

**eCos Portierung**  
unter besonderer Berücksichtigung der ARM-Architektur

Andreas Bürgel<sup>1</sup>

<http://www.andreas-buergel.de>

7. April 2003

<sup>1</sup>Mail: [andreas@andreas-buergel.de](mailto:andreas@andreas-buergel.de)

Version \$Id: Portierung.tex,v 1.1 2003/04/07 20:31:48 andreas Exp \$

eCos<sup>TM</sup> ist ein eingetragenes Markenzeichen von Red Hat, Inc.

RedHat<sup>®</sup>, RedBoot<sup>TM</sup>, GNUPro<sup>®</sup>, Cygwin<sup>TM</sup> sind eingetragene Markenzeichen von Red Hat, Inc.

Linux<sup>®</sup> ist ein eingetragenes Markenzeichen von Linus Torvalds

UNIX<sup>®</sup> ist ein eingetragenes Markenzeichen von The Open Group

Microsoft<sup>®</sup>, Windows<sup>®</sup>, Windows NT<sup>®</sup>, Windows 95<sup>®</sup>, Windows 98<sup>®</sup>, Windows 2000<sup>®</sup>, Windows XP<sup>®</sup>  
sind eingetragene Markenzeichen der Microsoft Corporation

GenoLogic<sup>®</sup> ist ein eingetragenes Markenzeichen der GenoLogic GmbH

ARM<sup>®</sup> ist ein eingetragenes Markenzeichen von ARM Ltd.

MIPS<sup>®</sup> ist ein eingetragenes Markenzeichen von MIPS, Inc.

Intel<sup>®</sup>, StrongARM<sup>®</sup> sind eingetragene Markenzeichen der Intel Corporation

ATMEL<sup>®</sup> und AT91<sup>®</sup> sind eingetragene Markenzeichen der Atmel Corporation

Alchemy Semiconductor<sup>®</sup>, Au1000<sup>®</sup>, Au1500<sup>®</sup> sind eingetragene Markenzeichen von Alchemy Semiconductor Inc.

HyperStone<sup>®</sup> ist ein eingetragenes Markenzeichen der HyperStone AG

Alle in dieser Aufzählung nicht genannten Markenzeichen sind ebenfalls eingetragene Markenzeichen ihrer Besitzer. Der Autor will sich diese auf keinen Fall zu eigen machen.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Über dieses Dokument . . . . .	1
1.2	Begriffsklärung . . . . .	1
<b>2</b>	<b>Der eCos-Boot-Vorgang</b>	<b>3</b>
<b>3</b>	<b>Portierung</b>	<b>5</b>
3.1	Verzeichnisstruktur eines Ports . . . . .	5
3.2	Plattform-Port . . . . .	6
3.2.1	hal_platform_setup.h . . . . .	6
3.2.2	<PLATFORM>_misc.c . . . . .	7
3.2.3	Speicher-Layout . . . . .	11
3.2.4	Einen „Glue“-Treiber für den Plattform Flash-Speicher schreiben . . . . .	13
3.2.5	Die CDL-Datei . . . . .	13
3.2.6	Einfügen von Paketen in die Datenbank-Datei ecos.db . . . . .	18
3.3	Varianten-Port . . . . .	19
3.4	Architektur-Port . . . . .	19
<b>A</b>	<b>Probleme und Lösungen</b>	<b>21</b>
A.1	Es wird nicht zum Beginn von PLATFORM_SETUP1 gesprungen (ARM) . . . . .	21



# Kapitel 1

## Einleitung

### 1.1 Über dieses Dokument

Dieses Dokument ist ein Fragment. Es erhebt z.Z. keinen Anspruch auf Vollständigkeit und Richtigkeit. Der Autor übernimmt keinerlei Haftung für Schäden, die durch Anwendung der in diesem Dokument enthaltenen Informationen entstanden sind.

Sollte ein Leser dieses Dokuments auf die glorreiche Idee kommen, die Steuerungs-Software für die Speisewasserpumpen des Primär-Kreislaufs eines Atomkraftwerks umzuschreiben und damit ein mehr oder weniger zivilisiertes Land von der Weltkarte entfernen, so ist das nicht die Schuld des Autors. Der Autor übernimmt auf keinen Fall irgendwelche Haftung.

Dieses Dokument darf frei benutzt und weiter verbreitet werden. Die kommerzielle Weiterverbreitung ist jedoch untersagt. D.h. man darf mit Hilfe der Informationen in diesem Dokument Produkte entwickeln und diese verkaufen ohne für das Dokument zu bezahlen. Es ist jedoch nicht gestattet für die Weiterverbreitung dieses Dokuments Geld zu verlangen. Ebenso ist die Veränderung des Dokuments sowie die Verwendung des Dokuments in eigenen Werken, dies betrifft auch Auszüge des Dokuments, nur mit schriftlicher Einwilligung des Autors gestattet.

Der Autor behält sich vor die Nutzungsbedingungen ohne vorherige Bekanntmachung jederzeit zu ändern.

### 1.2 Begriffsklärung

Als *binär* oder als *Binärdatei* wird hier alles verstanden, was Produkt eines Übersetzer, d.h. eines Assemblers oder eines Compilers, ist. *Executable* ist eine ausführbare Applikation für eine bestimmte Hardware und letztlich auch eine Binär-Datei. Eine *Objekt-Datei* ist Ergebnis des Übersetzungs-Vorgangs einer einzelnen Quell-Datei und damit ebenfalls binär. Das, was auf .o endet, ist meist eine Objekt-Datei. Eine *Bibliothek* ist eine Zusammenfassung mehrerer logisch zusammen gehörender Objekt-Dateien.

Als *Host* wird der Rechner bezeichnet, auf dem die Entwicklung stattfindet bzw. auf dem der Debugger läuft. Als *Target* oder *Ziel-Hardware* wird der Rechner bezeichnet, für den die Applikation entwickelt wird. Als *eCos-Repository* (< ECOS\_REPOSITORY >) wird der Verzeichnisbaum bezeichnet in dem die Quell-Dateien von eCos selbst, als da wären Kernel-, HAL- und Geräte-Treiber-Quellen, liegen. Als *(eCos-)Build-Verzeichnis* (<ECOS\_BUILD>) wird das Verzeichnis bezeichnet in dem die Konfigurationsdatei ecos.ecc und der Verzeichnisbaum der Header- und Binär-Dateien der eCos-Laufzeit-Bibliothek bzw. eines der auf eCos basierenden ROM-Monitore liegen.

Ein in spitze Klammern gefasster <BEGRIFF> ist ein Platzhalter, z.B. für einen Namen oder einen Programm-Parameter. <...> ist ein Platzhalter für beliebigen Text bzw. ein Auslassungssymbol.

Alle Dateinamen und Benutzer-Eingaben werden serifenlos gedruckt. In Schreibmaschinenschrift werden Programm- und Skript-Dateien, sowie Funktions-Namen und teilweise auch Ausgaben von Programmen gedruckt.

Als *Port* wird hier das Ergebnis der Portierungs-Bemühungen verstanden und nicht eine I/O-Schnittstelle der Hardware.



