Schnittstellenbeschreibung des Windows-NT Treibers

PCIVME.SYS

für das PCI-VME Interface von ARW-Elektronik

ARW - Elektronik
Schlehenweg 1
D-69168 Wiesloch
Germany
Fax+Tel. 06222/50816

Veränderungen

30.10.1999 Erstellung des Dokuments.
06.12.1999 Korrekturen: Pfade, pvmon, ..RESET, ..INIT, ..DEINIT

Inhaltsverzeichnis

1. LIEFERUMFANG

Auf der mitgelieferten Diskette finden sich folgende Inhalte:
Im Verzeichnis WINNT\DRIVER:

pcivmed.sys	-	der Treiber
install.exe	-	das Installations und Deinstallationsprogramm

Im Verzeichnis WINNT\DRIVER\SOURCE:

pcivme.h	-	die Header-Datei zur Schnittstelle zum Treiber.
vic.h	-	die Header-Datei zum VIC68A Baustein (identisch zu WIN95).
vme.h	-	die Header-Datei zu VME-Adreßbereichen (identisch zu WIN95).
	-	die Quellen zum „pcivme.sys“ Treiber

Im Verzeichnis COMMON\PVMON

pvmon.exe	-	ein nützliches Programm.

Im Verzeichnis COMMON\PVMON\SOURCE

	-	die Quellen zu pvmon.exe.

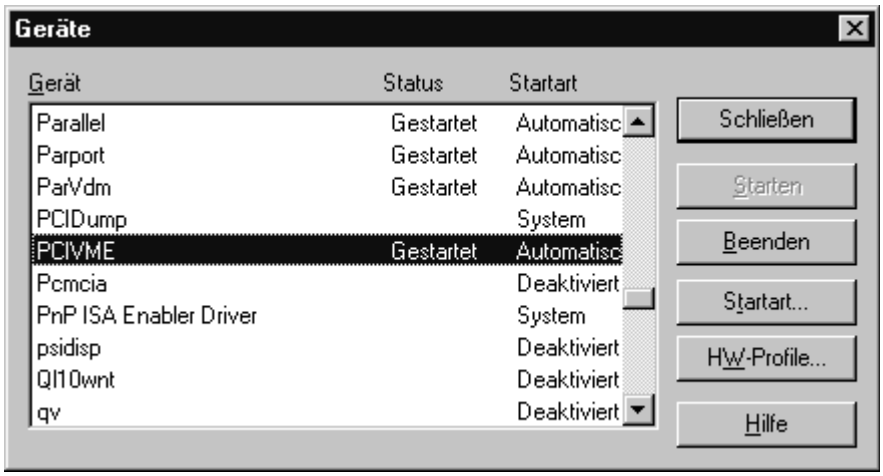
2. INSTALLATION

Hinweis: Der Treiber arbeitet nur unter Windows-NT. Er wurde mit Windows-NT 4.0 und installiertem Service-Pack 4 getestet. Da jedoch keine versionsspezifischen Anteile verwendet werden sollte der Treiber auch mit anderen Windows-NT Versionen zusammenarbeiten.

Die Schnittstelle zum Treiber ist großteils kompatibel zur Schnittstelle des WIN95/98 Treibers. Aufgrund der architekturspezifischen Unterschiede der Betriebssysteme konnte keine hundertprozentige Übereinstimmung erreicht werden.

Installieren Sie die PCIVME-Hardware nach Anleitung. Lassen Sie Ihren Rechner unter WINDOWS-NT starten. Melden Sie sich nach dem Boot als „Administrator“ an. Durch Starten des Programms „install.exe“ aus dem Pfad „WINNT\DRIVER“ wird der Treiber installiert.

Der erfolgreiche Start kann durch Start von „Systemsteuerung - Geräte“ verifiziert werden:



Dort erscheint nun obiger Eintrag. Der Gerätetreiber für das PCIVME Interface wird nun bei jedem Neustart des WINDOWS-NT automatisch geladen und gestartet.
Ebenso wird ein Eintrag in der Registry unter „HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\PCIVME“ erstellt.

2. DEINSTALLATION

Rufen Sie entweder über eine DOS-Box oder mittels der „Startleiste - Ausführen“ das Programm „install.exe“ im Pfad „WINNT\DRIVER“ mit der Optione „-r“ (remove) auf. Dies entfernt sowohl das Programm aus dem Treiberverzeichnis des Rechners als auch den zugehörigen Eintrag in der Registry. Der Treiber kann nicht entfernt werden, solange er noch von einem Programm genutzt wird.

