wählen (Read/Write) oder auch mit CTRL-R (Lese-Breakpunkt) oder CTRL-W (Schreib-Breakpunkt) setzen
 3. Ein gesetzter Breakpunkt wird orange markiert. (Programmbreakpunkte sind im Speicherfenster nicht sichtbar.)
- Über die Kommandozeile mit folgender Syntax: **b.s Adresse(hex) zugriffstyp Speicherbereich**

Beispiele:

b.s 0xF014 0x200 c	setzt auf die Adresse 0xF14 im CODE-speicher einen Lesezugriffsbreakpoint
b.s 0x14 0x400 d	setzt auf die Adresse 0x14 im DATA-speicher einen Schreibzugriffsbreakpoint
b.s 0x95A 0x600 x	setzt auf die Adresse 0x95A im XDATA-speicher einen Schreib+Lesezugriffsbreakpoint

➡ Wichtig zu beachten!

Das Setzen auch nur eines Zugriffsbreakpunktes oder Veränderung der Default-Speicherbelegung bewirkt dass die Simulation über eine andere, wesentlich langsamere Funktion läuft. Es muss ja jetzt bei jedem Zugriff überprüft werden ob ein Breakpunkt erreicht wurde. Das verlangsamt die Simulation um ungefähr 70-80% !!! Die Anzahl der Zugriffsbreakpunkte wirkt sich dann

allerdings nicht mehr aus. Also entweder gar keinen oder wenn schon dann nur solange wie notwendig.

Programmbreakpoints können deaktiviert werden. Das ist entweder über die Oberfläche für alle Breakpoints gleichzeitig möglich oder aber mit: **Bearbeiten-Breakpoint->Auswahl-Toggle** oder auf der Cursorposition mit **CTRL-F9**. Breakpoints werden im Workspacefile mit ihrem aktuellen Zustand gespeichert. **Beispiel:**

b.s.main setzt einen Breakpoint auf die Adresse "main"

Register-Fenster

Das Registerwindow wird entweder über das Menü **Ansicht-Register** bzw. den Toolbar-Button aufgerufen oder über die Kommandozeile mit "**w.r.**" Alle wesentlichen Prozessorregister werden hier dargestellt und können auch direkt modifiziert werden. Dazu wird der neue Wert eingegeben und mit "ENTER" bestätigt. Wird während dem Programmablauf der Inhalt eines Registers geändert, stellt es sich beim nächsten Halt Rot dar. Prinzipiell erfolgt ein Refresh nach jedem Programmhalt, nach Step und wenn ein Wert in einem anderen Watch- oder Speicher-Window verändert wurde.

➔ Wird der PC im Register-Fenster geändert erfolgt keine unmittelbare Umschaltung des Code-Fensters sondern erst beim nächsten Step!

➔ Das Ausgaberegister des Register P2 kann eigentlich nicht rückgelesen werden. Der Simulator setzt aber Ein- und Ausgabe in P2 gleich. Normalerweise dürfte das nicht stören da bei der Benutzung externen Speichers das Port nicht für Eingaben zur Verfügung steht.

Speicher-Fenster

Ein Memoryfenster wird über das Menü "**Ansicht-Speicher**" erzeugt. Es können maximal 10 Fenster angezeigt werden. Das sollte reichen. Die Alternative ist die Kommandozeile mit dem Kommando : "**w.m.viewmode.adresse /speicherbereich**". Die Adresse kann dezimal oder hexadezimal eingegeben werden. Auch ein Label oder ein qualifizierter C-Ausdruck können verwendet werden. Das bedeutet dann aber auch das sich der Bereich auf den das Fenster zeigt ständig ändern kann! Wichtig mindestens 1 Leerzeichen vor "/" !

Der Viewmodus wird mit folgenden Werten spezifiziert:

a	ASCII
b	Byte
s	short(16Bit)
l	long(32Bit)

Der Speicherbereich kann mit folgenden Werten spezifiziert werden:

x	XDATA
c (oder keine Angabe)	CODE
d	DATA

i	IDATA
p	PDATA (wird P2 geändert ändert sich auch der Bereich auf den das Speicherfenster zeigt)

Beispiele:

0x1F00 /x Xdata ab 0x1F00

20 /d Data ab 0x14

testarray Speicher ab dem Symbol testarray.

