PHYTEC

Softwaremanual

CPU- und Modultreiber

für

CPU8051 und CPU167

Version 1.0

Ausgabe Juni 1998

PHYTEC Elektronik GmbH • August-Bebel-Str. 29 • D-07973 Greiz
Telefon: +49(03661)62 79-0 • Telefax: +49(03661)62 79-48
WWW: http://www.phytec.de • E-Mail: info@phytec.de

Im Buch verwendete Bezeichnungen für Erzeugnisse, die zugleich ein eingetragenes Warenzeichen darstellen, wurden nicht besonders gekennzeichnet. Das Fehlen der ® Markierung ist demzufolge nicht gleichbedeutend mit der Tatsache, daß die Bezeichnung als freier Warenname gilt. Ebensowenig kann anhand der verwendeten Bezeichnung auf eventuell vorliegende Patente oder einen Gebrauchsmusterschutz geschlossen werden.

Die Informationen in diesem Handbuch wurden sorgfältig überprüft und können als zutreffend angenommen werden. Dennoch sei ausdrücklich darauf verwiesen, daß die Firma PHYTEC Elektronik GmbH weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgeschäden übernimmt, die auf den Gebrauch oder den Inhalt dieses Handbuches zurückzuführen sind. Die in diesem Handbuch enthaltenen Angaben können ohne vorherige Ankündigung geändert werden. Die Firma PHYTEC Elektronik GmbH geht damit keinerlei Verpflichtungen ein.

PHYTEC Elektronik GmbH
August-Bebel-Straße 29
D-07973 Greiz

1. Auflage Juni 1998

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

1 Treiberbibliotheken für die CPU8051 und CPU167

1.1 Überblick

Dieses Handbuch beschreibt die phyPS-Treiber für die CPU8051 und die CPU167. Die Treiber stellen für beide CPU-Baugruppen ein einheitliches Interface zur Verfügung, so daß die Funktionsbeschreibungen für beide Plattformen Gültigkeit besitzen. Unterschiede bestehen nur bei wenigen Funktionen und werden besonders hervorgehoben.

Die phyPS-Treiber ermöglichen einen einfachen Zugriff auf die Hardware der CPU-Platine und der verschiedenen Ein-/Ausgabemodule. Für die CPU stehen Funktionen zum Zugriff auf die Peripherie und den phyPS-Bus (Modul-Bus) zur Verfügung. Für die Module existieren Funktionen, die den Datenaustausch zwischen CPU und Ein-/Ausgabe-Modul realisieren.

1.2 Schichtenmodell der Treiber

Die Treiberbibliothek für die phyPS-CPU-Module untergliedert sich in zwei Schichten (siehe Bild 1):

- CPU-Treiber
- Modul-Treiber

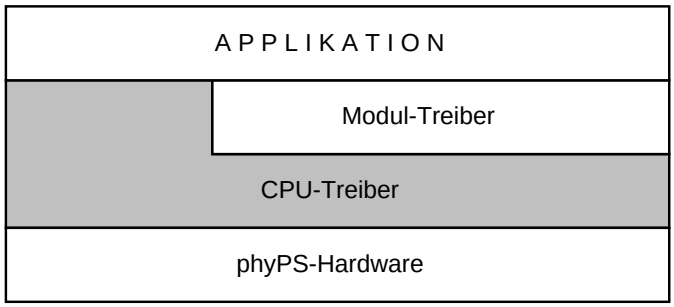

Bild 1: Schichtenmodell der Treiber

Wie in Bild 1 dargestellt, bilden die CPU-Treiber die Grundlage des Treibermodells. Sie stellen Grundfunktionen für den Zugriff auf den phyPS-Bus bereit, die von der darüberliegenden Schicht der Modultreiber für den Zugriff auf die angeschlossenen Module benötigt

werden. Neben diesen internen, nicht exportierten Funktionen, beinhalten die CPU-Treiber noch weitere Funktionen, die dem Anwender den Zugriff auf die Bedienelemente der CPU-Platine gestatten. Diese Funktionen werden im Abschnitt 3 erläutert.

Die Modultreiber ermöglichen dem Anwender einen einfachen Zugriff auf die am phyPS-Bus angeschlossenen Module. Sie kapseln das für den Datenaustausch notwendige Signalspiel auf dem phyPS-Bus, individuelle Moduleigenschaften und interne Datenformate. Dem Anwender wird es somit möglich, den kompletten Datenaustausch mit einem Ein-/Ausgabemodul durch einen einzigen Treiberaufruf abzuwickeln. Die Funktionen der Modultreiber werden im Abschnitt 4 erläutert. Für den Zugriff auf die Module bauen die Modultreiber auf Funktionen der CPU-Treiber auf.

