Megalock Handbuch

November 2007

SysTech

Entwicklungsges. für Mikroprozessorsysteme mbH

Winterstraße 48
D-53177 Bonn
Tel. : ++49 228 9711070
Fax : ++49 228 9711071
eMail : systech@megalock.de

SysTech

Entwicklungsges. für Mikroprozessorsysteme mbH

Inhaltsverzeichnis

Megalock Teil I

Installation und Schnelleinstieg

SysTech
Entwicklungsges. für Mikroprozessorsysteme mbH

INSTALLATION VON MEGALOCK

Legen Sie die Megalock-CD in ein CD/DVD-Laufwerk Ihres Computers.

Starten Sie das Installationsprogramm Setup.exe von der CD über ‚Eingabeaufforderung' oder über ‚Start' und dann ‚Ausführen' wie folgt.

Setup **COPYMEGALOCK** /s D: /d C:

/s LAUFWERK:	Quelllaufwerk
/d LAUFWERK:	Ziellaufwerk

Das Installationsprogramm legt auf dem Ziellaufwerk den Ordner MEGALOCK und die dazugehörigen Unterordner DRIVER, HANDBUCH, INCLUDE, LANGUAGE, EXAMPLES und PROJECT an.

MEGALOCK USB-DONGLE

Für den Megalock USB-Dongle ab Version 3.0 sind keine weiteren Installationen mehr notwendig.

~~Danach starten Sie das Setupprogramm erneut mit den folgenden Parametern, Systemdateien werden kopiert und Einträge in der Registry vorgenommen.~~

~~Setup **INSTALL** /i INFFILE [/p PROTOKOLL] [/w]~~

~~/I INFFILE~~	~~Path und Name der INF-Datei „MLockUSB.INF". z.B. c:\megalock\driver\mlockusb.inf~~
~~/p PROTOKOLL~~	~~Path und Name der Protokolldatei [Optional]~~
~~/w~~	~~Programmausführung wird verlangsamt [Optional]~~

~~Das Setupprogramm kann auch die Installation rückgängig machen. Hierzu wird das Programm wie folgt aufgerufen.~~

~~Setup **UNINSTALL** /k KEYNR [/p PROTOKOLL] [/w]~~

~~/k KEYNR~~	~~Ihre 6stellige Keynummer. Dies Keynummer ist identisch mit ihrer MLUCxxxxxx.DLL im Megalock-Ordner. Für die Demoversion wäre dies die 710200~~
~~/p PROTOKOLL~~	~~Path und Name der Protokolldatei. [Optional]~~
~~/w~~	~~Programmausführung wird verlangsamt. [Optional]~~

~~Das Setupprogramm können Sie natürlich auch für die Installation bei Ihren Kunden einsetzen.~~

~~Zwischen der UNINSTALL Funktion und einer erneuten Installation ist zwingend ein Neustart des Rechners notwendig.~~

Der MegalockUSB-Dongle wird an einem beliebigen USB-Port aufgesteckt. Es folgt ein Hinweis vom Betriebssystem das „Neue Hardware" gefunden wurde und die noch notwendigen Einträge in der Registry werden automatisch vom Betriebssystem vorgenommen. Je nach Einstellung erscheint unter Arbeitsplatz ein Laufwerkseintrag für den Megalock USB-Dongle.

MEGALOCK PARALLEL-DONGLE

Die Systemdateien für den Paralleldongle sind manuell zu kopieren, notwendige Einträge in der Registry werden beim erstmaligen Start von der Megalock.DLL vorgenommen. Kopieren Sie bitte MEGALOCK.SYS nach %Root%\system32\drivers und MEGALOCK.DLL nach %root%\system32, beide Dateien befinden sich im Ordner MEGALOCK\DRIVER.

Weitere Informationen über die Einträge in der Registry, notwendige Systemdateien und Inhalte der Megalock-CD sind im Megalock-Handbuch Teil IV Anhang enthalten. Dort sind auch die Optionen und Unterschiede zwischen Megalock Parallel- und USB-Dongle aufgeführt.

SCHNELLEINSTIEG

Der Schnelleinstieg sollte auf jeden Fall durchgeführt werden um ihnen zu verdeutlichen wie einfach die Grundstrukturen der Megalock Sprachelemente in Ihrem Quellcode eingebunden werden. In dem Schnelleinstieg wird eine Datei verwendet, die in drei Schritten erweitert wird. Wir werden die in dem Schnelleinstieg verwendeten Megalock Befehle in Kurzform beschreiben. Eine ausführliche Beschreibung finden Sie im Teil III des Megalock Handbuches.
Dabei werden Sie mit dem Megalock Compiler, -Debugger und -Toolbox arbeiten und hierbei die Grundfunktionen kennen lernen.

Starten Sie das Programm ML.EXE aus dem Megalock-Ordner. Nach dem Start des ML.EXE Programm ist die Oberfläche noch sehr leer und ohne ein Projekt anzulegen, geht gar nichts.
Stecken Sie den MegalockUSB Dongle mit Masterfunktion auf einen USB-Port oder den Paralleldongle auf einen Druckerport.

PROJEKT ANLEGEN

Wählen Sie die Option **„Projekt erstellen“** unter Menüauswahl **„Datei“** und wechsel zunächst in den Ordner **„\Megalock\EXAMPLES“**. Wählen Sie aus diesem Ordner den Ordner für Ihre Programmiersprache aus und tragen im Feld Dateiname **„Demo“** ein und betätigen den Button **„Öffnen“**. Jetzt erscheint Links ein Fenster, in diesem Fenster werden die Ordner und später auch die Dateien angezeigt, die zu diesem Projekt gehören. Setzen Sie den Mauszeiger auf den Eintrag **„Project“** und drücken die rechte Maustaste. Tragen Sie im nachfolgenden Dialogfenster den Path für den Megalock-Ordner ein, setzen die Einträge für beide Dongle-Typen auf **„Auto“** und betätigen den **„OK“-**Button.

Sollte ihre Programmiersprache im Ordner **„EXAMPLES“** nicht aufgeführt sein, setzte Sie sich bitte mit uns in Verbindung.

DER MEGALOCK COMPILER

Setzen Sie jetzt den Mauszeiger auf den Ordner **„Compiler“** und drücken die rechte Maustaste, wählen dann im erscheinenden Menü **„Einstellungen**“. Im nachfolgenden Dialogfenster werden die, zurzeit, unterstützten Programmiersprachen und die dazu gehörigen Steuerdateien, mit der Extension **MLL** angezeigt. Wählen Sie hier ihre Programmiersprache, durch einen Doppelklick auf die jeweilige Zeile, aus. Setzen Sie noch den Schalter **„Include Source to Output“** alle anderen Felder im Dialog lassen Sie leer und betätigen den **„OK“-**Button.

Die Dateien mit der Extension **MLL** sind im Ordner **„Megalock\Language“** untergebracht und steuern die Umsetzung der Megalock-Befehle, so das ihre Compiler die Megalock-Befehle fehlerfrei übersetzen kann. Änderungen an den MLL-Dateien sollten nicht vorgenommen werden, kopieren Sie eine Datei und legen diese unter einem anderen Namen im Ordner **„Megalock\Language“** ab. Diese erscheint dann auch in der Auswahlliste. Weitere Information und Aufbau der Steuerdateien finden Sie mit Handbuch Teil IV Anhang.und Steuerdateien.

Setzen Sie den Mauszeiger noch mal auf **„Compiler“** und drücken die rechte Maustaste, wählen dann **„Datei zum Ordner hinzufügen“** und wählen die Eingabe-Datei **„DEMO.SML“** über den rechten **„Auswahl“-**Button, dann legen Sie den Namen und Path der **„Ausgabedatei“** fest, für unser Beispiel wählen Sie den gleichen Path wie der, der Eingabedatei und ändern nur die Extension. Setzen Sie noch die Art der Eingabedatei im Feld **„Type der Eingabedatei“** auf „**Source-Datei“** und drücken den **„OK“**-Button.

Die Ausgabedatei ist die Eingabe- bzw. Source- Datei für ihren Compiler. Der Megalock-Compiler übersetzt die Megalock-Befehle so das ihr Compiler diese neuen Befehle fehlerfrei übersetzen kann.

Sichern Sie jetzt das Projekt über Menü **„Datei“** und dann **„Projekt speichern“**.

Setzen Sie den Mauszeiger auf die Datei **„DEMO.SML“** und mit einem Doppelklick wird die Datei im rechten Bildschirm angezeigt. In diesem Fenster kann der Inhalt der Datei bearbeite werden. Hierzu stehen ihnen einfache Editor-Funktion zur Verfügung.

In dem von Ihnen ausgewählten Ordner ist zusätzlich die Datei DEMOPRG.SML vorhanden, in der die nachfolgend beschriebenen Eingaben bereits eingefügt wurden. Diese Datei sollte nur zu Vergleichszwecken herangezogen werden, und sollte nicht den Schnelleinstieg ersetzen.

EXAMPLE 1

Sollte es Abweichungen zu dem nachfolgendem Beispiel, bzgl. des von Ihnen eingesetzten Compilers, geben, so sind diese zu Beginn der Datei DEMO.SML beschrieben.

In diese Datei wollen wir jetzt die Grundfunktionen (vier Stück an der Zahl) für den Aufruf des Megalock Dongles eingeben. Die jeweiligen Stellen sind mit **!!!EXAMPLE1**_ entsprechend gekennzeichnet und können über die Suchfunktionen Menü **„Bearbeiten“** und dann „**Suchen“** oder durch Scrollen mit den Cursortasten gesucht werden.

Bitte suchen und ersetzen Sie nachfolgende Einträge.

SUCHE	**ERSETZE**
!!!EXAMPLE1_MACRO	##DEF_MEGALOCK_MACRO
!!!EXAMPLE1_CALL	##DEF_MEGALOCK_CALL 32BIT
!!!EXAMPLE1_OPEN	a = ##OPEN_MEGALOCK (0)
!!!EXAMPLE1_CLOSE	##CLOSE_MEGALOCK (0)

Wenn Sie die oben angegebenen Eintragungen vollzogen haben, compilieren Sie den Quellcode über Menüpunkt „**Compile“** und **„Projekt kompilieren“** oder drücken die **„F7“**-Taste

Im Infofenster, unterhalb des Editorfensters, wird der Fortschritt des Kompiliervorgangs angezeigt. Etweilige Fehler die sich auf Megalock-Sprachelemente beziehen werden ebenfalls in diesem Fenster angezeigt. Sollten Fehler aufgetreten sein, prüfen Sie diese Zeile mit den zuvor gemachten Eingaben und korrigieren Sie diese entsprechend und wiederholen Sie den Compiliervorgang, bis dieser fehlerfrei verläuft.

Nun wollen wir uns das Ergebnis des Kompiliervorganges im Editor ansehen.
Setzen Sie den Mauszeiger noch mal auf **„Compiler“** und drücken die rechte Maustaste, wählen dann **„Datei zum Ordner hinzufügen“** und wählen Path und Name der zuvor als Ausgabedatei festgelegten Datei (Sourcedatei für ihren Compiler). Setzen Sie noch die Art der Datei im Feld **„Type der Eingabedatei“** auf **„nicht bearbeiten“** und drücken den **„OK“**-Button.
Anschließend laden Sie diese Datei in den ML-Editor.

Sie können natürlich auch ihren Eigenen oder den zu ihrem Compiler gehörenden Editor wählen.

Hinter den von Ihnen angegebenen Megalock-Anweisungen, die sich jetzt in Kommentarzeichen befinden, finden Sie die vom Megalock-Compiler generierten Anweisungen.

Mit der Anweisung „**##DEF_MEGALOCK_MACRO“** wurden hier für Ihren Compiler spezifische Vereinbarungen, sofern diese notwendig sind, eingefügt. Diese Anweisung muss in jeder Quelldatei in der Sie Megalock Befehle einsetzen an entsprechender Stelle eingetragen werden.

Anstelle der Anweisung „**##DEF_MEGALOCK_CALL 32BIT**“ wurden hier die externen Referenzen für Unterprogrammaufrufe der Megalock API eingefügt. Da es für die Megalockbefehle unterschiedlich zu übergebende Parameter gibt, gibt es auch mehrere Funktionsaufrufe in der Megalock API.

Der erste Parameter enthält in den unteren 16-Bit den kundenspezifisch codierten Funktionscode des Megalock Dongles, und die oberen 16-Bit enthalten Steuerinformationen für die Megalock API. Gegebenenfalls nachfolgend angegebene Parameter dienen zur Übertragung bzw. Aufnahme von funktionsspezifischen Daten.

Anstelle der Anweisung „**a = ##OPEN MEGALOCK(0)**“ wurde die Open Funktion mit Programmnummer 0 für einen Megalock Dongle eingebaut. Mit der Open Funktion wird die Suche nach den spezifischen Megalock Dongle ausgeführt. In der Testversion (710200) wird nach dem Dongle mit der Wert 0x01c9F600 + Programmnummer gesucht.
Wurde der entsprechende Dongle gefunden, wird eine Datenstruktur innerhalb der API angelegt und der Wert 0 in die Variable übertragen, ansonsten enthält die Variable einen entsprechenden Fehlercode.

Anstelle der Anweisung „**##CLOSE_MEGALOCK(0)“** wurde die Close Funktion für den Megalock Dongle eingebunden. Die Close Funktion löscht nur die innerhalb der API angelegten Datenstruktur für den jeweiligen Dongle, mit der entsprechenden Programmnummer die durch die Open Funktion angelegt worden ist.

Mit einem Megalock Dongle können Sie bis zu 32 Programme bzw. Programmteile, mit jeweils einer eigenen Programmnummer, verwalten. Hierdurch können Sie z.B. Ihr gesamtes Softwarepaket ausliefern und schalten jeweils die Programmnummern für Ihre entsprechenden Programmteile auf den Megalock Dongle frei.

COMPILIEREN MIT IHREM COMPILER.

Starten Sie ihren Compiler oder ihre Programmierumgebung und wechsel in das zuvor gewählten Ordner **„\Megalock\EXAMPLES\...“** und erstellen aus der, vom Megalock-Compiler erzeugten, Sourcedatei eine ausführbare EXE-Datei. Zu den meisten Programmiersprachen wurden bereits Projekt- oder Batch-Dateien erstellt, diese finden Sie auch in den gewählten Ordnern.

Sollten Fehler auftreten, überprüfen Sie die Path Anweisung, Include und Libraries und korrigieren Sie diese entsprechend.
Ist das Programm DEMO fehlerfrei erstellt, führen Sie dieses aus.
Bei aufgestecktem Dongle erhalten Sie die Meldung

“**Hello World I have found the Megalock Dongle**“

Somit ist dieses Programm nur in Verbindung mit dem Megalock Dongle korrekt ausführbar. Da der Megalock Dongle mehr zu bieten hat, beginnen wir jetzt mit dem zweiten Beispiel.

EXAMPLE 2

Im zweiten Beispiel werden wir zusätzlich Megalock Einzelbefehle in den Quellcode einbinden. Hierbei handelt es sich um Megalock Toolbox Befehle, deren Funktion im nachfolgenden Text kurz erläutert ist; weitergehende Informationen sollten Sie zu einem späteren Zeitpunkt im Teil III des Handbuches nachschlagen. Einzelbefehle müssen mit **ML..** gekennzeichnet werden, damit der Megalock Compiler diese erkennt und entsprechend generiert.

Starten Sie das Programm ML.EXE erneut, das zuletzt bearbeitete Projekt wird automatisch geladen. Setzen Sie den Mauszeiger auf die Datei **„DEMO.SML“** und mit einem Doppelklick wird die Datei im rechten Bildschirm angezeigt.

Löschen Sie die Beiden mit **!!!EXAMPLE2_COMMEND** gekennzeichneten Programmzeilen innerhalb des Unterprogrammes EXAMPLE2. Sollte Ihre Programmiersprache nicht über eine Blockkommentierung verfügen, entfernen Sie bitte nur die Kommentarzeichen in jeder Zeile des Unterprogrammes EXAMPLE2.

Ersetzen Sie jetzt die mit !!!EXAMPLE2_ gekennzeichneten Stellen im Quellcode wie folgt:

SUCHE	**ERSETZE**
!!!EXAMPLE2_PROGNO	ML..PROGNO=0

Mit **PROGNO** wird die Programmnummer (0-31) angegeben, unter der die nachfolgenden Megalock Befehle ausgeführt werden.

SUCHE	**ERSETZE**
!!!EXAMPLE2_SINGLE1	ML..ACount(0,CounterReg%)

Mit **ACount** wird eine Zählerfunktion eingefügt. Bei jeder Ausführung dieses Befehls wird der Inhalt der angegebenen Speicherstelle innerhalb des Toolbox-Speichers um 1 dekrementiert bis der Wert Null erreicht ist.

SUCHE	**ERSETZE**
!!!EXAMPLE2_SINGLE2	ML..CLock(0,32,StrTab$)
!!!EXAMPLE2_SINGLE3	ML..CLock(0,32,StrTab$)

Mit **CLock** wird eine Kryptierfunktion eingefügt. Die den Inhalt der Stringvariablen in der angegebenen Länge kryptiert bzw. dekryptiert. Der Kryptierschlüssel wird aus dem Inhalt der angegebenen Speicherstelle im Toolbox-Speicher gebildet.

SUCHE	**ERSETZE**
!!!EXAMPLE2_SINGLE4	ML..Rdata(0,RDataReg%)

Durch den Befehl **RData** wird bei Programmausführung der angegebene Inhalt der Toolbox-Speicherstelle in eine Variable übertragen.

Compilieren Sie den Quellcode erneut über Menüpunkt „**Compile“** und **„Projekt kompilieren“** oder drücken die **„F7“**-Taste.

Nun wollen wir uns erneut die Erweiterungen des Unterprogrammes in EXAMPLE2 im Editor ansehen.

Hier sehen Sie jetzt innerhalb der Kommentarzeilen die von Ihnen eingegebenen Megalock Befehle und nachfolgend die durch den Megalock Compiler erzeugte Programmzeilen.

Wie Sie bereits bei den externen Referenzen sehen konnten, gibt es, aufgrund unterschiedlicher Parameter, mehrere Funktionsaufrufe in der Megalock API. Der Compiler ordnet dem Befehl jeweils den richtigen Funktionsnamen zu und übergibt im ersten Parameter den kundenspezifischen verschlüsselten Befehlscode sowie die Steuerfunktionen für die API. Ggf. nachfolgende Parameter dienen zur Übertragung bzw. Aufnahme von funktionsspezifischen Daten.

Zu jeder Megalock Kunden-Version gehören:

- Kundenspezifische Megalock-Seriennummer
- Kundenspezifische kryptierte Befehlscodes
- Die speziell für Ihre Serie erstellte MLUCxxxxxx.DLL

Das ML.EXE Programm mit Compiler, Debugger, Toolbox ist kein Unikat, benötigt aber die MLUCxxxxxx.DLL um auf Ihre Dongle-Serie zugreifen zu können.

Somit können zwischen unseren Kunde A und B weder Software noch Dongle ausgetauscht werden.
Darüber hinaus kann der vergebene Befehlscode durch weitere Parameter beeinflusst werden.
Sie sollten auf keinen Fall Befehlscodes, die durch den Megalock Compiler erzeugt wurden, abändern oder von einem Megalock-Programmblock in einen anderen Block kopieren oder als Einzel-Befehle ausführen lassen, oder Einzel-Befehle in einen Programmblock kopieren. Die Kodierung ist nicht nur kundenspezifisch sondern auch je Kundenserie pro Programmblock und Einzelbefehlen anders.

Nun wiederholen wir, wie bereits zuvor beschrieben, den Compiliervorgang mit Ihrem Compiler und führen Sie das Programm aus.
Beim Ausführen des Programmes wird der Zählerinhalt, der kryptierte und dekryptierte String sowie der Inhalt der ersten Datenspeichertabelle innerhalb des Toolbox-Speicher angezeigt.
Bei wiederholtem Ausführen des Programmes wird sich der Inhalt des Zählers verändern, sofern dieser **> 0** ist.

DIE MEGALOCK TOOLBOX

Um die Inhalte des Toolbox-Speichers zu ändern, starten Sie das Programm ML.EXE und setzen Sie den Mauszeiger auf **„Toolbox“** und drücken die rechte Maustaste, wählen dann **„Datei neu erstellen“** und wählen den Ordner in dem sich auch die Beispieldateien befinden und geben als Dateiname **„DEMO.MLE“** an. Die Datei wir unter **„Toolbox“** eingetragen und rechts im Fenster angezeigt.

Lesen Sie den Toolbox-Speicher über Menüpunkt **„Toolbox“** und **„Lesen Toolbox-Speicher“** oder drücken die **„F4“**-Taste aus. Die Daten werden in den Fenstern **Lock**, **Counter**, **Data**, **String**, **Password** und ggf. **Memory** angezeigt.
Aktivieren Sie das Register „Lock“ und der Inhalt der Toolbox-Tabelle Lock wird angezeigt. Da wir in dem DEMO-Programm jeweils den ersten Tabellenplatz der entsprechenden Datengruppen verwenden, gehen Sie mit der Maus auf die Zeile, die die Kennzeichnung „000“ trägt und drücken die Linke Maustaste. Im nachfolgenden Fenster können Sie nach Belieben LockData und LockKey verändern. Hierdurch wird die Kryptierung innerhalb des Demoprogrammes über die Funktion CLock(0,...) verändert. Verändern Sie nun, wie zuvor beschrieben die Inhalte der ersten Counter- und Data-Tabelle.

Um die neu eingegebenen Daten in den Toolbox-Speicher zu schreiben, wählen Sie den Menüpunkt **„Toolbox“** und **„Schreiben Tollbox-Speicher“** oder **„Ctrl-F4**“ aus. Die geänderten Daten werden im Toolbox-Speicher auf dem Megalock-Dongle zurück geschrieben.

Führen Sie jetzt das zuvor erstellte ausführbare Programm erneut aus, um die Änderungen zu sehen.

EXAMPLE 3

Im dritten Beispiel werden wir zu dem bestehenden Programm einen Megalock Programmblock einfügen. Bei einem Megalock Programmblock haben Sie zusätzlich die Möglichkeit den Programmblock im Megalock Debugger zu prüfen.
Im ersten Schritt sollten Sie den Quellcode ergänzen bzw. ersetzen und anschließend wollen wir den compilierten Code im Megalock Debugger testen.

Starten Sie das Programm ML.EXE wie bisher, das aktuelle Projekt wird geladen. Setzen Sie den Mauszeiger auf die Datei **„DEMO.SML“** und mit einem Doppelklick wird die Datei im rechten Bildschirm angezeigt.

Entfernen Sie die mit **!!!EXAMPLE3_COMMEND** gekennzeichneten Zeilen innerhalb des Unterprogrammes EXAMPLE3 und ersetzen Sie die Zeilen **!!!EXAMPLE3** wie folgt:

SUCHE	**ERSETZE**
!!!EXAMPLE3_BEGIN	##BEGIN

Mit dieser Anweisung wird der Beginn eines Megalock Programmblockes gekennzeichnet.

SUCHE	**ERSETZE**
!!!EXAMPLE3_MODUL	MODUL=DEMO_EXAMPLES3

Für jeden Programmblock können Sie einen bis zu 16 Zeichen langen Namen verwenden. Bei mehreren Programmblöcken innerhalb einer Datei oder eines Projektes können Sie so, innerhalb des Debuggers, die jeweiligen Unterprogramme leichter identifizieren.

SUCHE	**ERSETZE**
!!!EXAMPLE3_PROGNO	PROGNO=0,MODULNR=1

Mit **PROGNO** wird die Programmnummer (0-31) angegeben, unter der der Programmblock ausgeführt wird. Für jede Programmnummer wird die Grundlage der Codierung des OPCODES festgelegt und über die **MODULNR** (0-255) wird die Codierung des Megalock OPCODES zusätzlich beeinflusst.

Zwei identische Befehle mit gleichen Parametern und Registern *innerhalb* eines Programmblockes erhalten die gleichen OPCODES.
Zwei identische Befehle mit gleichen Parametern und Registern *in unterschiedlichen Pro*grammblöcken mit unterschiedlicher **MODULNR**, erhalten verschiedene OPCODES. Zusätzlich können Sie die Codierung der Megalock OPCODES über die **PROGNO** beeinflussen.

SUCHE	**ERSETZE**
!!!EXAMPLE3_PARM	NOTURBO,NOREADCT NOWATCHDOG,STATUS=Status

Die zuvor eingegebenen Modul Parameter sind zusätzliche Anweisungen für den Megalock Compiler, auf die wir hier nicht weiter eingehen wollen.

SUCHE	**ERSETZE**
!!!EXAMPLE3_VARIABLE	LONG a,b

Mit der Variablendeklaration definieren wir zwei 32-Bit Variablen, die in dem Programmblock verwendet werden.

In den nachfolgenden Zeilen haben wir bereits einige Megalock Befehle eingegeben, auf die wir innerhalb des Debuggers näher eingehen wollen. Fügen Sie jetzt die Kennzeichnung für das Ende des Megalock Programmblockes ein.

SUCHE	**ERSETZE**
!!!EXAMPLE3_END	##END

Wenn Sie diese Eintragungen in den Quelltext eingefügt haben compilieren Sie den Quellcode erneut über Menüpunkt „**Compile“** und **„Projekt kompilieren“** oder drücken die **„F7“**-Taste.

Sollten Fehler aufgetreten sein, prüfen Sie diese Zeile mit den zuvor gemachten Eingaben und korrigieren Sie diese entsprechend und wiederholen Sie den Compiliervorgang, bis dieser fehlerfrei verläuft.

DER MEGALOCK DEBUGGER

Unter dem Ordner **„Debug“** werden alle fehlerfrei übersetzen Programmblöcke aufgelistet. Da wir nur einen Programmblock erstellt haben, fällt die Auswahl leicht. Setzen Sie den Mauszeiger auf den Eintrag **„DEMO_EXAMPLE3“** und mit einem Doppelklick, wird der Programmblock im rechten Bildschirm angezeigt. Neben dem Quelltext des Programmblockes werden noch die Megalock-Flags und Megalock-Register sowie die im Programmblock verwendeten Variablen angezeigt. Die Größe der einzelnen Fensterelemente können Sie nach Belieben anpassen.

Der Programmcounter steht bereits auf der ersten Befehlszeile **„Move(LR1,100)“**. Drücken Sie ein mal die Taste **„F10“**, der Programmcounter springt eine Zeile weiter und der Dezimalwert 100 wurde als 32-Bit Wert in das Register **R01** mit dem **Move** Befehl übertragen. Ein **L** oder **W** vor dem Register spezifiziert die Breite **W** = 16 Bit und **L** = 32-Bit.

Klicken Sie ein mal im Register-Fenster auf **„Wert“** in der Überschriftenzeile, die Darstellung ändert sich jeweils von Hexadezimal über Dezimal mit Vorzeichen nach Dezimal ohne Vorzeichen und wieder nach Hexadezimal. Die Darstellung können Sie nicht nur im Register-Fenster verändern sondern auch im Variablen-Fenster und auch bei fast allen Tabellen in der Toolbox.

Drücken Sie noch mal die Taste **„F10“**, der Befehl **“Move (LR2, 1)“** wird ausgeführt. Das Register **R02** enthält jetzt den Wert 1 und noch mal die Taste **„F10“** und das Register **R03** enthält den Wert 3.

Weitere Funktion zur Steuerung des Debugger sind über Menü **„Debug“** abzurufen. Die einzelnen Funktionen sind im **Megalock Handbuch Teil II** unter **„Der Megalock Debugger“** beschrieben.

Als nächstes folgt eine **For-Next-Schleife** (Initialisierung, Vergleich, Schrittweite). Innerhalb dieser Schleife befindet sich eine **If-Abfrage**, wobei je nach Vergleich addiert oder subtrahiert wird.

```
>       for  (LR4 = 0; LR4 < LR3; LR4 += 1 )
           if LR4 < 2
             Addition (LR1,LR4)
           else
             Subtract (LR2,LR1)
           endif
        next
```

Arbeiten Sie die Schleife steppweise mit der Taste **“F10“** ab und beachten Sie die Änderungen in den betroffenen Megalock Registern. Wenn Sie die Registerinhalte ändern möchten, setzen Sie den Mauszeiger auf die Registerbezeichnung z.B. R01 und drücken die linke Maustaste, im Fenster unterhalb den Variablen erscheint das gewählte Register mit dem aktuellen Inhalt. In dieser Zeile können Sie das Register mit einem neuen Wert belegen, mit der **„Return“**-Taste wird der Wert übernommen.
.

Wenn Sie das Schleifenende erreicht haben (die Bedienung LR4<LR3 ist falsch), gelangen Sie in eine **Do loop-Schleife**.

```
>       Move (LR0,50)
         do
           Addition (LR1,a)
           Increment (LR2)
           if LR1 > b
             Move (LR0,1)
```

```
    Getreg (a,LR1)
  endif
loop LR0
```

Do loop beschreibt eine Zählschleife, wobei der Zähler mit Register **LR0** angegeben ist und beim Erreichen des Wertes 0 wird die Schleife verlassen. Innerhalb der Schleife werden die beiden Variablen **a** und **b** verwendet, wobei die Variable **a** zum Megalock Register addiert wird und die Variable **b** zum Vergleich, um die Zählschleife vorzeitig zu beenden, herangezogen wird. **Getreg** überträgt den Inhalt des Megalock Register **LR1** in die Variable **a**.
Bevor Sie diesen Teil des Programmblockes ausführen setzen Sie die Variable **a**=2 und Variable **b**=10. Setzen Sie den Mauszeiger auf die jeweilige Variable, drücken die linke Maustaste, im Fenster unterhalb den Variablen erscheint die gewählte Variable mit dem aktuellen Inhalt. In dieser Zeile können Sie die Variable mit einem neuen Wert belegen, mit der **„Return“**-Taste wird der Wert übernommen.

Der Megalock Debugger bietet Ihnen die Möglichkeit, die Ausführung eines Programmblockes mit dem Menüpunkt **„Debug“** in verschiedenen Modis durchzuführen. Um einen Programmblock erneut ausführen, wählen Sie **“Erneut starten“** oder **„Shift+F5“**. Setzen nach belieben Haltepunkte und führen den Programmblock erneut aus.

Wenn Sie nun die Debug-Funktion ausprobiert haben, wollen wir uns anschließend den durch den Megalock Compiler generierten Code im Megalock Compiler ansehen.

Setzen Sie den Mauszeiger auf den Eintrag **„DEMO.SML“** und mit einem Doppelklick wird die Datei im rechten Bildschirm angezeigt.

Gehen Sie nun zum Unterprogramm mit der Kennzeichnung **EXAMPLE3**. Mit der ersten **if Abfrage** wird überprüft ob der Megalock Dongle vorhanden ist und die Programmnummer (**PROGNO**) freigeschaltet ist. Nur wenn diese Bedingungen erfüllt sind, wird der Programmblock ausgeführt. Andernfalls wird direkt zum Ende der **if Abfrage** verzweigt und der Fehlercode wird in die Variable **Status** übertragen.
In den nachfolgenden Programmzeilen innerhalb des if Blockes sehen Sie in Kommentarzeichen die eingegebenen Megalock Anweisungen und eine Zeile darunter den vom Megalock Compiler generierten Code.
Am Ende des Programmblockes wird, auch nach Abarbeitung der Befehle innerhalb der if Abfrage, der Fehlercode in die Variable Status übertragen. Treten während der Abarbeitung des Programmblockes Fehler auf (Dongle nicht mehr vorhanden, OPCODE Fehler, Überschreitung Datentabelle etc.), werden die restlichen Anweisungen des Programmblockes nicht mehr ausgeführt und der Fehlercode entsprechend gesetzt.

Änderungen sollten nur in der Ursprungsdatei *.SML durchgeführt werden und nicht in der vom Megalock Compiler erzeugten Datei, da diese bei einem erneuten Compilieren überschrieben wird.

Wir hoffen, dass wir Ihnen den ersten Einsatz mit Megalock gut beschrieben haben und bitten Sie jetzt, mit den weiteren zur Verfügung stehenden Megalock Befehlen ein bisschen herum zu experimentieren, damit Sie die Vielfältigkeit und den damit verbundenen Schutz für Ihre Software, effizient einsetzen.