3. INSTALLATIONSTEST

Mit Hilfe des Programms „pvmon“ können Sie einige einfache Tests auf dem VMEbus durchführen. Am Besten lassen sich diese Tests mit dem Test- und Diagnosewerkzeug VDIS von ARW durchführen.

Öffnen Sie eine DOS-Box unter Windwos95 und starten Sie dort die Konsole Applikation „pvmon.exe“. Die Abfrage „pvmon -?“ gibt eine Hilfe an der Kommandozeile. Die Eingabe von „?“ hilft dem Benutzer nach dem Aufruf weiter.

Der Treiberpfad muß angegeben werden: z.B. „\\.\pcivme:\vmemm2“ für den Treiber zum Interface mit der VMEMM-Modulnummer 2. Dann ist mit „c“ (wie configure) die Einstellung des pvmon zu setzen. Bitte beenden Sie pvmon nach der Einstellung und Sicherung der Konfiguration und starten Sie das Programm erneut.

4. ARBEITSWEISE DES TREIBERS

Der Treiber „pcivme.sys“ muß vor der Benutzung durch Applikationen geladen und gestartet sein. Am Besten nutzt man die Möglichkeit des WINDOWS-NT Betriebssystems Treiber automatisch beim Start des Systems zu starten. Die Installation legt diese Option an.

Der Treiber kann von mehreren Anwendungsprogrammen gleichzeitig genutzt werden. Er unterstützt synchrone und asynchrone Aufrufe. (Asynchrone Aufrufe kehren sofort aus dem Treiber zurück und bieten das Ergebnis des gegebenen Dienstes zum späteren Zeitpunkt an.)

Der Treiber bietet Ihren Anwendungsprogrammen eine gesicherte Schnittstelle zur Hardware. Er arbeitet im Ring-0 des Prozessors und bietet die Resource „PCI-VME-Interface“ gleichberechtigt allen Programmen, die das Interface nutzen wollen, an.

Der Treiber benötigt keine Information über die Adreßeinstellung des Interface. Sämtliche Information erfragt der Treiber von der PCI-Plug-and-Play Einstellung oder aus der „Windows-Registry“.

Der Treiber wird über Standard-IO-Funktionen von WINDOWS-NT angesprochen und ist daher unabhängig von jeder benutzten Programmiersprache. Die Strukturen und Konstanten der Header-Dateien sind jedoch nur für C oder C++ angeboten, können jedoch leicht auf andere Syntax angepaßt werden.

Der Treiber unterstützt gleichzeitig bis zu 16 Interface Einsteckkarten in den PC mit angeschlossenen VMEMM Einsteckkarten in den VME-BUS. Die Interface werden alleinig durch die Modul-Nummer der angeschlossenen VMEMM unterschieden. Die Modul-Nummer spiegelt sich beim Öffnen des Treiber-Pfades im Namen wieder. So öffnet z.B. der Pfad „\\.\pcivme:\vmemm2“ den Pfad zu dem VMEMM Interface mit der Modulnummer 2. (Hier zeigt sich ein wesentlicher Unterschied zum WINDOWS-95 Treiber!)
Jeder Pfad kann mehrfach geöffnet werden. Dies ermöglicht es, einzelne Pfade für den Zugriff auf bestimmte VME-Adreßbereiche oder alleinig für das Erfassen von Interrupts zu benutzen.

4.1. Beim Öffnen des Pfades

```
HANDLE handle = CreateFile(„\\.\pcivme:vmemm1“,
                           GENERIC_READ | GENERIC_WRITE,
                           0,
                           NULL,
                           OPEN_EXISTING,
                           FILE_ATTRIBUTE_NORMAL,
                           NULL);
```

erfaßt der Treiber die Anzahl der eingesteckten PCIADA-Module und durchsucht nun alle eingesteckten PCIADA-Module danach, ob ein VMEMM mit der passenden Modulnummer angeschlossen ist. (Die Modulnummer muß eindeutig sein!)