**xptr+2 /c* Code ab Inhalt des Zeigers xptr.
Solch ein Ausdruck ist nur verwendbar wenn OMF51-ext geladen wurde, ansonsten kennt man ja den Typ auf den xptr zeigt nicht.

MAIN Codespeicher ab Symbol MAIN.

Klick auf die rechte Maustaste öffnet ein Kontextmenü mit dem die Darstellungsarten *Byte - Short - Long - ASCII* ausgewählt werden können.

In allen Darstellungen können die Werte direkt geändert werden. Dazu auf den Eintrag klicken und den neuen Wert eintragen. Der Wert wird als Hexadezimalzahl interpretiert. Im ASCII-modus wird direkt der ASCII-code der Taste eingetragen. Bestätigt wird mit "ENTER". Ist die Eingabe gültig wird der Wert übernommen und in den Speicher geschrieben. Seit dem letzten Windowupdate geänderte Speicherinhalte werden rot dargestellt. (Also auch beim Step durch ein Programm das gerade angeeigte Speicherbereiche modifiziert.)

Copy und Paste im Speicherfenster

Es ist auch möglich ganze Speicherbereiche mittels Copy-Paste zu modifizieren. Um einen String einzutragen muß der Sichtmodus auf ASCII gestellt werden. Die Zelle ab der der String einzutragen ist wird mit der Maus selektiert und der String wie üblich mit Ctrl-V, SHIFT-Insert oder über das Kontextmenü eingefügt. **Die abschließende 0 wird mit kopiert!**


Will man binäre Daten in den Speicher füllen, muß der Sichtmodus auf BYTE gestellt werden. Der String muß dann in der Form: "01 dd E5 77 00 FF" als Text geschrieben und dann genauso wie ein Ascii-String in den Speicher kopiert werden.


Watch-Fenster


JSIM51 bietet leistungsfähige Möglichkeiten schnell und umfassend nicht nur die Werte beliebiger Variablen anzuzeigen sondern auch komplexere C-Ausdrücke zu evaluieren. Es gibt verschiedene Methoden sich die Werte von Variablen und Ausdrücken anzeigen zu lassen:

Tooltip

Einfach mit dem Mauszeiger im Code-Window über einem Variablennamen stehen bleiben oder mit Doppelklick markieren. Nach kurzer Zeit wird der aktuelle Wert der Variable angezeigt. Wenn es sich um einen eingebauten C-Datentyp handelt wird der Wert angezeigt. Ist es ein Pointer, der Speicherbereich auf den der Zeiger zeigt und sein Wert. Ist es ein komplexer Datentyp beispielsweise eine Struktur wird die Anfangsadresse und der Speicherbereich in geschweiften Klammern angezeigt. Kann der Wert des Labels auf den der Mauszeiger steht nicht angezeigt werden erscheint der Tooltip nicht.

 Diese Form der Anzeige sucht nach einer lokalen Variable entsprechenden Namens, d.h. der Variablenname wird automatisch in der Form: **Modul:Prozedur:Variablename** expandiert. Dazu wird die Stellung des Cursors im Programm ausgewertet. Das bedeutet, gibt es eine globale und lokale Variablen gleichen Namens, wird immer die lokale Variable angezeigt. Steht der Cursor außerhalb des Codes (z.B. auf einer Deklaration) wird die erste Variable mit dem Namen in dem Modul gesucht und angezeigt. Das ist, wenn vorhanden, die globale Variable oder aber eine ganz andere lokale Variable was aber aus dem Kontext nicht ohne weiteres erkennbar ist.

 **Tooltip auf markierten Text**
Ist der Ausdruck komplexerer Art (z.B. `testval[x]`) kann man den gewünschten Ausdruck mit der Maus markieren und anschließend den Mauszeiger darauf stellen. Die Anzeige erfolgt wie oben nur mit dem Unterschied das keine Erweiterung der Variablennamen vorgenommen wird.