Megalock Teil II

Das Megalock Programm

SysTech
Entwicklungsges. für Mikroprozessorsysteme mbH

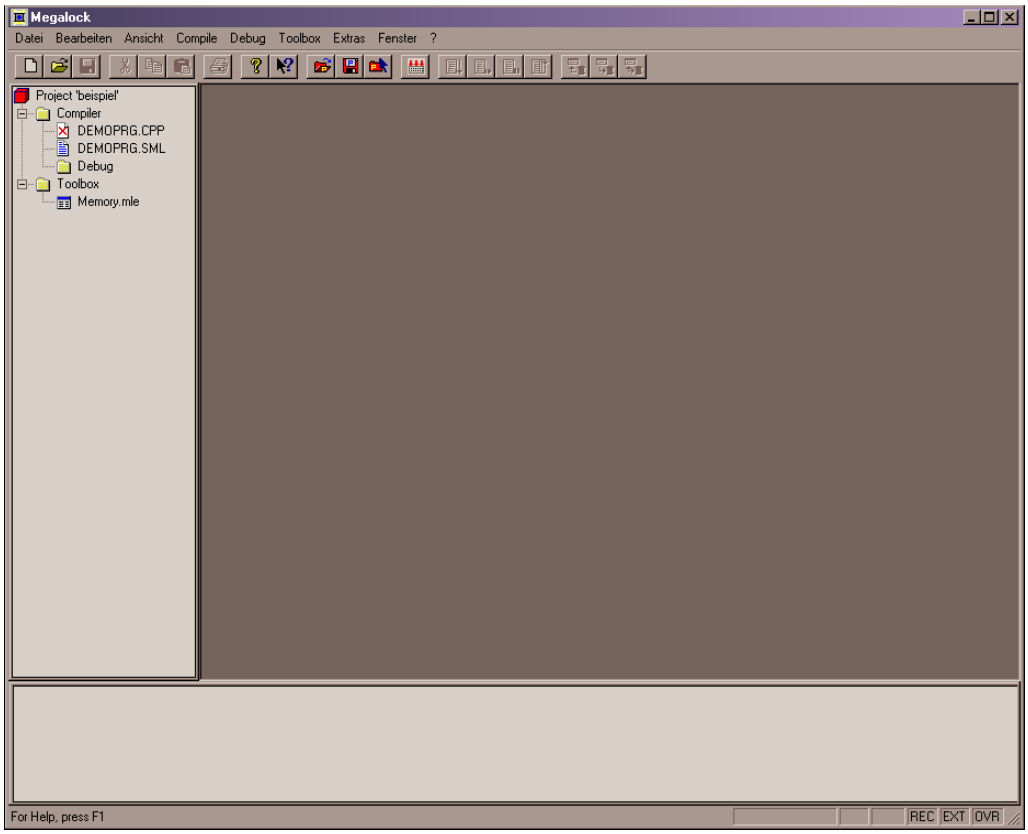
DAS MEGALOCK PROGRAMM

Das Megalock Programm (ML.EXE) ist für alle unserer Kunden gleich, benötigt aber eine kundenspezifische Datei MLUCxxxxxxx.DLL. Die MLUCxxxxxxx.DLL wir für Ihre kundenspezifische Megalock-Serie erstellt und ist auch nur mit dieser lauffähig. Auf der Megalock-CD im Ordner „Megalock“ befindet sich die MLUC710200.DLL, diese ist eine Test- und Demoversion. Ihre kundenspezifische DLL kopieren Sie auch in diesen Ordner. Das Programm ML.EXE wir ihre DLL vorrangig verwenden.

Sie können ihre MLUCxxxxxx.DLL aber auch auf einen Serverlaufwerk speichern und durch entsprechende Zugriffsrechte die Nutzen nur bestimmte Personen erlauben. Dann ist das Megalock Programm mit Path und Name der MLUCxxxxxx.DLL zustarten, z.B. ML.EXE j:\Projekt\Secur\MLUC712345.DLL.

Das heißt: Mit dem Megalock Programm und Ihrer Datei MLUCxxxxxxx.DLL können Sie keine anderen kundenspezifischen Megalock Dongles auslesen und verändern, andere Anwender von Megalock können Ihre Megalock Dongles somit auch nicht auslesen oder verändern.

Das Megalock Programm ist in drei Programmteile dem Megalock Compiler, -Debugger, -Toolbox unterteilt, mit denen Sie alle notwendigen Arbeitsschritte, von der individuellen Einbindung des Megalock Dongles in Ihren Programmen bis hin zur serienmäßigen Erstellung der Megalock Dongles nach Ihrer spezifischen Datenspeicherbelegung ausführen können. Sie können das Megalock Programm mit Angabe einer Projekt Datei „ML.EXE Projekt.mlp“ starten, wir keine Projekt Datei „ML.EXE“ angegeben wird das Megalock Programm mit dem zuletzt verwendeten Projekt gestartet.



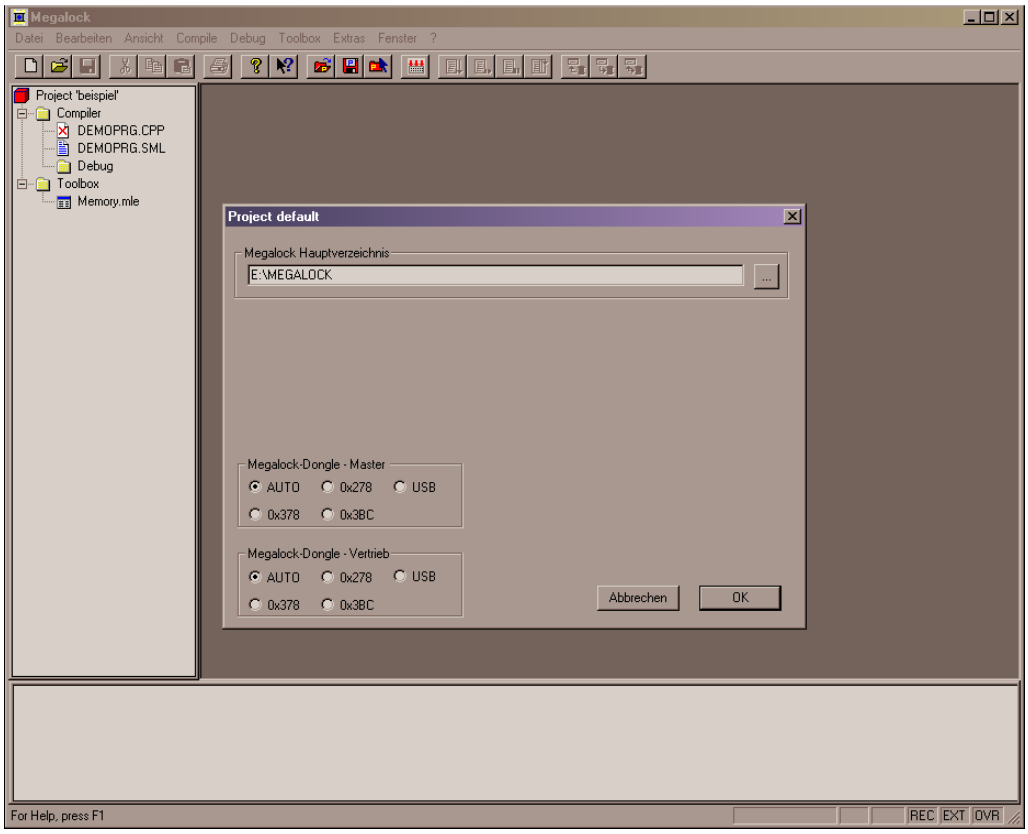
Links in Fenster wir das aktuelle Projekt mit den zu diesem Projekt gehörenden Dateien und Modulen angezeigt. Rechts im Fenster werden je nach Funktion, Quelldateien im Editorfenster, Module zum Testen im Debugfenster und Dialogfenster zur Bearbeitung des Datenspeichers dargestellt. Unten im Fenster werden Fehler- und Statusmeldungen angezeigt.

PROJEKT EINSTELLUNGEN

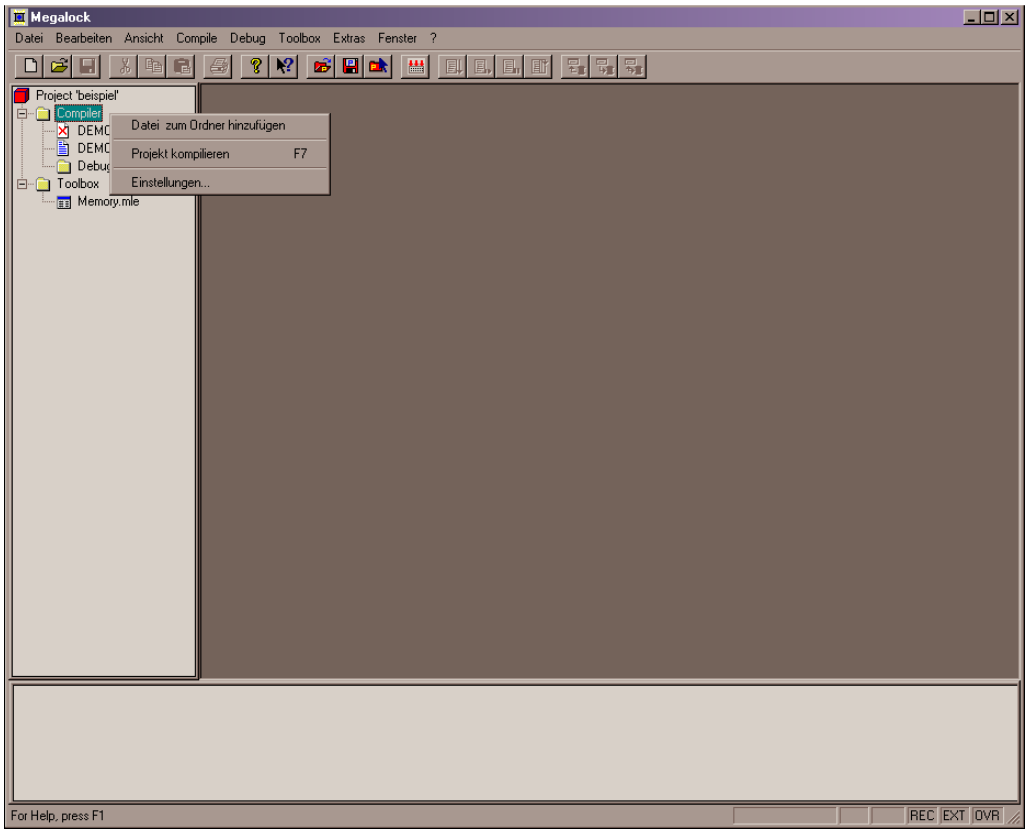
Projekt Einstellungen können durch drücken der rechten Maustaste über den Einträgen „Projekt“, “Compiler“ und „Toolbox“ und über den im Ordner befindlichen Einträgen vorgenommen werden.

Wir die Maustaste über **„Project“** gedrückt erscheint das Dialogfenster „Project default“. In diesem Fenster wird der Path des Megalock Ordners angegeben und wo die MegalockDongle zu suchen sind. Wird „AUTO“

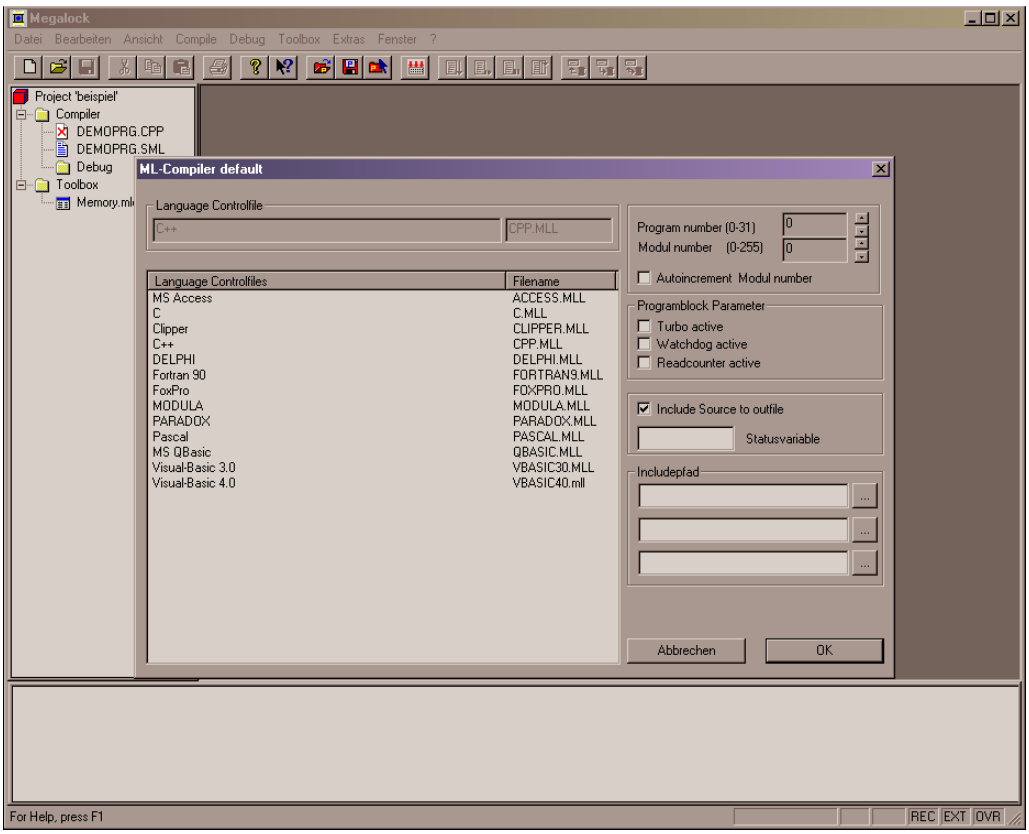
eingestellt, wird zuerst nach dem USB-Dongle gesucht, wird dort nicht der passenden Dongle gefunden wird auf den Parallel-Port gesucht, sofern vorhanden.



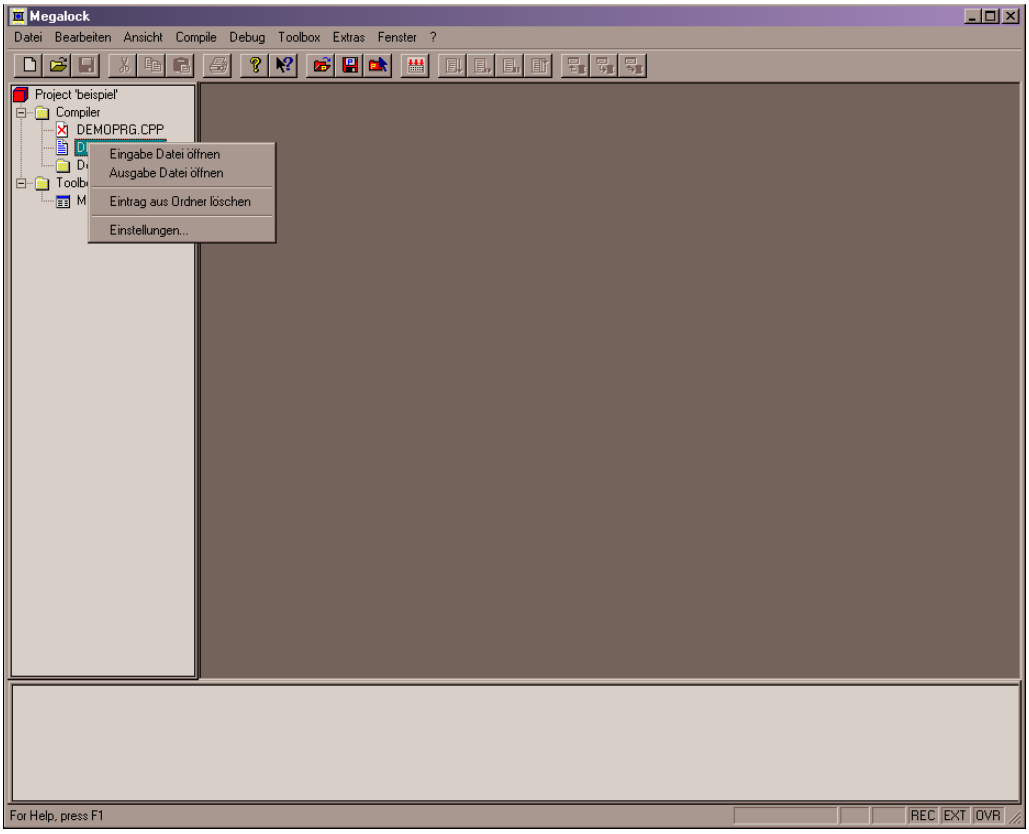
Wir die rechte Maustaste über **„Compiler“** gedrückt erscheint ein weiteres Menü. Über dass, Einstellung für den ML-Compiler vorgenommen werden können, es können weitere Dateien zum Projekt hinzugefügt werden und das gesamte Projekt kann kompiliert.



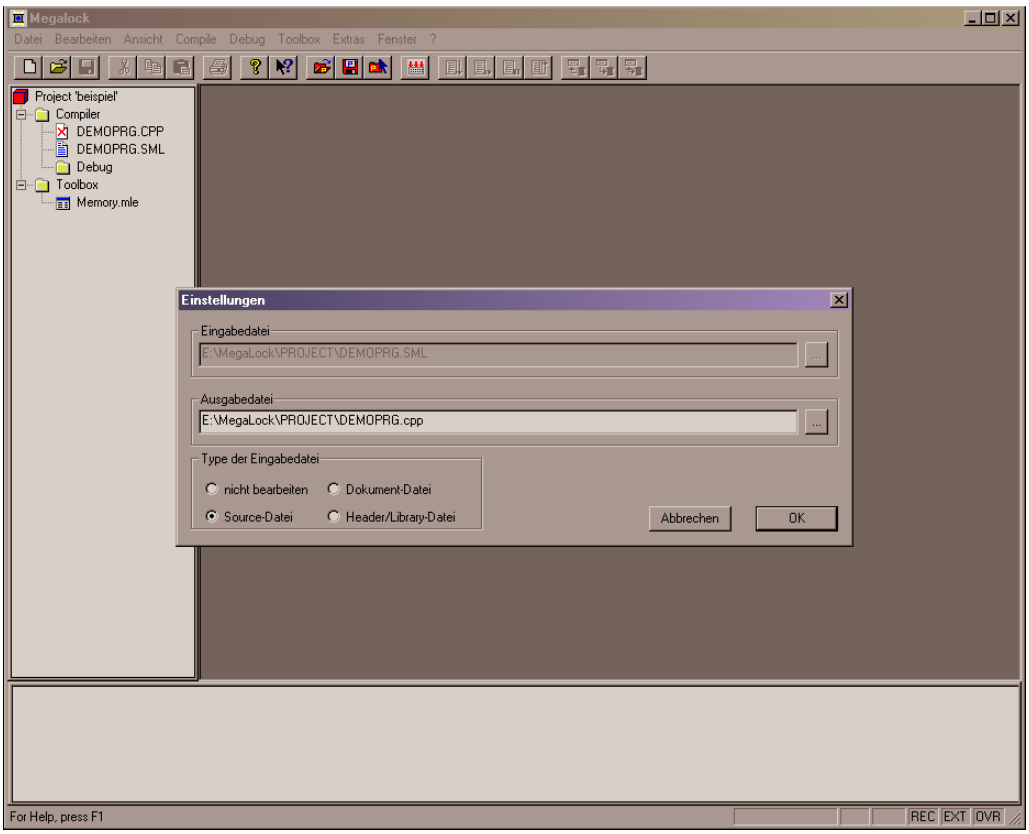
Wir der Menüpunkt „Einstellungen“ gewählt wird das nachfolgende Fenster angezeigt. Dort wird ihre Programmiersprache bzw. die Steuerdatei für ihre Programmiersprache festgelegt. Dann können Default-Einstellungen vorgenommen werden die dann für alle Megalock Programmteile gelten. Dann können Sie noch 3 Ordner definieren in denen nach Includedateien gesucht werden soll.



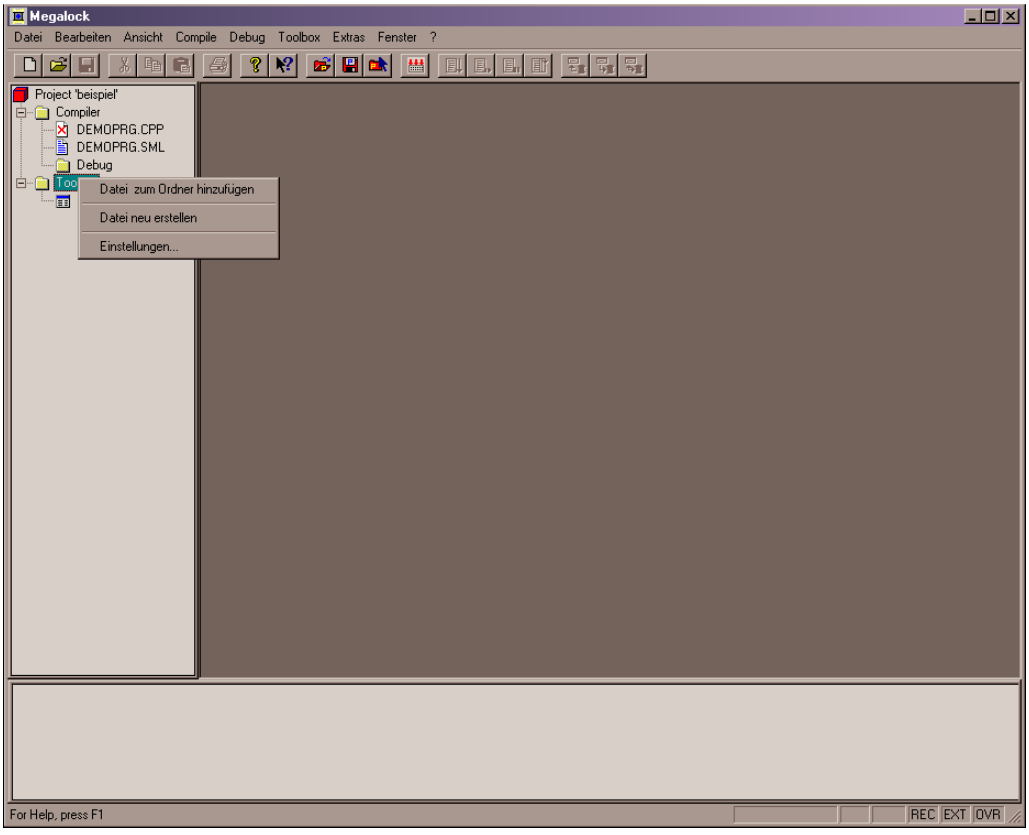
Wir die rechte Maustaste über den einzelnen Einträgen im „Compiler“-Ordner gedrückt erscheint ein weiteres Menü. Über die weitere Funktionen und Einstellungen zur jeweiligen Datei vorgenommen werden können.



Über **„Einstellungen“** wird Eingabe- und Ausgabedatei und Type zur jeweiligen Datei festgelegt.



Wir die rechte Maustaste über **„Toolbox“** gedrückt erscheint das Menü, über dass, Einstellung für die „Toolbox“ vorgenommen werden können, es können weitere Datei zum Projekt hinzugefügt werden und neue Dateien können erstellt werden.

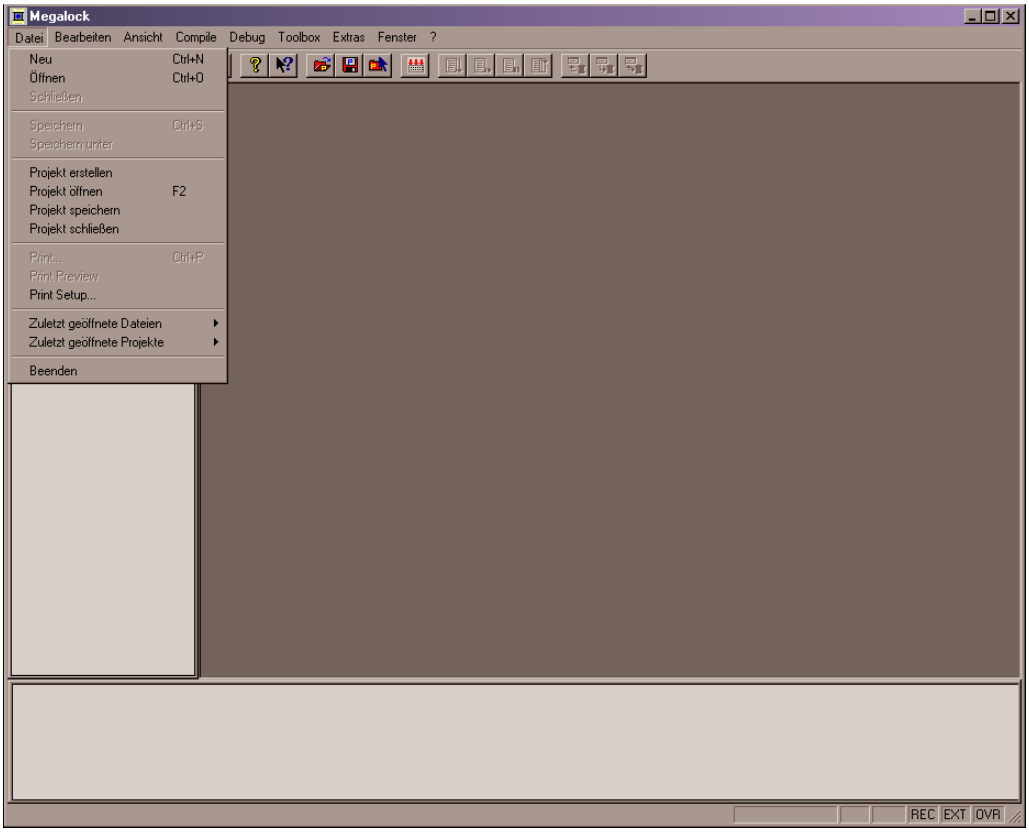


Wir die rechte Maustaste über den einzelnen Einträgen im „Toolbox“-Ordner gedrückt erscheint ein weiteres Menü zum Öffnen der Datei und zum Löschen des Eintrages.

MEGALOCK MENÜLEISTE

Die meisten Menüpunkte entsprechen Windows-Standard Funktionen und sollten daher nicht weiter zu erklären sein.

„Menü Datei“



Über **„Neu“** und **„Öffnen“** werden nur einzelne Dateien im Editorfenster angezeigt. Zum Projekt gehörende Dateien sollten im „Projekt“ eingetragen werden und dann durch einfaches Anklicken des Eintrags und „Compiler“ und „Toolbox“ geladen werden.

Projekte können unter **„Projekt erstellen“** und **„Projekt öffnen“** erstellt bzw. geladen werden.

„Menü Compiler“

Über „Projekt kompilieren“ werden die im Ordner „Compiler“ eingetragenen Quelldateien übersetzt. Weiter Informationen zum Compiler sind unter „Der Megalock-Compiler“ angegeben.

„Menü Debug“

Die Funktionen unter „Debug“ werden nur freigegeben, wenn zuvor ein Modul unter dem Ordner „Debugger“ ausgewählt wurde. Damit Module überhaupt im Ordner „Debugger“ erscheinen, müssen zuvor die Quelldateien kompiliert worden sein. Weiter Informationen zum Debugger und den Menü-Funktionen sind unter „Der Megalock-Debugger“ angegeben.

„Menü Toolbox“

Die Funktionen unter „Toolbox“ werden nur freigegeben, wenn zuvor eine Toolbox-Datei geladen wurde. Weiter Informationen zur Toolbox und den Menü-Funktionen sind unter „Die Megalock-Toolbox“ angegeben.

„Menü Extras“

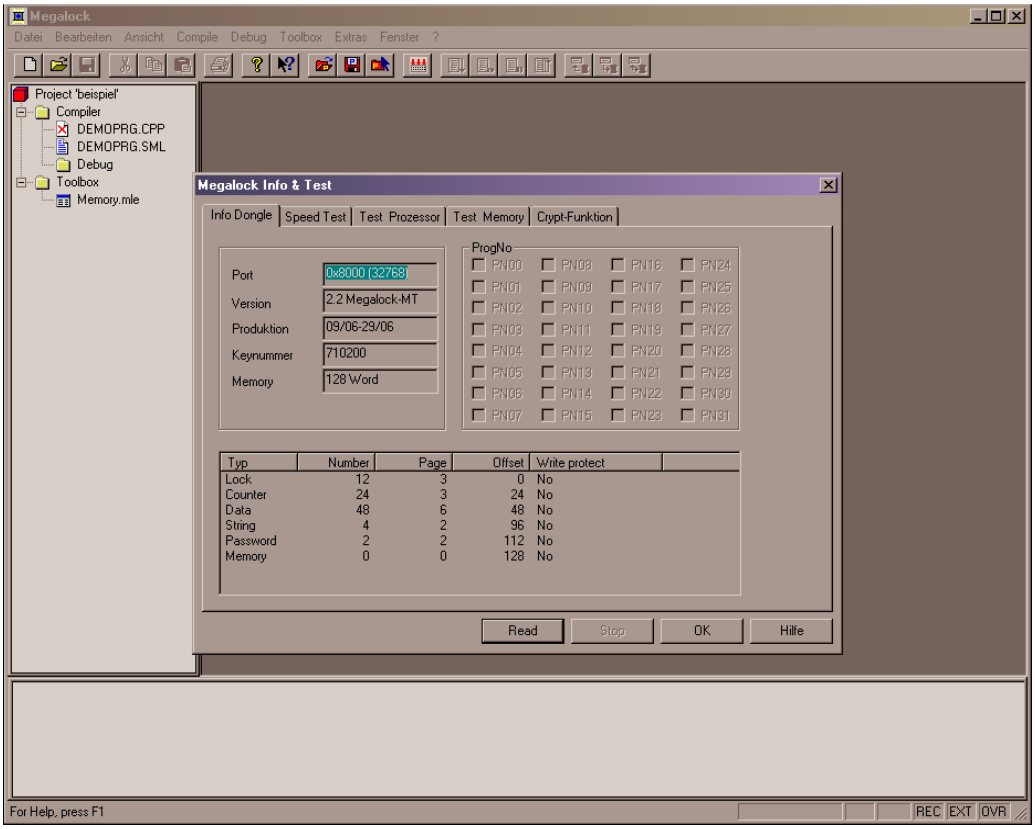
Über die Funktion „Dongle Info&Test“ werden Informationen über den angeschlossenen Dongle angezeigt und des weiteren besteht die Möglichkeit den angeschlossenen Dongle einem Funktionstest, zu unterzeihen. Weitere Informationen finden siehe nachfolgend unter „Megalock Info & Test“.

MEGALOCK INFO & TEST

Über das Dialogfenster „Megalock Info&Test“ werden Informationen über den angeschlossenen Dongle angezeigt und des weiteren besteht die Möglichkeit den angeschlossenen Dongle einem Funktionstest, zu unterzeihen.

Info Megalock

„Info Megalock“ zeigt die Megalock Systeminformationen und die Seitenaufteilung der einzelnen Datengruppen mit Zugriffsrechten an.



Port

Schnittstelle, an der der Megalock Dongle gefunden wurde. Ist der Wert > 0x8000 handelt es sich um den USB-Port.

Version

Hier wird die Versionsnummer und ob es sich um einen Master- (Megalock-MT) bzw. Vertriebsdongle (Megalock-VT) handelt, angezeigt.

Produktion

Hier wird das Fertigungsdatum der Hardware und das Datum, wann der Megalock Dongle mit Ihren kundenspezifischen Daten versehen wurde, angezeigt.

Keynumber

Hier wird die Megalock Donglenummer angezeigt.

Memory

Hier wird die Größe des Datenspeichers in 16-Bit angegeben die Ihnen zur Verfügung steht.

Die nachfolgende Tabelle zeigt Ihnen die Einstellung der Datenbereiche:

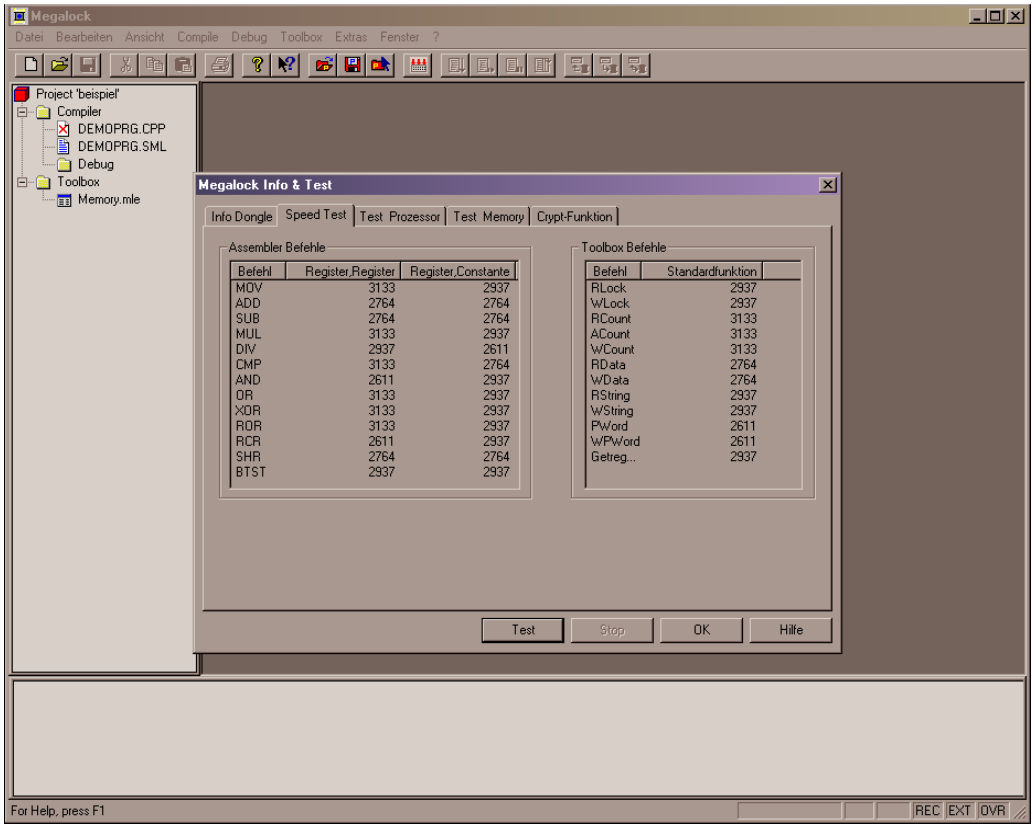
Number	Anzahl der Tabellenplätze
Page	Anzahl der Seiten
Offset	Startadresse der einzelnen Datenbereiche innerhalb des Datenspeichers
Write protect	Schreibschutz für den Datenbereich

ProgNo

Hier werden Ihnen die freigeschalteten Programmnummern angezeigt.