4.2. Nun kann mittels der Standardaufrufe

```
bNoError = ReadFile(hHandle, pBuffer, dwHowmany, &dwBytesRead, NULL);
bNoError = WriteFile(hHandle, pBuffer, dwHowmany, &dwBytesWrittn, NULL);
```

vom VME-BUS gelesen oder auf den VME-BUS geschrieben werden. Die notwendigen zusätzlichen Zugriffsparameter müssen zuvor für jeden Pfad mittels

```
dwResult = DeviceIoControl(hHandle, PCIVME_SET_ACCESS_PARA,
        &sAccess_parameter, sizeof(sAccess_parameter), NULL,
                0, &dwDIOC_count, NULL);
```

gesetzt werden. Das wird jedoch weiter unten erläutert.
Hinweis: Hier bestehen Unterschiede zum WINDOWS-95 Treiber.

4.3. Alle weiteren Einstellzugriffe auf den Pfad werden mit Hilfe der Standard-Funktionen

```
dwResult = DeviceIoControl(hHandle, ......);
```

durchgeführt.

4.4. Nach Beendigung der Nutzung muß der Pfad mit

```
CloseHandle(hHandle);
```

abgemeldet werden.

Es sind mehrere Dienste zur Nutzung mit den Einstellzugriffen (DeviceIoControl(....)) festgelegt. Diese Dienste werden durch festgelegte Nummern unterschieden (z.B. #define PCIVME_GET_STATIC_STATUS ....). Zu jedem Dienst assoziieren in der Regel noch eine Eingangs- und eine Ausgangsstruktur. Zum Aufruf der jeweiligen Dienste muß dann die Dienstekennung, ein Zeiger auf die Eingangsstruktur, die Größe der Eingangsstruktur, ein Zeiger auf eine Ausgangsstruktur, die Größe der Ausgangsstruktur und einen Zeiger auf die Größe der zurückgegebenen Daten mitgegeben werden.
Der Aufruf von „DeviceIOControl(..)“ liefert einen Ergebniswert zurück. Ein Ergebniswert von 0 zeigt die ordentliche Abarbeitung an. Ansonsten kann die Nummer des aufgetretenen Fehlers mit der Funktion „GetLastError()“ erfragt werden. Die Fehlernummern sind in „winerror.h“ definiert.

Beispiel:

```
DWORD dwResult;
DWORD dwDIOC_count;     // count of returned bytes of DeviceIoControl
PCIVME_ACCESS_COMMAND sAccess_parameter;

sAccess_parameter.bAddressModifier      = Short_NoPriv;
sAccess_parameter.bAccessType           = WORD_ACCESS;
sAccess_parameter.bIncrement            = WORD_ACCESS;

dwResult = DeviceIoControl(hHandle, PCIVME_SET_ACCESS_PARA,
                &sAccess_parameter, sizeof(sAccess_parameter), NULL,
                        0, &dwDIOC_count, NULL);

if (!dwResult)
        printf("PCIVME_SET_ACCESS_PARA error: 0x%08x\n",GetLastError());
```

Es wird der Dienst „PCIVME_SET_ACCESS_PARA“ aufgerufen.
Die Eingangsstruktur besteht hier aus der Struktur „sAccess_parameter“. Vor dem Aufruf wird die Struktur ausgefüllt. Es wird der Zeiger auf die Struktur übergeben.
Die Größe der Eingangsstruktur ist „sizeof(sAccess_parameter)“.
Eine Ausgangsstruktur wird nicht zurückerwartet. Daher die erwartete Länge „0“ und der Zeiger auf die Ausgangsstruktur „NULL“ übergeben.
In der Variablen „dwDIOC_count“ legt der Treiber die Größe der wirklich zurückgelieferten Daten ab. (Dies sollte hier 0 sein.)
Wenn der Rückgabewert gleich „0“ ist, dann kann mit „GetLastError()“ die zugehörige Fehlernummer erfragt werden. Die Fehlernumern sind in „winerror.h“ festgelegt.

5. DIE DIENSTE

Hinweis: Die verwendeten Konstanten sind in der Datei „pcivme.h“ definiert.