 **Temporärer Watchdialog** Die Darstellung mittels Tooltip erlaubt nur die Anzeige einfacher Datentypen. Will man eine Struktur oder den Inhalt eines Zeigers sehen, öffnet man ausgehend vom selektierten Ausdruck, einen temporären Watchdialog mit SHIFT-F9. Die Struktur wird in Form eines Strukturbaumes angezeigt. Elemente die ein vorangestelltes "+" aufweisen können durch Klick auf das Symbol weiterverfolgt werden. Einfache Datentypen und Zeiger können direkt editiert werden. Dazu wird der neue Wert eingegeben und mit "ENTER" bestätigt. Wünscht man die Übernahme in das permanente Watchwindow klickt man auf: **"Add to Watchwindow"**. Der selektierte Ausdruck wird dann direkt übernommen. Das Fenster wird mit Klick auf "OK" geschlossen.

Permanentes Watchwindow

Die Anzeige erfolgt über das Menü mit **"Ansicht-Watch"** über Toolbarbutton oder alternativ mittels Kommandozeile: **"w.w"**. Ein neuer Watchausdruck wird entweder über den temporären Watchdialog eingetragen oder aber direkt in die unterste freie Zeile geschrieben. Leerzeichen sind erlaubt. Ist der Ausdruck gültig wird er angezeigt, im anderen Fall eine Fehlerausschrift erzeugt. (Ein ungültiger Ausdruck muß nicht zwangsläufig immer ungültig bleiben. z.B Generic Pointer bei Keil-C51) Ein Ausdruck kann auch editiert werden. Bestätigt wird immer mit "ENTER". Entfernt wird ein Ausdruck durch Löschen des Ausdrucks und "ENTER". Ist unter dem zu löschenden oder geänderten Ausdruck ein Baum aufgeklappt wird dieser natürlich mit gelöscht. Um gleichnamige lokale und globale Symbole voneinander zu unterscheiden können Modulname und Prozedurname angegeben werden.

Beispiele für Watchausdrücke:

`x`

die Variable `x`, gibt es mehrere Variablen `x` dann ist entweder die global Definierte oder die Erste gefundene (also unsicher).


<code>_UPI</code>	die im Unterprogramm UP1 definierte lokale Variable x.
<code>sisal[4].perfdat->BERcnt[x-1]</code>	ein komplexer Ausdruck der auch mehrere Variablen beinhalten kann.
<code>{0x0200,x}</code>	den Inhalt der Speicherzelle 0x200 im XDATA bereich als Byte.(die Adresse kann nur hexadezimal angegeben werden). Diese Form ist zwar kein C-Ausdruck, kann aber nicht verwechselt werden und erlaubt so die Anzeige des Inhaltes direkter Speicherzellen.

Die Darstellung der berechneten Ausdrücke erfolgt analog zum temporären Watchdialog.

Beispiele:

<code>{XD: 0x3450}</code>	die Adresse eines komplexen Datentyps (nicht editierbar aber in der Regel aufzuklappen)
<code>{-> D:0x55}</code>	Zeiger auf die Adresse 0x55 im Datenbereich (editierbar und aufzuklappen)
<code>0x42"Beta"</code>	Ein Character (signed char). Um die Lesbarkeit eines Strings zu vereinfachen wird der ab dieser Adresse folgende C-String mit angezeigt. (das erste Zeichen ist editierbar)
<code>0x61</code>	Ein Unsigned Char. (editierbar)
<code>0x1234</code>	short (editierbar)

Über das Kontextmenü das mit der rechten Maustaste geöffnet wird läßt sich die Darstellung zwischen dezimal,hexadezimal un binär umstellen.Die Binärdarstellung ist nur für die Typen CHAR und UCHAR möglich. Die aktuelle Einstellung wird im Workspace mit abgespeichert.Ein Update des Fensters erfolgt bei jedem Code-Step, und bei Änderungen im Register- oder einem Speicherfenster.