Speed Test

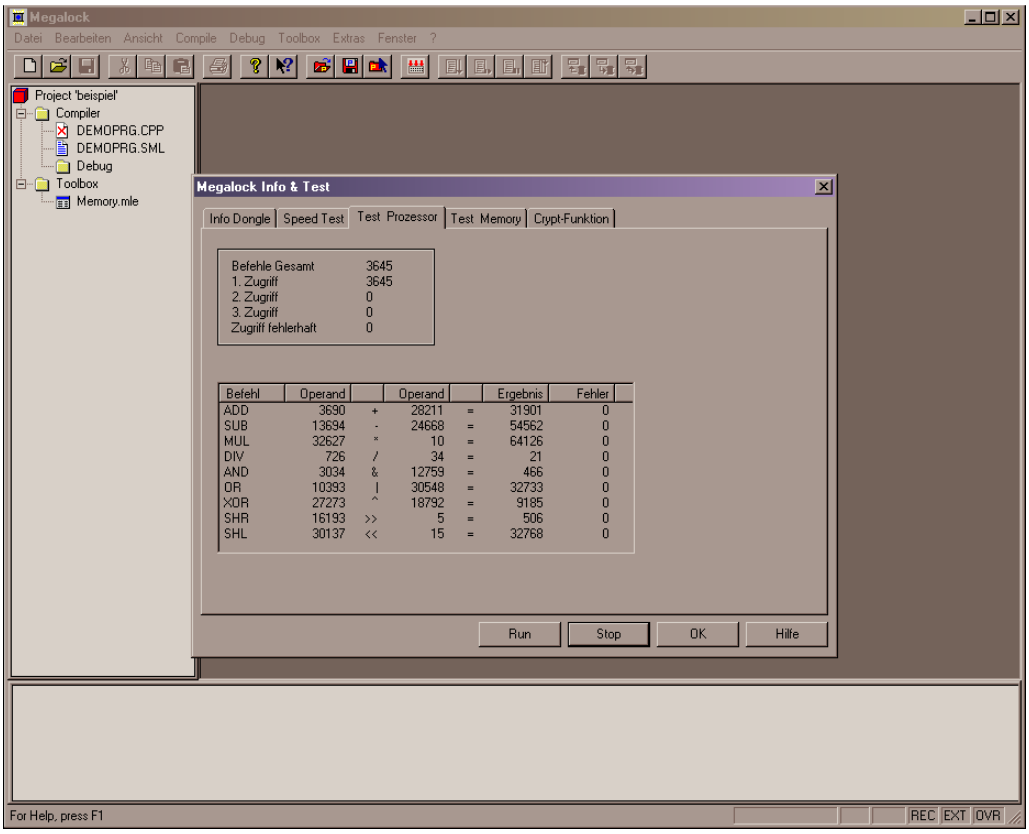
Hier können Sie die Performance des Megalock Dongle mit Assemblerbefehlen und Toolboxbefehlen ermitteln.



Die angegeben Werte beziehen sich auf Befehle pro Sekunde. Wir der Megalock Dongle an einem USB-Port betrieben liegt die Anzahl der Befehle bei 300 bis 2500 pro Sekunden. Die Performance wird durch den USB-Port gegrenzt und wenn der Megalock-Dongle über einen USB-HUB betrieben wird, ist die Performance meist besser als direkt am PC.

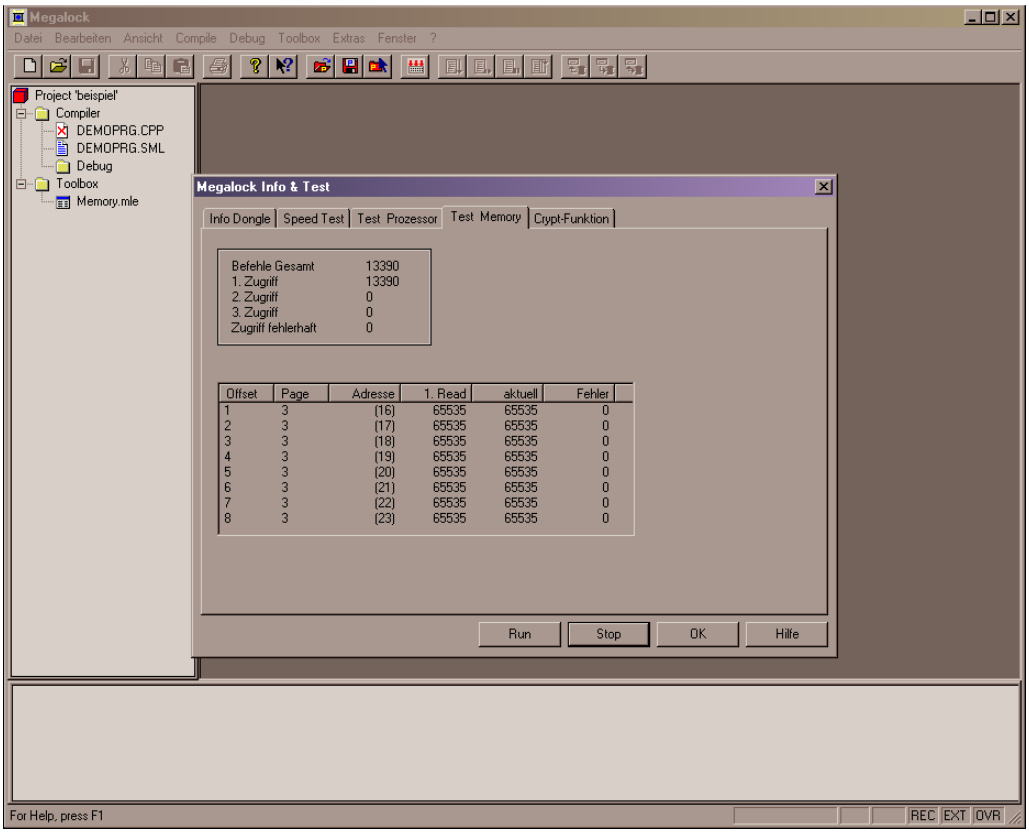
Test Prozessor

Mit Test Prozessor können Sie Dauertests auf dem Megalock Dongle durchführen. Es werden Assemblerbefehle auf dem Dongle durchgeführt, und es ist zu erkennen wie viele Zugriffe auf den Dongle durchgeführt wurden (Total access) und wie viele davon beim ersten, zweiten und dritten Versuch ausgeführt wurden (Access successful) bzw. nicht ausgeführt wurden (Error Access). In der Regel sollte der Zugriff bereits beim ersten Versuch durchgeführt werden. Innerhalb der Tabelle wird der Befehl, die beiden Werte, das Ergebnis sowie die aufgetretenen Fehler dargestellt.



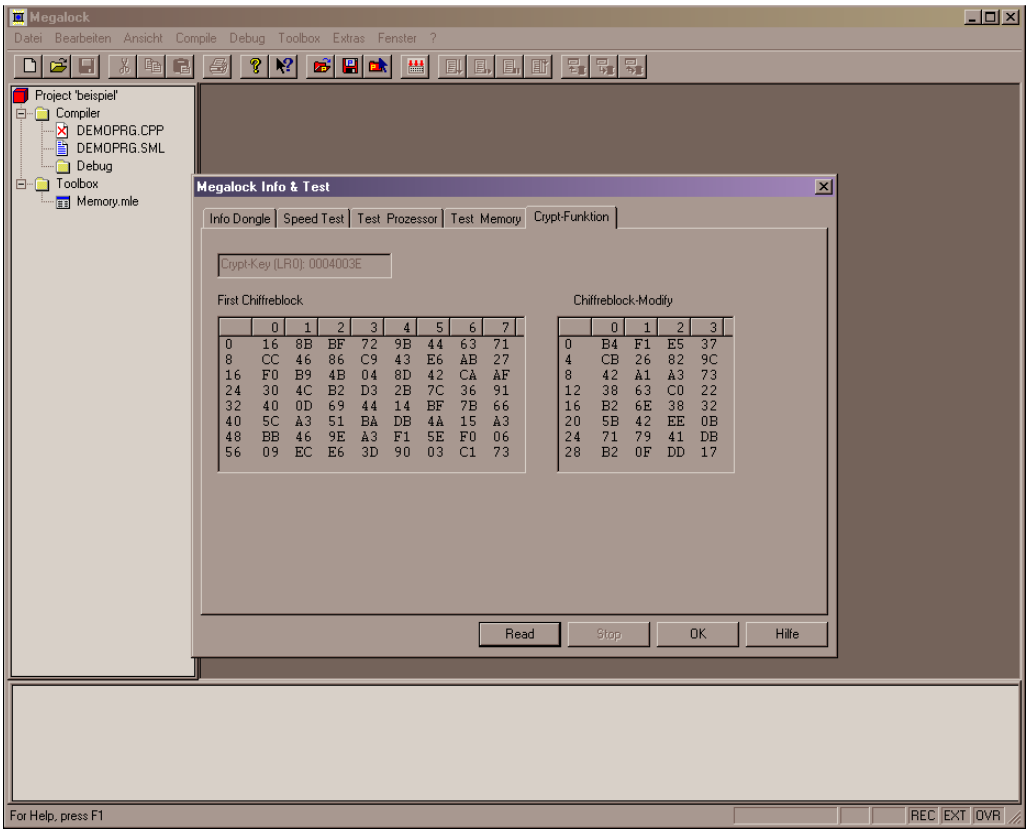
Test Memory

Mit dieser Testfunktion wird ein Dauerzugriff auf den Datenspeicher durchgeführt. Zu Beginn wird der gesamte Inhalt des Datenspeichers ausgelesen. Nachfolgend wird über eine Randomfunktion der Datenspeicher ausgelesen und mit dem zuvor gelesenen Wert verglichen. Treten Differenzen auf, werden diese unter Error angezeigt.



Test Cryptlock

In dem Fenster Cryptlock werden zwei Chiffrierblöcke angezeigt, die innerhalb des Megalock Dongles aus den 32 Bit des Registers LR0 und einen kundenspezifischen Schlüssel generiert werden.



First Chiffreblock

	0	1	2	3	4	5	6	7
0	16	8B	BF	72	9B	44	63	71
8	CC	46	86	C9	43	E6	AB	27
16	F0	B9	4B	04	8D	42	CA	AF
24	30	4C	B2	D3	2B	7C	36	91
32	40	0D	69	44	14	BF	7B	66
40	5C	A3	51	BA	DB	4A	15	A3
48	BB	46	9E	A3	F1	5E	F0	06
56	09	EC	E6	3D	90	03	C1	73

Chiffreblock-Modify

	0	1	2	3
0	B4	F1	E5	37
4	CB	26	82	9C
8	42	A1	A3	73
12	38	63	C0	22
16	B2	6E	38	32
20	5B	42	EE	0B
24	71	79	41	DB
28	B2	0F	DD	17

Über den in der linken Tabelle dargestellten Chiffrierblock wird ein Datenblock von 64 Byte Exklusivoder verknüpft. Mit der in der rechten Tabelle dargestellten 32 Byte wird die linke Tabelle nach jedem Datenblock (64 Byte) modifiziert.
Die Erzeugung des Chiffrierschlüssels und anschließender Verschlüsselung der Daten wird über die Toolbefehle Clock oder Chata durchgeführt.

Der Verschlüsselungsprozess:
Die eigentliche Verschlüsselung der von Ihnen angegebenen Daten wird innerhalb des Rechners durchgeführt. Jeder Datenblock (64 Byte) wird ‘exklusiv oder’ mit dem First Chiffreblock verschlüsselt. Nach der Verschlüsselung eines Datenblockes wird der First Chiffreblock anhand des Chiffreblock-Modify neu generiert, so dass für den nächsten zu verschlüsselten Datenblock ein neuer Chiffreblock zur Verfügung steht.
Auf diese Art und Weise können Datenmengen (1 - 64 KByte) schnell und sicher verschlüsselt bzw. entschlüsselt werden.

Mit der Option **„Next“** wird das Register LR0 um eins dekrementiert und die beiden Chiffrierblöcke werden neu generiert.

Mit dem USB-Dongle stehen weitere Möglichkeiten zur Verschlüsselung zur Verfügung (AES-128 Bit).

DER MEGALOCK COMPILER

Der Megalock Compiler ist in der Lage jede Quelldatei, unabhängig der von Ihnen eingesetzten Programmiersprache (PASCAL, C, C++, FORTRAN, Assembler, Modula, BASIC , FOXPRO etc.), zu bearbeiten, da der Megalock Compiler nur die Befehle und Anweisungen bearbeitet, die entsprechend gekennzeichnet sind. Alle anderen Textzeilen werden unbearbeitet in die Output Datei übernommen. Die Anweisungen und Befehle für den Megalock Compiler werden übersetzt und in eine für Ihren Compiler spezifischen Syntax ausgegeben. Die von Ihnen eingesetzt Programmiersprache ist zuvor anzugeben.

Der Compiler liest Zeile für Zeile aus dem Quellcode ein und sucht nach **##DEF.., ##OPEN..**, **##CLOSE..**, **##BEGIN** und Einzelbefehle, die mit **ML..** gekennzeichnet sind.
Bevor die dem Projekt zugeordneten Dateien compiliert werden, wird die Language Control Datei mit der Extension **„MLL“** aus dem aktuellen Ordner bzw. aus MEGALOCK\LANGUAGE geladen und umgesetzt.

Anschließend wird die Headerdatei MEGALOCK.MLH aus dem aktuellen Ordner bzw. aus MEGALOCK\INCLUDE, sofern vorhanden, bearbeitet.

Dann erst werden die Quelledateien vom Compiler bearbeitet. Die vom Megalock Compiler erzeugte Outputdatei enthält dann nur noch Programmcode, der für die entsprechende Programmiersprache verständlich ist. Der Dateiname der neuen Datei wird über (rechte Maustaste über Quelldateieintrag drücken) „Einstellungen“ festgelegt.
Anschließend können die Dateien mit Ihrem Compiler compiliert werden.

PROJEKTDATEI ERSTELLEN, ÖFFNEN

Über Menü „Datei“ können sie die Projekte erstellen, laden und dann bearbeiten. Das aktuelle Projekt wir im linken Fenster angezeigt. Durch drücken der rechten Maustaste können sie die Einträge in den Ordnern „Compiler“ und „Toolbox“ bearbeiten.

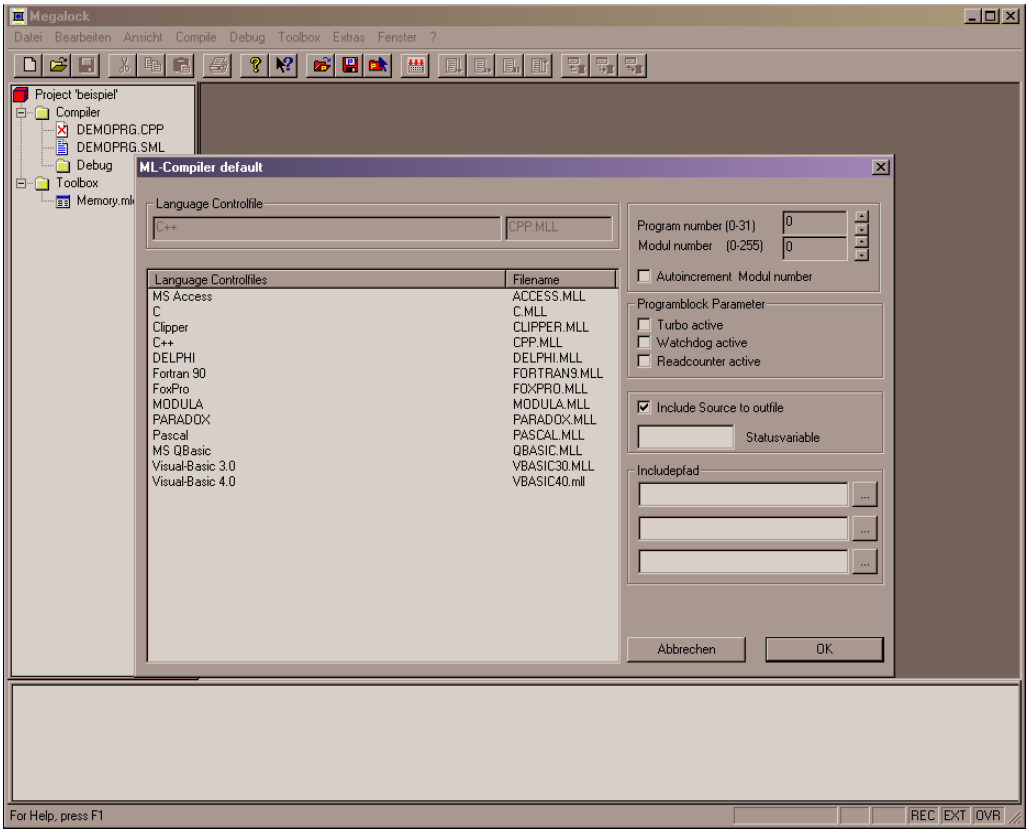
Alle Projekt spezifischen Dateien sollten im Ordner „Compiler“ eingetragen werden. Hier zu gehören auch mögliche Headerdatei (*.MLH). Die Headerdatei „Megalock.MLH“ wird vom Megalock Compiler automatisch eingebunden. In der Headerdatei „Megalock.mlh“ sollten auch nur Projekt unabhängige Definition gemacht werden.

PROJEKT KOMPILIEREN

Über das Menü **„Compile“** und dann **„Projekt kompilieren“** bzw. durch drücken der F7-Taste wird das aktuelle Projekt kompiliert. Eventuelle Fehler werden im Fenster unten angezeigt.

PROJEKT EINSTELLUNGEN

Einstellungen für den Megalock Compiler und die defination der eingesetzten Programmiersprache erfolgt im Fenster „ML-Compiler default“. Diese Fenster durch drücken der rechten Maustaste über dem Ordner **„Compiler“** angezeigt.



In diesem Dialogfenster werden ML-Compiler-Defaultwerte eingestellt. Die hier eingestellten Werte können innerhalb der Megalock Headerdateien und im Quellcode überschrieben werden.

Program Number

Durch die Anweisung PROGNO wird dem ML-Compiler mitgeteilt unter welcher Programmnummer die nachfolgenden Befehle ausgeführt werden sollen. Für die Programmnummer kann eine numerische Konstante im Bereich 0 bis 31 angegeben werden. Wird die Anweisung PROGNO im Quellcode nicht angegeben, so gilt die hier eingetragene PROGNO. Der als PROGNO spezifizierte Wert beeinflusst zusätzlich die Verschlüsselung der Befehlskennziffern (OPCODE). Eine Änderung der Programmnummer kann in einer Headerdatei oder im Quellcode mit der Anweisung PROGNO durchgeführt werden.

Modul number

Mittels der Anweisung MODULNR können Sie die Kryptierung der Befehlskennziffern (OPCODE) verändern. Die gültigen Modulnummern liegen im Bereich von 0 bis 255. Durch Angabe unterschiedlicher Modulnummern in Megalock Programmblöcken erhalten gleiche Megalock Befehle in unterschiedlichen Programmblöcken auch unterschiedliche Befehlskennziffern (OPCODE).

Turbo active

Durch die Anweisung TURBO wird die Ausführung der einzelnen Befehle des jeweiligen Megalock Programmblockes beschleunigt. Die Übertragung der Megalock Keynummer bzw. das Testen ob der Megalock Dongle noch am Rechner vorhanden ist, wird nur beim dem ersten Befehl durchgeführt. Die nachfolgenden Befehle bis einschließlich des Befehls ##END, werden dann ohne Übermittlung der Megalock Keynummer ausgeführt. Diese Funktion sollten Sie zurzeit nur dann aktivieren, wenn Sie sicherstellen können das während der Ausführung des jeweiligen Megalock Programmblockes keine Anwendung auf den jeweiligen COM oder LPT Port zugreift, auf dem sich der Megalock Dongle befindet.

Die Einstellung ist nur für den LPT Port wirksam. Auf den USB-Dongle hat diese Funktion keine Auswirkung.

Watchdog active

Mit der Anweisung WATCHDOG wird die Laufzeitüberwachung für das jeweilige Megalock Programmblockes innerhalb des Megalock Dongle aktiviert. Der WATCHDOG-Timer wird jeweils nach Erhalt des OPCODES ausgelesen und in das Register LR19 übertragen, auf 0 gesetzt und wieder gestartet. Ist der ausgelesene Wert größer als der Eingestellte, wird ein Error-Code (0x21) und das WATCHDOG-Flag im Systemregister gesetzt. Durch die WATCHDOG-Funktion kann zum Beispiel ein Debugger auf dem Rechner erkannt werden. In der ML-Shell wird die Anweisung TURBO und WATCHDOG eingesetzt.

Laufzeitschwankungen sollten Sie bei allen Betriebssystemen berücksichtigen.

Readcounter active

Die Anweisung **READCT** aktiviert den Schutzmechanismus zum Auslesen der Megalock Register (R0-R19). Bevor Sie den Inhalt eines Megalock Registers mit den Befehlen **Getreg** und **movs** auslesen, muss das RC-Register des Megalock Dongle mittels des Befehls **src** gesetzt werden. Die Befehle Getreg und movs werden dann nur noch ausgeführt, wenn das RC-Register > NULL ist. Das RC-Register wird jeweils um 1 dekrementiert. Ist der Inhalt des RC-Registers NULL, werden die Befehle nicht mehr ausgeführt und das Errorflag sowie der ErrorCode (0x33) werden gesetzt. Der Schutzmechanismus kann nur in Programmblöcken aktiviert werden.

Include Source to outfile

Mit dieser Defaulteinstellung haben Sie die Möglichkeit die Megalock Befehlszeilen mit Kommentarzeilen in der Outputdatei mitauszugeben.
Mit der Anweisung NOSOURCE haben Sie innerhalb der Quelldatei die Möglichkeit, die Ausgabe des Quellcodes in die Output-Datei, zu unterdrücken.

Name of Status Variable

Enthält das Feld einen Namen für die Statusvariable, so wird am Ende eines Programmblocks, die Anweisung zur Übertragung des Megalock Status in die Variable eingefügt. Enthält das Feld keinen Eintrag, erfolgt keine automatische Statusabfrage. Die Statusabfrage kann in einem Programmblock über STATUS aktiviert werden.

Includepfad

Hier können 3 Ordner angegeben werden in denen nach Includedateien gesucht werden soll.

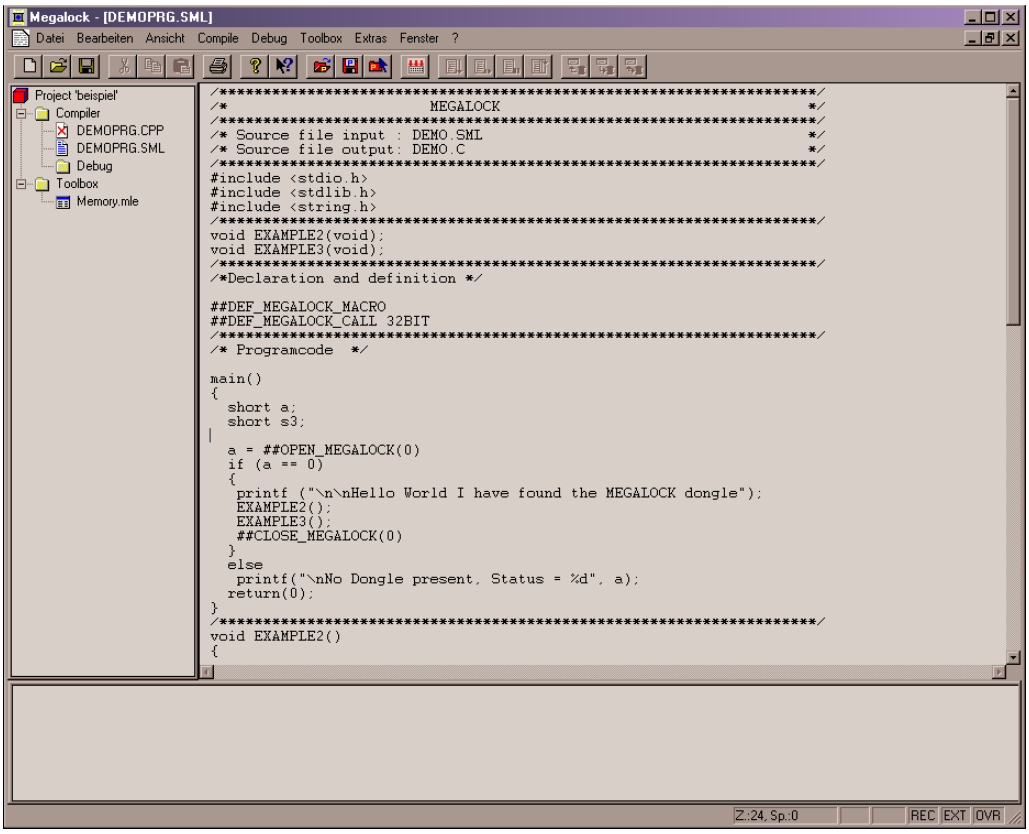
Language Controlfile

Das Language Controlfile dient zur syntaktischen Umsetzung der Megalock Befehle in die von Ihnen eingesetzte Programmiersprache. Hier werden die zurzeit zur Verfügung stehenden Steuerdateien für verschiedene Compiler dargestellt. Die Steuerdateien befinden sich in dem Ordner \MEGALOCK\LANGUAGE und haben die Endung ***.MLL**. Weitere Steuerdateien können in diesen Ordner untergebracht werden und werden automatisch in der Auswahlliste dargestellt. Änderungen an den vorhandenen Steuerdateien sollten nicht vorgenommen werden. Kopieren Sie ggf. eine Steuerdatei unter anderem Namen und passen diese auf Ihre Bedürfnisse an. In der Zeile „Language Controlfile“ ist der aktuelle Name der ausgewählten Steuerdatei zu sehen.

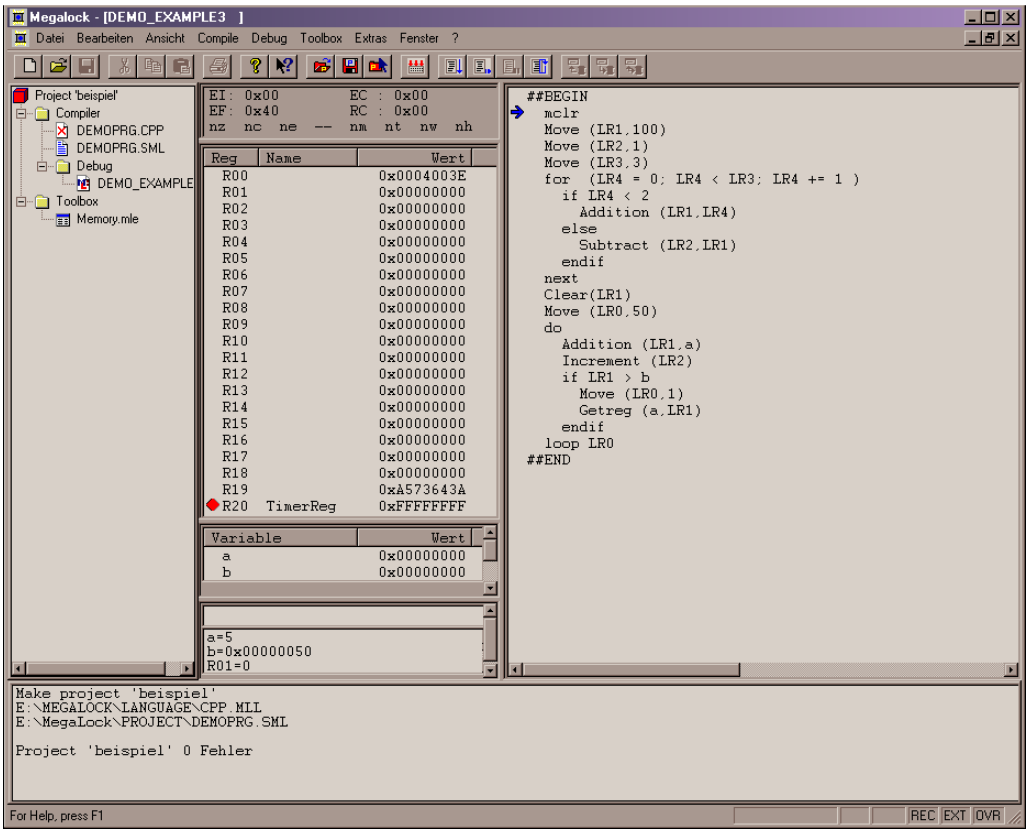
Sollte keine Control Datei für die von Ihnen verwendete Programmiersprache vorhanden sein, so setzten Sie sich bitte mit uns in Verbindung, und wir werden kurzfristig die notwendigen Dateien zur Verfügung stellen.

DER MEGALOCK DEBUGGER

Der Megalock Debugger ist ein Hilfsprogramm, mit dem Sie den logischen Ablauf innerhalb der Megalock Programmblöcke auf einfache Weise testen können, da Sie unmittelbar nach der Befehlsausführung das Ergebnis im Registerfenster oder im Dump Fenster sichtbar ist. Der Debugger wird nach Auswahl eines der Einträge im Ordner **„Debug“** gestartet. In dem Ordner **„Debug“** sind aber nur die Module eingetragen, die fehlerfrei durch den Megalock Compiler übersetzt werden konnten.



Wenn Sie einen Programmblock im Ordner **“Debug“** ausgewählt haben, werden neben dem Quelltext, die Megalock Flags und Register, die verwendeten Variablen und letzten 50 von Hand modifizierten Registern und Variablen Werte angezeigt.



Register

Im Fenster Register werden die Registerinhalte des Megalock Dongles sowie die Systemregister angezeigt. Veränderungen in den Registern werden farblich markiert. In dem oberen Teil des Fensters werden die Register (R1 bis R20) angezeigt. Innerhalb des Fensters kann mit den Cursortasten geblättert werden. Bis auf das Register R20 sind alle Register universell einsetzbar, wobei das Register R20 als Timer Register belegt ist. Änderungen der Registerinhalte können durch Anklicken des Registers mit der Maus durchgeführt werden.
Im oberen Teil des Fensters wird das Systemregister 1 und 2 angezeigt.

EI Error Id	EF Epromflag	EC Errorcounter	RC Readcounter

Die Error Id enthält den zuletzt aufgetretenen Fehlercode.
Das Epromflag enthält die Write protect Flags für die einzelnen Datenbereiche des Datenspeichers.

Der Error Counter enthält die Anzahl der Fehler.
Der Read Counter wird, sofern READCT aktiviert wurde, jeweils um 1 dekrementiert, wenn ein Megalock Register ausgelesen wird.

nh HAT	nw WD	nt TM	nm ME		ne ER	nc CY	nz ZR
Halt Flag	Watch dog Flag	Time Out Flag	Master Error Flag	NC nicht belegt	Error Flag	Carry Flag	Zero Flag

Sofern die Flags nicht aktiv sind, werden die Kürzel in Kleinbuchstaben angezeigt, ansonsten groß. Die Flags Halt, Watchdog, Timeout werden innerhalb des Debuggers nicht verändert.
Weitere Informationen über das Systemregister können im Anhang nachgelesen werden.

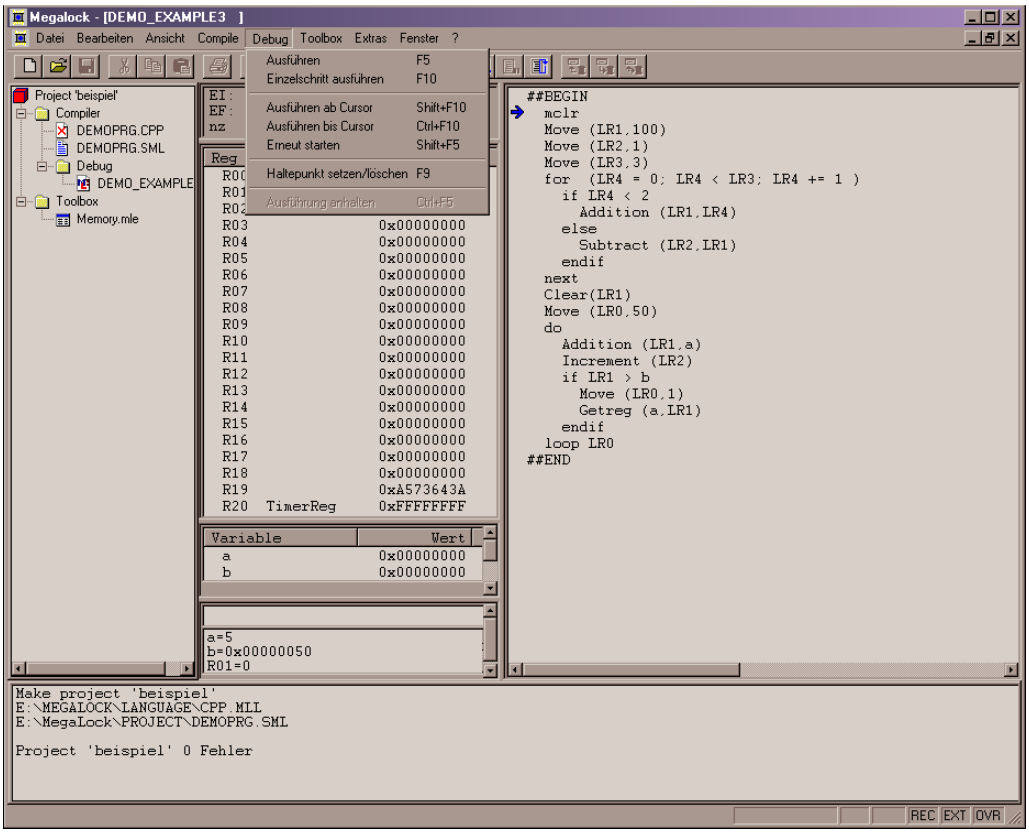
Variable

Im Fenster Variable werden die zum Programmblock gehörigen Variablen mit dem aktuellen Inhalt aufgelistet.

Änderungen an Variablen und Registern können im darunterliegenden Fenster vorgenommen werden. Die zuletzt verwendeten Werte werden gespeichert und können mit der RETURN-Taste übernommen.

Programmblock ausführen

Über die Funktion des Menüs „Debug“ und oder den entsprechenden Funktionstasten kann der Programmablauf gesteuert werden.