## Kapitel 2

# Der eCos-Boot-Vorgang

Die wichtigste Datei des Boot-Vorgangs ist `vectors.S`. Sie steuert die gesamte System-Initialisierung, wobei der konkrete Initialisierungs-Code hauptsächlich in anderen Modulen platziert ist, da alle CPUs einer Architektur die selbe `vectors.S` Datei benutzen. D.h. es muss für jede unterstützte Architektur eine spezielle Version von `vectors.S` geben.

Der Boot-Vorgang läuft in chronologischer Reihenfolge im Einzelnen in den folgenden Dateien ab<sup>1</sup>:

1. `vectors.S`:  
Sprung zum Hardware-Initialisierungs-Code an Label `reset_vector`. Der Initialisierungs-Code befindet sich als C-Makro in der Datei `hal_platform_setup.h` im Makro `PLATFORM_SETUP1`.
2. `hal_platform_setup.h`:  
Hardware-Initialisierung und Aufruf der MMU-Initialisierung `<PORT_NAME>.misc.c::hal_mmu_init` (falls die CPU eine MMU hat)
3. `<PLATFORM>.misc.c`:  
In diesem Modul ist, bei CPUs mit MMU, eine Funktion implementiert, die die MMU auf das endgültige Speicher-Layout programmiert.
4. `vectors.S`:  
Nun wird diverser, nur von der CPU-Architektur abhängiger Initialisierungs-Code ausgeführt. Was ausgeführt wird ist teilweise vom Startup-Typ bzw. davon abhängig ob eine Applikation oder ein ROM-Monitor startet. Während dieser Initialisierungs-Phase wird das Makro `LED` mit einer Zahl als Parameter aufgerufen. Zum ersten Mal geschieht dies direkt nach `PLATFORM_SETUP1` mit dem Parameter 7. Bei jedem folgenden Aufruf wird der Parameter um eins subtrahiert, sodass man einen Absturz während der Initialisierung eingrenzen kann, wenn die LED-Ausgabe Zahlen eindeutig darstellen kann. Sinnvoll ist hierbei natürlich die Benutzung einer evtl. vorhandenen Sieben-Segment-Anzeige. Nun wird die Funktion `<PLATFORM>.misc.c::hal_hardware_init` aufgerufen, die weitere Initialisierungs-Schritte ausführt.
5. `<PLATFORM>.misc.c`:  
Hier sind einige Plattform-spezifische Initialisierungs-Funktionen implementiert. Die Funktion `hal_hardware_init` initialisiert u.a. den Interrupt-Controller, startet den System-Timer, schaltet die Caches ein und ruft `hal_if.c::hal_if_init` auf.
6. `hal_if.c`:  
In diesem Modul werden verschiedene „Dienste“ der Betriebssystem-Infrastruktur implementiert. `hal_if_init` initialisiert diese Dienste und trägt sie in eine Datenstruktur ein.
7. `vectors.S`:  
Wird ein ROM-Monitor bzw. ein GDB-Stub gebaut so wird die Funktion `generic_stub.c::initialize_stub` aufgerufen.

---

<sup>1</sup>Die Schreibweise `<DATEI>::FUNKTION` referenziert die Funktion `<FUNKTION>` in der Datei `<DATEI>`.

8. `generic-stub.c`:  
Dieses Modul enthält die Funktionalität für Remote-Debugging, d.h. für die Kommunikation des Targets mit dem Debugger auf dem Host.
9. `vectors.S`:  
Ist die Unterstützung für Ctrl-C Break-Unterstützung für den Debug-Betrieb aktiviert, so wird die Funktion `hal_if.c::hal_ctrlc_isr_init` aufgerufen.
10. `halJf.c`:  
`hal_ctrlc_isr_init` aktiviert einen Interrupt-Handler, der die Applikation beim Empfang eines Break-Signals stoppt.
11. `vectors.S`:  
Nun werden die Konstruktoren aller statischen C++-Objekte aufgerufen. Dazu wird die Funktion `hal_misc.c::cyg_hal_invoke_constructors` aufgerufen.
12. `hal_misc.c`:  
Die Funktion `cyg_hal_invoke_constructors` ruft alle Konstruktoren in der Konstruktor-Tabelle, diese ist durch die vom Linker exportierten Labels `_CTOR_LIST_` und `_CTOR_END_` markiert.
13. `vectors.S`:  
Abhängig davon ob man einen ROM-Monitor oder eine Applikation baut, wird die Funktion `stubrom.c::cyg_start` bzw. die Funktion `startup.cxx::cyg_start` aufgerufen.
14. `stubrom.c`:  
Die Funktion `cyg_start` enthält lediglich eine Endlosschleife, die in jedem Durchlauf die Funktion `breakpoint` ausführt.

Baut man eine Applikation, so veranlasst `cyg_start` alle weiteren Start-Schritte des Betriebssystems und der Applikation. Näheres läßt sich auch in [4], Kapitel 4 nachlesen.

# Kapitel 3

## Portierung

Abhängig davon, wieviel Glück man hat, ist der Aufwand für einen Port recht unterschiedlich. Es gibt drei verschiedene Arten von Ports. Welchen man implementieren muss, hängt von den Unterschieden zwischen dem aktuellen Ziel-System und den bereits vorhandenen Ports ab.

Grundsätzlich zum Portieren kann man sagen, dass kopieren/anpassen/umbenennen weniger Arbeit macht als neu schreiben.

### 3.1 Verzeichnisstruktur eines Ports

Ein Port sollte in einem eigenen Verzeichnis mit dem Namen des Ports, z.B. `dimm_sa1110`, im lokalen CVS-Baum stehen. Unter diesem Basis-Verzeichnis sollten folgende Unterverzeichnisse angelegt werden:

<code>cdl</code>	In diesem Verzeichnis steht die CDL-Datei des Ports. Der Name dieser Datei sollte dem Schema <code>hal_&lt;ARCHITEKTUR&gt;_&lt;PLATTFORM&gt;.cdl</code> entsprechen. Bspw. heißt die CDL-Datei für das GenoLogic StrongARM DIMM <code>hal_arm_dimm_sa1110.cdl</code> .
<code>devs</code>	In diesem Verzeichnis sollten alle Treiber für eine Plattform abgelegt werden. Für die verschiedenen Typen von Treibern sollten hier verschiedene Unterverzeichnisse angelegt werden. Z.B. sollte man für Flash-Treiber ein Unterverzeichnis <code>flash</code> anlegen.
<code>include</code>	In diesem Verzeichnis liegen alle Header und Include-Dateien eines Ports, z.B. auch <code>hal_platform_setup.h</code> .
<code>include/pkgconf</code>	In diesem Verzeichnis liegen sämtliche Dateien, die das Speicher-Layouts beschreiben, d.h. hier liegen alle <code>mlt-</code> , <code>ldi-</code> und die entsprechenden Header-Dateien.
<code>misc</code>	Dies ist der Platz für Dinge, die nicht so recht in anderen Verzeichnisse passen wollen, z.B. fertige ROM-Images.
<code>src</code>	In diesem Verzeichnis liegen die C/C++ und Assembler-Quelldateien des Ports, z.B. <code>&lt;PLATTFORM&gt;_misc.c</code> .
<code>tests</code>	In diesem Verzeichnis sollten die Quelldateien von Test-Programmen speziell für diese Plattform abgelegt werden, so es denn überhaupt spezielle Tests gibt. Allgemeine Tests sind im Standardumfang von eCos enthalten.