5.1. Der Dienst „PCIVME_INIT_HARDWARE“ initialisiert eine Hardware eines speziellen VMEMM Interface. In der Regel reicht die Standard Initialisierungsfolge des Treibers aus. Der Benutzer kann jedoch ein terminiertes Array von Initialisierungen (PCIVME_INIT_COMMAND) der Standard Initialisierung hinzufügen. Die Terminierung des Arrays muß mit der STOP Konstante enden.
Beispiel:

```
PCIVME_INIT_COMMAND sUserInitStruct = {2, {{VIC,  BYTE_ACCESS, VIVR1, 200},
                                           {VIC,  BYTE_ACCESS, VIVR2, 202},
                                           {VIC,  BYTE_ACCESS, VIVR3, 203},
                                           {VIC,  BYTE_ACCESS, VIVR4, 204},
                                           {VIC,  BYTE_ACCESS, VIVR5, 205},
                                           {VIC,  BYTE_ACCESS, VIVR6, 206},
                                           {VIC,  BYTE_ACCESS, VIVR7, 207},
                                           {STOP, WORD_ACCESS, 0,     0}}};
```

Hinweis: Keine zusätzliche Initialisierung besteht nur aus einem Element mit der STOP Konstante als Inhalt.

Hinweis: Die Angabe des Strukturelements „dwInterfaceNumber“ ist ohne Bedeutung, wurde jedoch aus Kompatibilität zum WINDOWS-95 Treiber beibehalten.

Hinweis: Durch diesen Dienst wird die Einstellung der Hardware verändert. Daher kann dieser Dienst sollte nur verwendet werden, solange nur ein einziger Pfad auf das Interface geöffnet ist.

5.2. Mit Hilfe des „PCIVME_DEINIT_HARDWARE“ wird das spezielle PCIADA Interface mit seinem angeschlossenen VMEMM Interface deinitialisiert. Auch hier kann eine benutzerdefinierte Deinitialisierung der Standard Deinitialisierung hinzugefügt werden. Der Mechanismus ist der Gleiche wie bei der Initialisierung.

Hinweis: Durch diesen Dienst wird die Einstellung der Hardware verändert. Daher kann dieser Dienst verwendet werden, wenn nur noch ein einziger Pfad auf das Interface geöffnet ist.

5.3. Mit Hilfe des Dienstes „PCIVME_SET_ACCESS_PARA“ können die zusätzlichen Zugriffsparameter eines Pfades gesetzt werden. Für jeden Pfad wird getrennt festgelegt, ob der Zugriff die Zugriffsadresse inkrementieren soll, sowie der zugehörige „Address-Modifier“ und die Byte-Größe der folgenden Zugriffe.
Alle nachfolgenden Read(..) und Write(..) Zugriffe benutzen dann diese Einstellungen solange bis eine veränderte Einstellung folgt.

Mit Hilfe des Standard-Aufrufs

```
SetFilePointer(hHandle, dwNextAccessAddress, NULL, FILE_CURRENT);
```

kann die Basis des nächsten Zugriffs mittels Read(..) und Write(..) festgelegt werden.

5.5. Der Dienst „PCIVME_GET_STATIC_STATUS“ fordert statische Informationen zu einem speziellen PCIADA-VMEMM Interface an. Es ist die Struktur „PCIVME_STATIC_STATUS“ zugeordnet.
Hinweis: Hier bestehen Unterschiede zum WINDOWS-95 Treiber.

5.6. Der Dienst „PCIVME_GET_DYNAMIC_STATUS“ fordert im Betrieb sich ändernde Information an. Es ist die Struktur „PCIVME_DYNAMIC_STATUS“ zugeordnet.

5.7. Der Dienst „PCIVME_READ_VECTOR“ liest Informationen zu den Interrupts aus. Als Eingangsstruktur in diesen Dienst ist die Struktur „PCIVME_VECTOR_REQUEST“ festgelegt.

```
typedef struct                // to request vectors from a interface
{
  ULONG   dwInterface;        // here dummy 'cause of compatibility to WIN95..
  USHORT  wRequestCount;      // maximum number of vectors to requests
  BOOLEAN bPoll;              // no blocking allowed - poll always
} PCIVME_VECTOR_REQUEST;
```

Das Strukturelement „wRequestCount“ legt fest, wieviele aufgelaufene Vektoren vom Treiber maximal abgeholt werden sollen. Im Treiber ist für jeden offenen Pfad, dessen Interrupts zugelassen sind, ein Puffer für Vektoren angelegt. Dieser Puffer kann Element für Element oder in Blöcken entleert werden.
Mit Hilfe des Strukturelements „wPoll“ wird festgelegt, ob der Aufruf pollend oder blockierend beenden soll. Der Aufruf blockiert jedoch nur, wenn zum Zeitpunkt des Aufrufs noch kein Vektor (d.h. kein Interrupt) aufgelaufen ist.