Ausführen F5

Mit der Option „Ausführen“ wird der Programmblock ab dem aktuellen Programmcounter (>) Position im Debugger ausgeführt. Das Modul wird bis zum Erreichen eines Breakpoints bzw. bis zum Ende ausgeführt.

Einzelschritt ausführen F10

Der Programmblock wird schrittweise abgearbeitet.

Ausführen ab Cursor Shift+F10

Mit dieser Option wird der Programmcounter auf die aktuelle Cursorposition gesetzt. Wenn Sie anschließend Trace, Run oder Step auswählen, wird der Programmblock ab dieser Position durchgeführt.

Ausführen bis Cursor Ctrl+F10

Der Programmblock wird von dem Programmcounter bis zur aktuellen Cursorposition ausgeführt.

Erneut starten Shift+F5

Wenn Sie einen Programmblock mit Run oder Step ausgeführt haben und wollen es noch einmal ausführen, ist ein Programm Reset notwendig. Der Programmcounter wird auf den Programmanfang gesetzt.

Haltepunkt setzten/löschen F9

Mit dieser Option können Sie innerhalb eines Programmblockes an beliebigen Stellen Haltepunkte setzten, bei der die Programmausführung stoppt. Fahren Sie mit dem Cursor auf die Zeile wo Sie den Breakpoint setzen wollen und betätigen die Taste „F2“, wodurch diese Zeile farblich markiert wird. Um einen Breakpoint zu entfernen betätigen Sie in dieser Zeile erneut die Taste „F2“.

Ausführung anhalten Ctrl+F5

Mit dieser Option wird die Ausführung des Programmblocks angehalten.

DIE MEGALOCK TOOLBOX

In der Megalock Toolbox stehen Ihnen Funktionen zur Bearbeitung des Datenspeichers zu Verfügung. Die Aufteilung des Datenspeichers ist in Lock, Counter, Daten, String und Password aufgeteilt. Aus Sicht des Anwenders werden diese Bereiche als externe Datentabellen verwaltet. Die Größe der jeweiligen Datentabellen kann in 16-Byte Schritten (Page) vergeben werden, wobei die maximale Anzahl der zur Verfügung stehenden Seiten nicht überschritten werden darf.

Datengruppen	Speicherbedarf	Anzahl je Seite (Page)
LOCK	4 Byte	4
COUNTER	2 Byte	8
DATA	2 /4 Byte	4/8
STRING	8 Byte	2
PASSWORD	16 Byte	1
MEMORY	2 Byte	8

Die Größe und Beschaffenheit des Datenspeichers ist unterschiedlich, Der Parallel-Dongle verfügt über 480 Byte (30 Page) E²-Speicher. Der USB-Dongle verfügt über 256 Byte E²-Speicher (16 Page) und bis zu 7936 Byte Flashspeicher, für die Toolbox-Befehle stehen aber nur bis zu 4080 Byte (255 Page) insgesamt zur Verfügung.

Der E²-Speicher kann laut Hersteller bis zu 1 Million mal geändert werden. Also wenn Sie stündlich die Daten in diesen Speicher ändern hat der Dongle eine Lebenserwartung von gut 100 Jahren.

Bei dem Flashspeicher sieht das aber schon etwas anders aus, dieser kann laut Hersteller bis zu 100.000 mal geändert werden. Das wäre ja eigentlich auch noch kein Problem dann würde nach obigem Beispiel die Lebenserwartung des USB-Dongles gut 10 Jahre. Leider kann der Flashspeicher nicht Byte-weise geändert sondern nur in 64-Byte-Blöcken. Z. B. in einem 64-Byte-Bock sind 16 DATA-Bereiche (je 32 Bit) untergebracht und sie würden jeden der 16 Bereiche ein Mal verändern dann wären das aber bereits 16 Änderungen pro Speicherstelle und nach unserem Beispiel (stündliche Änderung) wäre die Lebenserwartung des Dongles nur noch 8 Monate. Würden die Daten hingegen nur ein Mal am Tag geändert, wäre die Lebenserwartung wieder gut15 Jahre.

Also Datenwerte, die mehr als ein Mal am Tag geändert werden, sollten in den Ersten 16 Page (256 Byte) des USB-Dongle untergebracht werden.

Sollten die 256 Byte E²-Speicher, des USB-Dongles, für häufig geänderte Daten nicht ausreichen, so stehen neben den Toolbox-Befehlen auch Befehle „FLoad“ und „FSave“ zur Verfügung mit denen Sie 64Byte (Flashspeicher Block) direkt lesen und scheiben können.

Die Funktion der einzelnen Datengruppen wird nachfolgend beschrieben und der Zugriff auf die Datengruppen erfolgt in Ihrem Anwendungsprogramm über die Toolboxbefehle. Die Beschreibung der Toolboxbefehle ist im Handbuch Teil III beschrieben.

LOCK

Die Locktabelle enthält pro Tabellenplatz zwei Felder; den LockKey und LockData.
Ein Zugriff auf LockData erfolgt nur dann, wenn der übergebene Schlüssel mit dem in der Tabelle enthaltenen Lockkey übereinstimmt.
Darüber hinaus wird die Locktabelle zur Bildung des Kryptierschlüssels verwendet.

COUNTER

Die Countertabelle enthält pro Tabellenplatz ein Feld, den Counter. Beim Zugriff auf die Countertabelle bzw. eines Tabellenplatzes wird der Inhalt jeweils um 1 dekrementiert, bis der Wert 0 erreicht ist.
Standardmäßig ist die Countertabelle mit 16-Bit pro Feld aufgeteilt.
Auf die Countertabelle kann auch ein 32-Bit-Zugriff erfolgen, hierbei werden jeweils zwei 16-Bit-Tabellenplätze zu einem 32-Bit-Tabellenplatz zusammengefasst.

COUNTER()	16-Bit	COUNTER()	32-Bit
0 1	0x1111 0x2222	0	0x22221111
2 3	0x3333 0x4444	1	0x44443333

DATA

Standardmäßig ist die Datatabelle mit 16-Bit pro Feld aufgeteilt. In der DATA könnten Sie z.B. Grundeinstellungen oder Programmparameter ablegen.
Auf die Datatabelle kann auch ein 32-Bit-Zugriff erfolgen, hierbei werden jeweils zwei 16-Bit Tabellenplätze zu einem 32-Bit-Tabellenplatz zusammengefasst.

DATA()	16-Bit	DATA()	32-Bit
0 1	0x1111 0x2222	0	0x22221111
2 3	0x3333 0x4444	1	0x44443333

STRING

Die Stringtabelle dient zur Speicherung von einer ASCII-Zeichenfolge. Die Tabellenbreite ist mit einer Größe von 8-Byte angelegt, längere Zeichenketten können auf mehrere Stringtabellenplätze abgelegt werden.

PASSWORD

Die Passwordtabelle ist in zwei Bereiche unterteilt. Im ersten Teil ist das Passwort (8-Byte) untergebracht, und im zweiten Teil befinden sich 4 Mal 16-Bit-Felder zur Aufnahme der Zugriffsberechtigung oder sonstiger benutzerspezifischen Informationen.

MEMORY

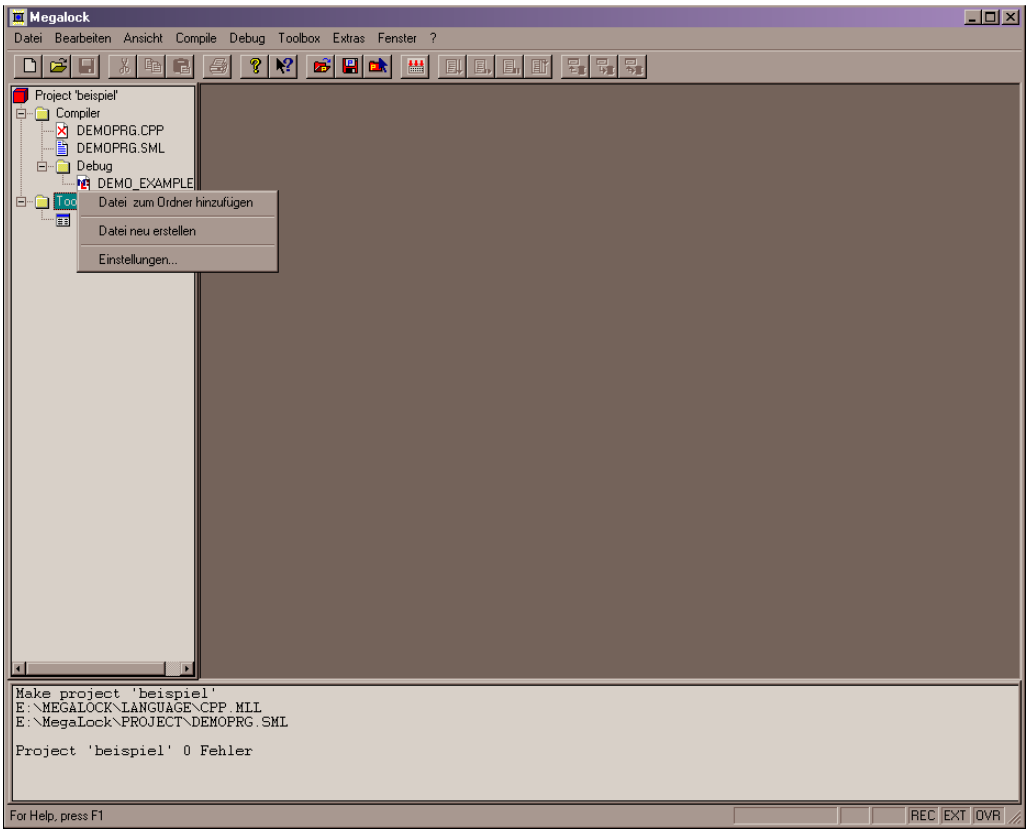
Die Memorytabelle überlagert alle zuvor beschriebenen Datengruppen und die nachfolgende Tabelle dient zur Verdeutlichung der Aufteilung innerhalb des Megalock Datenspeichers.

Megalock Datenspeicher

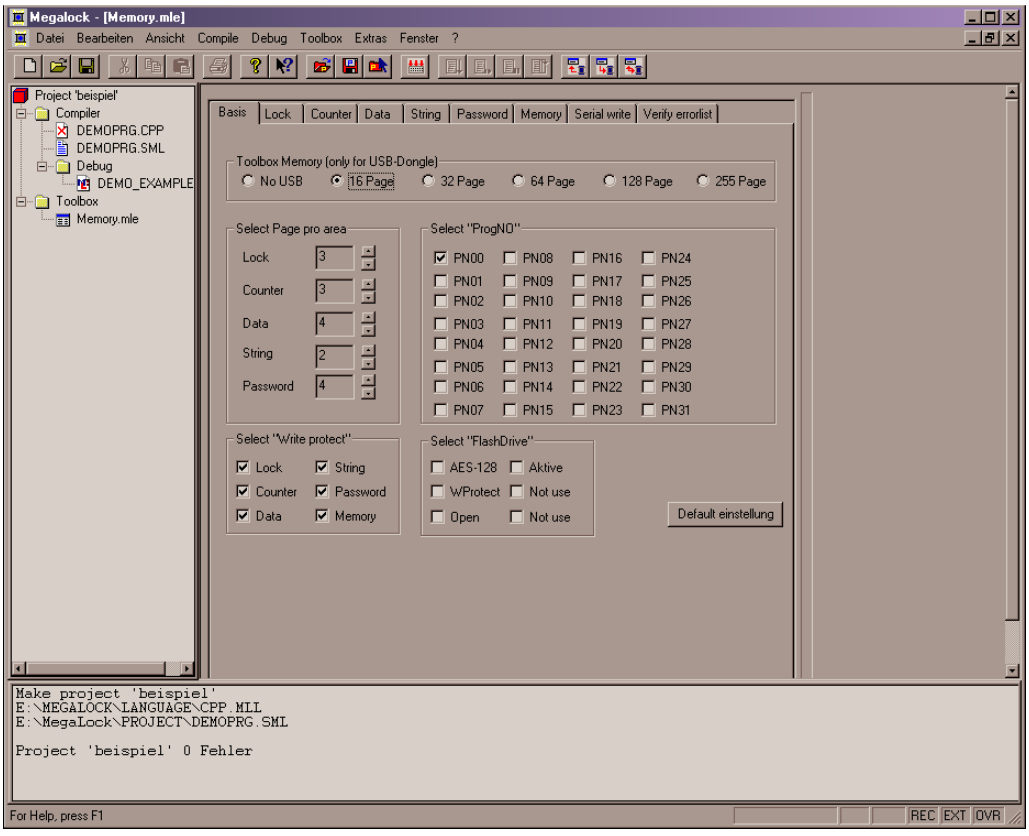
00-31	LockTabelle		00
32-63	CounterTabelle	Memory	
64-143	DataTabelle		
144-175	StringTabelle	Tabelle	
176-239	PasswordTabelle		239

Neben den Toolboxbefehlen kann auch mit dem Load- und Savebefehl über die Memorytabelle auf den gesamten Bereich des Datenspeichers zugegriffen werden.

Nach Drücken der rechten Maustaste über dem Eintrag **„Toolbox“** erscheint eine Funktionsauswahl um Dateien zum Ordner hinzuzufügen, Dateien neu anzulegen und Einstellungen vorzunehmen.

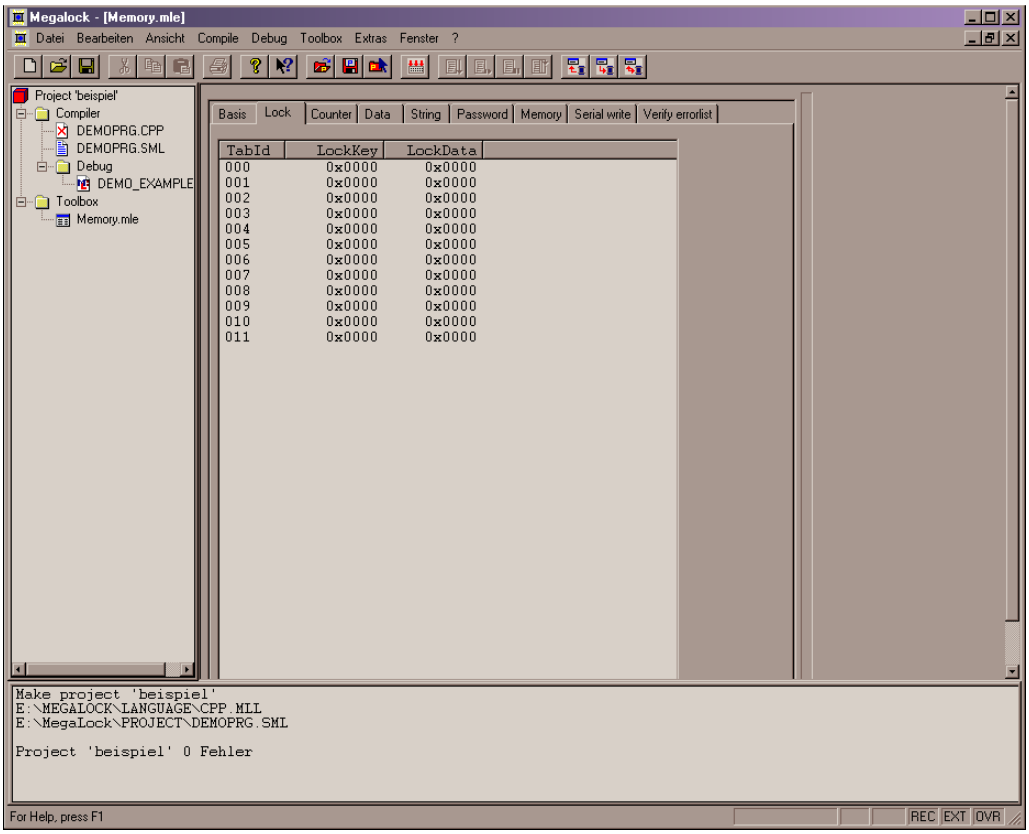


Durch Doppel-Klick über dem Dateieintrag im Ordner **„Toolbox“** oder über die rechte Maustaste wird die ausgewählte Datei ins Editorfenster geladen.



Jetzt kann der Inhalt bearbeitet werden und über das Menü “Toolbox“ können die Daten in den Megalock-Dongle geschrieben und gelesen werden. Die Daten können auch über Menü „Datei“ gespeichert und geladen werden.
Das Register zeigt die Aufteilung des Datenspeichers, die einzelnen Bereiche können für schreibende Toolbox-Befehle gesperrt werden und die Programmnummern können bestimmt werden über die dann der Megalock-Dongle angesprochen werden kann.
Die Gesamtanzahl der Seiten darf nicht größer sein als die des vorhanden Datenspeichers. Für die Daten-Speicheraufteilung stehen Ihnen in der Standardversion 30 bzw. bis 255 (USB-Dongle) Seiten zur Verfügung, die Sie individuell aufteilen können.
Bei der Seitenaufteilung muss mindestens je eine Lock- und Counter-Seite vergeben werden. Die restlichen Seiten können individuell eingeteilt werden. Wenn Sie nicht alle Seiten für die Datengruppen verwendet haben, werden die restlichen der Datengruppe **Memory** zugeordnet.

Lock, Counter, Data, String, Password



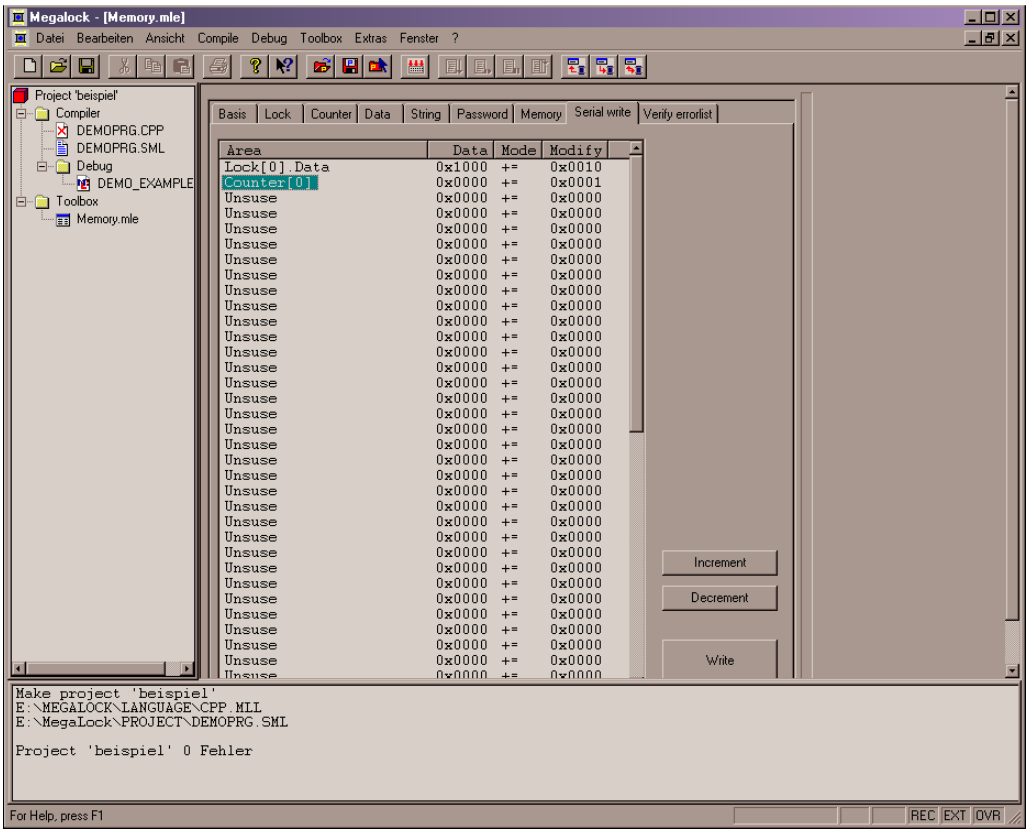
Die Register **Lock** bis **Password** zeigen die Inhalte der jeweiligen Datenbereiche und den Tabellenplatz „TabId“. Die TabId beginnt hier immer mit NULL.
Änderungen sind durch Doppelklick auf das jeweilige TabId-Feld möglich.

Memory

Das Register „Memory“ zeigt den Inhalt des nicht durch die zuvor genannten Bereiche belegten Speicher. Die TabId beginnt hier nicht bei NULL sondern wird ab Beginn des Datenspeichers gezählt. So wie es auch für die Befehle „Load“ und „Save“ notwendig ist.
Änderungen sind durch Doppelklick auf das jeweilige TabId-Feld möglich.

Serial Write

Über das Register haben Sie die Möglichkeit, die Inhalte der Datentabellen automatisch zu verändern. Hierzu können bis zu 64 Inhalte beliebiger Datengruppen mit Defaultwerten und einem Modifizierungswert belegen.



Increment

Durch Increment werden die Modifizierungen durchgeführt, jedoch werden die geänderten Daten nicht automatisch in den Megalock-Dongle übertragen.

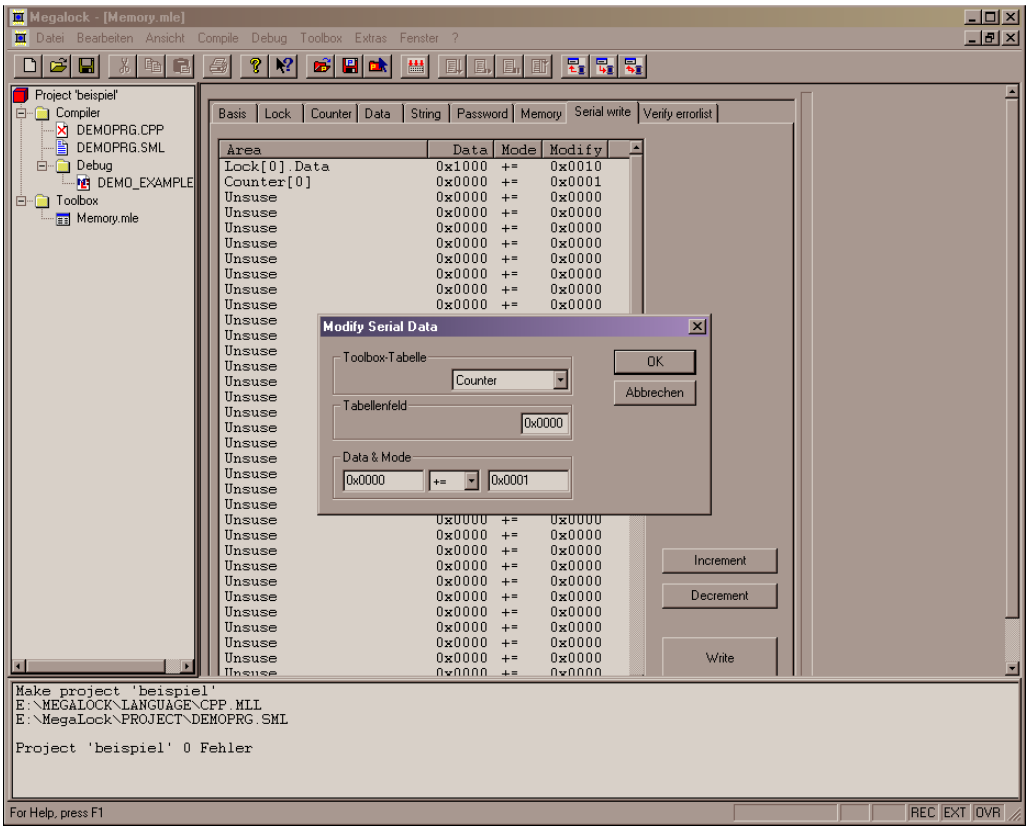
Decrement

Durch Decrement werden die Modifizierungen, die zuvor mit Write oder Increment durchgeführt wurden, rückgängig gemacht, jedoch werden die geänderten Daten nicht automatisch in den Megalock-Dongle übertragen.

Write

Mit der Option Write wird der aktuelle Datenbestand in den Megalock Dongle übertragen, und anschließend werden die Inhalte, der im linken Teil des Fensters definierten Bereiche, entsprechend den Vorgaben modifiziert.

Die einzelnen Einträge in der Tabelle können durch Doppel-Klick mit der linken Maustaste zur Modifizierung ausgewählt werden.
Nach Auswahl eines Eintrages erscheint nachfolgendes Dialogfenster.



Toolbox-Tabelle

Mit dieser Option wählen Sie eine der Datengruppen bzw. eines der Datenfelder, die automatisch verändert werden soll.

Tabellenfeld

Hier tragen Sie die entsprechende TabId der Datentabelle ein.

Data & Mode

Hier wird der Defaultwert bzw. Startwert für entsprechende Adresse eingetragen.
Dann wie der Wert (+=,-=, *= usw.) verändert werden soll.
Und dann womit der Wert verändert werden soll.

Megalock Teil III

Sprachelemente

SysTech
Entwicklungsges. für Mikroprozessorsysteme mbH

ALLGEMEINE DEFINITIONEN UND DEKLARATIONEN

Bei dem Einsatz von Megalock, sind, nachfolgend beschriebene, Grunddefinitionen für Ihren Compiler und zur Ausführung der verwendeten Megalock Befehle erforderlich, unabhängig davon ob Sie Einzelbefehle oder Programmblöcke verwenden.

##DEF_MEGALOCK_MACRO

Syntax **##DEF_MEGALOCK_MACRO**

Beschreibung Mit dieser Anweisung werden compilerspezifische Vereinbarungen, sofern diese notwendig sind, eingefügt. Ggf. weitere für Ihren Compiler notwendigen Informationen entnehmen Sie bitte dem Ordner \Megalock\Examples\.. .

Beispiel **##DEF_MEGALOCK_MACRO**

##DEF_MEGALOCK_CALL

Syntax **##DEF_MEGALOCK_CALL** *Parameter*

Beschreibung Mit der Anweisung **##DEF_MEGALOCK_CALL** werden Referenzen angelegt, die Ihr Compiler benötigt. Die Anweisung wird im Quellcode dort vorgenommen, wo Ihr Compiler die externen Referenzen erwartet. Als *Parameter* können angegeben werden:

16-Bit für 16-Bit Programme
32-Bit für 32-Bit Programme

Der Megalock Compiler fügt hinter die Anweisung **##DEF_MEGALOCK_CALL** die externen Referenzen in den Quellcode ein.
Ggf. weitere für Ihren Compiler notwendigen Informationen entnehmen Sie bitte dem Ordner \Megalock\Examples\.

Beispiel **##DEF_MEGALOCK_CALL 16BIT**
##DEF_MEGALOCK_CALL 32BIT

##OPEN_MEGALOCK

Syntax **##OPEN_MEGALOCK** (*ProgNo [,Access [,Port]]),*

Beschreibung Durch die Anweisung **##OPEN_MEGALOCK** erzeugt der Megalock Compiler eine Programmzeile zum aktivieren des Dongles mit der angegebenen *ProgNo*. Werden keine weiteren Parameter angegeben, sucht bei Programmausführung die Megalock API, auf allen vorhandenen Ports des lokalen Systems, nach dem Megalock Dongle.

Access

P sucht auf USB-Port und parallelen Schnittstellen
U sucht auf USB-Port und parallelen Schnittstellen
S sucht auf seriellen Schnittstellen
L sucht auf dem lokalen System
N sucht innerhalb des Netzwerkes

Derzeit stehen die Funktionen S und N **nicht** zur Verfügung.

Port

Bei *Port* können Sie die Suche auf eine bestimmte Portadresse beschränken. Wird der Port angeben, muss auch der Schnittstellentyp (*Access*) angegeben werden.
Die Angabe einer Portadresse wird beim USB-Dongle ignoriert.

Für jede benutzte *ProgNo* muss eine **##OPEN_MEGALOCK** Anweisung erfolgen. Es können gleichzeitig mehrere **OPEN**-Anweisungen aktiv sein. **##OPEN_MEGALOCK** muss so in Ihrem Quellcode untergebracht werden, dass bei Programmausführung die erzeugten Programmzeilen vor den eigentlichen Megalock Befehlen ausgeführt werden.

Darüber hinaus sollte nach der Anweisung **##OPEN_MEGALOCK** geprüft werden, ob der Dongle mit Ihrer Seriennummer und der Programmnummer gefunden wurde. Enthält der zurückgegebene Wert 0, dann ist der Dongle vorhanden, ansonsten wird der Fehlercode übertragen.

Beispiel a = **##OPEN_MEGALOCK(**1**)**
a = **##OPEN_MEGALOCK(**1,PL,0X278**)**

##CLOSE_MEGALOCK

Syntax **##CLOSE_MEGALOCK** (*ProgNo)*

Beschreibung Durch die Anweisung **##CLOSE_MEGALOCK** erzeugt der Megalock Compiler eine Programmzeile zum deaktivieren des Dongles, mit der angegebenen *ProgNo*.

Beispiel **##CLOSE_MEGALOCK(**1**)**

##DEF_MEGALOCK_INCLUDE

Syntax **##DEF_MEGALOCK_INCLUDE "***Dateiname***"**

Beschreibung Die Anweisung **##DEF_MEGALOCK_INCLUDE** veranlasst den Megalock Compiler eine zusätzliche HEADER-Datei einzubinden. Alle Anweisungen in der HEADER-Datei sind global und gelten für alle nachfolgenden Einzelbefehle und Programmblöcke. Der Dateiname muss ggf. mit Pfadnamen erfolgen.

Beispiel **##DEF_MEGALOCK_INCLUDE** “MyOwn.MLH“

PRÄPROZESSORSTEUERUNG

Mit der Präprozessorsteuerung haben Sie die Möglichkeit, Befehle bzw. Befehlsfolgen, für den eigentlichen Compilerdurchlauf, ein bzw. auszublenden. Dieses ist dann sinnvoll, wenn Sie in der Entwicklungsphase zusätzliche Befehle zur Überwachung von Programmabläufen verwenden, und diese über die bedingte Steuerung aktivieren bzw. deaktivieren.
Darüber hinaus können Sie über die Präprozessoranweisung **#include** Quelltexte einfügen und vorgefertigte Makros mit der Funktion **#macro** einbinden.
Die nachfolgend beschriebenen Präprozessoranweisungen gelten nur für den Megalock Compiler und sind in Headerdateien bzw. Programmblöcken anzugeben.

#define Bezeichner definieren

Syntax **#define** <*Bezeichner*>

Beschreibung Über die Anweisung **#define** werden Bezeichner definiert, die über **#ifdef** und **#ifndef** die Steuerung des Programmablaufes verändern. So können beispielsweise innerhalb der Programmentwicklungsphase zusätzliche Befehle zu Testzwecken ein- bzw. ausgeschaltet werden. Erfolgt die Definition innerhalb eines Programmblockes, gilt diese nur innerhalb des Blockes. Werden sie in einer Megalock Headerdatei angegeben, gilt sie für die gesamte Datei bzw. für alle Dateien die zum jeweiligen Projekt gehören.

Beispiel
```
#define     TestModus
```

#include Datei einbinden

Syntax **#include** *Includedatei*

Beschreibung Die Direktive **#include** fügt andere spezifizierte Dateien, sogenannte Includedateien, in den Quelltext ein. Der Megalock-Compiler entfernt die **#include**-Zeile und ersetzt diese im Quelltext durch den gesamten Text der angegebenen Datei. Die Quelltextdatei wird nicht wirklich geändert, es sieht nur vom Standpunkt des Compilers so aus. Die einzubeziehende Datei wird, sofern nicht angegeben, im aktuellen Ordner gesucht. Wird sie dort nicht gefunden, wird die Suche im Ordner MEGALOCK\INCLUDE bzw. in den unter Compiler-Einstellungen eingetragenen Pfad, fortgesetzt. Die Includedatei kann Variablendefinitionen und Megalock Befehle enthalten.

Beispiel
```
#include User.mlh
#include UnterPrg1.ccc
```

#ifdef #ifndef #else bedingte Steuerung

Syntax
```
#ifdef <Bezeichner> / #ifndef <Bezeichner>
  Anweisungen1
#else
  Anweisungen2
#endif
```

Beschreibung Mit Hilfe der bedingten Direktiven **#ifdef** und **#ifndef** können Sie prüfen, ob ein Bezeichner definiert ist, das heißt, ob eine **#define**-Direktive für diesen Bezeichner existiert. Die bedingten Direktiven ermöglichen einen oder mehrere Befehle nur dann wirksam werden zu lassen, wenn eine Bedingung erfüllt ist. Die bedingten Direktiven können jedoch nicht verschachtelt eingesetzt werden.

Beispiel
```
#ifdef TestModus
  LR1 = LR2
#endif

#ifndef TestModus
```

```
  WR0 = 1
#else
  WR1 = 2
#endif
```

#macro Makro einbinden

Syntax

```
#macro <Bezeichner>
  Anweisungen
#endm
```

Beschreibung

Der Megalock-Compiler bietet Ihnen die Möglichkeit mehrzeilige Macros zu erstellen. Damit können Sie einen Block von Befehlen definieren, der bei jedem Aufruf des Makros in Ihren Quellcode aufgenommen wird. Sie können Argumente an die Makros übergeben, die der Megalock-Compiler in den Macroblock einsetzt. Die Makro-Definition beginnt mit der Direktiven **#macro** mit nachfolgendem Bezeichner, der Bezeichner ist der Name, über den das Macro aktiviert wird. Die Parameterübergabe (max. 10 Parameter) geschieht innerhalb der jeweiligen Anweisungen, durch Angabe des %-Zeichen, gefolgt von der Ziffer zwischen 0 und 9. Die Ziffer 0 steht für den ersten Parameter, die 1 für den zweiten Parameter usw.
Die Definition von Makros sollte innerhalb der ML-Headerdateien MLH erfolgen. Als Parameter können Konstanten, arithmetische Operationen, Variablen, Registername und Befehle übergeben werden. Der Befehl **#endm** schließt die Beschreibung eines Makros ab. Der Macronamen stehen Ihnen 15 Zeichen alphanumerisch zur Verfügung.