+++ FIXME Link vom Repository auf CVS+++

## 3.2 Plattform-Port

Der Plattform-Port ist das einfachste Portierungs-Unternehmen. Die Voraussetzung hierfür ist, dass die Ziel-CPU bzw. die Familie der Ziel-CPU bereits von eCos unterstützt werden, d.h. dass es bereits einen Port für diese CPU-Familie gibt. Familie ist hier in einem relativ engen Sinn zu verstehen. Die Intel StrongARM- oder die ATMEL AT91-CPU's kann man hierbei jeweils zu einer Familie zählen, da sowohl CPU-Core als auch die onchip-Peripherie jeweils identisch oder zumindest sehr ähnlich sind.

Folgende Punkte sind bei einem Plattform-Port mindestens zu erledigen:

- hal\_platform\_setup.h
- <PLATTFORM>\_misc.c
- Speicher-Layout
- CDL-Datei
- Eintragen des Ports in die Datenbank-Datei ecos.db

### 3.2.1 hal\_platform\_setup.h

Diese Datei besteht aus, in C-Makros verpackten, Assembler-Code. Das zentrale Makro hat den Namen PLATFORM\_SETUP1 und sollte folgenden Low-Level Initialisierungs-Schritte ausführen:

1. Initialisierung von I/O-Ports für die Ansteuerung von auf dem Board befindlichen LEDs; Implementation der LED-Ansteuerung in Form des C-Makros CYGHWR\_LED\_MACRO
2. Programmierung des Speicher-Controllers hinsichtlich Speicher-Timing aller genutzten Speicher-Bänke, was auch die ROM/Flash-Bänke mit einschließt
3. Programmierung des CPU-Taktes (bei programmierbarer PLL)
4. Initialisierung von I/O-Ports, sodass die zum Debuggen benutzte serielle Schnittstelle funktioniert (für den Fall, dass die I/O-Ports der seriellen Schnittstelle auch anderweitig genutzt werden können)
5. Umkopieren der Applikation aus dem ROM ins RAM bei ROMRAM-Startup
6. Aufruf der MMU-Initialisierungs-Funktion

Ist der Startup-Typ gleich RAM, so sollte PLATFORM\_SETUP1 als leeres Makro definiert werden, da die System-Initialisierung ja bereits von einem ROM-Monitor ausgeführt worden ist. Beispiel:

```
#if defined(CYG_HAL_STARTUP_ROM) || defined(CYG_HAL_STARTUP_ROMRAM)
#define PLATFORM_SETUP1 _platform_setup1
#else // defined(CYG_HAL_STARTUP_ROM v CYG_HAL_STARTUP_ROMRAM)
#define PLATFORM_SETUP1
#endif

// This macro represents the initial startup code for the platform
.macro _platform_setup1
    _setupLEDGPIO
    _blink 500000

    <...>

.endm
```

Für ROMRAM-Startup muss in einer bestimmten Phase von PLATFORM\_SETUP1 der Applikations-Code vom ROM ins RAM kopiert werden. Dies sollte unmittelbar vor der Programmierung der MMU, also relativ am Ende von PLATFORM\_SETUP1 getan werden. Glücklicherweise ist dieser Reloziierungs-Code lediglich CPU-Architektur-abhängig, sodass man ihn leicht übernehmen kann. Ein Beispiel für ARM-Reloziierungs-Code:

```

#if defined(CYG_HAL_STARTUP_ROMRAM)
#define RELOCATE_CODE \
    ldr    r2, =SA11X0_RAM_BANK0_BASE ;\
    ldr    r3, =__exception_handlers ;\
    ldr    r4, =SA11X0_ROM_BANK0_BASE ;\
    ldr    r5, =_end ;\
    add    r5, r5, r2 ;\
    add    r2, r2, r3 ;\
15:      ;\
    ldr    r6, [r4], #4 ;\
    str    r6, [r2], #4 ;\
    cmp    r2, r5 ;\
    bne    15b ;\
    nop    ;\
    nop    ;\
    nop    ;\
    nop    ;\
    nop    ;\
    nop    ;\
    nop    ;\
20:      ;\
    nop    ;\
    nop    ;\
    _blink
#else
#define RELOCATE_CODE
#endif

```

Der obige Code ist noch nicht optimal. Man kann ihn durch Verwendung von Multi-Register-Befehlen noch beschleunigen. Ebenso ist er noch nicht ganz von der speziellen Ziel-CPU abstrahiert, was in diesem Fall allerdings hilft das Beispiel verständlicher zu machen.

### 3.2.2 <PLATFORM>\_misc.c

Dieses Modul implementiert folgende Funktionen:

- void hal\_hardware\_init ( void)
 

Diese Funktion führt Hardware-Initialisierungen aus, die nicht von hal\_platform\_setup.h gemacht worden sind. Typisch wäre folgendes:

  - Interrupts löschen
  - Interrupt-Controller einschalten
  - hal\_if\_init aufrufen
  - plf\_hardware\_init aufrufen (z.B. bei den StrongARM-Ports)
  - Cache einschalten
- void hal\_clock\_initialize ( cyg\_uint32 pPeriod)
 

Diese Funktion initialisiert einen Timer als Hardware-Uhr. pPeriod ist die Anzahl der Timer-Takte, nach denen ein Interrupt ausgelöst wird.
- void hal\_clock\_reset (cyg\_uint32 pVector, cyg\_uint32 pPeriod)
 

Diese Funktion wird von der Interrupt-Behandlung des Timer/Uhren-Interrupts aufgerufen, um den Timer für den nächsten Interrupt neu zu programmieren, wenn die Timer-Hardware nicht selbständig fortlaufend Interrupts generiert. pVector ist die Nummer des Timer-Interrupts.