Die Ausgangsstruktur „PCIVME_VECTOR_RESPONSE“ enthält alle Informationen nach der Rückkehr aus einem Aufruf. Dabei ist es unabhängig, ob der Aufruf pollend oder blockierend durchgeführt wurde.

```
typedef struct                // the response to the above request
{
  ULONG  dwInterface;         // here dummy 'cause of compatibility to WIN95..
  USHORT wPendingCount;       // represents the number of vectors pending
  USHORT wCount;              // actual delivered count of vectors
  BOOLEAN bOverflow;          // there was a irq overflow @ this channel
  UCHAR  bStatusID;           // base of following vector array
} PCIVME_VECTOR_RESPONSE;
```

Das Strukturelement „wPendingCount“ kennzeichnet die Anzahl der noch anstehenden Vektoren. In „Count“ ist die wirkliche Anzahl der gelieferten Vektoren abgelegt. Bei einer zu hohen Interruptrate zwischen 2 Aufrufen können die internen Vektor-Puffer überlaufen und Vektoren verloren gehen.. Dies wird in „bOverflow“ mitgeteilt. „bStatusID“ ist das erste Element eines Arrays „UCHAR bStatusID[wCount]“. Hier werden in zeitlich sequentieller Reihenfolge die aufgelaufenen Vektoren abgelegt.

Hinweis: Bei blockierender Abfrage muß vor dem Schließen des zugehörigen Pfads der IO-Vorgang beendet werden. (z.B. mit CancelIo(...)).

Hinweis: Da einzelnen Vektoren nicht bestimmten Pfaden zugeordnet werden können, muß der aufrufende Thread die Vektoren nach „seinen eigenen“ untersuchen und alle anderen verwerfen.

Hinweis: Da auch „BUS-Error-„ oder „Spurious-Interrupt-„ Ereignisse einen Interrupt auslösen, werden auch diese in dem Vektor-Puffer gehalten. Hierdurch ist manchmal eine genaue Zuordnung eines „BUS-Errors“ zu einem bestimmten Zugriff nur schwer möglich.

Hinweis: Hier bestehen Unterschiede zum WINDOWS-95 Treiber.

5.8. Mit Hilfe des Dienstes „PCIVME_ACCESS_VIC68A“ kann direkt der VIC68A Baustein eines VMEMM-Interface programmiert und ausgelesen werden. Es ist die Struktur „PCIVME_VIC68A_ACTION“ als Ein- und Ausgangsstruktur zugeordnet. Um die Vielfältigkeit des VIC68A-Bausteins nutzen zu können wurde der direkte Zugriff darauf freigegeben. Mit einem Zugriff kann der übergebene Inhalt gelesen, geschrieben, verodert oder verundet geschrieben und das Ergebnis erneut zurück gelesen werden. Es darf jedoch kein VIC68A Register verändert werden, welches direkten Einfluß auf das Adreß-Modifier Register des VIC68A hat.

5.9. Mit Hilfe des Dienstes „PCIVME_CONTROL_INTERRUPTS“ lassen sich für einzelne Pfade die Interrupts zu oder abschalten. Hierbei werden nicht wirklich die Hardware-Interrupts unterdrückt, sondern es wird nur der Vektor-Puffer zu oder abgeschaltet. Beim Abschalten bleibt der Inhalt des Vektor-Puffers erhalten.

Hinweis: Hier bestehen Unterschiede zum WINDOWS-95 Treiber.

5.10 Mittels des Aufrufs PCIVME_TAS kann ein ununterbrechbarer Lese-Schreib-Zyklus, ähnlich der TAS Instruktion der 68K Prozessoren auf dem VME-BUS ausgelöst werden.

5.11 Der Dienst PCIVME_RESET löst je nach Parameter einen VME-BUS oder einen VME-BUS- und eine VMEMM-Reset aus. Über die Eingangsstruktur „PCIVME_RESET_COMMAND“ wird dieses unterschieden.

```
typedef struct
{
  ULONG  dwInterface;         // here dummy 'cause of compatibility to WIN95
  USHORT wCommand;            // the approriate reset command
} PCIVME_RESET_COMMAND;
```

In der Ausgangsstruktur „PCIVME_RESET_RESULT“ wird das Ergebnis des Reset zurückgegeben:

```
typedef struct
{
  ULONG  dwInterface;         // here dummy 'cause of compatibility to WIN95
  USHORT wResult;
} PCIVME_RESET_RESULT;        // polling result: in progress if (wResult != 0)
```

Hinweis: Nach dem Auslösen eines Reset muß die Hardware neu initialisiert werden. Daher kann der Reset-Dienst nur verwendet werden, solange nur ein einziger Pfad auf das Interface geöffnet ist.