Beispiel

```
#macro RegInit
  mov WR1,%0
  mov WR2,%1
  mov %2,%3
#endm
```

Nach dem oben beschriebenen Macro sind 4 Parameter hinter dem Macroaufruf erforderlich. Der Aufruf innerhalb eines Programmblockes könnte wie folgt aussehen:

```
RegInit     KundenNr,MwSt_Satz,LR3,5*0x4711
```

Der Megalock Compiler wird das Macro wie folgt ergänzen und compilieren:

```
  mov WR1,KundenNr
  mov WR2,MwSt_Satz
  mov LR3,5*0x4711
```

EINZELBEFEHLE

Bei Verwendung von Megalock Einzelbefehlen ist es erforderlich, dass die Zeichenfolge **ML..** vor jedem Befehl gesetzt wird. Der Schreibaufwand ist bei Einzelbefehlen geringer als bei Programmblöcken und kann an entsprechenden Stellen in Ihrem Programmcode angegeben werden. Kontrollstrukturen (if, then, for next, do while usw.) können nicht als Einzelbefehle eingegeben werden. Einzelbefehle können nicht im Debugger überprüft werden.
Die Variablen, die für Einzelbefehle verwendet werden, müssen für den Megalock Compiler gekennzeichnet sein, sofern Sie nicht in einer Headerdatei definiert wurden.

Die Kennzeichnung für die Variablen sind :

% numerische Variablen	16-Bit z.B.: zahl**%**
# numerische Variable	32-Bit z.B.: zahl**#**
$ String	z.B.: text**$**
@ Pointervariable	z.B.: @zahl%, @text$

Der Einsatz von Einzelbefehlen wurde bereits im Schnelleinstieg beschrieben, bzw. ist in der Datei DEMOPRG.SML im Ordner Megalock\Examples\. Ersichtlich.

PROGRAMMBLOCK

Der Programmblock bietet sich dann an, wenn mehrere Megalock Befehle in Folge eingesetzt werden.
In dem nachfolgend aufgeführten Schema wird der Aufbau eines Programmblocks dargestellt. Der Anfang eines Programmblockes wird im Quellcode mit ##BEGIN gekennzeichnet und ##END schließt ihn ab.

##**BEGIN**
  <Programmblockparameter>
  <Variablen, Konstanten und Registernamen definieren>
  <Megalock Befehle>
##END

Nach der Anweisung **##BEGIN** folgen die Definitionen der Programmblockparameter, die zur Steuerung des Megalock Compiler benötigt werden.
Danach werden die im Programmblock benötigten Variablen, Konstanten und Register definiert. Variablen die innerhalb des Programmblockes eingesetzt und als Schnittstelle zwischen Megalock und Ihrem Programmcode fungieren, werden entweder lokal oder global definiert. Werden die Variablendefinitionen innerhalb eines Programmblocks angegeben, so gelten diese nur innerhalb des Blockes, werden die Definitionen in einer Megalock Headerdatei angegeben, gilt sie für die gesamte Datei bzw. für alle Dateien die zum jeweiligen Projekt zusammengefaßt sind. Nach dem Definitionsteil folgen die Megalock Befehle, die ohne für die Einzelbefehle notwendige Kennzeichnung ML.. eingeben werden. Der Programmblock wird jeweils mit **##END** abgeschlossen.
Innerhalb eines Programmblockes kann es notwendig sein, Anweisungen anzugeben, die nicht vom Megalock Compiler ausgewertet sollen (z.B. Unterprogrammaufrufe, Berechnungen mit Gleitkommawerten etc.). Diese werden in der jeweiligen Zeile an erster Position mit einem Doppelpunkt (..) gekennzeichnet.
Sie können mehrere Programmblöcke in Ihrem Anwendungsprogramm angegeben werden, jedoch ist es nicht möglich, innerhalb eines Programmblockes durch Unterprogrammaufruf, einen weiteren Programmblock oder Einzelbefehle auszuführen.
In einem Quelltext können beliebig viele Programmblöcke angegeben werden.
Der Einsatz von Programmblöcken wurde bereits im Schnelleinstieg beschrieben, bzw. ist in der Datei DEMOPRG.SML im Ordner Megalock\Examples\Sprache ersichtlich.

PROGRAMMBLOCKPARAMETER

Werden Anweisungen zur Programmblocksteuerung nicht angegeben, so gelten die unter Option\ML-Compiler default spezifizierten Einstellungen bzw. die in den Headerdateien global definierten.

MODUL

Syntax **MODUL**=<*Stringconstante*>

Beschreibung Durch die Anweisung MODUL wird dem jeweiligen Megalock Programmblock einen, bis zu 15 alphanumerische Zeichen langen Namen zugewiesen. Wird die Anweisung MODUL nicht angegeben, so generiert der Megalock Compiler einen Modulnamen. Der Modulname wird im Debugger zur Modulauswahl verwendet. Daher sollte die Anweisung MODUL in jedem Programmblock angeben werden.

Beispiel **MODUL**=MeinModul,MODULNR=10,PROGNO=1

MODULNR

Syntax **MODULNR**=<*numerische Konstante(0..255)*>

Beschreibung Mittels der Anweisung **MODULNR** können Sie die Kryptierung der Befehlskennziffern (OPCODE) verändern. Die gültigen Modulnummern liegen im Bereich von 0 bis 255. Durch Angabe unterschiedlicher Modulnummern im Megalock Programmblock, erhalten gleiche Megalock Befehle in unterschiedlichen Programmblöcken auch unterschiedliche Befehlskennziffern (OPCODE). Wird die Anweisung **MODULNR** nicht angegeben, so gilt die unter **COMPILER-EINSTELLUNG** spezifizierte **MODULNR**, die dann für jeden nachfolgenden Programmblock um 1 erhöht wird.

Beispiel MODUL=MeinModul,**MODULNR**=10,PROGNO=1

PROGNO

Syntax **PROGNO**=<*numerische Konstante(0..31)*>

Beschreibung Durch die Anweisung **PROGNO** wird dem Megalock Compiler mitgeteilt unter welcher Programmnummer die nachfolgenden Befehl ausgeführt werden sollen. Für die Programmnummer kann eine *numerische Konstante* im Bereich 0 bis 31 angegeben werden. Wird die Anweisung **PROGNO** nicht angegeben, so gilt die unter **COMPILER-EINSTELLUNG** spezifizierte **PROGNO**. Der als **PROGNO** spezifizierte Wert beeinflusst zusätzlich die Verschlüsselung der Befehlskennziffern (OPCODE).
Die **PROGNO** 0 ist auf jedem Megalock Dongle freigeschaltet, und weitere verwendete **PROGNO**'s werden unter **ML-TOOLBOX\BASIS** freigeschaltet. Erhält der Megalock Dongle einen Programmblock der unter einer nicht freigeschalteten **PROGNO** ausgeführt werden soll, werden die Befehle in diesem Programmblock nicht ausgeführt. Über **PROGNO** bzw. über Freischaltung der Programmnummer auf dem Megalock Dongle, können unterschiedliche Programme und/oder Programmausbaustufen mit nur einem Megalock Dongle geschützt werden.

Beispiel MODUL=MeinModul,MODULNR=10,**PROGNO**=1

TURBO

Syntax **TURBO** **(Keine Auswirkung bei USB-Dongle)**

Beschreibung Durch die Anweisung **TURBO** wird die Ausführung der einzelnen Befehle des jeweiligen Megalock Programmblockes beschleunigt. Die Übertragung der Megalock Keynummer bzw. das Testen ob der Megalock Dongle noch am Rechner vorhanden ist, wird nur beim dem ersten Befehl durchgeführt. Die nachfolgenden Befehle, bis einschließlich die ##END Anweisung, werden dann ohne Übermittlung der Megalock Keynummer ausgeführt.

Diese Funktion sollten Sie zurzeit nur dann aktivieren, wenn Sie sicherstellen können, dass während der Ausführung des jeweiligen Megalock Programmblockes keine Anwendung auf den Port zugreift, auf dem sich der Megalock Dongle befindet. Die Auswirkung bezüglich der Ausführungsgeschwindigkeit können Sie unter **EXTRAS\INFO&TEST** testen.

Beispiel MODUL=MeinModul,PROGNO=1,**TURBO**

NOTURBO

Syntax **NOTURBO** **(Keine Auswirkung bei USB-Dongle)**

Beschreibung Durch die Anweisung **NOTURBO** wird vor jeder Megalock Befehlsausführung die Megalock Keynummer übertragen bzw. getestet, ob der Megalock Dongle noch am Rechner vorhanden ist. Diese Art der Befehlsausführung ist jedoch etwas zeitintensiver als die Anweisung TURBO. Die Auswirkung bezüglich der Ausführungsgeschwindigkeit können Sie unter **EXTRAS\INFO & TEST** testen.

Beispiel MODUL=MeinModul,PROGNO=1,**NOTURBO**

WATCHDOG

Syntax **WATCHDOG**

Beschreibung Mit der Anweisung **WATCHDOG** wird die Laufzeitüberwachung für den jeweiligen Megalock Programmblock, innerhalb des Megalock Dongles, aktiviert. Die Timerfunktion wird beim Starten eines Programmblockes aktiviert und der Timecounter wird auf 0 gesetzt. Erhält der Megalock Dongle einen Befehl zur Ausführung, so wird die bisher verstrichene Zeit mit dem im Timer Register enthaltene Maximalzeit verglichen. Ist die verstrichene Zeit größer, wird der Errorcode 0x21 und das **WATCHDOG** Flag im Flag Register gesetzt.

Durch die **WATCHDOG**-Funktion kann zum Beispiel ein Debugger auf dem Rechner erkannt werden.

Beispiel MODUL=MeinModul,**WATCHDOG**

NOWATCHDOG

Syntax **NOWATCHDOG**

Beschreibung Mit der Anweisung **NOWATCHDOG** wird die Laufzeitüberwachung für den jeweiligen Megalock Programmblock nicht aktiviert.

Beispiel MODUL=MeinModul,**NOWATCHDOG**

READCT

Syntax **READCT**

Beschreibung Die Anweisung **READCT** aktiviert den Schutzmechanismus zum Auslesen der Megalock Register (R0-R19). Bevor Sie den Inhalt eines Megalock Registers mit den Befehlen **Getreg** und **movs** auslesen, muss das RC-Registers des Megalock Dongles mittels des Befehls **src** gesetzt werden. Die Ausführung der Befehle **Getreg** und **movs** wird das RC-Register um 1 dekrementiert sofern der Wert in diesem > NULL ist. Wenn der Inhalt des RC-Registers NULL ist, werden die Befehle nicht mehr ausgeführt und das Errorflag sowie der ErrorCode 0x33 werden gesetzt.

Beispiel MODUL=MeinModul,MODULNR=10,**READCT**

NOREADCT

Syntax **NOREADCT**

Beschreibung Die Anweisung **NOREADCT** deaktiviert den Schutzmechanismus zum Auslesen der Megalock Register (R0-R19).

Die Befehle **Getreg** und **movs** werden immer ausgeführt. Das RC-Register wird aber auch hier jeweils um 1 dekrementiert. Es wird jedoch beim erreichen von NULL kein ErrorCode gesetzt.

Beispiel MODUL=MeinModul,PROGNO=1, **NOREADCT**

SOURCE

Syntax **SOURCE**

Beschreibung Wird die Anweisung **SOURCE** in der Definitionsbeschreibung angegeben, wird der Quellcode der Megalock Befehle als Kommentarzeilen in die Output-Datei übertragen.

Beispiel MODUL=MeinModul,MODULNR=10,**SOURCE**

NOSOURCE

Syntax **NOSOURCE**

Beschreibung Wird die Anweisung **NOSOURCE** in der Definitionsbeschreibung angegeben, wird der Quellcode der Megalock Befehle nicht als Kommentarzeilen in die Output-Datei übertragen.

Beispiel MODUL=MeinModul,**NOSOURCE**

STATUS

Syntax **STATUS**=<Variablenname>

Beschreibung Wird die Anweisung **STATUS** in der Definitionsbeschreibung angegeben, wird am Ende des Programmblocks die Megalock Statusvariable, der vorangegangen Befehl in die Variable, die hinter **STATUS** angegeben wurde, automatisch übertragen.

Die Anweisung zur Ermittlung des Status führt keine Operationen mit dem Megalock Dongle durch, sie überträgt lediglich den Status vorangegangener Befehle.

Beispiel MODUL=MeinModul,**STATUS**=RetStatus

NOSTATUS

Syntax **NOSTATUS**

Beschreibung Wird die Anweisung **NOSTATUS** in der Definitionsbeschreibung angegeben, erfolgt am Ende des Programmblocks keine Übertragung der Megalock Statusvariable.

Beispiel MODUL=MeinModul,**NOSTATUS**

REGISTERDEFINITION

Wird die Registerdefinitionen innerhalb eines Programmblockes angegeben, gelten diese nur innerhalb des Blockes. Werden die Anweisungen in einer Megalock Headerdatei angegeben, gelten sie für die gesamte Datei bzw. für alle Dateien die zum jeweiligen Projekt zusammengefasst sind.

LONGREG 32 Bit Register

Syntax **LONGREG**Nr Bezeichner

Beschreibung Die Megalock Register LR0-LR19 können über die Anweisung **LONGREG** mit sprechenden Namen versehen werden.

Beispiel
```
MODUL=MeinModul,MODULNR=10,PROGNO=1
LONGREG0  X
LONGREG1  BX
LONGREG2  Counter
LONGREG3  Ergebnis
```

WORDREG 16 Bit Register

Syntax **WORDREG**Nr *Bezeichner*

Beschreibung Die Megalock Register WR0-WR19 können über die Anweisung **WORDREG** mit sprechenden Namen versehen werden. Bei der Registerbelegung mit Word werden nur die unteren 16-Bit des entsprechenden Registers angesprochen.

Beispiel
```
MODUL=MeinModul,MODULNR=10,PROGNO=1
WORDREG0  X
WORDREG1  BX
WORDREG2  Counter
WORDREG3  Ergebnis
```

VARIABLENDEFINITION

Variablen dienen als Schnittstelle zwischen den Megalock Programmblöcken und Ihrem Programmcode. Die verwendeten Variablen müssen für den Megalock Compiler und zusätzlich für Ihren Compiler definiert werden. Erfolgt die Definition innerhalb eines Programmblockes, gilt diese nur innerhalb des Blockes. Werden sie in einer Megalock Headerdatei angegeben, gilt sie für die gesamte Datei bzw. für alle nachfolgenden Megalock Anweisungen.

LONG	**32 Bit Variable**

Syntax **LONG** *Variablenname*

Beschreibung **LONG** definiert eine 32-Bit-Variable

Beispiel
```
MODUL=MeinModul,MODULNR=10,PROGNO=1
LONG varLong, varLongErgebnis
```

LONGPTR	**32 Bit Pointer**

Syntax **LONGPTR** *Variablenname*

Beschreibung **LONGPTR** definiert einen Zeiger (Adresse) auf eine 32-Bit Variable

Beispiel
```
MODUL=MeinModul,MODULNR=10,PROGNO=1
LONGPTR ptrLong, ptrLongErgebnis
```

WORD	**16 Bit Variable**

Syntax **WORD** *Variablenname*

Beschreibung **WORD** definiert eine 16-Bit Variable

Beispiel
```
MODUL=MeinModul,MODULNR=10,PROGNO=1
WORD vCounter,vErgebnis
```

WORDPTR	**16 Bit Pointer**

Syntax **WORDPTR** *Variablenname*

Beschreibung **WORDPTR** definiert einen Zeiger (Adresse) auf eine 16-Bit Variable

Beispiel
```
MODUL=MeinModul,MODULNR=10,PROGNO=1
WORDPTR vCounter,vErgebnis
```

STRING	**Stringvariable**

Syntax **STRING** *Variablenname*

Beschreibung **STRING** definiert eine Stringvariable.

Beispiel
```
MODUL=MeinModul,MODULNR=10,PROGNO=1
STRING vPassword
```

STRINGPTR	**Stringpointer**

Syntax **STRINGPTR** *Variablenname*

Beschreibung **STRINGPTR** definiert einen Zeiger (Adresse) auf eine Stringvariable.

Beispiel
```
MODUL=MeinModul,MODULNR=10,PROGNO=1
STRINGPTR vPassword1
```

CONST Konstante

Syntax **CONST** *Name Wert*

Beschreibung Über die Anweisung **CONST** werden numerische Konstanten (16/32 Bit) Symbolen zugeordnet.

Beispiel
```
MODUL=MeinModul,MODULNR=10,PROGNO=1
CONST eepKndNr 10
```

KONTROLLSTRUKTUREN

Kontrollstrukturen sind Hauptbestandteile eines Programmes. Es gibt verschiedene Arten von Anweisungen, die einen Programmablauf beeinflussen:

Fallunterscheidungen (**if-else-endif**, **select-case**),
Schleifen (**for**, **until**, **while**, **loop**).
Die Anweisungen können nach belieben verschachtelt werden.
Zur Bildung einer Bedingung stehen folgende Vergleichsoperatoren zur Verfügung:

==	entspricht 'gleich'
!=	entspricht 'ungleich'
>	entspricht 'größer'
<	entspricht 'kleiner'
>=	entspricht 'größer gleich'
<=	entspricht 'kleiner gleich'

if / else/ endif

Syntax

if *Bedingung*
  Anweisungen
else
  Anweisungen
endif

Beschreibung

Die **if** *Bedingung* ermöglicht einen oder mehrere *Anweisungen* nur dann wirksam werden zu lassen, wenn eine *Bedingung* erfüllt ist.

Beispiel 1

if WR0 > 100
  WR2 = WR3
endif

In diesem Fall ist 'WR0 > 100' die *Bedingung*. Die *Anweisung* in der Zeile zwischen **if** und **endif** wird nur abgearbeitet, wenn diese *Bedingung* WAHR ist. Ist sie FALSCH, so wird der Programmablauf hinter dem Befehlswort **endif** fortgesetzt.

Beispiel 2

if WR0 > 100
  WR1 = WR2
else
  WR1 = 0
endif

Im zweiten Beispiel wird die *Anweisung* zwischen if und else abgearbeitet, wenn die *Bedingung* hinter if WAHR ist. Anschließend fährt die Programmausführung hinter dem Kommando endif fort. Ist die *Bedingung* hinter if nicht erfüllt, so werden die *Anweisungen* zwischen else und endif wirksam.

_if / _endif

Syntax

_if *Bedingung*
  Anweisungen
_endif

Beschreibung

Die **_if** *Bedingung* entspricht der Funktion der standardmäßigen if Bedingung. Der gravierende Unterschied liegt darin begründet, dass in der standardmäßigen if Abfrage die Ausführung des WAHR Zweiges bzw. nicht Ausführung des WAHR Zweiges auf dem Rechnersystem durchgeführt wird. Bei der **_if** *Bedingung* werden immer alle Befehle zwischen **_if** und **_endif** zum Megalock Dongle übertragen. Der Megalock Dongle entscheidet aufgrund der zuvor ausgewerteten Bedingung, ob diese Befehle ausgeführt werden oder nicht.

Nachfolgend das Beispiel einer standard if Abfrage,

```
if WR0 > WR1
    WR2 = WR3
endif
```

und dem vom Megalock Compiler erzeugten C-Programmcode.

```
MEGALOCK1 (0x0040d782L);
if (MEGALOCK10(0x0040c5b2L)== -1)
{
     MEGALOCK1(0x0040ae93L);
}
```

Die entstandene if Abfrage ist im Source Code bzw. für einen versierten Hacker auch im Debugger zu erkennen.

Nachfolgend das Beispiel einer der **_if** Abfrage,

```
_if WR0 > WR1
    WR2 = WR3
_endif
```

und dem vom Megalock Compiler erzeugten Programmcode:

```
MEGALOCK1(0x0040d782L);
MEGALOCK1(0x00401aceL);
MEGALOCK1(0x0040ae93L);
```

Die Funktion ist die gleiche, jedoch ist keine if Abfrage mehr erkenntlich und damit für einen Hacker unverständlich.

select/case/break/default/endsel

Syntax

```
select Registervariable
  case Operand1
      Anweisungen1
      break
  default
      Anweisungen
endsel
```

Beschreibung Der Befehl **select** ermöglicht Verzweigungen in Abhängigkeit von numerischen Werten bzw. Registerinhalten.

Beispiel

```
LR1 = 20
select LR1
  case 10
      Getreg (p1,WR0)
      break
  case 20
      RCounter(0,p3)
      break
  default
      mov R0,0xffffffff
endsel
```

Zunächst wird ein Register hinter **select** bestimmt, der die Verzweigungsbedingung (hier LR1) enthält.
Anschließend werden die **case**-Anweisungen von oben nach unten durchgegangen und geprüft, ob diese übereinstimmen. In diesem Beispiel steht hinter der ersten **case**-Anweisung der Wert 10. Da die Verzweigungsbedingung (LR1 = 20) ist, springt die Programmausführung zum nächsten **case**. Hinter dem zweiten **case** steht der Wert 20.
Diese Bedingung ist also erfüllt. Im vorliegenden Fall führt dies dazu, dass die Anweisungen zwischen dem zweiten **case** und dem **break**-Befehl abgearbeitet werden. Sofern Ihre Programmiersprache über einen Sprungbefehl verfügt (goto). wird die Programmausführung

hinter dem Kommando **endsel** fortgesetzt. Andernfalls wird verfahren, als wäre der **break** Befehl nicht angegeben worden. Fehlt der **break**-Befehl wird die Suche nach dem Verzweigungskriterium mit dem nächsten **case** wieder aufgenommen. Ist die aktuelle Ausprägung des Verzweigungskriteriums hinter keinem **case** aufgeführt, so werden die Befehle zwischen **default** und **endsel** verarbeitet, sofern eine **default**-Anweisung vorhanden ist.
Hinter **case** können jedoch nicht nur numerische Konstanten, sondern auch numerische Variablen und Register vorhanden sein.

for / next

Syntax

for (*Initialisierung; Bedingung; Inkrementierung*)
  Anweisungen
next

Beschreibung

Die **for-next**-Schleife dient dazu, die zwischen den Befehlsworten **for** und **next** stehenden *Anweisungen* wiederholt abzuarbeiten. Zuerst wird die *Initialisierung*sanweisung ausgeführt. Anschließend wird die *Bedingung* geprüft. Ist diese FALSCH, wird mit dem Befehl hinter **next** fortgefahren. Ist die Bedingung WAHR, werden die *Anweisungen* bis zum **next** Befehl ausgeführt. Anschließend wird die *Inkrementierungs*anweisung ausgeführt und danach wird die *Bedingung* erneut geprüft. Ist die *Bedingung* noch WAHR, werden die *Anweisungen* erneut abgearbeitet.

Beispiel

```
for (LR1 = 1 ; LR1 < 100; LR1 += 3)
  LR2 += 0x12
  LR3++
next
```

while / wend

Syntax

while *Bedingung*
  Anweisungen
wend

Beschreibung

(Kopfgesteuerte Schleife) Die Befehle **while** und **wend** können eine Gruppe von Anweisungen einschließen, die solange abgearbeitet werden, wie die Bedingung WAHR ist. Stößt die Programmausführung auf den Befehl **while**, so wird die dahinter stehende *Bedingung* geprüft. Wenn sie WAHR ist, so werden die *Anweisungen* zwischen **while** und **wend** ausgeführt. Beim Erreichen von **wend** springt die Programmausführung wieder zu **while** und der Zyklus beginnt von neuem, bis die *Bedingung* FALSCH wird.

Beispiel

```
while LR1 <= 100
  LR1 += 10
wend
```

do / until

Syntax

do
  Anweisungen
until Bedingung

Beschreibung

(Fußgesteuerte Schleife) Die Befehle **do** und **until** können eine Gruppe von *Anweisungen* einschließen, die solange abgearbeitet werden, solange die *Bedingung* FALSCH ist. Wird im Programm die Anweisung **do** erreicht, so werden die *Anweisungen* bis zum Erreichen von **until** ausgeführt. Nun wird geprüft, ob die hinter **until** stehende *Bedingung* WAHR ist. Wenn dies der Fall ist, so werden die Befehle hinter **until** ausgeführt. Ist die Bedingung FALSCH, dann springt die Programmausführung wieder zum **do**-Schleifenanfang. Die *Anweisungen* zwischen **do** und **until** werden mindestens einmal abgearbeitet.

Beispiel

```
do
  LR1+= 10
```

until LR1 > 100

do / loop

Syntax
```
do
  Anweisungen
loop Register
```

Beschreibung Die Befehle **do** und **loop** *Register* erzeugen eine Zählschleife, wobei der Inhalt des Registers bei jedem erreichen von **loop** um 1 dekrementiert wird. Ist der Inhalt des *Registers* = 0, wird die Schleife nicht mehr ausgeführt, andernfalls werden die *Anweisungen* zwischen **do** und **loop** durchgeführt.

Beispiel
```
do
  LR2 += 10
  if LR2 > 100
      LR3 = 0
loop LR1
```

goto

Syntax **goto** *Marke*

Beschreibung Mit Hilfe einer *Marke* kann man Stellen im Programm festlegen, die mit dem Befehl **goto** angesprungen werden können. Die Programmausführung wird dann an der *Marke*nposition fortgesetzt. **goto** kann nur dann eingesetzt werden, wenn Ihre Programmiersprache über einen Sprungbefehl verfügt.

Beispiel
```
goto L12Auswahl
  ....
L12Auswahl:
```

Marke

Syntax **Marke:**

Beschreibung Die **Marke** muss mit einem Buchstaben beginnen und kann nachfolgend aus Buchstaben, Ziffern und Unterstriche bestehen und muss mit einem Doppelpunkt enden. Beim Ansprung der **Marke** mit **goto** wird der Doppelpunkt allerdings nicht angegeben. **Marke:** kann nur dann eingesetzt werden, wenn Ihre Programmiersprache über einen Sprungbefehl verfügt.

Beispiel
```
goto L12Auswahl
  ...
L12Auswahl:
```

Exit

Syntax **Exit**

Beschreibung Mit Hilfe von **Exit** kann ein Programmblock vorzeitig beendet werden.

Beispiel
```
Exit
...
```

TOOLBEFEHLE

Die Toolbefehle sind aus Sicht des Megalock Dongles und der vielfältigen Befehlsmöglichkeiten als einfache Befehle (Funktionen) anzusehen. Die Toolbefehle greifen auf den Datenspeicher innerhalb des Megalock Dongles zu. Die Aufteilung der Toolbefehle bzw. des Datenspeichers ist in Lock, Counter, Daten, String und Password Bereiche aufgeteilt. Aus Sicht des Anwenders werden diese Bereiche als externe Datentabellen verwaltet.
Innerhalb der Befehlsbeschreibung werden nachfolgende Namen verwendet:

ML_LOCKTAB(0..n)
ML_COUNTERTAB (0..n)
ML_DATATAB (0..n)
ML_STRINGTAB (0..n)
ML_PASSWORDTAB (0..n)

Die Größe jeder einzelnen Tabelle können Sie auf Ihre Bedürfnisse in der Megalock Toolbox (BASIS) festlegen, wobei der Gesamtumfang der Tabellen nicht größer sein darf als die Größe des Datenspeichers.

Da auf den Megalock Dongle nur über kundenspezifischen Befehlscode zugegriffen werden kann, gibt es für jede einzelne Speicherstelle des Datenspeichers einen eigenen kryptierten Befehlscode. Z.B.:

Tabelle	Befehl	Befehlscode Kunde 1	Befehlscode Kunde 2
ML_LOCKTAB(0)	RLock	4711	9910
ML_LOCKTAB(1)	RLock	4812	9829
ML_DATATAB(0)	RData	1020	3040
ML_DATATAB(1)	RData	3311	5511

Durch den kryptierten Befehlscode ist es für dritte nicht mehr möglich, weder Funktion noch der verwendete Tabellenplatz zu erkennen.
Damit der Megalock Compiler bereits beim compiliern den entsprechenden Befehlscode erzeugen kann, ist der Zugriff auf die Datentabellen nur über Konstante oder indirekt über ein Megalock Register, welches die Nummer des Tabellenplatzes enthält, möglich.

Beschreibung und Zugriff auf die einzelnen Tabellen:

ML_LOCKTAB:
Die Locktabelle enthält pro Tabellenplatz zwei Felder; den LockKey und LockData.
Ein Zugriff auf LockData erfolgt nur dann, wenn der übergebene Schlüssel mit dem in der Tabelle enthaltenen LockKey übereinstimmt.
Darüber hinaus wird die LockTabelle zur Bildung des Kryptierschlüssels verwendet.

ML_COUNTERTAB:
Die Countertabelle enthält pro Tabellenplatz ein Feld, den Counter.
Beim Zugriff auf die Countertabelle bzw. eines Tabellenplatzes wird der Inhalt jeweils um 1 dekrementiert, bis der Wert 0 erreicht ist.
Standardmäßig ist die Countertabelle mit 16-Bit pro Feld aufgeteilt.
Die Tabelle kann als 16 oder 32-Bit Counter eingesetzt werden.

ML_COUNTERTAB()	16-Bit	ML_COUNTERTAB()	32-Bit
0 1	0x1111 0x2222	0	0x22221111
2 3	0x3333 0x4444	1	0x44443333

ML_DATATAB:
Die Datatabelle dient zur Speicherung von 16 oder 32-Bit Daten.
Standardmäßig ist die Datatabelle mit 16-Bit pro Feld aufgeteilt

ML_DATATAB()	16-Bit	ML_DATATAB()	32-Bit
0 1	0x1111 0x2222	0	0x22221111
2 3	0x3333 0x4444	1	0x44443333

In der ML_DATATAB könnten Sie z.B. Grundeinstellungen oder Programmparameter abgelegen.

ML_STRINGTAB:
Die Stringtabelle dient zur Speicherung von Strings in der Größe von 8-Byte.In der Stringtabelle könnten Sie z.B. Anwendernamen ablegen.

ML_PASSWORD:
Die Passwordtabelle ist in zwei Bereiche unterteilt. Im ersten Teil ist das Passwort (8-Byte) untergebracht, und im zweiten Teil befinden sich 4 mal 16-Bit Felder zur Aufnahme der Zugriffsberechtigung oder sonstiger benutzerspezifischen Informationen.

BLock — Bitvergleich auf LockKey

Syntax **BLock**(*Index*,*BitKey,Variable*)

Beschreibung Der Inhalt von ML_LOCKTAB*(Index).*LockData wird in die *Variable* übertragen, sofern die UND-Verknüpfung von ML_LOCKTAB*(Index)*.LockKey und *BitKey* ungleich 0 ist. Der Befehl **BLock** kann z.B. dafür eingesetzt werden, das bestimmte Programmfunktionen aufgrund der zurückgegebenen Variable ausgeführt werden.

Funktion
```
if (ML_LOCKTAB(Index).LOCKKEY & Bitkey) != 0
  Variable = ML_LOCKTAB(Index).LOCKDATA
else
  Variable = 0
```

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

BitKey Word oder Konstante

Variable Word

Beispiel
```
BLock(0,4711,DataVar)
BLock(LockConst,LockVar ,DataVar)
BLock(WR0,LockVar,DataVar)
```

RLock — Lock-Registers lesen

Syntax **RLock** (*Index,LockKey,Variable*)

Beschreibung Der Inhalt von ML_LOCKTAB(*Index*).LockData wird in die Variable übertragen, sofern *LockKey* und ML_LOCKTAB(*Index*).LockKey übereinstimmen.
Der Befehl **RLock** kann z.B. dafür eingesetzt werden, das bestimmte Programmfunktionen aufgrund der zurückgegebenen Variable ausgeführt werden.

Funktion
```
if ML_LOCKTAB[Index].Lockkey = Lockkey
  Variable = LockData
else
  Variable = 0
```

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

LockKey Konstante oder Word

Variable Word oder Long

Beispiel **RLock**(0,4711,DataConst)
RLock(LockConst,KeyConst,DataConst)
RLock(WR1,KeyConst,DataConst)

CLock Datenbereich kryptieren (LOCKTAB

Syntax **CLock**(*Index,Anz,Variable*)

Beschreibung Kryptiert den Inhalt der *Variable* in der angegebenen Länge(*anz*). Der kundenspezifische Kryptierschlüssel von 96 Byte wird durch den Inhalt von ML_LOCKTAB(*Index*).LockData und LockKey gesteuert. Die Daten können nur mit der gleichen Megalock Kundenserie und dem gleichen Inhalt von ML_LOCKTAB(*Index*) entschlüsselt werden.

Index Konstante oder Register (Registerbereich 0-n);
indirekt über Megalock Register (WR0-WR15)

Anz Konstante oder Word

Variable Word, Long oder String

Beispiel **CLock**(0,1024,KryptVar)
CLock(LockRegConst,1024,KryptVar)
CLock(WR1, 1024, KryptVar)

WLock Schreibt Daten in das LockData-Register

Syntax **WLock**(*Index, Variable*)

Beschreibung Überträgt den Inhalt der 32-Bit*Variable* in die ML_LOCKTAB(*Index*). Die unteren 16 Bit der Variable werden nach LockData und die oberen 16-Bit werden nach LockKey übertragen.
Ist ML_LOCKTAB schreibgeschützt wird der Fehlercode 0x32 im Errorregister gesetzt und die Funktion nicht ausgeführt.

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Funktion ML_LOCKTAB(*Index*).LockData = (*Variable* & 0xffff*)*
ML_LOCKTAB(*Index*).LockKey = (*Variable* >>16)

Variable Long

Beispiel **WLock**(0,DataVar)
WLock(LockConst,DataVar)
WLock(WR1, DataVar)

CData Datenbereich kryptieren (ML-Register)

Syntax **CData**(*Register*,*Anz,Variable*)

Beschreibung Kryptiert den Inhalt der *Variable* in der angegebenen Länge(*Anz*).
Der kundenspezifische Kryptierschlüssel von 96 Byte wird durch den Inhalt des Megalock Registers (32 Bit) gesteuert. Die Daten können nur mit der gleichen Megalock Kundenserie und dem gleichen Inhalt des Registers entschlüsselt werden.