- `void hal_clock_read ( cyg_uint32* pValue)`  
Diese Funktion liefert die momentane, rohe Uhrzeit in der Variablen zurück auf die `pValue` zeigt und sollte nicht von der Applikations-Ebene aus benutzt werden.
- `void hal_delay_us ( cyg_int32 pUSecs)`  
Dies ist eine mikrosekunden-genaue Verzögerungs-Funktion, die normalerweise einen Timer in einer Busy-Wait-Schleife abfragt. D.h. während dieser Verzögerungs-Funktion ist keine andere nicht interrupt-getriebene Systemaktivität möglich.
- `int hal_IRQ_handler ( void)`  
Diese Funktion liefert die Nummer des gerade aufgetretenen Interrupts zurück. Einige CPU-Architekturen, wie z.B. ARM, haben nur zwei Interrupt-Leitungen in den Kern, sodass die konkrete Quelle des Interrupts beim Interrupt-Controller per Software ermittelt werden muss. Meist hat der Interrupt-Controller ein Register in dem ein gesetztes Bit einen aufgetretenen Interrupt anzeigt. Diese Funktion sollte möglichst zeiteffizient implementiert werden.
- `void hal_interrupt_mask ( int pVector)`  
Diese Funktion maskiert die Interrupt-Quelle `pVector` aus, d.h. die Interrupt-Erzeugung durch diese Quelle wird abgeschaltet. `pVector` ist die Nummer der Interrupt-Quelle und im CPU-Handbuch dokumentiert.
- `void hal_interrupt_unmask ( int pVector)`  
Diese Funktion hebt die Maskierung der Interrupt-Quelle `pVector` wieder auf, d.h. diese Quelle kann wieder Interrupts erzeugen.
- `void hal_interrupt_acknowledge ( int pVector)`  
Diese Funktion bestätigt einen Interrupt beim Interrupt-Controller. Bei den meisten Interrupt-Controllern müssen Interrupts bestätigt werden, um diese zu löschen. Anderenfalls würde ein einmal aufgetretener Interrupt immer wieder ein Interrupt-Signal auslösen.
- `void hal_interrupt_configure (int pVector, int pLevel, int pRaise)`  
Diese Funktion konfiguriert den Interrupt-Controller auf welche Veränderungen der Interrupt-Quelle `pVector` der Controller reagieren soll. Manche Interrupt-Controller sind programmierbar, ob sie einen Pegel-Wechsel oder eine Flanke als Interrupt auslösendes Signal interpretieren, was interessant ist, wenn die Interrupt-Quellen I/O-Leitungen sein können.  
Ein `pRaise > Null` sollte den Controller so programmieren, dass er auf eine Flanke reagiert, anderenfalls sollte er auf einen bestimmten Pegel reagieren. Ist `pRaise > Null`, so bedeutet `pLevel > Null`, dass der Controller auf eine steigende Flanke reagieren soll. Ist `pLevel <= Null`, so soll der Controller auf eine fallende Flanke reagieren.  
Ein `pRaise <= Null` sollte den Controller so programmieren, dass er auf Pegel reagiert. Ist `pRaise <= Null` und `pLevel > Null` sollte den Controller so programmieren, dass er auf einen High-Pegel reagiert und umgekehrt.
- `void hal_mmu_init ( void)`  
Hat die Ziel-CPU eine MMU, so sollte diese in dieser Funktion auf das spätere Speicher-Layout programmiert werden. Eingeschaltet wird die MMU jedoch nicht in dieser Funktion, sondern an einer anderen Stelle, üblicherweise in `hal_platform_setup.h`.

eCos stellt, zumindest für die ARM-Architektur, Makros zur Beschreibung der MMU-First-Level-Tabelle zur Verfügung.

Die Anpassung dieser Funktion an eine bestimmte Plattform beschränkt sich meist auf die Anpassung von Adressen und die Erweiterung um bestimmte Chip-Selects.

```
void hal_mmu_init(void) {
    unsigned long ttb_base = SA11X0_RAM_BANK0_BASE + 0x4000;
    unsigned long i;
```



```

        ARM_BUFFERABLE,
        ARM_ACCESS_PERM_RW_RW);
#endif // FLASH - size
/* FPGA at CS2 */
X_ARM_MMU_SECTION(0x100,
                  0x100,
                  1,
                  ARM_UNCACHEABLE,
                  ARM_UNBUFFERABLE,
                  ARM_ACCESS_PERM_RW_RW);
/* PLD at CS3 */
X_ARM_MMU_SECTION(0x180,
                  0x180,
                  1,
                  ARM_UNCACHEABLE,
                  ARM_UNBUFFERABLE,
                  ARM_ACCESS_PERM_RW_RW);
/* StrongARM(R) Registers */
X_ARM_MMU_SECTION(0x800,
                  0x800,
                  0x400,
                  ARM_UNCACHEABLE,
                  ARM_UNBUFFERABLE,
                  ARM_ACCESS_PERM_RW_RW);
/* DRAM Bank 0 */
X_ARM_MMU_SECTION(0xC00,
                  0x000,
                  16,
                  ARM_CACHEABLE,
                  ARM_BUFFERABLE,
                  ARM_ACCESS_PERM_RW_RW);
/* DRAM Bank 0 raw access */
X_ARM_MMU_SECTION(0xC00,
                  0xC00,
                  16,
                  ARM_UNCACHEABLE,
                  ARM_BUFFERABLE,
                  ARM_ACCESS_PERM_RW_RW);
/* DRAM Bank 1 */
X_ARM_MMU_SECTION(0xC80,
                  0x010,
                  16,
                  ARM_CACHEABLE,
                  ARM_BUFFERABLE,
                  ARM_ACCESS_PERM_RW_RW);
/* DRAM Bank 1 raw access */
X_ARM_MMU_SECTION(0xC80,
                  0xC80,
                  16,
                  ARM_UNCACHEABLE,
                  ARM_BUFFERABLE,
                  ARM_ACCESS_PERM_RW_RW);
/* Zeros (Cache Clean) Bank */
X_ARM_MMU_SECTION(0xE00,

```

```

        0xE00,
        128,
        ARM_CACHEABLE,
        ARM_BUFFERABLE,
        ARM_ACCESS_PERM_RW_RW);
}

```

Das Beispiel zeigt die Implementierung von `hal_mmu_init` für ein GenoLogic StrongARM DIMM. Der interessanteste Teil ist der Aufbau der Tabelle am Ende der Funktion. Hier werden die den einzelnen Chip-Selects zugeordneten Adress-Bereiche in den späteren, logischen Adressraum abgebildet. Auch für diese Tabelle sind bereits C-Makros definiert, die die Erzeugung der Tabellen-Einträge erleichtern.

`X_ARM_MMU_SECTION` erzeugt einen neuen Tabellen-Eintrag mit den in Klammern stehenden Parametern. Der erste Parameter ist die physikalische Adresse eines Speicherbereichs, der zweite Parameter ist die vorgesehene Ziel-, bzw. logische Adresse dieses Speicher-Bereichs nach dem Einschalten der MMU. Hierbei werden nur die höchsten drei Stellen der Adressen in hexadezimaler-Form geschrieben. Diese werden automatisch um die fehlenden Stellen zu einer vollständigen 32-Bit Adresse ergänzt. Der dritte Parameter ist die Größe des an die Ziel-Adresse abgebildeten Speicher-Bereichs. Die 32 MB RAM des Moduls liegen an zwei Chip-Selects à 16 MB. Die MMU wird so programmiert diese beiden Speicherblöcke in den logischen Speicher-Bereich zwischen 0 und 32MB des logischen Adress-Bereichs einzublenden.