1.3 Unterstützung von Speichermodellen

Die Bibliotheken mit den Treiberfunktionen sind unabhängig vom verwendeten Speichermodell (Small, Medium, Large). Die Übergabe von Parametern bzw. Returnwerten erfolgt stets in Registern des Prozessors. Durch die explizite Angabe von Speicherbezeichnern (XDATA, FAR) werden die Default-Einstellungen des Compilers für den Aufruf der Treiberfunktionen modifiziert, so daß stets ein korrekter Code generiert wird.

1.4 Einbindung der Treiber

Die beiden Treiberschichten für CPU- und Modultreiber werden in einer gemeinsamen Bibliothek mit der Bezeichnung PPSDRVxx.LIB ausgeliefert, wobei "xx" durch die Kennung der jeweiligen CPU zu ersetzen ist:

- PPSDRV51.LIB
- PPSDRV67.LIB

Diese Bibliothek ist mit in das Steuerfile für den Linker aufzunehmen (Einbeziehung in die Liste der zu linkenden Module).

Weiterhin sind die Treiberfunktionen dem C-Compiler bzw. Assembler durch die Einbindung der entsprechenden Header- bzw. Include-Dateien bekannt zu geben. Es ist darauf zu achten, daß stets die Definitionsdateien für CPU- und Modultreiber einzubinden sind. Die Zuordnung der Header- bzw. Include-Dateien zeigt Tabelle 1.

Entwicklungumgebung	CPU8051	CPU167
C166	CPUDRV51.H PPSDRV51.H PHYPSID.H	CPUDRV67.H PPSDRV67.H PHYPSID.H
A166	CPUDRV51.INC PPSDRV51.INC PHYPSID.INC	CPUDRV67.INC PPSDRV67.INC PHYPSID.INC

Tabelle 1: Zuordnung der Definitionsdateien

Die Definitionsdateien besitzen folgende Bedeutung:

CPUDRVxx: Definitionsdatei für CPU-Treiber
PPSDRVxx: Definitionsdatei für Modultreiber
PHYPSID: Festlegung der Modulkennbytes für die phyPS-Module (siehe Abschnitt 2.3)

1.5 Besonderheiten bei Verwendung der CPU154

Bei der CPU154 werden die höheren Adreßleitungen für RAM und Flash (A15 und A16 für RAM und Flash) über ein Latch auf der CPU-Platine beeinflußt, um so mit Hilfe von Bank-Switching jeweils 128k verwalten zu können (siehe Hardwaremanual zur CPU). Gleichzeitig werden aber über zwei weitere Bits desselben Latches auch die beiden LED's an der Frontplatte gesteuert. Da das Latch selbst jedoch nicht rücklesbar ist, dupliziert der CPU-Treiber beim Schalten einer LED den aktuellen Zustand des Latches in einer DATA-Variablen. Um ein unbeabsichtigtes Umschalten von Speicherbänken beim Setzen einer LED zu vermeiden, sind dem CPUDRV51 die aktuellen Zustände der o.g. Adreßleitungen bekannt zu geben. Dies erfolgt mit Hilfe der Funktion

void PPSSetAddrLatch (BYTE AddrLatchValue);

Die Bitstellen des Übergabeparameters haben die in Tabelle 2 angegebene Bedeutung. Beim Aufruf von **PPSSetAddrLatch** wird der übergebene Wert gleichzeitig das Latch sowie dessen Duplikat im DATA geschrieben.

Bitstelle	Bedeutung
Bit0	Adreßleitung A15 für RAM
Bit1	Adreßleitung A16 für RAM
Bit2	Adreßleitung A15 für Flash
Bit3	Adreßleitung A16 für Flash
Bit4	-
Bit5	-
Bit6	RUN-LED (0=ON, 1=OFF)
Bit7	ERROR-LED (0=ON, 1=OFF)

Tabelle 2: Bitstellen des Adreßlatch der CPU154

Es ist erforderlich, vor dem Aufruf der Funktion **PPSInitialize** das Adreßlatch mit Hilfe von **PPSSetAddrLatch** auf einen definierten Wert zu setzen.