Register Register LR0 - LR15.

Anz Konstante oder Variable

Variable Word, Long oder String

Beispiel **CData**(WR0,1024,KryptVar)
CData(WR1,AnzVar,KryptVar)

ACount Dekrementieren des Counter-Registers

Syntax **ACount**(*Index,Variable*)

Beschreibung Der Inhalt ML_COUNTERTAB(*Index)* wird in die *Variable* übertragen.
Ist der Inhalt der ML_COUNTERTAB(*Index)* größer Null, wird der Inhalt um 1 dekrementiert und das Ergebnis im ML_COUNTERTAB(*Index)* abgelegt.

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Funktion *Variable* = ML_COUNTERTAB(*Index*)
if ML_COUNTERTAB(*Index*) > 0
ML_COUNTERTAB(*Index*) - 1

Variable Word oder Long

Beispiel **ACount** (3,CounterVar)
ACount(LockConst,CounterVar)
ACount(WR1, CounterVar)

RCount Counter-Registers lesen

Syntax **RCount**(*Index,Variable*)

Beschreibung Der Inhalt von ML_COUNTERTAB(*Index)* wird in die *Variable* übertragen.
Im Gegensatz zu ACount wird der Wert nicht verändert.

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Funktion *Variable* = ML_COUNTERTAB(*Index*)

Variable Word oder Long

Beispiel **RCount**(0,CounterVar)
RCount(CountConst,CounterVar)
RCount(WR1,CounterVar)

WCount Counter-Registers schreiben

Syntax **WCount**(*Index,Variable*)

Beschreibung Überträgt den Inhalt der Variable in die ML_COUNTERTAB(Index). Ist ML_COUNTERTAB schreibgeschützt wird der Fehlercode 0x32 im Errorregister gesetzt und die Funktion nicht ausgeführt.

Funktion ML_COUNTERTAB(*Index*) = *Variable*

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Variable Word oder Long

Beispiel **WCount**(0,CounterVar)
WCount(CountConst,CounterVar)
WCount (WR1,CounterVar)

RData	**Data-Registers lesen**

Syntax **RData**(*Index,Variable*)

Beschreibung Überträgt den Inhalt der ML_DATATAB(*Index*) in die *Variable*.

Funktion *Variable* = ML_DATATAB(*Index*)

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Variable Word oder Long

Beispiel **RData**(0,DataVar)
RData(DRegNr,DataVar)
Rdata (WR1,DataVar)

WData	**Data-Registers schreiben**

Syntax **WData**(*Index,Variable*)

Beschreibung Überträgt den Inhalt der *Variable* in die ML_DATATAB(Index). Ist ML_DATATAB schreibgeschützt wird der Fehlercode 0x32 im Errorregister gesetzt und die Funktion nicht ausgeführt.

Funktion ML_DATATAB(*Index*) = *Variable*

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Variable Word oder Long

Beispiel **WData** (0,DataVar)
WData (DRegNr,DataVar)
WData (WR1,DataVar)

RString	**String-Registers lesen**

Syntax **RString**(*Index,Variable*)

Beschreibung Überträgt die 8 Byte das ML_STRINGTAB(*Index*) zur *Variable*.

Funktion *Variable* = ML_STRINGTAB(*Index*)

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Variable String (8-Byte)

Beispiel **RString**(0,StringVar)
RString(SRegNr,StringVar)

WString	**String-Registers schreiben**

Syntax **WString**(*Index,Variable*)

Beschreibung Überträgt 8 Byte der *Variable* in die ML_STRINGTAB(*Index*). Ist ML_STRINGTAB schreibgeschützt wird der Fehlercode 0x32 im Errorregister gesetzt und die Funktion nicht ausgeführt.

Funktion ML_STRINGTAB(*Index*) = *Variable*

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Variable String (8-Byte)

Beispiel **WString**(0,StringVar)
WString(SRegNr,StringVar)

PWord Passwortüberprüfung

Syntax **PWord**(*Index,Variable*)

Beschreibung Überprüft das in *Variable* angegebene Passwort mit dem ML_PASSWORDTAB(*Index*). Sind diese identisch, wird die Zugriffsberechtigung (4 x 16 Bit), die für dieses Password hinterlegt wurden, in die Variable, die zuvor das Passwort enthielt, übertragen, andernfalls bleibt der Inhalt der Variablen unverändert.

Funktion if ML_PASSWORD(*Index)*.Password = *Variable*
Variable = Zugriffsberechtigung

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Variable String (8-Byte) oder 16-Bit Tabelle

Beispiel **PWord**(0,PassVar)
PWord(PwdConst,Passvar)
PWord (WR1,PassVar)

WPWord ändert das Passwort

Syntax **WPWord**(*Index,Variable*)

Beschreibung Überträgt das in der *Variable* angegebene Passwort, in den durch Index angegebenen Tabellenplatz ML_PASSWORDTAB(*Index*).Password
Ist ML_PASSWORDTAB schreibgeschützt wird der Fehlercode 0x32 im Errorregister gesetzt und die Funktion nicht ausgeführt.

Funktion ML_PASSWORD(*Index).*Password = *Variable*

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Variable String (8-Byte)

Beispiel **WPWord**(0,PassVar)
WPWord(PwdConst,Passvar)

WPAccess ändert die Zugriffsberechtigung

Syntax **WPAccess** (*Index,Variable*)

Beschreibung Überträgt die in der *Variable* enthaltenen Zugriffsrechte in den durch Index angegebenen Tabellenplatz ML_PASSWORDTAB(*Index*).Access.
Ist ML_PASSWORDTAB schreibgeschützt wird der Fehlercode 0x32 im Errorregister gesetzt und die Funktion nicht ausgeführt.

Funktion ML_PASSWORD(*Index)*.Zugriffsberechtigung =*Var.*

Index Konstante oder indirekte Adressierung über ein Megalock Register (0-15), welches den *Index* enthält.

Variable String (8-Byte) oder 16-Bit Tabelle

Beispiel **WPAccess**(0,StringVar)
WPAccess(PwdConst,AccessTab)

Getreg Megalock-Register 0-20 lesen

Syntax **Getreg**(*Variable,Register*)

Beschreibung Der Inhalt des *Register*s (R0-R20) wird in die angegebene *Variable* übertragen.

Funktion Variable = Register

Register Register R0 - R20, Registervariable.

Variable Word oder Long

Beispiel **Getreg**(WR0,WordVar)
Getreg(LR1,LongVar)

Geterr Fehlermeldung (Error ID) übertragen

Syntax **Geterr**(*Variable*)

Beschreibung Die Error_ID und das Statusregister wird in die angegebene *Variable* übertragen. Das Statusregister wird in den oberen 8-Bit und die Error_ID in den unteren 8-Bit abgelegt. Die Error_ID und die Flags im Statusregister bis auf Zero, Carry und Mastererror werden anschließend gelöscht.

Funktion *Variable* = (Statusregister << 8) + Error_ID
Error_ID = 0
Statusregister &= (Zero, Carry, Mastererror)

Variable Word

Beispiel **Geterr**(p1)

Getmls Status-Informationen übertragen

Syntax **Getmls**(*Variable*)

Beschreibung Überträgt den Status der Megalock API in die angegebene *Variable*. Anschließend wird der Status innerhalb der Megalock API auf 0 gesetzt. Der Befehl **Getmls** überprüft nicht ob zur Ausführungszeit der Megalock Dongle vorhanden ist. Er gibt vielmehr den aufgetretenen Errorcode einer der vorangegangenen Zugriffe auf den Megalock Dongle an. Mögliche aufgetretene Errorcodes sind im Anhang ersichtlich.

Funktion *Variable* = ML_Status

Variable Word

Beispiel **Getmls** (p1)

MEGALOCK BASIC

Die Basicbefehle greifen auf die 20 Megalock Register (32-Bit) zu. Aus Sicht des Programmierers können die Register als externe Variablen angesehen werden, die innerhalb des Megalock Dongles weiter bearbeitet werden können (Addition, Subtraktion etc.). Das Ergebnis der Bearbeitung kann zu einem beliebigen Zeitpunkt wieder an Ihr Programm zurückgegeben werden.
Die Register 0-15 können von jedem Befehl direkt angesprochen werden. Die Register 16- 20 können nur über die Befehle **=, Move** und **Getreg** angesprochen werden.
Sofern nichts anderes erwähnt, können alle Befehle in 16- oder 32-Bit Breite angewendet werden. Sollen die Register auf 16-Bit angewendet werden, so lautet die Defaultbezeichnung für das Register **WR**nr, für 32-Bit Register **LR**nr. Die Operationslänge des jeweiligen Befehls wird durch die Bezeichnung des *Zielregisters* (**WR**nr, **LR**nr) bestimmt.
Die in den nachfolgenden Befehlsbeschreibungen verwendeten Bezeichnungen für *Zielregister,* definieren immer ein Megalock Register, die Bezeichnung *Operand* definiert eine Konstante, Variable oder ein Megalock Register, wobei die Breite des *Operanden* dem des *Zielregisters* entsprechen muss.
Einiger der nachfolgend beschriebenen Befehle verändern das FLAG-Register. Da diese Veränderungen normalerweise in einer Hochsprache nicht von Bedeutung sind, haben wir bei der Beschreibung der Befehle darauf verzichtet. Die Veränderungen sind zu dem äquivalenten Assemblerbefehl beschreiben.

= Übertragen

Syntax Zielregister **=** Operand

Beschreibung Überträgt den Inhalt des *Operanden* in das *Zielregister*. Über den **=** Befehl können auch die oberen Register (16-20) angesprochen werden, jedoch darf nur das *Zielregister* oder das *Quellregister* gleichzeitig auf die oberen Register verweisen.
Beispiele:
LR18 **=** LR1 erlaubt
LR1 **=** LR18 erlaubt
LR16 **=** LR18 Compiler erzeugt Fehlermeldung

Assemblerbefehl **mov**

Beispiel
```
WR0 = 1234
WR1 = WR2
LR1 = LongVariable
```

+= Addieren

Syntax Zielregister **+=** Operand

Beschreibung Addiert den *Operanden* zum Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **add**

Beispiel
```
WR0 = 1234
WR1 = WR2
LR1 = LongVariable
```

-= Subtrahieren

Syntax *Zielregister* **-=** *Operand*

Beschreibung Subtrahiert den *Operanden* vom Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **sub**

Beispiel WR0 **-=** 1234
WR1 **-=** WR2
LR1 **-=** LongVariable

*= Multiplizieren

Syntax *Zielregister* ***=** *Operand*

Beschreibung Multipliziert den *Operand* (16-Bit) mit dem Inhalt des *Zielregisters* (16-Bit) und speichert das Ergebnis (32-Bit) im *Zielregister.*

Assemblerbefehl **mul**

Beispiel WR0 ***=** 1234
WR1 ***=** WR2
LR1 ***=** LongVariable

/= Division

Syntax *Zielregister* **/=** *Operand*

Beschreibung Dividiert den *Operand* (16-Bit) vom Inhalt des *Zielregister* (16-Bit) und speichert das Ergebnis im *Zielregister*. Ist der Inhalt des *Operanden* gleich 0, bleibt das *Zielregister* unverändert.

Assemblerbefehl **div**

Beispiel WR0 **/=** 1234
WR1 **/=** WR2
LR1 **/=** LongVariable

&= UND-Verknüpfung

Syntax *Zielregister* **&=** *Operand*

Beschreibung Verknüpft den *Operanden* bitweise, nach den Regeln der logischen UND-Verknüpfung, mit dem Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **and**

Beispiel WR0 **&=** 1234
WR1 **&=** WR2
LR1 **&=** LongVariable

|= ODER-Verknüpfung

Syntax *Zielregister* **|=** *Operand*

Beschreibung Verknüpft nach den Regeln des logischen ODER den *Operanden* mit dem Inhalt des *Zielregister* und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **or**

Beispiel WR1 **|=** 1234
WR1 **|=** WR2
LR1 **|=** LongVariable

^= EXKLUSIV-ODER-Verknüpfung

Syntax *Zielregister* **^=** *Operand*

Beschreibung Verknüpft die beiden *Operanden* nach den Regeln des logischen EXKLUSIV-ODER und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **xor**

Beispiel
```
WR0 ^= 1234
WR1 ^= WR2
LR1  ^= LongVariable
```

<<= Bitweise SHIFTEN nach links

Syntax *Zielregister* **<<=** *Operand*

Beschreibung Shiftet den Inhalt des *Zielregister* bitweise nach links und rechts wird ein 0-Bit eingefügt. Die Anzahl der Verschiebungen wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Assemblerbefehl **shl**

Beispiel
```
WR0 <<= 3
WR1 <<= WR2
LR1  <<= LongVariable
```

>>= Bitweise SHIFTEN nach rechts

Syntax *Zielregister* **>>=** *Operand*

Beschreibung Shiftet den Inhalt des *Zielregister* bitweise nach rechts und links wird ein 0-Bit eingefügt. Die Anzahl der Verschiebungen wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Assemblerbefehl **shr**

Beispiel
```
WR0 >>= 3
WR1 <<= WR2
LR1  >>= LongVariable
```

Addition Addieren

Syntax **Addition** (*Zielregister*, *Operand*)

Beschreibung Addiert den *Operanden* zum Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **add**

Beispiel
```
Addition (WR0,1234)
Addition (WR1,WR2)
Addition (LR1,LongVariable)
```

And Logische UND-Verknüpfung

Syntax **And** (*Zielregister, Operand*)

Beschreibung Verknüpft den *Operanden* bitweise, nach den Regeln der logischen UND-Verknüpfung, mit dem Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **and**

Beispiel **And** (WR0,1234)

And (WR1, WR2)
And (LR1,LongVariable)

Bitclear	**Löschen eines Bit**

Syntax **Bitclear** (*Zielregister*, *Operand*)

Beschreibung Ein Bit (0-15/31), welches durch den *Operand* spezifiziert ist, wird im *Zielregister* gelöscht.

Assemblerbefehl **bclr**

Beispiel **Bitclear** (WR0,0x5)
Bitclear (WR1, WR2)
Bitclear (LR1,LongVariable)

Bitset	**Setzen eines Bits**

Syntax **Bitset** (*Zielregister, Operand*)

Beschreibung Ein Bit (0-15/31), welches durch den *Operand* spezifiziert ist, wird im *Zielregister* gesetzt.

Assemblerbefehl **bset**

Beispiel **Bitset** (WR0,0x5)
Bitset (WR1, WR2)
Bitset (LR1,LongVariable)

Bittest	**Überprüfen ob ein Bit gesetzt ist**

Syntax **Bittest** (Zielregister, Operand)

Beschreibung Ein Bit, welches durch den *Operand* spezifiziert ist, wird im *Zielregister* getestet. Ist dieses Bit gesetzt, dann wird das Zero-Flag gelöscht, ansonsten wird das Zero-Flag gesetzt. Der Inhalt des *Zielregisters* wird nicht verändert.

Assemblerbefehl **btst**

Beispiel **Bittest** (WR0,0x5)
Bittest (WR4,WR1)
Bittest (LR1,LongVariable)

Clear	**Löschen des Zielregister**

Syntax **Clear** (*Zielregister*)

Beschreibung Der Inhalt des *Zielregisters* wird gelöscht.

Assemblerbefehl **clr**

Beispiel **Clear** (WR0)
Clear (LR1)

Complement	**Bilden des 1er Komplements**

Syntax **Complement** (*Zielregister*)

Beschreibung Bildet vom Inhalt des *Zielregisters* das 1 er Komplement. Das Ergebnis wird im *Zielregister* abgelegt.

Assemblerbefehl **not**

Beispiel **Complement** (WR0)
Complement (LR1)

Decrement Dekrementieren

Syntax **Decrement** (*Zielregister*)

Beschreibung Subtrahiert 1 vom Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **dec**

Beispiel **Decrement** (WR0)
Decrement (LR1)

Division Vorzeichenlose Division

Syntax **Division** (*Zielregister, Operand*)

Beschreibung Dividiert den *Operand* (16-Bit) vom Inhalt des *Zielregisters* (16-Bit) und speichert das Ergebnis im *Zielregister.*

Assemblerbefehl **div**

Beispiel **Division** (1234, WR0)
Division (WR1, WR2)
Division (LongVariable, LR1)

Exklusive EXKLUSIVE-ODER Verknüpfung

Syntax **Exklusive** (*Zielregister*, *Operand*)

Beschreibung Verknüpft die beiden *Operanden* nach den Regeln des logischen EXKLUSIV-ODER und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **xor**

Beispiel **Exklusive** (WR0,1234)
Exklusive (WR1, WR2)
Exklusive (LR1,LongVariable)

Extsign Übertragen mit Vorzeichenerweiterung

Syntax **Extsign** (*Zielregister*, *Operand*)

Beschreibung Überträgt den *Operanden* (Byte bzw. Wort) vorzeichengerecht erweitert in das Zielregister.

Assemblerbefehl **movsx**

Beispiel **Extsign** (WR0,0x2)
Extsign (LR1,0x12)
Extsign (LR2,LongVar)

Extunsign Übertragen mit Nullerweiterung

Syntax **Extunsign** (*Zielregister*, *Operand*)

Beschreibung Erweitert den *Operanden* (Byte bzw. Wort) ohne Beachtung des Vorzeichens und überträgt diesen in das Zielregister.

Assemblerbefehl **movzx**

Beispiel **Extunsign** (WR0,0x5)
Extinsign (LR1,LongVar)

Geterr Fehlernummer übertragen

Syntax **Geterr** (*Variable)*

Beschreibung Die Error_ID und das Statusregister wird in die angegebene *Variable* übertragen. Das Statusregister wird in den oberen 8-Bit und die Error_ID in den unteren 8-Bit abgelegt. Die Error_ID und die Flags im Statusregister bis auf Zero, Carry und Mastererror werden anschließend gelöscht.

Assemblerbefehl **moves**

Beispiel **Geterr** (intvar)

Getmls Statusregister übertragen

Syntax **Getmls**(*Variable)*

Beschreibung Überträgt den Status der Megalock API in die angegebene *Variable*. Anschließend wird der Status innerhalb der Megalock API auf 0 gesetzt. Der Befehl **Getmls** überprüft nicht ob zur Ausführungszeit der Megalock Dongle vorhanden ist. Er gibt vielmehr den aufgetretenen Errorcode einer der vorangegangenen Zugriffe auf den Megalock Dongle an. Mögliche aufgetretene Errorcodes sind im Anhang ersichtlich.

Beispiel **Getmls** (p1)

Getreg Register in Variable übertragen

Syntax **Getreg** (*Register, Variable*)

Beschreibung Der Inhalt des *Register*s (R0-R20) wird in die angegebene *Variable* übertragen.

Assemblerbefehl **movs**

Beispiel **Getreg** (WR1, WordVar)
Getreg (LR2, LongVariable)

Increment Inkrementieren

Syntax **Increment** (*Zielregister*)

Beschreibung Addiert 1 zum Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **inc**

Beispiel **Increment** (WR0)
Increment (LR1)

Load Datenspeicher lesen

Syntax **Load** (*Zielregister*, *Operand*)

Beschreibung Der Inhalt des *Operanden* spezifiziert den Tabellenplatz innerhalb der Datenspeichertabelle. Der Inhalt des Speicherplatzes wird in das *Zielregister* übertragen. Dem **Load** Befehl steht der gesamte Datenspeicher zur Verfügung, d.h. Sie können über den **load** Befehl auch die Bereiche der Toolbefehle lesen. Die Adressierung der Datenspeichertabelle beginnt mit 0. Weitere Informationen zur Speicheraufteilung können Sie im Handbuch Teil II Megalock Toolbox und unter Toolbefehle erhalten.

Assemblerbefehl **load**

Beispiel **Load** (WR0,5)
Load (WR1,WordVar)
Load (LR2,LongVar)

Move Übertragen

Syntax **Move** (*Zielregister, Operand*)

Beschreibung Überträgt den Inhalt des *Operanden* in das *Zielregister*. Über den **Move** Befehl können die oberen Register (16-20) angesprochen werden, jedoch darf nur das *Zielregister* oder das *Quellregister* gleichzeitig auf die oberen Register verweisen.
Beispiele:
Move(LR18,LR1) erlaubt
Move(LR1,LR18) erlaubt
Move(LR16,LR18) Compiler erzeugt Fehlermeldung

Assemblerbefehl **mov**

Beispiel **Move** (WR0, WR1)
Move (WR1,0x1234)
Move (LR19,LongVariable)

Multiply Multiplizieren

Syntax **Multiply** (*Zielregister, Operand*)

Beschreibung Multipliziert den *Operand*(16-Bit) mit dem Inhalt des *Zielregisters*(16-Bit) und speichert das Ergebnis (32-Bit) im *Zielregister*.

Assemblerbefehl **mul**

Beispiel **Multiply** (WR0,0x0a)
Multiply (WR1, WR0)
Multiply (LR1,LongVar)

Negation Bilden des 2er-Komplement

Syntax **Negation** (*Zielregister*)

Beschreibung Bildet vom Inhalt des *Zielregisters* das 2 er-Komplement und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **neg**

Beispiel **Negation** (WR0)
Negation (LR1)

Or Logische ODER-Verknüpfung

Syntax **Or** (*Zielregister, Operand*)

Beschreibung Verknüpft nach den Regeln des logischen ODER, den *Operanden* mit dem Inhalt des *Zielregister* und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **or**

Beispiel **Or** (WR0,12)
Or (WR1,WR0)
Or (LR1,LongVar)

Reset — Megalock aktiv setzen

Syntax **Reset** ()

Beschreibung Veranlaßt den Megalock Prozessor aus dem Halt-Modus zu gehen. Megalock nimmt alle Befehle an und arbeitet sie wieder ab.
Siehe auch : **Stop()** Befehl.

Assemblerbefehl **reset**

Beispiel **Reset** ()

Rotateleft — Rotieren nach links

Syntax **Rotateleft** (*Zielregister*, *Operand*)

Beschreibung Rotiert den Inhalt des *Zielregister* bitweise nach links. Die Anzahl der Rotierung wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Assemblerbefehl **rol**

Beispiel **Rotateleft** (WR0,0x3)
Rotateleft (WR1,IntVar)
Rotateleft (LR1,LongVar)

Rotateright — Rotieren nach rechts

Syntax **Rotateright** (*Zielregister*, *Operand*)

Beschreibung Rotiert den Inhalt des *Zielregisters* bitweise nach rechts. Die Anzahl der Rotierung wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Assemblerbefehl **ror**

Beispiel **Rotateright** (WR0,0x3)
Rotateright (WR1,WR0)
Rotateright (LR1,LongVar)

Save — Datenspeicher schreiben

Syntax **Save** (*Register*, *Operand*)

Beschreibung Der Inhalt des *Operanden* spezifiziert den Tabellenplatz innerhalb der Datenspeichertabelle. Dieser Speicherplatz enthält nach Ausführung des Befehls den Inhalt des Registers. Ist der Memory-Speicher schreibgeschützt, dann wird der Fehlercode 0x32 im Error Register gesetzt und der Befehl wird nicht ausgeführt. Dem **Save** Befehl steht der gesamte Datenspeicher zur Verfügung, d.h Sie können über den **Save** Befehl auch die Bereiche der Toolbefehle verändern. Die Adressierung der Datenspeichertabelle beginnt mit 0. Weitere Informationen zur Speicheraufteilung können Sie im Handbuch Teil II Megalock Toolbox und unter Toolbefehle erhalten.

Assemblerbefehl **save**

Beispiel **Save** (WR0,5)
Save (WR0,WR1)
Save (LR1,6)

Shiftleft Logisches Shiften nach links

Syntax **Shiftleft** (*Zielregister*, *Operand*)

Beschreibung Shiftet den Inhalt des *Zielregisters* bitweise nach links. Rechts wird ein 0-Bit eingefügt. Die Anzahl der Verschiebungen wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Assemblerbefehl **shl**

Beispiel **Shiftleft** (WR0,0x3)
Shiftleft (WR1,WordVar)
Shiftleft (LR1,LongVar)

Shiftright Logisches Shiften nach rechts

Syntax **Shiftright** (*Zielregister*, *Operand*)

Beschreibung Shiftet den Inhalt des *Zielregisters* bitweise nach Rechts. Links wird ein 0-Bit eingefügt. Die Anzahl der Verschiebungen wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.
Assemblerbefehl **shr**

Beispiel **Shiftright** (WR0,0x3)
Shiftright (WR1, WR0)
Shiftright (LR1,LongVar)

Stop Befehlsausführung wird angehalten

Syntax **Stop** ()

Beschreibung Veranlaßt den Megalock Prozessor in den Halt**-**Modus zu gehen. Dieser Zustand wird nur durch den **Reset**-Befehl aufgehoben. Megalock nimmt weiterhin alle Befehle an, arbeitet sie jedoch nicht ab.

Assemblerbefehl **stop**

Beispiel **Stop** ()

Subtract Subtrahieren

Syntax **Subtract** (*Zielregister, Operand*)

Beschreibung Subtrahiert den *Operanden* vom Inhalt des *Zielregister* und speichert das Ergebnis im *Zielregister*.

Assemblerbefehl **sub**

Beispiel **Subtract** (WR0,1234)
Subtract (WR1, WR0)
Subtract (LR1,LongVar)

Swap	**Tauschen von Register-Inhalten**

Syntax **Swap** (*Zielregister*)

Beschreibung Wenn das *Zielregister* 32-Bit groß ist, werden die Bits 0-15 mit 16-31 vertauscht und bei 16-Bit Größe werden die Bits 0-7 mit 8-15 vertauscht.

Assemblerbefehl **swap**

Beispiel **Swap** (WR0)
Swap (LR1)

ASSEMBLERBEFEHLE

Die Assemblerbefehle greifen auf die 20 Megalock Register (32-Bit) zu. Aus Sicht des Programmierers können die Register als externe Variablen angesehen werden, die innerhalb des Megalock Dongles weiter bearbeitet werden können (Addition, Subtraktion etc.). Das Ergebnis der Bearbeitung kann zu einem beliebigen Zeitpunkt wieder an Ihr Programm zurückgegeben werden.
Die Register 0-15 können von jedem Befehl direkt angesprochen werden. Die Register 16- 20 können nur über die Befehle **mov, movis, movid, xchg** angesprochen werden.
Neben den 20 zur freien Verfügung stehenden Register verfügt der Megalock Dongle noch über weitere 3 Register, die über spezielle Befehle angesprochen werden können.
Diese sind:

Register 20 (Timer Register)

time max		time counter	
31	16	15	0

Das Timer Register ist ein 32-Bit Register. In den unteren 16-Bit wird die Zeit (eine Einheit 8 µs) eingetragen, die seit dem Starten des Timers vergangen ist. In den oberen 16-Bit wird der Wert eingetragen, der bei Überschreitung das TM-Flag im Statusregister setzt, sofern die Watchdog-Funktion aktiviert wurde.

Register 21 (ProgNoTab)

ProgNo (0 - 31)	
31	0

Das Register ProgNo enthält die Informationen darüber, welche der max. 32 Programmnummern auf einem Megalock Dongle zugelassen sind. Das Register 21 ist ein Duplikat aus der Tabelle ProgNo des Datenspeichers. Änderungen im Register 21 führen dazu, dass auch automatisch die Tabelle im Datenspeicher angepasst wird.

Register 22 (System Register 1)

Read Counter		Error Counter		Status Register		Error ID	
31	24	23	16	15	8	7	0

Das System Register 1 enthält im Bereich Error ID den letzten aufgetretenen Fehlercode (siehe Fehlertabelle). Der Error Counter enthält die Anzahl der Fehler. Das Read Counter Register wird bei aktivem READCT durch Ausführen der Befehle movs und Getreg um 1 dekrementiert. Das Statusregister ist weiter unterteilt und enthält folgende Informationen:

Status Register

HT	**WD**	**TM**	**ME**	**NC**	**ER**	**CY**	**ZR**
Halt Flag	Watch dog Flag	Time Out Flag	Master Error Flag	nicht belegt	Error Flag	Carry Flag	Zero Flag
7	6	5	4	3	2	1	0

Ist das jeweilige Bit im Flag Register gesetzt, ist dieses aktiv. D.h. enthält das Error-Flag eine “1“, so ist zuvor ein Fehler festgestellt worden.

Register 23 (System Register 2)

reserviert		reserviert		reserviert		EF-Register	
31	24	23	16	15	8	7	0

EF-Register

		MP	**PP**	**SP**	**DP**	**CP**	**LP**
		Memory Protect Flag	Password Protect Flag	String Protect Flag	Data Protect Flag	Counter Protect Flag	Lock Protect Flag
7	6	5	4	3	2	1	0

Ist das jeweilige Bit “1“, so ist der entsprechende Datenbereich im Datenspeicher schreibgeschützt.
Sofern nichts anderes erwähnt, können alle Assemblerbefehle auf 16- oder 32-Bit angewendet werden. Sollen die Register auf 16-Bit angewendet werden, so lautet die Defaultbezeichnung für das Register **WR**nr, für 32-Bit

Register **LR**nr. Die Operationslänge des jeweiligen Befehls wird durch die Bezeichnung des *Zielregisters* (**WR**nr, **LR**nr) bestimmt. Die in den nachfolgenden Befehlsbeschreibungen verwendeten Bezeichnungen für *Zielregister,* definieren immer ein Megalock Register, die Bezeichnung *Operand* definiert eine Konstante, Variable oder ein Megalock Register, wobei die Breite des Operanden dem des Zielregisters entsprechen muss.

adc	**Addition mit Übertrag**

Syntax **adc** *Zielregister,<Operand>*

Beschreibung Addiert den *Operanden* zum Inhalt des *Zielregisters*, addiert 1 dazu, falls das Carry-Flag zuvor gesetzt war und speichert das Ergebnis im Zielregister.

Funktion ZRn = (ZRn + Carry) + Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: wird bei Übertrag gesetzt, ansonsten gelöscht

Beispiel
```
adc  WR1,0x1234
adc  WR3,WR4
adc  LR1,LongVariable
```

add	**Addition**

Syntax **add** *Zielregister,<Operand>*

Beschreibung Addiert den *Operanden* zum Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Funktion ZRn = ZRn + Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: wird bei Übertrag gesetzt, ansonsten gelöscht

Beispiel
```
add  WR1,0x1234
add  WR1,WR2
add  LR1,0x0101ffee
add  LR2,LR3
```

and	**Logisches UND**

Syntax **and** *Zielregister,<Operand>*

Beschreibung Verknüpft den *Operanden* bitweise, nach den Regeln der logischen UND-Verknüpfung, mit dem Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Funktion ZRn = ZRn and Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
and  WR1,0x1234
and  WR0,WR1
and  LR2,LongVariable
```

bclr	**Bit löschen**

Syntax **bclr** *Zielregister,<Operand>*

Beschreibung Ein Bit (0-15/31), welches durch den *Operand* spezifiziert ist, wird im Zielregister gelöscht.

Funktion if LongBef

```
  ZRn = ZRn and (0xffff - (1 << Operand & 0x0f))
else
  ZRn = ZRn and (0xffffffff - (1 << Operand & 0x1f)
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
bclr WR1,0
bclr WR2,WR3
bclr LR1,LongVariable

bset Bit setzen

Syntax **bset** *Zielregister,<Operand>*

Beschreibung Ein Bit (0-15/31), welches durch den *Operand* spezifiziert ist, wird im *Zielregister* gesetzt.