 OMF-51 kennt keine PDATA-Variablen. Das bedeutet eine in PDATA deklarierte Variable erscheint im Watchfenster als in XDATA deklariertes Symbol. Das stimmt nur solange P2 nicht geändert wird. Um das Problem beim Debuggen zu beheben kann über das Kontextmenü (linke Spalte, nicht markierte Variable !!) der Speicherbereich zwischen XDATA und PDATA . Ebenso möglich über die Kommandozeile mit: "**c.p Symbolname**" ändert zu PDATA oder "**c.x Symbolname**" zu XDATA. umgeschaltet werden. Das Watchfenster folgt dann dem Register P2. Diese Einstellung wird **nicht** im Workspace gespeichert.

Datentypen ohne "OBJECTEXTEND"

Ist ein Absolutfile nicht mit der Option "OBJECTEXTEND" übersetzt worden sind keine Typinformationen für Symbole vorhanden. Alle oben beschriebenen komfortablen Möglichkeiten der Betrachtung von komplexen Datentypen stehen damit nicht zur Verfügung. Um das Debuggen dennoch zu erleichtern kann einem Symbol ein einfacher Datentyp zugewiesen werden. Dazu trägt man zunächst das Symbol in das Watchfenster ein, stellt den Mauszeiger auf das Feld in dem das Symbol steht und öffnet mit der rechten Maustaste ein Dialog der es ermöglicht dem Symbol einen Typ zuzuweisen. Verfügbar sind:

- signed char
- unsigned char
- signed short
- unsigned short
- signed long
- unsigned long
- float
- Pointer auf vorangestellte Datentypen in jeden Speicherbereich

(Nicht unterstützt werden die generischen Keil-C51-Zeiger. Wer die benutzt kann auch mit OBJEXT übersetzen!) Der Typ kann auch während der Debugsession geändert werden, wird aber NICHT im Workspacefile gespeichert. Diese Möglichkeit der Datentypmanipulation steht nur zur Verfügung wenn kein erweitertes Objektformat vorhanden ist, d.h. nur wenn alle Module ohne "OBJECTEXTEND" übersetzt wurden!!

Locals

Eine sehr bequeme Einrichtung wie bei VisualC++. Das Fenster zeigt immer alle lokalen Variablen einer Prozedur an. Das bedeutet, die angezeigten Variablen ändern sich jedesmal wenn ein Prozedurwechsel auf Grund von Call oder Return erfolgt. Die Anzeige erfolgt über das Menü mit "**Ansicht-locals**", Toolbarbutton oder alternativ mittels Kommandozeile: "**w.l.**". In das Fenster können keine weiteren Einträge aufgenommen werden. Ansonsten unterscheiden sich Bedienung und Verhalten nicht von der des permanenten Watchwindow.<>

Breakpoint-Fenster

Das Breakpoint-fenster wird über das Menü "**Bearbeiten-Breakpoints**" geöffnet. Es zeigt alle aktuell gesetzten Breakpoints an. Ein Breakpoint kann hier gelöscht, deaktiviert oder aktiviert werden. Deaktivierte Breakpoints werden im Code-Window durch einen roten Kreis dargestellt, Aktive durch einen roten Punkt. Zugriffsbreakpunkte (ab Version 2.00) werden mit einem orangen Quadrat gekennzeichnet. Sie können nur gelöscht und nicht deaktiviert werden.

Speicherkonfigurations-Fenster

Das Fenster wird über das Menü "**Bearbeiten-Speicher Konfiguration**" geöffnet. In diesem Fenster kann der Speicher den das Programm scheinbar zur Verfügung hat konfiguriert werden. Beliebige Aufteilungen sind möglich, auch die Segmentierung eines Bereiche in mehrere Stücke. Damit ist es möglich Zugriffe in Bereiche zu erkennen die in der späteren Applikation nicht verfügbar sind. Die Änderung der Default-Speicherkonfiguration hat allerdings auch [Konsequenzen](#) auf die Simulationgeschwindigkeit die sich dann deutlich verringert.