6. INTERRUPT VEKTOREN

Von VMEMM erzeugte oder weitergeleitete Interrupts müssen vektorisiert werden. Per Konvention werden die Interrupt Vektoren so aufgeteilt, daß die Vektoren 0x00..0x3f der internen (d.h. vom Treiber vorbelegten) Nutzung vorbehalten ist. Die Vektoren 0x40..0xff können von Peripherie auf dem VMEbus genutzt werden.
Hinweis: Der von PCIADA generierte Timeout Interrupt wird von der Software auf die Interrupt Vektor Nummer 1 gespiegelt.

Interrupt Ursache	**Vektor Nr.**
PCIADA erzeugter Interrupt (Timeout)	1 (aktiviert)
Clock Tick Interrupt Generator	2
Reset Taster an der Frontplatte	6 (aktiviert)
VMEbus Timeout (Bus-Error)	7 (aktiviert)
Interprocess communication global switch #0	8
Interprocess communication global switch #1	9
Interprocess communication global switch #2	10
Interprocess communication global switch #3	11
Interprocess communication module switch #0	12
Interprocess communication module switch #1	13
Interprocess communication module switch #2	14
Interprocess communication module switch #3	15
ACFAIL asserted	16
Write post Fail	17
Arbitration Timeout	18
SYSFAIL asserted	19
VMEbus Interrupter acknowledge	20

Hinweis: Der von dem Reset Taster an der Front des VMEMM Interface ausgelöste Interrupt kann dazu verwendet werden, eine benutzerdefinierte Reset- oder Initialisierungsfunktion auszulösen.

7. DIE STANDARD INITIALISIERUNG DES TREIBERS

Der Treiber nutzt folgendes Array von PCIVME_INIT_COMMANDs zur Initialisierung des Interface. Wie weiter oben gezeigt kann diese Initialisierung von dem Anwenderprogramm mit Hilfe des Dienstes PCIVME_INIT_HARDWARE ergänzt oder überschrieben werden.