Funktion
```
if LongBef
  ZRn = ZRn or (1 << Operand & 0x0F)
else
  ZRn = ZRn or (1 << Operand & 0x1f)
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
bset WR1,0
bset WR0,WR1
bset LR3,LongVariable

btst Bit testen

Syntax **btst** *Zielregister,<Operand>*

Beschreibung Ein Bit, das durch den *Operand* spezifiziert ist, wird im *Zielregister* getestet. Ist dieses Bit gesetzt dann wird das Zero-Flag gelöscht, ansonsten wird das Zero-Flag gesetzt. Der Inhalt des *Zielregisters* wird nicht verändert.

Funktion ZERO = (ZRn and (1 << Operand))

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
btst WR0,0
btst WR0,WR1
btst LR1,LongVariable

clc Carry-Flag löschen

Syntax **clc**

Beschreibung Das Carry-Flag im Statusregister wird gelöscht.

Funktion C = 0

Statusregister CY: wird gelöscht

Beispiel **clc**

clr Registerinhalt löschen

Syntax **clr** *Zielregister*

Beschreibung Der Inhalt des *Zielregisters* wird gelöscht.

Funktion ZRN = 0

Statusregister ZR: wird gesetzt

Beispiel **clr** WR0
clr LR1

clw Watchdog Flag löschen

Syntax **clw**

Beschreibung Das Watchdog-Flag wird gelöscht. Dieser Befehl wird nur im Programm-Modus ausgeführt.

Statusregister WD: wird auf 0 gesetzt

Beispiel **clw**

cmc Carry-Flag komplementieren

Syntax **cmc**

Beschreibung Wenn das Carry-Flag auf 0 steht, wird es auf 1 und wenn es auf 1 steht, wird es auf 0 gesetzt.

Funktion C = (C ^1)

Statusregister CY: wird komplementiert

Beispiel **cmc**

cmp Zwei Operanden vergleichen

Syntax **cmp** *Zielregister,<Operand>*

Beschreibung Subtrahiert den *Operand* vom Inhalt des *Zielregister*, ohne das Ergebnis im *Zielregister* zu speichern. Es werden nur die Flags verändert.

Funktion FLAGS = ZRn - Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: wird bei Übertrag gesetzt, ansonsten gelöscht

Beispiel **cmp** WR1,0x1234
cmp WR1,WR2
cmp LR1,LongVariable

crypt Register kryptieren

Syntax **crypt** *Zielregister,<Operand>*

Beschreibung Der Inhalt des *Zielregisters* wird über die Funktion random(*Operand*) EXKLUSIV-ODER verknüpft.

Funktion ZRn = (ZRn xor random(Operand))

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel mov WR0,0x1234
mov WR1,0x4711
crypt WR0,WR1

dec	**Dekrementieren des Zielregisters um 1**

Syntax **dec** *Zielregister*

Beschreibung Subtrahiert 1 vom Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Funktion ZRn = ZRn - 1

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: wird bei Übertrag gesetzt, ansonsten gelöscht

Beispiel **dec** LR1

div	**Vorzeichenlose Division**

Syntax **div** Zielregister,<Operand>

Beschreibung Dividiert den *Operand* (16-Bit) vom Inhalt des *Zielregisters* (16-Bit) und speichert das Ergebnis im *Zielregister.*

Funktion ZRn = ZRn / Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: wird bei Übertrag gesetzt, ansonsten gelöscht

Beispiel **div** WR0,0x1234
div WR0,WR2
div LR1,LongVariable

idiv	**Vorzeichenbehaftete Division**

Syntax **idiv** *Zielregister,<Operand>*

Beschreibung Dividiert den *Operand*(16-Bit), unter Berücksichtigung des Vorzeichens, vom Inhalt des *Zielregisters*(16-Bit). Das Ergebnis wird im Zielregister gespeichert.

Funktion ZRn = ZRn / Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: wird bei Übertrag gesetzt, ansonsten gelöscht

Beispiel **idiv** WR0,0x1234
idiv WR0,WR2
idiv LR2,LongVariable

imul	**Vorzeichenbehaftete Multiplikation**

Syntax **imul** *Zielregister,<Operand>*

Beschreibung Multipliziert, unter Berücksichtigung des Vorzeichens, den *Operand*(16-Bit) mit dem Inhalt (16-Bit) des *Zielregister.* Das Ergebnis (32-Bit) wird im Zielregister gespeichert.

Funktion ZRn = ZRn * Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: wird bei Übertrag gesetzt, ansonsten gelöscht

Beispiel **imul** WR0,0x1234
imul WR0,WR2
imul LR1,LongVariable

inc	**Inkrementieren des Zielregisters um 1**

Syntax **inc** *Zielregister*

Beschreibung Addiert 1 zum Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister* .

Funktion ZRn = ZRn + 1

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: wird bei Übertrag gesetzt, ansonsten gelöscht

Beispiel
```
inc   WR1
inc   LR2
```

ja	**Sprung, wenn gößer**

Syntax **ja** *Marke*

Beschreibung Verzweigt zur angegebenen *Marke* wenn das Zero-Flag = 0 und das Carry-Flag = 0 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion
```
if ZR = 0 and CY = 0
  IP = Marke
```

Statusregister unverändert

Beispiel
```
ja    Label
```

jae	**Sprung, wenn größer oder gleich**

Syntax **jae** *Marke*

Beschreibung Verzweigt zur angegebenen *Marke,* wenn das Carry-Flag = 0 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion
```
if CY = 0
  IP = Marke
```

Statusregister unverändert

Beispiel
```
jae   Label
```

jb	**Sprung, wenn kleiner**

Syntax **jb** *Marke*

Beschreibung Verzweigt zur angegebenen *Marke,* wenn das Carry-Flag = 1 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion
```
if CY = 1
  IP = Marke
```

Statusregister unverändert

Beispiel
```
jb    Label
```

jbe	**Sprung, wenn kleiner gleich**

Syntax **jbe** *Marke*

Beschreibung Verzweigt zur angegebenen *Marke* wenn das Zero-Flag = 1 oder das Carry-Flag = 1 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion if ZR = 1 or CY = 1
IP = Marke

Statusregister unverändert

Beispiel **jbe** Label

jc Sprung, wenn Carry gesetzt

Syntax **jc** *Marke*

Beschreibung Verzweigt zur angegebenen *Marke*, wenn das Carry-Flag = 1 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion if CY = 1
IP = Marke

Statusregister unverändert

Beispiel **jc** Label

jmp Unbedingter Sprung

Syntax **jmp** *Marke*

Beschreibung Bewirkt einen unbedingten Sprung zur angegebenen *Marke*.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion IP = Marke

Statusregister unverändert

Beispiel **jmp** Label

jna Sprung, wenn nicht größer

Syntax **jna** *Marke*

Beschreibung Verzweigt zur angegebenen *Marke*, wenn das Zero-Flag = 1 oder Carry-Flag = 1 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion if ZR = 1 or CY = 1
IP = *Marke*

Statusregister unverändert

Beispiel **jna** Label

jnae Sprung, wenn nicht größer gleich

Syntax **jnae** *Marke*

Beschreibung Verzweigt zur angegebenen *Marke*, wenn das Carry-Flag = 1 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion if CY = 1
IP = *Marke*

Statusregister unverändert

Beispiel **jnae** Label

jnb — Sprung, wenn nicht kleiner

Syntax **jnb** *Marke*

Beschreibung Verzweigt zur angegebenen *Marke*, wenn das Carry-Flag = 0 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion if CY = 0
IP = *Marke*

Statusregister unverändert

Beispiel **jnb** Label

jnbe — Sprung, wenn nicht kleiner gleich

Syntax **jnbe** *Marke*

Beschreibung Verzweigt zur angegebenen *Marke*, wenn das Zero-Flag = 0 und das Carry-Flag = 0 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion if ZR = 0 and CY = 0
IP = *Marke*

Statusregister unverändert

Beispiel **jnbe** Label

jnc — Sprung, wenn nicht Carry

Syntax **jnc** *Marke*

Beschreibung Verzweigt zur angegebenen *Marke*, wenn das Carry-Flag = 0 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion if CY = 0
IP = Marke

Statusregister unverändert

Beispiel **jnc** Label

je / jz — Sprung, wenn gleich

Syntax **jz** *Marke*
je Marke

Beschreibung Verzweigt zur angegebenen *Marke* wenn das Zero-Flag = 1 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion if ZR = 1
IP = Marke

Statusregister unverändert

Beispiel **jz** Label

je Label

jne / jnz	**Sprung, wenn nicht gleich**

Syntax **jnz** *Marke*
jne Marke

Beschreibung Verzweigt zur angegebenen *Marke*, wenn das Zero-Flag = 0 ist.
Dieser Befehl ist nur verwendbar, wenn Ihr Compiler den Goto-Befehl ausführen kann.

Funktion
```
if ZR = 0
  IP = Marke
```

Statusregister unverändert

Beispiel
```
jnz  Label
jne  Label
```

ldv	**Megalock-Version laden**

Syntax **ldv** *Zielregister*

Beschreibung Die Megalock Dongle-Versionsnummer wird in die unteren 8-Bit des Zielregisters übertragen, die restlichen Bits bleiben unverändert. Bit 0-3 geben die Revision an. Bit 4-7 enthält die Versionsnummer.

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
ldv  WR1
ldv  LR2
```

load	**Datenspeicher lesen**

Syntax **load** *Zielregister, <Operand>*

Beschreibung Der Inhalt des *Operanden* spezifiziert den Tabellenplatz innerhalb der Datenspeichertabelle. Der Inhalt des Speicherplatzes wird in das *Zielregister* übertragen. Dem load Befehl steht der gesamte Datenspeicher zur Verfügung, d.h. Sie können über den load Befehl auch die Bereiche der Toolbefehle lesen. Die Adressierung der Datenspeichertabelle beginnt mit 0. Weitere Informationen zur Speicheraufteilung können Sie im Handbuch Teil II Megalock Toolbox und unter Toolbefehle erhalten.

Funktion
```
if Long
  if Operand < (DatenSpeicher /2)
       ZRn = Datenspeicher[Operand]
  else
       Errorr ID = 0x31
else
  if Operand < (Datenspeicher / 4)
       ZRn = Datenspeicher[Operand]
  else
       Error ID = 0x31
```

Statusregister wird die Error ID gesetzt, wird auch das ER und ME im Statusregister gesetzt.

Beispiel
```
load  WR1,WR2
load  LR3, LR4
```

lpn	**PNR (Programnumber-Register) übertragen**

Syntax **lpn** *Zielregister*

Beschreibung Überträgt den Inhalt des Registers 21 (ProgNoTab) in das *Zielregister*.

Funktion ZRn = RPN

Statusregister unverändert.

Beispiel **lpn** WR1
lpn LR2

mov	**Übertragen**

Syntax **mov** *Zielregister,<Operand>*

Beschreibung Überträgt den Inhalt des *Quellregister* bzw. *Operanden* in das *Zielregister*.
Über den mov Befehl können die oberen Register (16-20) angesprochen werden, jedoch darf nur das *Zielregister* oder das *Quellregister* gleichzeitig auf die oberen Register verweisen.
Beispiele:
mov LR18,LR1 erlaubt
mov LR1,LR18 erlaubt
mov LR16,LR18 Compiler erzeugt Fehlermeldung

Funktion ZRn = Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel **mov** WR17,0x1234
mov LR16,LR15
mov LR1,LongVariable

move	**Statusregister übertragen**

Syntax **move** *Zielregister*

Beschreibung Überträgt den Inhalt des Registers 21 in das *Zielregister*. Wird als Zielregister (16-Bit) angegeben, werden nur die unteren 16-Bit übertragen und die Error Id wird auf 0 und die Flags werden gelöscht.
Wird als Zielregister (32-Bit) abgegeben, werden alle 32 Bit übertragen und zusätzlich wird der Error Counter auf 0 gesetzt.
Überträgt den Status der Megalock API in die angegebene *Variable*. Anschließend wird der Status innerhalb der Megalock API auf 0 gesetzt. Der Befehl **move** überprüft nicht ob zur Ausführungszeit der Megalock Dongle vorhanden ist. Er gibt vielmehr den aufgetreten Errorcode einer der vorangegangenen Zugriffe auf den Megalock Dongle an. Mögliche aufgetretene Errorcodes sind im Anhang ersichtlich.

Beispiel **move** WR0
move LR1

moves	**Error-ID und -Statusregister laden**

Syntax **moves** *Variable*

Beschreibung Überträgt den Inhalt der unteren 16-Bit des Registers 21 in die Variable(16-Bit). Die Error Id wird auf 0 gesetzt.

Statusregister ZR, CY und ME bleiben unverändert,
die restliche Flags werden auf 0 gesetzt.

Beispiel **moves** mlStatus

movid Destination indirekte übertragen

Syntax **movid** *Zielregister,Quellregister*

Beschreibung Der Inhalt des *Quellregisters* wird in das Register übertragen, welches durch den Inhalt des *Zielregisters* angegeben ist.

Funktion ZRn[0-20] = QRn

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
mov  R1,10
movidR1,R0  //Reg0 wird nach R10 übertragen
```

movis Source indirekt übertragen

Syntax **movis** *Zielregister,Quellregister*

Beschreibung Der Inhalt des Registers, welches durch den Inhalt (0-20) des *Quellregisters* angegeben ist, wird in das *Zielregister* übertragen.

Funktion ZRn = Register[QRn]

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
mov  R1,10
movisR0,R1  //Reg10 wird nach R0 übertragen
```

movlx Übertragen der unteren 8-Bit

Syntax **movlx** *Zielregister,<Operand>*

Beschreibung Überträgt die unteren 8 Bit bzw. 16 Bit des *Operanden* ohne Erweiterung in das Zielregister.

Funktion
```
if LongBef
  ZRn = ((ZRn &0xFFFF0000) or (QRn and 0x0000FFFF))
else
  ZRn = ((ZRn &0xFFFFFF00) or (QRn and 0x000000FF))
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
movlxWR1,WR2
movlxLR3,LR4
```

movs Register in Variable übertragen

Syntax **movs** *Variable,Quellregister*

Beschreibung Der Inhalt des *Quellregisters* wird in die angegebene Variable übertragen.

Funktion Variable = ZRn

Beispiel
```
movs Variable,WR2
movs LongVariable,LR1
```

movsx	**Übertragen mit Vorzeichenerweiterung**

Syntax **movsx** *Zielregister,<Operand>*

Beschreibung Überträgt den *Operanden* vorzeichengerecht erweitert in das Zielregister.

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel **movsx** WR0,WR1

movzx	**Übertragen mit Nullerweiterung**

Syntax **movzx** *Zielregister,<Operand>*

Beschreibung Erweitert den *Operanden* ohne Beachtung des Vorzeichens und überträgt diesen in das Zielregister.

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel **movzx** WR1,WR0

mul	**Vorzeichenlose Multiplikation**

Syntax **mul** *Zielregister,<Operand>*

Beschreibung Multipliziert den *Operand*(16-Bit) mit dem Inhalt des *Zielregisters*(16-Bit) und speichert das Ergebnis (32-Bit) im *Zielregister.*

Funktion ZRn = ZRn * Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
mul  WR0,0x1234
mul  WR0,WR1
mul  LR1,LongVariable
```

neg	**Negieren**

Syntax **neg** *Zielregister*

Beschreibung Bildet vom Inhalt des *Zielregisters* das 2 er-Komplement und speichert das Ergebnis im *Zielregister*.

Funktion
```
if LongBef
  ZRn = (0xffffffff - ZRn) + 1
else
  ZRn = (0xffff -ZRn) +1
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
neg  WR1
neg  LR1
```

nop	**Keine Operation**

Syntax **nop**

Beschreibung **nop** bewirkt keine Operation.

Funktion keine.

Statusregister unverändert

Beispiel **nop**

not Logisches Nicht

Syntax **not** Zielregister

Beschreibung Bildet vom Inhalt des *Zielregisters* das 1 er-Komplement. Das Ergebnis wird im *Zielregister* abgelegt.

Funktion
```
if LongBef
  ZRn = 0xffffffff - ZRn
else
  ZRn = (ZRn and 0xffff0000) or (0xffff - ZRn)
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
not  WR0
not  LR1
```

or Logisches ODER

Syntax **or** *Zielregister,<Operand>*

Beschreibung Verknüpft nach den Regeln des logischen ODER den *Operanden* mit dem Inhalt des *Zielregister* und speichert das Ergebnis im *Zielregister*.

Funktion ZRn = ZRn or Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
or   WR1,0x1234
or   WR0,WR1
or   LR2,LongVariable
```

rcl Rotieren nach links über das Carry-Flag

Syntax **rcl** *Zielregister,<Operand>*

Beschreibung Rotiert den Inhalt des *Zielregisters* bitweise nach links unter Einbeziehung des Carry-Flags. Die Anzahl der Rotierung wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Funktion
```
if LongBef
  ZRn = (( C <- ZRn <- C)  loop (Operand &0x1f))
else
  ZRn = (( C <- ZRn <- C)  loop (Operand &0x0f))
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: enthält das letzte Bit, das hinausrotiert wurde.

Beispiel
```
rcl  WR0,5
rcl  WR0,WR1
rcl  LR2,LongVariable
```

rcr	**Rotieren nach rechts über das Carry-Flag**

Syntax **rcr** *Zielregister,<Operand>*

Beschreibung Rotiert den Inhalt des *Zielregisters* bitweise nach rechts unter Einbeziehung des Carry-Flags. Die Anzahl der Rotierung wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Funktion
```
If LongBef
  ZRn = ((CR -> ZRn -> CR) loop (Operand &0x1f))
else
  ZRn = ((CR -> ZRn -> CR) loop (Operand &0x0f))
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: enthält das letzte Bit, das hinausrotiert wurde.

Beispiel
```
rcr   WR0,5
rcr   WR0,WR2
rcr   LR1,LongVariable
```

reset	**Megalock aktive setzen**

Syntax **reset**

Beschreibung Veranlaßt den Megalock Prozessor aus dem Halt-Modus zu gehen. Der Megalock Dongle nimmt alle Befehle an und arbeitet sie wieder ab.

Statusregister RC wird auf ff gesetzt,
die restlichen Flags werden auf 0 gesetzt

Beispiel **reset**

rnd	**Zufallszahl erzeugen**

Syntax **rnd** *Zielregister*

Beschreibung Dem Inhalt des Zielregisters wird über die Random Funktion eine neue Zufallszahl übertragen.

Funktion ZRn = Random(ZRn)

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel **rnd** WR1

rol	**Rotieren nach links**

Syntax **rol** *Zielregister,<Operand>*

Beschreibung Rotiert den Inhalt des *Zielregisters* bitweise nach links. Die Anzahl der Rotierung wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Funktion
```
if LongBef
  ZRn = ( ( +<- ZRn <-+ )  loop ( Operand &0x1f) )
else
  ZRn = ( ( +<- ZRn <-+ )  loop ( Operand &0x0f) )
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
rol   WR0,5
rol   WR0,WR1
```

rol LR1,LongVariable

ror	**Rotieren nach rechts**

Syntax **ror** *Zielregister,<Operand>*

Beschreibung Rotiert den Inhalt des *Zielregisters* bitweise nach rechts. Die Anzahl der Rotierung wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Funktion
```
if Long
  ZRn = ( ( +-> ZRn ->+ )  loop ( Operand &0x1f) )
else
  ZRn = ( ( +-> ZRn ->+ )  loop ( Operand &0x0f) )
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
ror   WR0,5
ror   WR0.WR1
ror   LR1,LongVariable
```

save	**Datenspeicher schreiben**

Syntax **save** Register, <Operand>

Beschreibung Der Inhalt des *Operanden* spezifiziert den Tabellenplatz innerhalb der Datenspeichertabelle. Dieser Speicherplatz enthält nach Ausführung des Befehls den Inhalt des Registers. Ist der Memory-Speicher schreibgeschützt, dann wird der Fehlercode 0x32 im Error Register gesetzt und der Befehl wird nicht ausgeführt. Dem **save** Befehl steht der gesamte Datenspeicher zur Verfügung, d.h. Sie können über den save Befehl auch die Bereiche der Toolbefehle verändern. Die Adressierung der Datenspeichertabelle beginnt mit 0. Weitere Informationen zur Speicheraufteilung können Sie im Handbuch Teil II Megalock Toolbox und unter Toolbefehle erhalten.

Funktion
```
if Memory <> write protect
  if LongBef
      if Operand < (max. Datenspeicher /2)
            Datenspeicher[Operand] = ZRn
      else
            Error ID = 0x31
  else
      if Operand < (max. Datenspeicher / 4)
            Datenspeicher[Operand] = ZRn
      else
            Error ID = 0x31
else
  Errorregister = 0x32
```

Statusregister wird die Error ID gesetzt, wird auch das ER und ME im Statusregister gesetzt.

Beispiel
```
save  WR1,WR2
save  LR3,LR4
```

sbb	**Subtraktion mit Brogen**

Syntax **sbb** *Zielregister,<Operand>*

Beschreibung Subtrahiert den *Operanden* vom Inhalt des Zielregisters, zieht 1 ab, falls das Carry-Flag gesetzt war. Das Ergebnis wird im *Zielregister* gespeichert.

Funktion ZRn = (ZRn - Operand) - CR)

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: wird bei Übertrag gesetzt, ansonsten gelöscht

Beispiel **sbb** WR0,5
sbb WR0,WR1
sbb LR1,LongVariable

sef EEPFlag Register lesen und setzen

Syntax **sef** *Zielregister*

Beschreibung Das EEPFlag Register wird in das Highbyte des *Zielregisters* übertragen und das Lowbyte des *Zielregisters* wird in das EEPFlag Register übertragen. Änderungen in diesem Register führen nicht dazu, das dass Duplikat im Datenspeicher verändert wird.

Funktion ZRn = (EEPFlag <<8) + (ZRn &0x00ff)
EEPFlag = (ZRn &0x00ff)

Beispiel **sef** WR1
sef LR1

shl Logisches Shiften nach links

Syntax **shl** *Zielregister,<Operand>*

Beschreibung Shiftet den Inhalt des *Zielregisters* bitweise nach links. Rechts wird ein 0-Bit eingefügt. Die Anzahl der Verschiebungen wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Funktion
```
if LongBef
  ZRn = ((CR <- ZRn <- 0) loop ( Operand &0x1f))
else
  ZRn = ((CR <- ZRn <- 0) loop ( Operand &0x0f))
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: enhält das letzte Bit, das hinausshiftet wurde.

Beispiel **shl** WR0,5
shl WR0,WR1
shl LR2,LongVariable

shr Logisches Shiften nach rechts

Syntax **shr** *Zielregister,<Operand>*

Beschreibung Shiftet den Inhalt des *Zielregisters* bitweise nach rechts. Links wird ein 0-Bit eingefügt. Die Anzahl der Verschiebungen wird im *Operanden* angegeben. Das Ergebnis wird im *Zielregister* gespeichert.

Funktion
```
if LongBef
  ZRn = (( 0 -> ZRn -> CR) loop ( Operand &0x1f))
else
  ZRn = (( 0 -> ZRn -> CR) loop ( Operand &0x0f))
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: enthält das letzte Bit, das hinausgeshiftet wurde

Beispiel **shr** WR0,5
shr WR0,WR1
shr LR2,LongVariable

spn	**PNR (Programnumber-Register) setzen**

Syntax **spn** *Rn*

Beschreibung Überträgt den Inhalt des *Rn* in das Progno Register (21) und speichert die Änderungen auch in dem Duplikat innerhalb des Datenspeichers.

Funktion PNR = ZRn)

Statusregister Das Statusregister wird nicht verändert.

Beispiel **spn** WR1
spn LR2

src	**RC Register lesen und setzen**

Syntax **src** *Zielregister*

Beschreibung Das RC Register wird in das Highbyte des *Zielregisters* übertragen und das Lowbyte des *Zielregisters* wird in das RC Register übertragen.

Funktion ZRn = (RC <<8) + (ZRn &0x00ff)
RC = (ZRn &0x00ff)

Statusregister unverändert

Beispiel **src** WR1

stc	**Carry-Flag setzen**

Syntax **stc**

Beschreibung Das Carry-Flag im Statusregister wird gesetzt.

Funktion C = 1

Statusregister CY: wird gesetzt

Beispiel **stc**

stop	**Befehlsausführung wird angehalten**

Syntax **stop**

Beschreibung Veranlaßt den Megalock Prozessor in den Halt-Modus zu gehen. Dieser Zustand wird nur durch den **reset**-Befehl aufgehoben. Megalock nimmt weiterhin alle Befehle an, arbeitet sie jedoch nicht ab.

Statusregister HT wird gesetzt

Beispiel **stop**

stw	**Watchdog Flag setzen**

Syntax **stw**

Beschreibung Das Watchdog-Flag wird gesetzt. Dieser Befehl wird nur im Programm-Modus ausgeführt.

Statusregister WD wird gesetzt

Beispiel **stw**

sub Subtrahieren

Syntax **sub** *Zielregister,<Operand>*

Beschreibung Subtrahiert den *Operanden* vom Inhalt des *Zielregisters* und speichert das Ergebnis im *Zielregister*.

Funktion ZRn = ZRn - Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht
CY: wird bei Übertrag gesetzt, ansonsten gelöscht

Beispiel
```
sub  WR0,5
sub  WR0,WR1
sub  LR1,LongVariable
```

swap Tauschen von Register-Werten

Syntax **swap** *Zielregister*

Beschreibung Wenn das *Zielregister* 32-Bit groß ist, werden die Bits 0-15 und 16-31 vertauscht und bei 16-Bit Größe werden die Bits 0-7 und 8-15 vertauscht.

Funktion
```
if LongBef
  ZRN = ( ZRn >> 16) + (ZRn << 16)
else
  ZRN = ( ZRn >> 8) + (ZRn << 8)
```

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
swap WR1
swap LR2
```

tmr Timer run

Syntax **tmr**

Beschreibung Setzt das Timer Counter auf 0 und aktiviert bei Zeitüberschreitung, des im oberen 16 Bit des Registers 20 definierten Zeitwert das TM-Flag. Wird innerhalb eines Programmblockes die Watchdog-Überwachung aktiviert, wird beim setzen des TM-Flags gleichzeitig das Error-Flag gesetzt. Die Timerfunktion erhöht alle 8µs den Timer Counter um 1, und beim Erreichen des voreingestellten Timer Max wird das TM-Flag gesetzt.

Statusregister TM :wird beim erreichen von Timer Max gesetzt

Beispiel **tmr**

tms Timer stop

Syntax **tms**

Beschreibung Stoppt den Timer Counter und speichert die verstrichene Zeit zwischen **tmr** und **tms** in die unteren 16 Bit vom Register 20.

Beispiel **tmr**

test	**Testen oder logischer Vergleich**

Syntax **test** *Zielregister,<Operand>*

Beschreibung Führt eine bitweise Verknüpfung der Operanden nach den Regeln der logischen UND-Verküpfung durch, speichert jedoch das Ergebnis nicht ab. Es wird nur das Zero-Flag verändert.

Funktion FLAGS = ZRn and Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
test   WR1,5
test   WR01,WR1
test   LR1,LongVariable
```

xchg	**Tauschen von Register-Inhalten**

Syntax **xchg** *Zielregister, Quellregister*

Beschreibung Tauscht die Inhalte des *Quell-* und *Zielregisters*.
Über den xchg Befehl können die oberen Register (16-20) angesprochen werden, jedoch darf nur das *Zielregister* oder das *Quellregister* gleichzeitig auf die oberen Register verweisen.
Beispiele:
xchg LR18,LR1 erlaubt
xchg LR1,LR18 erlaubt
xchg LR16,LR18 Compiler erzeugt Fehlermeldung

Funktion
```
temp = ZRn
ZRn  = QRn
QRn  = temp
```

Statusregister ZR: wird gesetzt bei ZRn 0, ansonsten gelöscht

Beispiel **xchg** R1,R2

xor	**Exklusives Oder**

Syntax **xor** *Zielregister,<Operand>*

Beschreibung Verknüpft die beiden *Operanden* nach den Regeln des logischen EXKLUSIV-ODER und speichert das Ergebnis im *Zielregister*.

Funktion ZRN = ZRn **xor** Operand

Statusregister ZR: Ergebnis = 0 gesetzt, ansonsten gelöscht

Beispiel
```
xor   WR0,5
xor   WR0,WR1
xor   Lr2,LongVariable
```

FLASHDRIVE BEFEHLE

Optional verfügt der MegalockUSB-Dongle über einen FlashDrive. Dieser Datenspeicher fungiert wie ein Memorystick und wird auch so vom Betriebssystem erkannt und als zusätzliches Laufwerk ins System eingebunden. Die Daten werden beim schreiben automatisch verschlüsselt wahlweise AES-128 oder schneller mit einer X-OR Tabelle. Das Laufwerk kann immer aktiviert sein oder durch ein Password geschützt, eben so kann ein Schreibschutz aktiviert werden.

Die Voreinstellungen werden über das Dialog-Fenster **Toolbox-Basis** vorgenommen. Hier stehen im Rahmen „FlashDrive“ nachfolgende Schalter.

Aktive

Der Massenspeicher wird vom Betriebssystem erkannt, wird dieser Schalter nicht gesetzt kann der Datenspeicher nur über die Megalock Befehle **fdLoad** und **fdSave** gelesen und geschrieben werden.

Open

Das Laufwerk ist nicht mit einem Password geschützt und steht somit schon während des Bootvergang zur Verfügung.

AES-128

Aktiviert die AES-128 Bit Verschlüsselung, wird dieser Schalter nicht gesetzt werden die Daten mit einer schnellen variablen X-OR Tabelle verschlüsselt. Die Codier-Schlüssel werden sowohl bei AES-128 als auch bei der X-OR Tabelle bereits bei Herstellung des Dongles festgelegt und sind **Unikate**-Codier-Schlüssel. Die Art der Verschlüsselung ist gar nicht so wichtig. Ist das Laufwerk nicht geöffnet (Open-Funktion) werden auch keine Daten übertragen die decodiert werden könnten. Ein unerlaubter Zugriff auf die codierten Daten kann nur von Hardware-Seite erfolgen. Der Aufwand ist erheblich und nur mit spezieller Hardware möglich. Wenn Sie jedoch ein Angriff von Hardware-Seite beführten dann wählen Sie die AES-128 Verschlüsselung.
Die Zugriffszeit ist bei AES-128 geringer als bei der X-OR Tabelle.

Protect

Wird dieser Schalter aktiviert kann das Laufwerk nur gelesen werden.

fdOpen	**FlashDrive öffen**

Syntax **fdOpen**(Register,Modus$)

Beschreibung Ist das Laufwerk durch ein Password geschützt, so kann über den Befehl **fdOpen** der Zugriff auf das Laufwerk erlaubt werden. Das Password ist immer 16 Byte lang und ist in 4 aufeinanderfolgenden Registern angegeben. Ist das Password mit einen der beiden gespeicherten Passwörtern identisch, wir das Laufwerk freigegeben.

Register Als Register kann R0,R4,R8 und R12 angegeben werden, wird z.B. das Register R4 gewählt so sind zuvor die Register R4,R5,R6 und R7 mit dem Password zu laden.

Modus$ “R” Das Laufwerk ist schreibgeschützt.
“RW“ Lesen und Schreiben ist erlaubt.

Beispiel
```
Move(LR4,0x11112222)
Move(LR5,0x33334444)
Move(LR6,0x555566666)
Move(LR7,0x77778888)
fdOpen(LR4,“RW“)
```

fdClose	**FlashDrive schliessen**

Syntax **fdClose**()

Beschreibung Ist das Laufwerk durch ein Password geschützt, dann kann über **fdClose** der Zugriff auf das Laufwerk wieder gespeert werden.

Wichtig Es ist aber darauf zu achten das alle zu schreibenden Daten vom Betriebssystem bereits auf das Laufwerk geschrieben wurden.

Beispiel **fdClose**()

fdPassword FlashDrive Password ändern

Syntax **fdPassword**(Register,Id$)

Beschreibung Der Befehl ändert das Password „1“ oder „2“. Das aktuelle Password wird wie beim **fdOpen**-Befehl zuvor in 4 aufeinanderfolgende Register geladen und das neue Password in die nächsten 4 Register. Es werden also 8 aufeinanderfolgende Register benötigt. Zu erst wird geprüft ob das aktuelle Password übereinstimmt und dann wird das neue Password gesichert. Mit dem Password „2“ (Master-Password) können beide Passwörter geändert werden, mit dem Password „1“ kann nur das Password „ 1“ geändert.

Beide Passwörter sind jeweils pro Byte mit 0xff vorinstalliert.
Nach Aufruf des **fdPassword**-Befehls wird das Laufwerk geschlossen und ggf. erneuet mit einem **fdOpen**-Befehl und dem neuen Password geöffnet werden.

Register Als Register kann R0,R4,R8 und R12 angegeben werden, wird z.B. das Register R4 gewählt so sind zuvor die Register R4,R5,R6 und R7 mit dem aktuellen Password und R8,R9,R10 und R11 mit dem neuen Password zu laden.

Modus$ “1” Password 1 soll geändert werden.
“2“ Das Master-Password soll geändert werden.

Beispiel
```
Move(LR4,0xffffffff)
Move(LR5,LR4)
Move(LR6,LR4)
Move(LR7,LR4)
// neues Password
Move(LR8,0xaaaaaaaa)
Move(LR9,0xbbbbbbbb)
Move(LR10,0xcccccccc)
Move(LR11,0xdddddddd)

fdPassword(LR4,“1“)
```

fdEnable FlashDrive aktivieren

Syntax **fdEnable**()

Beschreibung Das FlashDrive Laufwerk wird wieder aktiviert und der Zugriff auf das Laufwerk ist wieder möglich.

Beispiel fdEnable()

fdDisable FlashDrive deaktivieren

Syntax **fdDisable**()

Beschreibung Das FlashDrive Laufwerk wird deaktiviert und der Megalock Dongle kann anschließend, ohne Fehlermeldung des Betriebssystems, von der USB-Schnittstelle abgezogen werden. Zuvor ist aber sicherzustellen das keine Anwendung mehr auf das FlashDrive Laufwerk zugreifen.

Beispiel fdDisable()

fdStart	**FlashDrive starten**

Syntax **fdStart**()

Beschreibung Der Treiber für das Flashdrive Laufwerk wird gestartet. Der Zugriff auf das Laufwerk ist aber erst nach einem Neustart des Betriebssystems möglich.

Beispiel fdStart()

fdStop	**FlashDrive anhalten**

Syntax **fdStop**()

Beschreibung Der Treiber für das FlashDrive Laufwerk wird gestopt. Ein Zugriff ist nicht nicht mehr möglich.

Beispiel fdStop()

fdLoad	**FlashDrive lesen**

Syntax **fdLoad**(*Register*,*Variable*)

Beschreibung Überträgt einen Datenblock, 512 Byte, in die *Variable*. Der jeweilige Datenblock wird im *Register* angegeben. *Das Register* muss als Long-Register angegeben werden und kann bei 2MB Datenspeicher Werte von Null bis 0x00000FFF bzw. bei 4MB Datenspeicher Werte von Null bis 0x00001FFF enthalten. Der Datenspeicher darf nicht als FlashDrive aktiviert sein und muss zuvor durch **fdOpen** geöffnet werden.

Funktion *Variable* = Datenspeicher[*Register]*

Beispiel
```
Move(LR0,0)
fdLoad(LR0,Buffer512)
```

fdSave	**FlashDrive schreiben**

Syntax **fdSave**(*Register*,*Variable*)

Beschreibung Überträgt 512 Byte aus der Variable in den Datenspeicher. Der jeweilige Datenblock wird im *Register* angegeben. *Das Register* muss als Long-Register angegeben werden und kann bei 2MB Datenspeicher Werte von Null bis 0x00000FFF bzw. bei 4MB Datenspeicher Werte von Null bis 0x00001FFF enthalten. Der Datenspeicher darf nicht als FlashDrive aktiviert sein und muss zuvor durch **fdOpen** geöffnet werden.

Funktion Datenspeicher[*Register]* = *Variable*

Beispiel
```
Move(LR0,0)
fdSave(LR0,Buffer512)
```

FLASHSPEICHER BEFEHLE

Optional verfügt der MegalockUSB-Dongle über einen FlashSpeicher. Dieser Datenspeicher hat eine Größe von max. 7936 Byte also fast 8 KByte. Der Zugriff erfolgt in 64 Byte Blöcken. Die Anzahl der Datenblöcke ist abhängig von der gewählten Größe des Toolbox-Speichers.