Analysator-Fenster

Das Analysatorfenster erlaubt verschiedene Einstellungen zur Laufzeitmessung und Konfiguration des Tracelogs. Mit Prozessortakt wird die Taktfrequenz der Zielplattform eingestellt. Die Zeiten im Fenster "realeProgrammlaufzeit" sind davon abhängig. Eine Verringerung der Taktfrequenz bewirkt sofort eine Erhöhung der Laufzeit. Die Angabe der Simulationslaufzeit ist mehr informativer Art sie wird in Abhängigkeit von den Traceoptionen und dem Rechner auf dem der Simulator läuft abhängen.

"**Autostart**" bewirkt das bei jedem Step, Step-Over, Run usw. die Zähler neu gestartet werden. Damit ist es möglich für einzelne C-Anweisungen Programmteile bis hin zu einzelnen Assemblerbefehl die Zeiten zu messen. Ist "Autostart" nicht gesetzt werden die Zeiten akkumuliert auch wenn in einzelnen Schritten simuliert wird. Die Zähler können explizit mit "Reset" gelöscht werden. Die Berechnung der Simulationszeit erfolgt nur im Autostartmodus.

Wenn man Informationen über minimale, mittlere und maximale Laufzeit eines Programmteils benötigt, (etwa bei Schleifen die über verschiedene Pfade durchlaufen werden) ist es möglich im Codefenster einen Messpunkt zu setzen und diesen im Analysatorfenster freizugeben (Messpunkt EIN). Immer wenn dieser Messpunkt erreicht wird der Wert des aktuellen Zykluszählers abgespeichert. Der Zähler wird beim Durchlaufen eines jeden, auch deaktivierten (!) Breakpoints auf Null gesetzt. Eine Messung wird also folgendermaßen durchgeführt:

1. Setze einen deaktivierten Breakpoint auf den Beginn der Schleife
2. Setze den Meßpunkt an die interessierende Stelle (**F3** im Codefenster oder über den Toolbar)
3. Setze einen Breakpoint hinter die Schleife
4. Aktiviere die Messung mit "Messpunkt EIN" im Analysatorfenster und starte das Programm

Nach Erreichen des Ausgangsbreakepoints werden die Anzahl der Durchläufe, kürzeste, mittlere und längste Ausführungszeit für das Programmstück angezeigt.

Es gibt zwei verschiedene Arten Trace zu schreiben:

- Jeden Assemblerbefehl mit den wesentlichsten Registern (PC- PSW- A- R0..R7- B- DPH- DPL- P2- SP)
- Für einen ausgewählten Programmpunkt (Tracepoint) beliebige Variablenwerte. Prinzipiell wäre es kein großes Problem eine beliebige Zahl von Tracepunkten mit ebenso beliebigen Variablen zu realisieren. Allerdings muß dann natürlich nach jedem Maschinenbefehl nachgesehen werden ob ein Tracepunkt erreicht wurde und wenn ja die ganze Maschine zum Interpretieren und Berechnen der Ausdrücke angelassen werden. Das verlangsamt die Simulation dann kolossal was mich dazu bewegt hat nur einen Tracepunkt zuzulassen. Das sollte auch reichen da man ja oft nur eine Folge von Variablen- veränderungen oder Portausgaben zu einem Zeitpunkt verfolgen will. Um den Tracepunkt zu nutzen sind zwei Dinge zu tun:
 1. unter "**Bearbeiten-Tracepoint**" die zu überwachenden Variablen eintragen.
 2. den Tracepunkt im Codefenster mit F4 setzen oder mittels Toolbarbutton auf die aktuelle Cursorposition.

Die Traceaufzeichnung wird mit "Enable Trace" gestartet. Der Trace wird in die Datei "temptrace.log" geschrieben. Diese Datei speichert den ASM-Trace binär, den Variablen-Trace als

ASCII ab, ist also nur bedingt mit einem Editor lesbar. Um den aufgezeichneten Trace anzusehen benutzt man besser den Trace-Viewer. Dieser wird mit "**Ansicht-Trace**", einem Toolbar- button oder alternativ über die Kommandozeile mit "**w.t.**" geöffnet. Die Länge des Trace ist fest auf 1000 Einträge begrenzt. Danach wird das File wieder von vorn beginnend geschrieben. Das Tracefenster ist nur verfügbar wenn "Enable Trace" ausgeschaltet ist.