Der vierte Parameter legt fest, ob der Speicherbereich bei Lesezugriffen gecached werden darf (`ARM_CACHEABLE`), oder ob Zugriffe darauf am Cache vorbei (`ARM_UNCACHEABLE`) stattfinden müssen. Der fünfte Parameter legt entsprechend fest, ob Schreibzugriffe am Cache vorbei (`ARM_UNBUFFERABLE`) gehen oder gecached (`ARM_BUFFERABLE`) werden. Handelt es sich um eingebundene Hardware, wie z.B. das FPGA an Chip-Select 2, so sollten die Zugriffe normalerweise nicht gecached stattfinden. Der sechste Parameter schließlich kennzeichnet den Speicherbereich als les-/schreibbar. Im Beispiel sind alle Speicher-Bereiche les- und schreibbar (`ARM_ACCESS_PERM_RW_RW`);

Wird eine Plattform in unterschiedlichen Bestückungs-Varianten hergestellt, wie das als Beispiel dienende DIMM, das z.B. mit 4 MB Flash oder 8MB Flash erhältlich ist, so sollten die entsprechenden MMU-Einträge, wie im Beispiel, mit C-Makros variabel gestaltet werden. Die entsprechenden Variablen (hier: `CYGHWR_HAL_ARM_SA11X0_SA1110DIMM_FLASH_SIZE`) werden in der CDL-Datei des Ports definiert und die richtige Ausstattung kann dann in der Datei `ecos.ecc` eingestellt werden.

- `void <PLATFORM>_program_new_stack (void* pFunc)`  
+++FIXME+++

Glücklicherweise muss man bei einem üblichen Plattform-Port nur die Funktionen `void hal_mmu_init`, `plf_hardware_init` - wenn diese Funktion überhaupt benötigt wird - und `<PORT_NAME>_program_new_stack` - hier ist zumindest der Funktions-Name zu ändern - anpassen. Die restlichen Funktionen können direkt übernommen werden.

Der eCos-StrongARM-Port teilt `<PLATFORM>_misc.c` - in diesem speziellen Fall `sa11x0_misc.c` geheißen - nochmals in zwei Dateien auf. Die im vorangegangenen Absatz aufgezählten Funktionen liegen hierbei in der Datei `+++FIXME+++`

### 3.2.3 Speicher-Layout

Als Speicher-Layout wird die Beschreibung an welchen Adressen im Adress-Raum der CPU sich welcher physikalische Speicher der Plattform befindet bezeichnet. In der Windows-Version von `ecosconfig` ist ein grafischer Editor enthalten, um ein Speicher-Layout zu beschreiben. Aus dem Produkt dieses Editors werden dann alle anderen Speicher-Layout Dateien generiert, man kann diese Dateien jedoch auch von Hand erzeugen, was im Falle komplexerer Layouts auch zu empfehlen ist.

Ein Speicher-Layout wird durch drei verschiedene Dateien, hier benannt nach der Datei-Endung, beschrieben:

- mlt-Datei Diese Datei ist Produkt des grafischen Speicher-Layout-Editors und wird nur benutzt, wenn auch der Editor benutzt wurde. D.h. wenn man die anderen Dateien von Hand erzeugt ist diese Datei unwichtig.
- ldi-Datei Diese Datei ist die Rohversion des späteren Linker-Skripts. Um diese Datei zu verstehen hilft Wissen über GNU-Linker-Skripte.
- h-Datei In dieser C-Include-Datei stehen Informationen über die Speicher-Bereiche hinsichtlich Start-Adresse im CPU-Adressraum, Größe und Schreib-/Lese-Fähigkeit in Form von C-Makros. Zusätzlich wird die Größe des Heap als C-Makro definiert.

Jeder Startup-Typ der Plattform benötigt einen eigenen Satz von Beschreibungs-Dateien, da sich das Speicher-Layout vor und nach dem Einschalten der MMU - bzw. dem üblichen Verschieben von Bänken bei Nicht-MMU-CPU's - unterscheiden. Allerdings ist das Speicher-Layout für RAM-Startup und ROMRAM-Startup meist identisch.

Die folgende ldi-Datei (mlt\_arm\_sa11x0\_sa1110dimm\_ram.ldi) wurde für RAM-Startup für ein GenoLogic StrongARM DIMM mit 32MB RAM und 4MB Flash-Speicher (ROM) geschrieben:

```
#include <cyg/infra/cyg_type.inc>

MEMORY
{
    ram : ORIGIN = 0, LENGTH = 0x2000000
    rom : ORIGIN = 0x40000000, LENGTH = 0x400000
}

SECTIONS
{
    SECTIONS_BEGIN
    SECTION_fixed_vectors (ram, 0x20, LMA_EQ_VMA)
    SECTION_rom_vectors (ram, 0x20000, LMA_EQ_VMA)
    SECTION_text (ram, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_fini (ram, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_rodata (ram, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_rodata1 (ram, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_fixup (ram, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_gcc_except_table (ram, ALIGN (0x4), LMA_EQ_VMA)
    SECTION_data (ram, ALIGN (0x4), LMA_EQ_VMA)
    .sec1 ALIGN (0x4) : { __fpga_data_start = ABSOLUTE (.); *(.sec*) \
        __fpga_data_end = ABSOLUTE (.); } > ram
    SECTION_bss (ram, ALIGN (0x4), LMA_EQ_VMA)
    CYG_LABEL_DEFN(__heap1) = ALIGN (0x8);
    SECTIONS_END
}
```

Aus dieser Datei wird durch Verarbeitung durch den C-Präprozessor das Linker-Skript target.ld erzeugt. SECTIONS\_BEGIN, SECTIONS\_END, SECTION\_\* und LMA\_EQ\_VMA sind C-Makros, die in der Datei +++FIX-ME+++ definiert sind.

Für die üblichen Sektionen eines ARM-Binary sind Makros definiert, für die etwas unüblicheren FPGA-Daten muss man hier selbst Hand anlegen. Ansonsten muss man diese Datei für einen neuen Port lediglich kopieren und entsprechend den Gegebenheiten der neuen Plattform anpassen. Dies gilt natürlich ebenso für die Dateien für die anderen Startup-Typen.

Die nachfolgende Datei ist die zur obigen ldi-Datei gehörende Header-Datei:

```
ifndef __ASSEMBLER__
#include <cyg/infra/cyg_type.h>
#include <stddef.h>
```

```

#endif
#define CYGMEM_REGION_ram      (0)
#define CYGMEM_REGION_ram_SIZE (0x2000000)
#define CYGMEM_REGION_ram_ATTR (CYGMEM_REGION_ATTR_R | CYGMEM_REGION_ATTR_W)
#ifndef __ASSEMBLER__
extern char CYG_LABEL_NAME      (__heap1) [];
#endif
#define CYGMEM_SECTION_heap1    (CYG_LABEL_NAME (__heap1))
#define CYGMEM_SECTION_heap1_SIZE (CYGMEM_REGION_ram_SIZE - \
                                   (size_t) CYG_LABEL_NAME (__heap1))

```

Wie man sieht wird das in der ldi-Datei definierte Label `__heap1` hier benutzt, um die Größe des Heap zu berechnen. Auch diese Dateien sind einfach von einem anderen Port zu übernehmen und lediglich leicht zu modifizieren.