Hinweis: Die verwendeten Konstanten sind in der Datei „pcivme.h“ definiert.

```
{LCR,  WORD_ACCESS, 0x4c, 0x0009}            // disable interrupts
{LCR,  WORD_ACCESS, 0x50, 0x4180}            // enable interface

{VIC,  BYTE_ACCESS, (WORD)0x03, 0xf8+1}    // VIICR

{VIC,  BYTE_ACCESS, (WORD)0x07, 0x78+1}    // VICR1
{VIC,  BYTE_ACCESS, (WORD)0x0b, 0x78+2}
{VIC,  BYTE_ACCESS, (WORD)0x0f, 0x78+3}
{VIC,  BYTE_ACCESS, (WORD)0x13, 0x78+4}
{VIC,  BYTE_ACCESS, (WORD)0x17, 0x78+5}
{VIC,  BYTE_ACCESS, (WORD)0x1b, 0x78+6}
{VIC,  BYTE_ACCESS, (WORD)0x1f, 0x78+7}     // VICR7

{VIC,  BYTE_ACCESS, (WORD)0x23, 0xf8+0}     // DSICR

{VIC,  BYTE_ACCESS, (WORD)0x27, 0xf8+1}     // LICR1
{VIC,  BYTE_ACCESS, (WORD)0x2b, 0xf8+2}
{VIC,  BYTE_ACCESS, (WORD)0x2f, 0xf8+3}
{VIC,  BYTE_ACCESS, (WORD)0x33, 0xf8+4}
{VIC,  BYTE_ACCESS, (WORD)0x37, 0xf8+5}
{VIC,  BYTE_ACCESS, (WORD)0x3b, 0x38+6}
{VIC,  BYTE_ACCESS, (WORD)0x3f, 0x38+7}     // LICR7

{VIC,  BYTE_ACCESS, (WORD)0x43, 0xf8+2}     // ICGS
{VIC,  BYTE_ACCESS, (WORD)0x47, 0xf8+3}     // ICMS

{VIC,  BYTE_ACCESS, (WORD)0x4b, 0xe8+6}     // EGICR

{VIC,  BYTE_ACCESS, (WORD)0x4f, 0x08}        // ICGS-IVBR (!)
{VIC,  BYTE_ACCESS, (WORD)0x53, 0x0c}        // ICMS-IVBR (!)

{VIC,  BYTE_ACCESS, (WORD)0x57, 0x00}        // LIVBR (!)

{VIC,  BYTE_ACCESS, (WORD)0x5b, 0x10}        // EGIVBR (!)
```

```
    {VIC,  BYTE_ACCESS, (WORD)0x5f, 0x00}       // ICSR

    {VIC,  BYTE_ACCESS, (WORD)0x63, 0x00}       // ICR0
    {VIC,  BYTE_ACCESS, (WORD)0x67, 0x00}
    {VIC,  BYTE_ACCESS, (WORD)0x6b, 0x00}
    {VIC,  BYTE_ACCESS, (WORD)0x6f, 0x00}
    {VIC,  BYTE_ACCESS, (WORD)0x73, 0x00}       // ICR4

    {VIC,  BYTE_ACCESS, (WORD)0x83, 0xfe}       // VIRSR

    {VIC,  BYTE_ACCESS, (WORD)0x87, 0x0f}       // VIVR1
    {VIC,  BYTE_ACCESS, (WORD)0x8b, 0x0f}
    {VIC,  BYTE_ACCESS, (WORD)0x8f, 0x0f}
    {VIC,  BYTE_ACCESS, (WORD)0x93, 0x0f}
    {VIC,  BYTE_ACCESS, (WORD)0x97, 0x0f}
    {VIC,  BYTE_ACCESS, (WORD)0x9b, 0x0f}
    {VIC,  BYTE_ACCESS, (WORD)0x9f, 0x0f}       // VIVR7

    {VIC,  BYTE_ACCESS, (WORD)0xa3, 0x3c}       // TTR  - 16 usec

    {VIC,  BYTE_ACCESS, (WORD)0xb3, 0x40}       // ARCR
    {VIC,  BYTE_ACCESS, (WORD)0xb7, 0x29}       // AMSR
    {VIC,  BYTE_ACCESS, (WORD)0xd3, 0x00}       // RCR

    {IFR,  LONG_ACCESS, (WORD)ADRHL, 0xF0F0F0F0}   // ADR-H, ADR-L
    {IFR,  WORD_ACCESS, (WORD)CSR  , 0x0000}  // Contr-Reg

    {VIC,  BYTE_ACCESS, (WORD)0x7f, 0x80}       // ICR7

    {LCR,  WORD_ACCESS, 0x4c, 0x0009}           // disable interrupts

    {STOP, WORD_ACCESS, 0,     0}
```

8. DIE STANDARD DEINITIALISIERUNG DES TREIBERS

Die eingebaute Deinitialisierung des Treibers erfolgt in zwei Schritten. Zwischen diesen beiden Schritten wird die Deinitialisierungs Sequenz des Benutzers eingefügt.

Erste Deinitialisierungs Sequenz:

```
{VIC,  BYTE_ACCESS, (WORD)0x7f, 0x00},      // ICR7 - set SYSFAIL
{LCR,  WORD_ACCESS, 0x4c, 0x0009},          // disable interrupts
{STOP, WORD_ACCESS, 0,    0}};
```

Zweite Deinitialisierungs Sequenz:

```
{LCR,  WORD_ACCESS, 0x50, 0x4080},          // disable interface
{STOP, WORD_ACCESS, 0,    0}};
```

9. SONSTIGES

Es empfiehlt sich innerhalb einer Applikation für jeden genutzten Adressbereich des VME-BUS einen eigenen Pfad zu öffenen. Dann müssen für jeden Pfad nur einmal die Zugriffsparameter gesetzt werden. Ebenso kann für jeden Interrupt-Handler ein eigener Pfad geöffnet werden und jeder Pfad wartet mit blockierendem Zugriff auf „seine“ nächsten Vektoren.
Hinweis: Nur der erste geöffnete Pfad darf die Hardware initialisieren und einen VME-RESET auslösen. Auch darf nur der letzte geöffnete Pfad die Hardware deinitialisieren.

10. WEITERE LITERATUR

1. Hardware Handbuch des PCI-VME Interface
2. Programmierhandbuch zum VIC68A der Firma Cypress