● Stimulationsfile

Neu ab Version 3.01 ist die Möglichkeit einen sogenannten Stimulationspunkt zu setzen. Das bedeutet während dem Lauf des Programms wird bei jedem Passieren des Stimulationspunktes eine Zeile aus einem Skriptfile gelesen und über die Kommandozeile eingegeben. Das Skriptfile ist ein Textfile das alle zulässigen [Kommandozeilenstrings](#) enthalten kann. Mehrere Anweisungen in einer Zeile werden durch Komma getrennt. Der Sinn dieses Stimulationsfiles ist zum Beispiel Eingaben über Ports zu simulieren. Ein Beispiel könnte so aussehen:

```
$m.d 0x20 0x05 , s.d 0x22   setzt den Datenspeicher an Adresse 0x20 auf 5 und auf Adresse
0xAA                       0x22 auf 0xAA
$m.x 0x1000 0xAB          setzt die XDATA-Zelle 0x1000 auf 0xAB
w.m.l 0x9F0 /x           öffnet einSpeicherfenster
b.s 0x3400               setzt eine Breakpoint auf die Adresse 0x3400
```

Bei Erreichen des Fileendes wird wieder von vorn begonnen. Ein Stimulationspunkt kann selbst auch über die Kommandozeile gesetzt werden bzw. vom Toolbar aus. Der Stimulationspunkt wie auch der Pfad des Simulationsfiles werden im Workspace gespeichert.

● Interrupt -Fenster

Der Simulator kann nicht die vollständig periphere Hardware des Prozessor simulieren. Das bedeutet z.B. das zwar die Timer seit der DLL-Version 1.011 unterstützt werden, die serielle Schnittstelle aber nur mehr funktional vorhanden ist.(Problem Baudrate..) Dennoch sollen Aktionen die durch die Peripherie über Interrupts in Gang gesetzt werden auch testbar sein. Dazu dient das Interruptfenster. Es erlaubt die 5 Standardinterrupts des 8051 zu konfigurieren und auszulösen. Für den seriellen Interrupt wird noch zwischen Receive und Transmit-Interrupt unterschieden.

Um einen Interrupt zu erzeugen muß der freigegeben sein und auch das globale Interruptfreigabeflag muß gesetzt sein. Es ist auch möglich mehrere Interrupts gleichzeitig auszulösen, der Simulator sollte die Reihenfolge über die Priorität richtig auflösen. Ein anstehender aber noch nicht bearbeiteter Interrupt ist daran erkennbar das der Interruptbutton unten bleibt. Der Simulator setzt den Interrupt bei Annahme selbständig zurück.(Ausnahme die Flags RI, TI des seriellen Interrupts die durch das Programm zu löschen sind). Das Fenster wird nur aktualisiert wenn das Programm anhält. Das Ändern der Interrupteinstellungen im Programm wird also erst sichtbar nach dem nächsten Halt!

Stack-Fenster

Das Stackfenster zeigt die Tiefe der Unterprogrammaufrufe. Mit jedem Call oder auch Interruptannahme wird oben ein Eintrag für die aktuelle Prozedur und ihre Aufrufadresse angefügt. Durch Doppelklick auf einen Listeneintrag kann man die Sicht zu dem jeweiligen Aufrufpunkt schalten. Die Sache wird natürlich problematisch wenn in irgendeinem Programmteil auf den Stack geschrieben, der Stackzeiger selbst verändert oder Inhalte gezielt verändert werden. Das Stackfenster versucht zwar solche einzelnen Stackaktionen mitzuverfolgen aber es gehört nicht viel dazu es durcheinanderzubringen. Wer es benutzt muß wissen worauf er sich einläßt wenn er PUSH und POP benutzt.

Wenn der Stackpointer überläuft wird der Programmlauf unterbrochen und eine Warnung ausgegeben werden. Sollte das tatsächlich beabsichtigt sein kann natürlich fortgesetzt werden.