### 3.2.4 Einen „Glue“-Treiber für den Plattform Flash-Speicher schreiben

Glücklicherweise sind Hardware-Treiber in eCos, soweit möglich, von der konkreten Plattform abstrahiert. Viele Bausteine, wie Ethernet-, Seriell- oder Flash-Bausteine, sind unabhängig von einer bestimmten Plattform. Um diese benutzen zu können, muss man dem Treiber eigentlich nur noch Dinge wie Basis-Adresse oder Bus-Breite mitteilen.

```

#undef  CYGPKG_DEVS_FLASH_AMD_AM29XXXXX

#ifndef CYGHWR_HAL_ARM_SA11X0_DIMM_SA1110_FLASH_SIZE
#define CYGHWR_HAL_ARM_SA11X0_DIMM_SA1110_FLASH_SIZE ( 4)
#endif

#if ( CYGHWR_HAL_ARM_SA11X0_DIMM_SA1110_FLASH_SIZE == 4)
#define CYGHWR_DEVS_FLASH_AMD_AM29LV800
#elif ( CYGHWR_HAL_ARM_SA11X0_DIMM_SA1110_FLASH_SIZE == 8)
#define CYGHWR_DEVS_FLASH_AMD_AM29LV160
#endif

#define CYGNUM_FLASH_INTERLEAVE (2)
#define CYGNUM_FLASH_SERIES      (2)
#define CYGNUM_FLASH_BASE        (0x40000000)
#define CYGNUM_FLASH_BLANK       (1)
#define CYGNUM_FLASH_WIDTH       (16)
#define CYGNUM_FLASH_ID_MANUFACTURER FLASHWORD(0x4)

#include "cyg/io/flash_am29xxxxx.inl"

```

Das obige Beispiel ist der Glue-Flash-Treiber für das GenoLogic StrongARM DIMM. Er berücksichtigt zwei Bestückungs-Varianten, einmal mit vier 29LV800-Chips und einmal mit vier 29LV160-Chips. Die Chips sind hierbei sowohl zweifach in Serie (Makro `CYGNUM_FLASH_SERIES`) als auch zweifach parallel, für 32Bit-Zugriff, organisiert.

### 3.2.5 Die CDL-Datei

CDL-Dateien beschreiben Pakete für das eCos-Gesamtsystem und sind nötig diese Pakete in eCos zu integrieren. Hat man für ein selbstentwickeltes Paket eine CDL-Datei geschrieben und diese in die Datenbank-Dateien `ecos.db` eingetragen, siehe Abschnitt 3.2.6, so kann man das selbstentwickelte Paket genau so wie ein originäres eCos-Paket mit `ecosconfig` verwalten. D.h. man kann es zu einer Konfiguration hinzufügen bzw. entfernen oder das Paket konfigurieren.

Am Beispiel der CDL-Datei für den eCos-Port für das GenoLogic StrongARM DIMM wird der Aufbau einer CDL-Datei erläutert:

```
#
# Begin of CDL-Package for the GenoLogic StrongARM DIMM
#

cdl_package CYGPKG_HAL_ARM_SA11X0_SA1110DIMM {
  display      "ARM SA1110/GenoLogic SA1110 DIMM board"
  parent       CYGPKG_HAL_ARM_SA11X0
  hardware
  include_dir  cyg/hal
  define_header hal_arm_sa11x0_dimm_sa1110.h
  description  "
    This HAL platform package provides generic
    support for the Intel StrongARM SA1110 based
    GenoLogic DIMM board."

  compile      dimm_sa1110_misc.c

  implements   CYGINT_HAL_DEBUG_GDB_STUBS
  implements   CYGINT_HAL_DEBUG_GDB_STUBS_BREAK
  implements   CYGINT_HAL_VIRTUAL_VECTOR_SUPPORT

  implements   CYGHWR_HAL_ARM_SA11X0_UART1
  implements   CYGHWR_HAL_ARM_SA11X0_UART3

  define_proc {
    puts $::cdl_system_header "#define CYGBLD_HAL_TARGET_H \
      <pkgconf/hal_arm.h>"
    puts $::cdl_system_header "#define CYGBLD_HAL_VARIANT_H \
      <pkgconf/hal_arm_sa11x0.h>"
    puts $::cdl_system_header "#define CYGBLD_HAL_PLATFORM_H \
      <pkgconf/hal_arm_sa11x0_sa1110dimm.h>"
    puts $::cdl_header "#define HAL_PLATFORM_CPU \
      \"StrongARM 1110\""
    puts $::cdl_header "#define HAL_PLATFORM_BOARD \
      \"GenoLogic SA1110 DIMM development system\""
    puts $::cdl_header "#define HAL_PLATFORM_EXTRA \"\""
    puts $::cdl_header "#define HAL_PLATFORM_MACHINE_TYPE 25"
    puts $::cdl_header "#define HAL_ARCH_PROGRAM_NEW_STACK \
      sa1110dimm_program_new_stack"
  }
}
```

Schlüsselwort	Wert/Bedeutung
<code>display</code>	Der hier abgelegte String wird in den grafischen Versionen von <code>ecos-config</code> in der Liste der vorhandenen Pakete ausgegeben.
<code>parent</code>	Dies ist der Name des Pakets von dem dieses Paket abgeleitet worden ist. In diesem speziellen Fall ist der Port für das GenoLogic StrongARM DIMM nur eine Erweiterung des allgemeinen eCos StrongARM Ports.
<code>hardware</code>	Das Paket bietet Unterstützung für Hardware.
<code>include_dir</code>	Die zu diesem Paket gehörenden Header-Dateien, außer der durch <code>define_header</code> definierten, werden im Verzeichnis <code>&lt;ECOS_BUILD&gt;/install/include/cyg/hal</code> abgelegt. D.h. dieses Schlüsselwort beeinflusst den Speicherort der zu einem Paket gehörenden Header-Dateien.
<code>define_header</code>	Diese Datei wird durch den Aufruf von <code>ecosconfig tree</code> aus Informationen aus <code>ecos.ecc</code> und der CDL-Datei generiert. Die Datei besteht aus C-Defines (siehe auch <code>define_proc</code> ). Die Datei wird im Verzeichnis <code>&lt;ECOS_BUILD&gt;/install/include/pkgconf</code> abgelegt.
<code>compile</code>	Dies ist eine durch Leerzeichen getrennte Liste der Quell-Dateien des Pakets, die für den Bau eines ROM-Monitors oder der Laufzeit-Bibliothek zu übersetzen sind. Dieser Port enthält nur eine einzige „richtige“ Quell-Datei. Alle anderen Dateien sind entweder Konfigurations-Dateien oder Header-/Include-Dateien.
<code>implements</code>	Bestimmte Funktionalitäten in eCos sind von der konkreten Implementierung durch eine Schnittstelle ( <code>cdl_interface</code> ) abstrahiert. D.h., dass der Anwender der Funktionalität diese benutzen kann ohne sich Gedanken machen zu müssen, wer die konkrete Funktionalität implementiert und wie diese implementiert ist. Um anzuzeigen, dass ein Paket eine bestimmte Schnittstelle implementiert, ist das Schlüsselwort <code>implements</code> mit dem Namen der implementierten Schnittstelle in die CDL-Datei einzufügen. Im Beispiel werden die Schnittstellen für die Funktionalität des GDB-Stub und für zwei serielle Schnittstellen von diesem Paket bzw. den zu diesem Paket gehörenden Paketen implementiert.
<code>define_proc</code>	Alle durch den Zusatzschlüssel <code>\$::cdl_system_header</code> markierten C-Defines werden landen in der Datei <code>&lt;ECOS_BUILD&gt;/install/include/pkgconf/system.h</code> . Diese Datei enthält ebenfalls Konfigurationen, die aus <code>ecos.ecc</code> und den CDL-Dateien aller Pakete der Konfiguration extrahiert worden sind. Alle durch den Zusatzschlüssel <code>\$::cdl_header</code> markierten C-Defines landen in der vom Schlüssel <code>define_header</code> spezifizierten Header-Datei.
<code>cdl_component</code>	<pre> CYG_HAL_STARTUP {   display      "Startup type"   flavor       data   default_value {"RAM"}   legal_values {"RAM" "ROM" "ROMRAM"} } </pre>
<code>no_define</code>	<pre> define -file system.h CYG_HAL_STARTUP description  "   When targetting the GenoLogic SA1110 DIMM board it is possible to build </pre>

```

the system for either RAM bootstrap or ROM bootstrap(s). Select
'ram' when building programs to load into RAM using eCos GDB
stubs. Select 'rom' when building a stand-alone application
which will be put into ROM, or for the special case of
building the eCos GDB stubs themselves. Select 'romram' for
standalone applications starting from ROM and running in RAM"
}