Beispiel 1:
Toolbox-Speicher = 256 Byte (256 Byte EEProm-Speicher).
8192 Byte – Toolbox-Speicher = 7936 Byte, ergibt 7936 / 64 = 124 Blöcke
Beispiel 2:
Toolbox-Speicher = 1024 Byte (256 Byte EEProm +768 Byte Flashspeicher).
8192 Byte – Toolbox-Speicher = 7168 Byte, ergibt 7168 / 64 = 112 Blöcke.

Der Hersteller garantiert 100.000 Lösch- und Schreiboperationen, also wenn Sie stündlich die Daten pro Block ändern hat der Speicher ein Lebenserwartung von mehr als 11 Jahren. Der Datenerhalt ist mit 100 Jahren angegeben.

Wenn Sie mehrmals stündlich die Daten ändern müssen sollten Sie diese im E²Prom-Speicher der Toolbox unterbringen. Hier garantiert der Hersteller 1.000.000 Lösch- und Schreiboperationen.

fLoad — FlashSpeicher lesen

Syntax **fLoad**(*Register*,*Variable*)

Beschreibung Überträgt einen Datenblock, 64 Byte, in die *Variable*. Der jeweilige Datenblock wird im *Register* angegeben. *Das Register* muss als Long-Register angegeben werden.

Funktion *Variable* = Datenspeicher[*Register]*

Beispiel
```
Move(LR0,0)
fLoad(LR0,Buffer64)
```

fSave — FlashSpeicher schreiben

Syntax **fSave**(*Register*,*Variable*)

Beschreibung Überträgt 64 Byte aus der Variablen in den Datenspeicher. Der jeweilige Datenblock wird im *Register* angegeben. *Das Register* muss als Long-Register angegeben werden. Die Schreibschutzsperre für den Toolbox-Speicher **„Memory“** gilt auch für diesen Speicher und kann im Dialog-Fenster **„Toolbox“-„Basis“** voreingestellt und mit dem Befehl **sef** temporäre abgeändert werden.

Funktion Datenspeicher[*Register] = Variable*

Beispiel
```
Move(LR0,0)
fSave(LR0,Buffer512)
```

AES128BIT BEFEHLE

Der MegalockUSB-Dongle verfügt AES-128Bit Cryptierung. Der Schlüssel wird zuvor in 4 aufeinander folgende Register beginnend mit LR0, LR4, LR8 oder LR12 geladen. Die Inhalte der Register werden während der Cryptierung nicht verändert. Die Länge des zu cryptierenden Datenbereich kann bis zu 65536 Byte groß sein. Die Crytierung erfolgt in 64 Byteblöcken ggf. muss der zu cryptierende Datenbereich entsprechend erweitert werden.

Encrypt	**Daten verschlüsseln**

Syntax **Encrypt**(*Register*,*Size*,V*ariable*)

Beschreibung Verschlüsselt den durch *Variable* angegeben Datenblock in der mit *Size* definierten länge.

Register Als Register kann R0,R4,R8 und R12 angegeben werden.

Funktion *Variable* = Variable

Beispiel
```
Move(LR0,0x12345678)
Move(LR1,0x12345678)
Move(LR2,0x12345678)
Move(LR3,0x12345678)
Encrypt(LR0,4096,Tabelle)
```

Decrypt	**Daten entschlüsseln**

Syntax **fDecrypt**(*Register*,*Size*,*Variable*)

Beschreibung Entschlüsselt den durch *Variable* angegeben Datenblock in der mit *Size* definierten länge.

Register Als Register kann R0,R4,R8 und R12 angegeben werden.

Funktion *Variable* = Variable

Beispiel
```
Move(LR0,0x12345678)
Move(LR1,0x12345678)
Move(LR2,0x12345678)
Move(LR3,0x12345678)
Decrypt(LR0,4096,Tabelle)
```

Megalock Teil IV

Anhang

SysTech
Entwicklungsges. für Mikroprozessorsysteme mbH

FEHLER- U. RETURNCODES

FEHLER- U. RETURNCODES DES MEGALOCK DONGLE

Innerhalb des Megalock Dongles können nachfolgende Fehler- bzw. Returncodes auftreten, die mit **move** innerhalb des Dongles ausgewertet werden können. Über die Megalock API werden ggf. aufgetretene Fehler in der Gruppe 4 (0x04..) angegeben.

Code	Name	Beschreibung
0x10	ML_DEFAULT_EEP	Die Prüfsumme über die Speicheraufteilung des Datenspeichers stimmt nicht überein. Die Defaulteinstellung wird aktiviert. Unwahrscheinlich oder Datenspeicher defekt.
0x11	ML_ERR_HWEEP1	Externer Datenspeicher defekt
0x12	ML_ERR_HWEEP2	Externer Datenspeicher defekt
0x13	ML_ERR_HWEEP3	Externer Datenspeicher defekt
0x14	ML_ERR_HWEEP4	Externer Datenspeicher defekt
0x15	ML_ERR_HWEEP5	Externer Datenspeicher defekt
0x20	ML_RESTART	Megalock Dongle wurde abgezogen und wieder aufgesteckt
0x21	ML_TIMEOUT	Timeout Fehler bei aktiver WATCHDOG Funktion
0x30	ML_OPCODE_ERR	Ungültiger Befehlscode
0x31	ML_EEPADR_ERR	Adresse auf Datentabelle größer als definiert
0x32	ML_WRPROTECT	Schreibversuch auf schreibgeschützte Datentabelle
0x33	ML_RDCOUNTCLR	Ungültiger Lesezugriff auf ein Register bei aktivem READCT. READCT = 0
0x34	ML_FD_NOTOPEN	FlashDrive wurde nicht geöffnet oder fdOpen() fehlerhaft
0x35	ML_FD_PW_ERR	Password stimmt nicht überein.

FEHLER- U. RETURNCODES DER MEGALOCK API

In den nachfolgenden Tabellen sind die möglichen Return- und Fehlercodes aufgeführt, wobei das Highbyte die Gruppe zuordnet und das Low Byte den Wert näher spezifiziert.

Returncodes der API bzw. des Megalock Dongles

Code	Name	Beschreibung
0x0000	ML_READY	Megalock Dongle mit angegebener Programmnummer vorhanden und freigeschaltet.
0x0100	NOKEYFOUND	Megalock Dongle mit angegebener Programmnummer nicht vorhanden bzw. nicht freigeschaltet
0xFFFF		Megalock Flag Abfrage, Bedingung erfüllt
0xFFFE		Megalock Flag Abfrage, Bedingung nicht erfüllt

Programmier- und Megalock Compilerfehler

Code	Name	Beschreibung
0x0201	ERRPGMNR	Programmnummer ist größer als 31, wahrscheinlich wurde Programmcode in EXE File überschrieben
0x0202	ERRNOTOPEN	Für die angegebene Programmnummer wurde kein Open durchgeführt
0x0203	ERRFUNCTION	ungültiger Funktionsparameter, Megalock Compilerfehler unwahrscheinlich, wahrscheinlich wurde Programmcode in EXE File überschrieben
0x0204	ERRRETDATA	Differenz Return Datacode zwischen Aufrufparameter und Megalock Dongle, Megalock Compilerfehler unwahrscheinlich, wahrscheinlich wurde Programmcode in EXE File überschrieben

Fehlercodes der Megalock API

Code	Name	Beschreibung
0x0301	IERRBREAK1	Kommunikation zwischen Megalock API und Megalock Dongle abgebrochen. Ursache: Genereller Hardwaredefekt, PC-Port oder Megalock Dongle, wahrscheinlicher: Dongle wurde während des Zugriffes abgezogen
0x0302	IERRBREAK2	
0x0303	IERRBREAK3	
0x0304	IERRBREAK4	
0x0305	IERRBREAK5	
0x0306	IERRBREAK6	
0x0307	IERRGETDATA	
0x0308	IERRRETCOM	
0x0309	IERRRETLEN	
0x030A	IERRRUNTOUT	

Fehlercodes des Megalock Dongles

Code	Name	Beschreibung
0x0410	ML_DEFAULT_EEP	Die Prüfsumme über die Speicheraufteilung des Datenspeichers stimmt nicht überein. Die Defaulteinstellung wird aktiviert. Unwahrscheinlich oder externer Datenspeicher defekt.
0x0411	ML_ERR_HWEEP1	Externer Datenspeicher defekt, unwahrscheinlich
0x0412	ML_ERR_HWEEP2	Externer Datenspeicher defekt, unwahrscheinlich
0x0413	ML_ERR_HWEEP3	Externer Datenspeicher defekt, unwahrscheinlich
0x0414	ML_ERR_HWEEP4	Externer Datenspeicher defekt, unwahrscheinlich
0x0415	ML_ERR_HWEEP5	Externer Datenspeicher defekt, unwahrscheinlich
0x0420	ML_RESTART	Megalock Dongle wurde abgezogen und wieder aufgesteckt
0x0421	ML_TIMEOUT	Timeout Fehler bei aktiver WATCHDOG Funktion
0x0430	ML_OPCODE_ERR	Ungültiger Befehlscode
0x0431	ML_EEPADR_ERR	Adresse auf Datentabelle größer als definiert
0x0432	ML_WRPROTECT	Schreibversuch auf schreibgeschützte Datentabelle
0x0433	ML_RDCOUNTCLR	Ungültiger Lesezugriff auf ein Register bei aktivem READCT. READCT = 0
0x0434	ML_FD_NOTOPEN	FlashDrive wurde nicht geöffnet oder fdOpen() fehlerhaft
0x0435	ML_FD_PW_ERR	Password stimmt nicht überein.

STEUERDATEIEN

Die Steuerdateien werden benötigt, um Megalock Befehle in ein für Ihren Compiler (PASCAL, C, etc.) sprachspezifisches Format umzusetzen, sodass Ihr Compiler die Megalock Befehle als externe Unterprogrammaufrufe akzeptiert. Die Steuerdateien sind im Ordner Megalock\Language untergebracht. Die Namensgebung der Dateien entsprechen der Bezeichnung bzw. Kürzel Ihres Compilers oder Programmiersprache und die Extension lautet *.MLL. Die jeweilige Steuerdatei für Ihren Compiler sind unter Compilereinstellungen auszuwählen, also Mauszeiger auf den Ordner **„Compiler“** und drücken die rechte Maustaste, wählen dann im erscheinenden Menü **„Einstellungen**“ und dann die gewünschte Steuerdatei.

Sollten Sie dort die von Ihnen eingesetzte Programmiersprache nicht vorhanden sein, setzen Sie sich bitte mit uns in Verbindung!
Änderungen an den vorhandenen Steuerdateien sollten nicht vorgenommen werden. Kopieren Sie ggf. die zu bearbeitende Steuerdatei unter einen anderen Namen. Die neue MLL-Datei wird dann automatisch unter „Language Controlfile“ aufgelistet. In der ersten Zeile der Datei (//NAME) sollte eine zusätzliche Kennzeichnung erfolgen.

Nachfolgend sehen Sie den Aufbau einer Steuerdatei für einen C-Compiler:

Anweisung	Parameter	Beschreibung
SML_SINGLE	“ML..“	Kennzeichnung Einzelbefehle
SML_MARKER	“’##“	Kennzeichnung für DEF-, OPEN-, CLOSE-, BEGIN- und END- Anweisung
SML_COMMENT_ON	“/*“	Kommentaranfang
SML_COMMENT_OFF	“*/“	Kommentarende
SML_EXTENSION1	“ccc,c“	Extension Input, Output
SML_EXTENSION2	“cml,c“	Extension Input, Output
SML_EXTENSION3	“sml,c“	Extension Input, Output
SML_COMPILE	“:()“	Kennzeichnung für nicht Megalock Befehle
SML_VARPOINTER	“@“	Kennzeichen für Pointervariable
SML_VARWORD	“%“	Kennzeichen für Wordvariable
SML_VARLONG	“#“	Kennzeichen für Longvariable
SML_VARSTRING	“$“	Kennzeichen für Stringvariable

Nachfolgend die Syntaxbeschreibung für den Aufbau der unterschiedlichen Megalock Befehlszeilen bzw. die Anweisungen für die Darstellung der einzelnen Parameter.

Anweisung	Syntax	Beschreibung
CSP_ON	“MEGALOCK%d(“	Funktionsname
CSP_SET	“=“	Zuweisungszeichen
CSP_OFF	“)“	Ende der Parameterliste
CSP_ENDBEFEHL	“;“	Befehlsende
CSP_TRENN	“,“	Parameter Trennzeichen
CSP_CONSTWORD	"0x%04x"	Wordkonstante
CSP_CONSTLONG	"0x%08lxL"	Longkonstante
CSP_VARWORD	“%s“	Variable, Word
CSP_VARWORDPTR	“*%s“	Variable, Wordpointer
CSP_VARLONG	“%s“	Variable, Long
CSP_VARLONGPTR	“*%s“	Variable, Longpointer
CSP_PTRWORD	“&%s“	Adresse, Wordvariable
CSP_PTRWORDPTR	“%s“	Adresse, Wordpointer
CSP_PTRLONG	“&%s“	Adresse, Longvariable
CSP_PTRLONGPTR	“%s“	Adresse, Longpointer
CSP_PTRSTRING	“char *)%s“	Adresse, Stringvariable
CSP_PTRSTRINGPTR	“%s“	Adresse, Stringpointer
CSP_COMPILE	“%s“	nicht Megalock Anweisung

Nachfolgend die Syntaxbeschreibung für if-Abfragen und Schleifen.

Anweisung	Syntax	Beschreibung
CSP_COMMODE	“C“	Compiler Steuerkennzeichen
CSP_IF	“if (%s == -1)“	if- Abfrage
CSP_IFBEGIN	“{“	Beginn WAHR Zweig
CSP_IFENDBEGIN	“}“	Ende WAHR Zweig
CSP_ELSE	“}“ else	Ende WAHR Zweig, ELSE
CSP_ELSEBEGIN	“{“	Beginn ELSE Zweig
CSP_ELSEENDBEGIN	“}“	Ende ELSE Zweig

CSP_LABEL	“%s:“	Labeldarstellung
CSP_IFGOTO	“if (%s==-1) goto %s;“	if Abfrage mit Sprungbefehl
CSP_GOTO	“goto %s;“	Sprungbefehl
CSP_LOOP	“for (;;)“	Endlosschleife
CSP_LOOPBEGIN	“{“	Beginn Endlosschleife
CSP_LOOPEND	“}“	Ende Endlosschleife
CSP_LOOPBREAK	“if (%s != -1) break;“	Abbruch für Endlosschleife

Die Def_.. Anweisungen legen die Erzeugung der externen Referenzen für die Unterprogrammaufrufe der Megalock API fest.

```
DEF_MLMODE      "1"
DEF_MLCALL      ".16BIT=far.32BIT=_stdcall"
DEF_MLDATA      ".16BIT=far.32BIT=far"

DEF_MLCI1       "void  %s MEGALOCK1(ulong OpCode);"

DEF_MLCI11      "short %s MEGALOCK11(ulong OpCode,ulong ldat1);"
```

Die DEF_MLCI1..11 Anweisungen stellen die Grundreferenzen der Unterprogramme dar. Diese werden je nach Aufrufmodus (16 / 32 Bit) und Parameterübergabe erweitert. Die Beschreibung der Erweiterungen ist in DEF_MLCALL und DEF_MLDATA angegeben. Die Anweisung DEF_MLMODE ist ein weiterer Steuermechanismus, derzeit ist nur die“1“ gültig.

Die Programmzeilen hinter DEF_MACRO bis zum Ende der Datei werden jeweils durch die Anweisung ##DEF_MEGALOCK_MACRO in Ihren Quellcode eingebunden.

```
DEF_MACRO
#define ushort unsigned short
#define ulong  unsigned long
```

REGISTRY-EINTRÄGE

Wird das Megalock Programm Setup.exe mit der Option „INSTALL“ ausgeführt, erstellt dieses die Grundeinträge in der Registry und nach erster Inbetriebnahme des MegalockUSB-Dongle werden die Einträge in der Registry ergänzt. Nachfolgend eine Übersicht der Einträge, für den MegalockUSB-Donge, in der Registry.

Betriebsystem: Win2000, WinXP

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class\
- **{E5338F04-7930-45BA-9CFB-0071EE9BE005}**
  - **0000**
  - **0001**
  - Die Unterordner „0000“ und folgende verweisen Unteranderem auf die OEM??.INF-Datei.

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB\
- **Vid_04d8&Pid_ffb8**
  - **6&1b5b324b&2&2 (Beispiel)**
  - Eintrag für USB-Verbundgerät
- **Vid_04d8&Pid_ffb8&Mi_00**
  - **7&3374173&10&0 (Beispiel)**
  - Eintrag für MegalockUSB-Dongle Interface 0
- **Vid_04d8&Pid_ffb8&Mi_01**
  - **7&3374173&10&1 (Beispiel)**
  - Eintrag für MegalockUSB-Dongle Interface 1
- **Vid_04d8&Pid_ffb8&Mi_02**
  - **7&3374173&10&2 (Beispiel)**
  - Eintrag für MegalockUSB-Dongle Interface 2
  - (Nur wenn Dongle mit Massenspeicher ausgestattet ist)

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USBSTOR
- **Disk&Ven_MegaLock&Prod_ProtectFDx710200&Rev_1.10**
  - **8&268a3c99&0 (Beispiel)**
  - Eintrag für Datenträger
  - (Nur wenn Dongle mit Massenspeicher ausgestattet ist)
  - Die Kennzeichnung FDx710200 bzw. 710200 wäre dann ihre 6stellige Keynummer. Dies Keynummer ist identisch mit ihrer MLUCxxxxxx.DLL im Megalock-Ordner.
  - Für die Demoversion wäre dies die 710200

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\
- **MLOCKUSB**
  - **Control**
  - **Security**
  - Unter dem Ordner “Control” ist dann auch wieder Ihre KeyNummer in Form MLx710200 (Demoversion) eingetragen. An den Einträgen im Ordner „Control“ kann man dann auch erkenne ob weitere MegalockUSB-Dongle mit anderen KeyNummer auf dem Rechner installiert wurden.

Alle in der zuvor aufgeführten Tabelle **Fett** dargestellten Ordner können wenn notig gelöscht werden, ggf. sind zuvor „Berechtigungen“ für den Löschvorgang zu vergeben.

Es ist aber sinnvoller die Einträge mit UNINSTALL zu entfernen. Zwischen der UNINSTALL-Funktion und erneuter INSTALL-Funktion ist zwingend ein Neustart des Rechners notwendig.

DIE HARDWARE

MEGALOCK PARALLEL DONGLE

Der Megalock Parallel-Dongle wird auf die Parallel-Schnittstelle gesteckt. Da alle Leitungen des Parallel-Ports durchgeschleift sind, kann ein Drucker der Buchse des Megalock Parallel-Dongle betrieben werden.

Technischen Daten:

Security MC	20 x 32-Bit Register + 4 x 32-Bit Systemregister 16-Bit Opcode kundenspezifisch kodiert, 2^{24} kodierte Befehlssätze.
Datenspeicher	512 Byte EEProm **//** Optional 2KB - bzw. 8 KB E³Prom Programmierzyklen > 1.000.000 Datenerhalt > 40 Jahre
Geschwindigkeit	Megalock Basic Befehle 1000 - 3000 Befehle pro Sekunde Toolbox-Befehle - Read 500-800 Befehle - Write 200-400 Befehle pro Sek. **Abhängig von der Taktfrequenz des PCs**
Dongle-Adressen	2^{18} Adressen wird von SysTech je Kunde einmal vergeben, plus 32 Adressen pro Kundenserie durch den Kunden zu vergeben.
Betriebsspannung	min. 1.98 Volt
Benutzte Signalleitungen	D0 .. D7,STROBE,AUTO FEED, INIT, SLCT IN
Rückgeführte Signalleitungen	ACK,BUSY,PE, FAN OUT 10LSTTL
max. Anzahl in Reihe	theoretisch beliebig
Temperaturbereich	0 ° bis +70 ° C **// Optional -40 ° bis +85 °C //**
Gehäuseabmessung	ca. 62 x 55 x 16 mm
Steckverbindung	standardmäßig Centronicsport - DB25
Gewicht	ca. 40 g
Technische Änderungen ohne Vorankündigung möglich.	

Artikelnummer	Artikelbezeichnung
120140-512	Megalock-VTP, kundenspezifisch, 512Byte E²Prom, Parallel
120150-2048	Megalock-VTP, kundenspezifisch, 2KByte E²Prom, Parallel
120160-4096	Megalock-VTP, kundenspezifisch, 4KByte E²Prom, Parallel

MEGALOCK USB DONGLE

Der Megalock USB-Dongle wird an der USB-Schnittstelle betrieben, die Datenleitungen sind hier aber nicht durchgeschleift. Der Megalock USB-Dongle kann direkt am Rechner oder über USB-HUB betrieben werden.

Technischen Daten:

Security MC	20 x 32-Bit Register + 4 x 32-Bit Systemregister 16-Bit Opcode kundenspezifisch kodiert, 2^{24} kodierte Befehlssätze.
Datenspeicher	256 Byte EEProm **Programmierzyklen > 1.000.000 Datenerhalt > 40 Jahre** Optional 7936 Byte FlashSpeicher **Programmierzyklen >100.000 Datenerhalt >40 Jahre** 2MB - 64MB FlashSpeicher **Programmierzyklen >100.000 Datenerhalt>20 Jahre**
Geschwindigkeit	Megalock Basic Befehle 400 - 3000 Befehle pro Sekunde Toolbox-Befehle Read+Write 400 - 3000 Befehle pro Sek. **Abhängig von der Taktfrequenz des PCs**
Dongle-Adressen	2^{18} Adressen wird von SysTech je Kunde einmal vergeben, plus 32 Adressen pro Kundenserie durch den Kunden zu vergeben.
Betriebsspannung	min. 3.00 Volt
Benutzte Signalleitungen	D+, D-, VCC, GND
Temperaturbereich	0 ° bis +70 ° C **// Optional -40 ° bis +85 °C //**
Gehäuseabmessung	ca. 62 x 25 x 13 mm (mit Schutzkappe) ca. 58 x 25 x 11 mm (ohne Schutzkappe)
Steckverbindung	standardmäßig USB-Port
Gewicht	ca.12 g
Technische Änderungen ohne Vorankündigung möglich.	

Datenspeicher:
Die 256-Byte-E2Promspeicher und der kleine Flashspeicher von 7936 Byte sind auf dem Mikrocontrollerchip vorhanden und so mit nicht direkt zugängig.
Der Massenspeicher von 2MByte bis zurzeit von 64MByte ist als eigenständiger Baustein auf der Leiterplatte untergebracht, die Daten sind auf dem externen Speicher cyptiert gespeichert, wahlweise AES128 oder einfache XOR-Cryptierung. Der Massenspeicher kann, wahlweise, als Datenlaufwerk (USB-DatenStick) angesprochen werden.

Der Unterschied zwischen E^2Prom- und Flash-Speicher liegt in den vom Hersteller garantierten Änderungen der Daten. So kann der Flashspeicher bis zu 100.000 und der E^2Prom-Speicher bis zu 1.000.000 mal geändert werden.

Geschwindigkeit:
Wird der Megalock USB-Dongle über einen USB-HUB betrieben, können mehr Befehle pro Sekunde abgearbeitet werden als wenn der Dongle direkt am PC aufgesteckt ist.

Artikelnummer	Artikelbezeichnung
120230-256	Megalock-VTU, kundenspezifisch, 256Byte E²Prom, USB
120240-8196	Megalock-VTU, kundenspezifisch, 8KByte Datenspeicher, USB
120250-2MB	Megalock-VTU, kundenspezifisch, 8KByte Datenspeicher, 2MB FlashDrive,USB
120260-8MB	Megalock-VTU, kundenspezifisch, 8KByte Datenspeicher, 8MB FlashDrive,USB
	Versionen mit mehr FlashDrive-Speicher auf Anfrage.

MEGALOCK USB <-> PARALLEL

Die Befehle und Funktionen des Megalock USB-Dongle sind identisch, es gibt jedoch 2 technisch bedingte Änderungen die, wenn Sie beide Versionen einsetzen möchten, zu berücksichtigen sind.

Über den Befehl **ldv** können Sie die Dongle-Version feststellen. Ist die Versionnummer 1 dann handelt es sich um einen Parallel-Dongle, ist die Versionnummer 2 ist es USB-Dongle.

Die erste Änderung betrifft die TIMER-Funktion bzw. die Befehle **tmr/tms** bis lang wurde alle 8us der Counter um 1 erhöht, dies geschieht jetzt, bei dem USB-Dongle alle 20us. Es spielt aber keine Rolle ob ein Debugger schon nach ½ Sekunde oder erst nach 1 ¼ Sekunde erkannt wird. Somit kann für beide Versionen der Maximalwert 0xffff eingesetzt werden. Ansonsten muss, nach Abfrage **ldv**, der Zeitwert durch 2,5 geteilt werden.

Die zweite Änderung ist schon etwas gravierender und betrifft den Toolbox-Speicher.
Der Parallel-Dongle hat extern 512(480)Byte E^2Promspeicher und der USB-Dongle hat intern 256 Byte E^2Promspeicher beide erlauben laut Datenblatt bis zu 1.000.000 Änderungen. In der 8 KByte Version hat der USB-Dongle zusätzlich noch 7936 Byte Flashspeicher, der erlaubt aber nur 100.000 Änderungen. Der Flashspeicher lässt ich mit dem ML.EXE-Programm genauso einstellen bzw. aufteilen als wäre es der Parallel-Dongle mit 512 Byte Speicher.
Es gibt nur ein Problem wenn Sie über die ersten 256 Byte hinaus, sehr häufig die Daten ändern. 100.000 Änderungen sind zwar schon eine Menge aber in dem Flashspeicher können die Daten nur in 64-Byte-Böcken geändert werden. Wenn Sie im ungünstigsten Fall eine Word-Tabelle von 32 Einträgen haben und Sie diese Tabelle immer komplett ändern, wären das gleich 32 Änderungen. Somit 100.000 /32 = 3125 Änderungen, wenn Sie die Tabelle 1mal am Tag ändern wären das zwar noch kein Problem, dann wäre die Lebenserwartung 3125/365 = 8,5 Jahre. Aber wenn diese Tabelle mehrmals am Tag ändern, dann sinkt die Lebenserwartung des Dongle doch erheblich.

Wenn sie den Toolbox-Speicher so aufteilen das die Datenfelder, die mehrmals täglich geändert werden, in den ersten 256 Byte unterbringen sollte eine Programmentwicklung ohne Berücksichtigung ob ein Parallel- und USB-Dongle eingesetzt wird, möglich sein.

Die MEGALOCK.DLL sucht zuerst nach einen passenden USB-Dongle, wird kein passender Dongle auf den USB-Port gefunden, wird geprüft ob die MEGALOCK.SYS auf dem System installiert ist, dann wird geprüft ob ein passender Parallel-Dongle vorhanden ist.

MEGALOCK CD UND ORDNER

Nachfolgend eine Übersicht der Ordner und Dateien auf der Megalock-CD.Die **Fett** dargestellten Dateien werden für die Installation und den Zugriff auf den Megalock-Dongle benötigt und sind daher auch bei Ihren Kunden zu installieren. Im Anschluss finden Sie eine Übersicht wie und für welches Betriebssystem die Dateien benötigt werden und in welchem Ordner diese kopiert werden.

Ab Megalock USB-Dongle Version 3.x wird nur noch die Megalock.DLL benötigt. Die Megalock.DLL in den Windows-Ordner System32 oder in dem aktuelle Arbeitsordner kopieren, fertig.

Systemdateien	**<= Win98**	**>= Win2000**
MLockUSB.sys **(nur USB-Dongle V2.x)**	%root%\system32\drivers	%root%\system32\drivers
WDMStub.sys **(nur USB-Dongle V2.x)**	%root%\system32\drivers	
Megalock.sys (nur Parallel-Dongle)	%root%\system32\drivers	%root%\system32\drivers
Megalock.vxd (nur Parallel-Dongle)	%root%\system32\drivers	
Megalock.dll	%root%\system32	%root%\system32

Installationdateien	**<= Win98**	**>= Win2000**
MLockUSB.inf **(nur USB-Dongle V2.x)**	%root%\inf\OEM??.inf	%root%\inf\OEM??.inf
MLockUSB.cat **(nur USB-Dongle V2.x)**		

Die MLUC710200.DLL ist eine kundenspezifische DLL, die zu Test und Demozwecken erstellt wurde. Sie als Kunde erhalten eine eigene MLUCxxxxxx.DLL und einen dazu passenden Megalock-Master Dongle mit der nur Sie ihre Megalock-Dongle bearbeiten können. Es sollte in Ihrem Interesse liegen, das Sie diese DLL und den Master-Dongle nur ausgewählten Person zur Verfügung stellen

Die unterstützen Programmiersprachen werden ständig erweitert und die hier angegeben Inhalte der Ordner **Examples** und **Language** stellen nur eine kleine Auswahl dar. Sollte keine Steuerdatei oder kein Beispiel für die von ihre eingesetzte Programmiersprache auf der CD vorhanden sein, setzten Sie ich bitte mit uns in Verbindung.

Übersicht der Megalock-CD bzw. des Megalock Ordners

\Megalock		**Kurzbeschreibung**
	ML.exe	Megalock-Programm
	ML.HLP	Megalock-Programm Helptext
	ML.ini	Megalock-Programm Parameter und Einstellungen
	MEGALOCK.DLL	API,ermöglicht Zugriff auf Parallel Dongle, USB V2.x und V3.x
	MLUC710200.dll	Kundenspezifische DLL für die Megalock-Demo-Serie
	Setup.exe	Installationsprogramm
	DRIVER	Ordner mit Systemdateien (USB+Parallel)
	DRIVER\MEGALOCK.DLL	API,ermöglicht Zugriff auf Parallel Dongle, USB V2.x und V3.x
	DRIVER\MEGALOCK.lib	Informationen zur API für ihren Compiler/Linker
	DRIVER\MEGALOCK.sys	Systemtreiber **nur für Parallel-Dongle** ab Win98
	DRIVER\MEGALOCK.vxd	Systemtreiber **nur für Parallel-Dongle** bis WIN95
	DRIVER\MLockUSB.cat	USB Catalog-Datei wird zur Installation benötigt **(NUR VERSION 2.X)**
	DRIVER\MLockUSB.inf	INF-Datei, wird zur Installation benötigt. **(NUR VERSION 2.X)**
	DRIVER\MLockUSB.sys	Systemtreiber für USB-Dongle ab WIN98. **(NUR VERSION 2.X)**
	DRIVER\WDMStub.sys	Systemtreiber für USB-Dongle (nur für WIN98). **(NUR VERSION 2.X)**
	DRIVER\Megalock_PAR	Ordner mit Dateien nur für Parallel-Dongle und DOS
	EXAMPLES	Ordner mit Beispiel
	EXAMPLES\access70	MS ACCESS 7.0
	EXAMPLES\BC	Borland C
	EXAMPLES\BPASCAL	Borland Pascal
	EXAMPLES\clipper	Clipper
	EXAMPLES\DELPHI40	Delphi 4.0
	EXAMPLES\FoxPro30	FoxPro 3.0

	EXAMPLES\MSVC60	MS Visual C 6.0
	EXAMPLES\vbasic50	MS Visual Basic 5.0
	EXAMPLES\WATCOM_C	Watcom C
	EXAMPLES\...	Weitere Beispiele
	HANDBUCH	Ordner mit Dokumenation
	HANDBUCH\MLGesamt.pdf	Megalock Handbuch (diese Datei)
	INCLUDE	Ordner für Header-Dateien (.MLH)
	INCLUDE\MEGALOCK.MLH	Header-Datei für den Megalock-Compiler
	LANGUAGE	Ordner für Steuerdateien (.MLL)
	LANGUAGE\ACCESS.MLL	Steuerdatei für MS ACCESS
	LANGUAGE\C.MLL	Steuerdatei für C-Compiler
	LANGUAGE\CLIPPER.MLL	Steuerdatei für Clipper-Compiler
	LANGUAGE\CPP.MLL	Steuerdatei für C++-Compiler
	LANGUAGE\DELPHI.MLL	Steuerdatei für Delphi-Compiler
	LANGUAGE\FORTRAN9.MLL	Steuerdatei für Fortran9-Compiler
	LANGUAGE\FOXPRO.MLL	Steuerdatei für FoxPro-Compiler
	LANGUAGE\MODULA.MLL	Steuerdatei für Modula-Compiler
	LANGUAGE\PARADOX.MLL	Steuerdatei für Paradox-Compiler
	LANGUAGE\PASCAL.MLL	Steuerdatei für Pascal-Compiler
	LANGUAGE\QBASIC.MLL	Steuerdatei für QuickBasic
	LANGUAGE\VBASIC30.MLL	Steuerdatei für Visual Basic 3.0
	LANGUAGE\VBASIC40.mll	Steuerdatei für Visual Basic 4.0
	LANGUAGE\...	Weitere Steuerdateien
	PROJECT	Ordner für Beispiel-Projekt
	PROJECT\beispiel.mlp	Projekt-Date für Megalock-Compiler
	PROJECT\DEMOPRG.CPP	Outdatei bzw. Inputdatei für Ihren Compiler
	PROJECT\DEMOPRG.SML	Inputdatei für den Megalock-Compiler
	PROJECT\Memory.mle	Kopie des Datenspeicher

Megalock Teil V

Stichwortverzeichnis

SysTech

Entwicklungsges. für Mikroprozessorsysteme mbH

STICHWORTVERZEICHNIS

A

B

D

E

F

I

K

L

M

O

P

R

S

T

V

W