Terminal-Fenster

Da sehr oft Programme zur Kommunikation über die serielle Schnittstelle zu entwickeln sind habe ich hier eine begrenzte Ausnahme gemacht was die Hardwaresimulation des Controllers betrifft gemacht. Wohl aber nur qualitativ. Auf Zeitbedingungen wurde kein Wert gelegt. Auch die Baudrate hat keine Bedeutung. Die korrekte Initialisierung des Timers wird nicht überprüft. Das Terminalfenster wird über das Menü mit "**Ansicht-Terminal**" oder mittels Kommandoeingabe "**w.c.**" geöffnet Das Terminalfenster simuliert einen Receive-Interrupt wenn ein Zeichen eingegeben wird. Umgekehrt simuliert es einen Transmit- interrupt wenn das Register S0BUF geändert wird. Es wird nur eine Übertragung mit 8Bit-Breite simuliert. Das Starten des Terminalfensters verlangsamt den Simulationslauf etwas da es einen eigenen Thread benutzt, sollte also nur geöffnet werden wenn es wirklich benötigt wird.

Die Einstellung "lokales Echo" stellt jedes eingegebene Zeichen direkt auf dem Fenster dar. Ist die Option ausgeschaltet werden eingegebene Zeichen direkt in S0BUF geschrieben und, wenn freigegeben (ES=1 in IE0 + REN=1 in S0CON), der Interrupt ausgelöst (RI=1 in S0CON).

Will man auch binäre Zeichen darstellen ist die Option "Binär String" einzuschalten. Der empfangene String hat dann das Format 00 02 09 41 FF ...usw. Auch die Eingabe muss in dieser Form erfolgen. Alle Zeichen, ausgenommen Backspace die keine hexadzimalen Ziffern darstellen werden ignoriert.

Es ist nicht möglich Strings über Copy-Paste an das Terminalfenster zu schicken. Wie hätte man das auch behandeln sollen. Das Terminalfenster müßte ja dann darauf warten das der Interrupt für ein Zeichen bearbeitet wurde. Das allerdings entspräche nicht den realen Verhältnissen und man käme leicht in Gefahr mit dem Simulator etwas "Gut" zu testen.

Anmerkungen zum Laden von "SDC51/ASxxxLink"-Dateien

Im Gegensatz zu anderen Compilern/Linkern erzeugen SDC51/ASxxxLink nicht nur eine einzelne Absolut-Objektdatei sondern die zum Debuggen notwendigen Informationen sind in mehreren Textdateien verteilt. Die Symboltypinformationen befinden sich in den ".cdb"-Dateien (je eine für ein C-Quellfile). Die Symboladressen findet man im Mapfile. Der eigentliche Code steht in einer

".ihx"-Datei (normales Intel-Hexformat).

Um also ein SDC51-Projekt zu debuggen benötigt man folgende Dateien im gleichen Verzeichnis:

- ".c" die C-Quellen
- ".ihx" ein normales Intel-Hexfile
- ".cdb" ein Symbolfile für jede C-Quelldatei
- ".map" das Projekt- Mapfile.

Um das Projekt zu laden, öffnet man die **.ihx**-Datei. JSIM weiss dann das ein SDC51-Projekt geladen werden soll und sucht selbständig zunächst nach der Map-Datei (**.map**) mit dem gleichen Namen wie die Hex-Datei und dann nach den **CDB**-Dateien und C-Quelldateien.

Die Quellen müssen mit der **Option --debug** compiliert werden! Jede Quelldatei muss einzeln übersetzt und anschließend alle gemeinsam gelinkt werden!

Hier ein Beispiel für ein Batchfile:

```
..\bin\sdcc -c
--debug mod1.c
..\bin\asx8051 -olj aprog.a51
..\bin\sdcc -c --debug main.c
..\bin\sdcc -Wl-mjkc:\sdcc\sdcc51lib main.rel aprog.rel mod1.rel
```



Es gibt einige Einschränkungen beim Debuggen von SDC51-Projekten:

- Man kann dem Compiler das Optimieren nicht ganz verbieten dadurch werden lokale Variablen in Registern gehalten, bzw. überlagert. Das Mapfile enthält dazu aber keine Informationen. Diese Variablen sind dann einfach nicht darstellbar.
- Von den Libraryfunktionen gibt es nur die Startadresse im Mapfile aber nicht das Ende. Deshalb kann man diese nicht direkt über den Debugger anfahren.
- Der Compiler kennt keine Variablen im PDATA -Bereich.