```

<...>

```

cdl_option CYGHWR_HAL_ARM_SA11X0_SA1110DIMM_PROCESSOR_CLOCK {
  display      "SA1110 DIMM processor clock speed"
  flavor       data
  legal_values 132700 206400
  default_value 206400
  description  "
    The GenoLogic SA1110 DIMM may be equipped with different type of
    StrongARM CPUs. Those with a BD printed on will run with 206MHz,
    those with an AD printed on will run with 133MHz."
}

```

```

cdl_option CYGHWR_HAL_ARM_SA11X0_DIMM_SA1110_FLASH_SIZE {
  display      "SA1110 DIMM flash size"
  flavor       data
  legal_values 4 8
  default_value 4
  description  "
    The GenoLogic SA1110 DIMM may be equipped with different type of
    flash chips. If 29LV800 chips are mounted, then the total flash size
    is 4MB. If 29LV160 chips are mounted, then the total flash size is
    8MB."
}

```

<...>

Das GenoLogic StrongARM DIMM wird in verschiedenen Varianten gefertigt, unter anderem werden verschiedene Versionen des Prozessors hinsichtlich des maximalen Taktes und unterschiedliche Flash-Bestückungen geliefert.

Um nicht für jede Bestückungs-Variante einen eigenen Port schreiben zu müssen, kann man im Code C-Defines berücksichtigen, wie z.B. in `dimmsa1110.c` in Abschnitt 3.2.2. Dort wird das C-Makro `CYGHWR_HAL_ARM_SA11X0_DIMM_SA1110_FLASH_SIZE` abgefragt um die MMU entsprechend der Flash-Ausstattung zu programmieren. Der Wert dieses Makros wird durch die CDL-Option mit dem Namen des Makros gesteuert. Wählt man in der grafischen Version von `ecosconfig` Flash-Größe 8 aus, bzw. setzt man diese in `ecos.ecc` auf `user_value 8`, so wird während des Laufs von `ecosconfig tree` obengenanntes Makro mit dem Wert 8 erzeugt. Mit dem Konstrukt `cdl_option` lassen sich also bequem Makros definieren.

Schlüsselwort	Wert/Bedeutung
flavor	Mit diesem Schlüsselwort wird die Art der Daten beschrieben, die hier eingestellt werden können.
legal_values	Dies ist eine durch Leerzeichen getrennte Liste von erlaubten Werten für die Option. In der grafischen Version von ecosconfig wird daraus eine Liste generiert aus der man den gewünschten Wert auswählen kann. Wählt man per <code>user_value</code> durch direkte Manipulation von <code>ecos.ecc</code> einen Wert, der nicht in der Liste der <code>legal_values</code> enthalten ist, so wird beim Ausführen von <code>ecosconfig tree</code> eine Fehlermeldung ausgegeben.
default_value	Hier wird der voreingestellte Wert eingetragen, den diese Option hat, wenn der Benutzer keinen anderen Wert ausgewählt hat.

```

cdl_component CYGHWR_MEMORY_LAYOUT {
    display "Memory layout"
    flavor data
    no_define
    calculated { CYG_HAL_STARTUP == "RAM" ? "arm_s11x0_dimm_s1110_ram" : \
                CYG_HAL_STARTUP == "ROM" ? "arm_s11x0_dimm_s1110_rom" : \
                "arm_s11x0_dimm_s1110_romram" }

cdl_option CYGHWR_MEMORY_LAYOUT_LDI {
    display "Memory layout linker script fragment"
    flavor data
    no_define
    define -file system.h CYGHWR_MEMORY_LAYOUT_LDI
    calculated { CYG_HAL_STARTUP \
                == "RAM" ? \
                "<pkgconf/mlt_arm_s11x0_s1110dimm_ram.lds>" : \
                CYG_HAL_STARTUP \
                == "ROM" ? \
                "<pkgconf/mlt_arm_s11x0_s1110dimm_rom.lds>" : \
                "<pkgconf/mlt_arm_s11x0_s1110dimm_romram.lds>" }
}

cdl_option CYGHWR_MEMORY_LAYOUT_H {
    display "Memory layout header file"
    flavor data
    no_define
    define -file system.h CYGHWR_MEMORY_LAYOUT_H
    calculated { CYG_HAL_STARTUP \
                == "RAM" ? \
                "<pkgconf/mlt_arm_s11x0_s1110dimm_ram.h>" : \
                CYG_HAL_STARTUP \
                == "ROM" ? \
                "<pkgconf/mlt_arm_s11x0_s1110dimm_rom.h>" : \
                "<pkgconf/mlt_arm_s11x0_s1110dimm_romram.h>" }
}
}
}
<...>

```

```

}
#
# End of CDL-Package for the GenoLogic StrongARM DIMM
#

```

Eine CDL-Komponente (`cdl_component`) kann, wie ein CDL-Package, selbst wieder aus Komponenten und Optionen bestehen. Da der Startup-Typ Einfluss auf verschiedene Optionen, sind diese unter der Komponente `CYGHWR_MEMORY_LAYOUT` zusammengefasst. Interessant ist hier die Bestimmung der Werte für die Optionen durch den Ausdruck `calculated`. Hier wird ein aus C und TCL - CDL basierender auf TCL - bekannter verkürzter `if-then-else` Ausdruck verwendet, um das vom Inhalt von `CYG_HAL_STARTUP` abhängige Ergebnis zu berechnen. Der Ausdruck hinter `define` bewirkt, dass in der Header-Datei `system.h` die Makros `CYGHWR_MEMORY_LAYOUT_H` und `CYGHWR_MEMORY_LAYOUT_LDI` mit den entsprechenden Werten definiert werden.