JFE eignet sich gut um solche Projekte zu editieren, übersetzen und korrigieren. Dazu muss man lediglich ein eigenes Tool definieren was die Batchdatei aufruft, den Compilerfilter auf "MS C/C++" setzen und die Ausgabe zu JFE umleiten.

Man kann dann direkt per Doppelklick vom Ausgabefenster zu den Fehlern springen.

Kommandozeilen Syntax

Die meisten Fenster und Parameter welche im Workspacefile ab gespeichert werden benutzen dafür auch die folgende Syntax. Über die Combobox können jeweils die letzten 10 Eingaben der Kommandozeile zurückgeholt werden. Alle Kommandos können auch im Stimulationsfile verwendet werden. Die rot gekennzeichneten Parameter sind 8051 spezifisch, d.h bei anderen Prozessorimplementierungen können da andere Werte oder Kommandos vorhanden sein.

Kommando	Parameter		Beschreibung
d.p	Adresse (Label)		Anzeige des Programmcodes ab Adresse/Label

w.r				Öffnet das Registerfenster
w.m	viewmode l= long s=short b=byte	Adresse (Label)	/memtyp d=data x=xdata i=idata c=code	Öffnet ein neues Speicherfenster im angegebenen Darstellungsmodus. Der Speichertyp kann nur angegeben werden wenn eine Adresse angegeben wird und kein Label.
w.w				Öffnet das Watchfenster.
w.l				Öffnet das Fenster zur Anzeige der lokalen Variablen
w.s				Öffnet das Stackfenster.
w.a				Öffnet das Analysatorfenster.
w.t				Öffnet das Traceview-Fenster
w.b				Öffnet das Browserfenster. Das Fenster wird voreingestellt angezeigt.
b.s	Label			Setzt auf die Adresse des Labels einen aktiven Breakpoint. Das Display wird NICHT automatisch aktualisiert.
b.c	Label			Löscht wenn vorhanden den Breakpoint an der Adresse des Labels. Das Display wird NICHT automatisch aktualisiert.
t.s	Adresse(hex)			Setzt den Tracepoint auf die absolute Adresse. Ist diese Adresse nicht der Anfang eines Befehls, ist der Tracepoint nicht sichtbar und wird auch nie erreicht. Also vorsichtig! Dieses Kommando ist für die Benutzung durch das Workspacefile gedacht.
t.a	Variable (Ausdruck)			Fügt dem Tracepunkt eine neue Variable hinzu.
m.	Adresse(hex)			setzt den Messpunkt auf die angegebene Adresse. Wird keine Adresse angegeben wird der Messpunkt gelöscht.
s.	Speicherbereich d=data x=xdata i=idata c=code	Anfangsadresse	Endadresse	Konfiguriert den Speicher als vorhanden im angegebenen Bereich. Sind Anfangs- und Endadresse = 0 wird der ganze Bereich gelöscht.
s.0				Die Defaulteinstellung der Prozessor-DLL wird eingestellt.
c.	x	Symbol		setzt den Speicherbereich des angegebenen Symbols auf XDATA
c.	p	Symbol		setzt den Speicherbereich des angegebenen Symbols auf PDATA
stm	Adresse (hex)			setzt den Stimulationspunkt auf die angegebene Adresse
stop				unterbricht den Programmlauf
update				aktualisiert alle Watchfenster
spezifische Kommandos der DLLs: 8051.dll und 80c320.dll				

\$...				immer ein Kommando was durch die Prozessor DLL ausgewertet werden muss. Hier die 8051-spezifischen Kommandos
\$m.	Speicherbereich d=data x=xdata i=idata c=code	Adresse (hex)	neuer Wert	Setzt die selektierte Speicherzelle (wenn vorhanden) mit dem angegebenen Wert. Das Display wird nicht aktualisiert.
w.c				Öffnet das Terminalfenster
w.i				Öffnet das Interruptfenster