### 3.2.6 Einfügen von Paketen in die Datenbank-Datei `ecos.db`

In der Datenbank-Datei `ecos.db` steht eine Liste von allen Paketen, die im eCos-Repository enthalten sind. `ecosconfig` benutzt diese Datei und ein Paket, das nicht in `ecos.db` aufgelistet ist, existiert faktisch für `ecosconfig` nicht. Ein Eintrag in `ecos.db` ist dabei nicht viel mehr als ein Verweis auf die CDL-Datei eines Pakets, versehen mit Kommentar und einem Alias-Namen für das Paket. Das folgende Beispiel zeigt den `ecos.db`-Eintrag für das GenoLogic StrongARM DIMM HAL-Paket:

```

package CYGPKG_HAL_ARM_SA11X0_DIMM_SA1110 {
  alias { "GenoLogic SA1110 DIMM" hal_arm_sa11x0_dimm_sa1110 }
  directory hal/arm/sa11x0/dimm_sa1110
  script hal_arm_sa11x0_dimm_sa1110.cdl
  hardware
  description
    "This package provides eCos support for the GenoLogic SA1110 DIMM."
}

```

Zuerst wird mit dem Schlüsselwort `alias` ein Alias für das Paket definiert. Das Paket kann dann später über den eigentlichen Paket-Namen `CYGPKG_HAL_ARM_SA11X0_DIMM_SA1110` oder über den Alias `hal_arm_sa11x0_dimm_sa1110` referenziert werden. Dem Schlüsselwort `directory` wird der Pfad zum Basis-Verzeichnis des Pakets ausgehend vom Verzeichnis `packages` im eCos-Repository zugeordnet. Da, zumindest auf Unix-Datei-Systemen, ein relativer Pfad ein beliebiges Verzeichnis referenzieren kann, sind die in Abschnitt 3.1 gemachten symbolischen Datei-Links nicht unbedingt nötig.

Dem Schlüsselwort `script` wird der Name der CDL-Datei des Pakets zugeordnet. Das Schlüsselwort `hardware` bedeutet, dass es sich bei diesem Paket um Hardware-Unterstützung handelt.

Um die neue Plattform auch als Target des Befehls `ecosconfig new <TARGET>` benutzen zu können, muss zusätzlich ein neues Target in `ecos.db` definiert werden.

```

target dimm_sa1110 {
  alias { "GenoLogic SA1110 DIMM" sa1110dimm dat_komische_board_mit_sa1110 }
  packages { CYGPKG_HAL_ARM
             CYGPKG_HAL_ARM_SA11X0
             CYGPKG_HAL_ARM_SA11X0_DIMM_SA1110
             CYGPKG_IO_SERIAL_ARM_SA11X0 }
  description
    "This package provides eCos support for the GenoLogic SA1110 DIMM."
}

```

Auch hier werden wieder Alias-Namen für das Target definiert. D.h. man kann `dimm_sa1110`, `sa1110dimm` oder `dat_komische_board_mit_sa1110` für `<TARGET>` einsetzen, um für das GenoLogic StrongARM DIMM einen ROM-Monitor oder die Laufzeit-Bibliothek `libtarget.a` zu bauen.

Beim Schlüsselwort `package` sind die Pakete aufgezählt, die mindestens in eine sinnvolle Umgebung für das Target gehören. Zu beachten ist, dass jedes Paket durch definierte Abhängigkeiten die Einbeziehung weiterer Pakete in die Umgebung induzieren kann.

Abschließend noch ein Beispiel für die Integration des Flash-Pakets für das GenoLogic StrongARM DIMM:

```
package CYGPKG_DEVS_FLASH_DIMM_SA1110 {
  alias      { "FLASH memory support for the GenoLogic SA1110 \
               based DIMM" flash_dimm_sa1110 }
  directory  devs/flash/arm/dimm_sa1110
  script     flash_dimm_sa1110.cdl
  hardware
  description "
    This package contains hardware support for FLASH memory on the
    SA1110 GenoLogic DIMM."
}
```

### 3.3 Varianten-Port

Der Varianten-Port ist deutlich aufwändiger als der Plattform-Port. Voraussetzung für einen Varianten-Port ist, dass die CPU-Architektur, wie ARM oder MIPS, bereits von eCos unterstützt wird. Bspw. wird die MIPS-Architektur bereits von eCos unterstützt, nicht jedoch die Aurum-CPU-Familie von Alchemy, die auf einem MIPS-Kern basieren.

Zusätzlich zu den in Abschnitt 3.2 mindestens zu implementierenden Teilen, sind noch folgende Dinge zu unterstützen

- Interrupt-, MMU- und Timer-Unterstützung
- Register-Konstanten/Register-Adressen
- evtl. Treiber für die serielle Schnittstelle, mindestens aber ein „Glue“-Treiber

### 3.4 Architektur-Port

Zusätzlich zu den in Abschnitt 3.3 mindestens zu implementierenden Teilen, sind folgende Teile zu implementieren:

- `vectors.S`
- Makros für Register- und I/O-Zugriffe



# Anhang A

## Probleme und Lösungen

### A.1 Es wird nicht zum Beginn von PLATFORM\_SETUP1 gesprungen (ARM)

Dieser Fehler ist zugegebenermaßen schwierig zu diagnostizieren, da die Hardware meist „wie tot“ daliegt, wenn PLATFORM\_SETUP1 nicht angesprungen wird. Die erste Instruktion in der ARM-Variante von vectors.S ist eine unbedingte Sprung-Anweisung (`b reset_vector`) oder eine unbedingte Lade-Anweisung (`ldr pc, .reset_vector`), die beide von der Wirkung her identisch sein sollten.

Bei einigen CPUs gibt es jedoch Probleme, insbesondere mit dieser speziellen Lade-Anweisung an dieser speziellen Stelle, sodass in vectors.S das C-Makro `CYGSEM_HAL_ROM_RESET_USES_JUMP` entscheidet, ob die Sprung- oder die Lade-Anweisung ausgeführt wird. Ist das Makro gesetzt, so wird die Sprung-Anweisung benutzt, anderenfalls die Lade-Anweisung.



# Literaturverzeichnis

- [1] Anthony J. Massa:  
**eCos Porting Guide**  
<http://www.embedded.com/story/OEG20011220S0059>
  
- [2] RedHat-eCos-Team:  
**eCos Porting Guide**  
<http://sources.redhat.com/ecos/docs-latest/porting/index.html>
  
- [3] Bart Veer, John Dallaway:  
**The eCos Component Writer's Guide**  
<http://sources.redhat.com/ecos/docs-latest/cdl/cdl-guide.html>  
<http://sources.redhat.com/ecos/docs-latest/pdf/cdl-guide.pdf>
  
- [4] RedHat-eCos-Team:  
**eCos Reference Manual**  
<http://sources.redhat.com/ecos/docs-latest/ref/ecos-ref.1.html>  
<http://sources.redhat.com/ecos/docs-latest/pdf/ecos-ref.pdf>
  
- [5] **Archiv der eCos Mailing-Liste**  
<http://sources.redhat.com/ml/ecos-discuss>
  
- [6] Andreas Bürgel:  
**Die GNUPro Tool-Chain**  
[http://www.andreas-buergel.de/documents/gnupro\\_toolchain.pdf](http://www.andreas-buergel.de/documents/gnupro_toolchain.pdf) Handbuch der GenoLogic GmbH