Es ist empfehlenswert, die Einstellung des Latches - und damit letztendlich des Speichermodells - bereits im Startup vorzunehmen, um somit noch vor dem Ansprung der Main-Funktion die gewünschte RAM-Bank einzublenden. Es ist jedoch darauf zu achten, daß sämtlicher, vor dem Aufruf von **PPSSetAddrLatch** abzuarbeitender Code im Bereich unterhalb 0x8000 anzusiedeln ist. Erreicht werden kann dies durch explizite Adreßvorgabe für die betreffenden Segmente. Das folgende Beispiel demonstriert die Einbindung des Aufrufes der Funktion **PPSSetAddrLatch** in den Startup und zeigt das zugehörige Linker-Steuerfile (mit dem Wert #0C5H werden jeweils ein linearer Bereich von 64k für RAM und Flash selektiert sowie die beiden LED's ausgeschalten):

Erweiterung des Startup:

```
RSEG    ?STARTUP?START

?C_STARTUP: ljmp    SetDecoderLatch

RSEG    ?STARTUP?INIT

;------------------------------------------------
; Speichermodell für die CPU154 setzen:
;
; Flash:   Segment0 = 0000..7FFFF
;          Segment1 = 8000..FFFFF
; RAM:     Segment0 = 0000..7FFFF
;          Segment1 = 8000..FFFFF
;------------------------------------------------

SetDecoderLatch:
            CLR     EA          ;Disable Ints
            MOV     R7,#0C5h    ;A16F=0 A15F=1,
                                ;A16R=0, A15R=1
            CALL    _PPSSetAddrLatch
            ljmp    STARTUP1

RSEG    ?C_C51STARTUP?STARTUP

STARTUP1:
```

zugehöriges Linker-Steuerfile:

```
ppsdemo.obj, ppsdrv51.lib, startup.obj NOOL &
  DA(24h) &
  XD(0C000h) &
  CO(08100h, ?STARTUP?START(08000h),
           ?STARTUP?INIT(080A0h),
           ?PR?_PPSSETADDRLATCH?CPUDRV(080B0h))
```


2 Grundlagen

2.1 Ablage der Moduldaten

Die phyPS-Modultreiber verwalten die Moduldaten grundsätzlich im RAM (XDATA beim 8051). Der zur Aufnahme der Prozeßdaten benötigte Speicherbereich ist vom Anwenderprogramm zur Verfügung zu stellen. Den Treiberfunktionen ist beim Aufruf jeweils ein Zeiger auf den Anfang des zum Modul gehörigen Speicherbereiches zu übergeben. In Bild 2 ist der Aufbau eines Prozeßabbildes am Beispiel dargestellt.

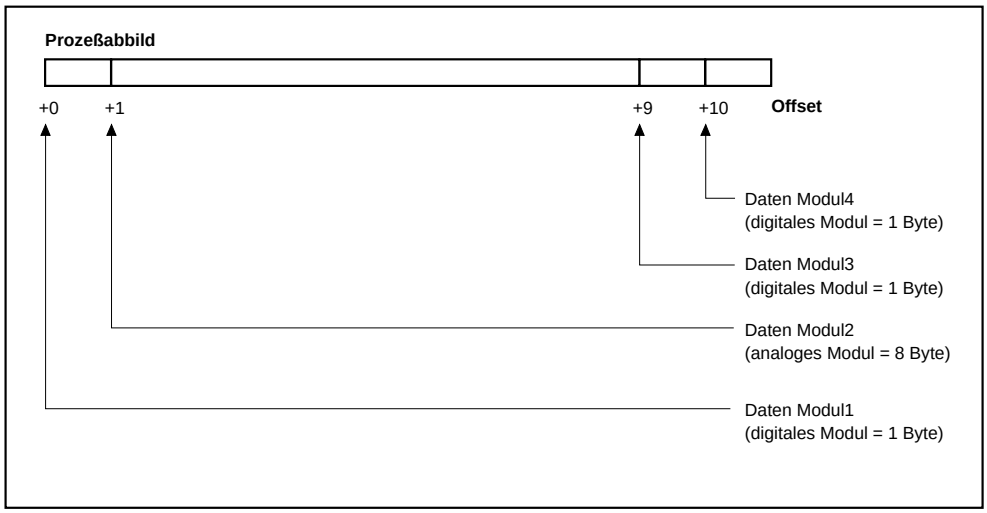

Bild 2: Aufbau Prozeßabbild

Die Speicherbereiche für die Moduldaten können beliebig angeordnet werden. Es ist aber darauf zu achten, daß die zur Aufnahme des Prozeßabbildes notwendige Größe reserviert wird. Die von einem Modul benötigte Anzahl von Bytes ist dem jeweiligen Modulhandbuch zu entnehmen (digitale Module typ. 1 Byte, analoge Module typ. 8 Bytes).

Die Tabelle 3 enthält die Prozeßabbildgrößen für die wichtigsten phyPS-Module. Die Anzahl der vom jeweiligen Modul belegten Bytes läßt sich auch über die in Abschnitt 6 beschriebene Funktionalität der Modultreiber ermitteln.

Modul	Bezeichnung	Daten-Bytes
DI 024.8	phyPS-204	1
DI 024.4	phyPS-205	1
DOR 250.4	phyPS-206	1
DOT 024P.8	phyPS-207	1
CT 024.2	phyPS-209	4
AIV 005U.4	phyPS-301	8
AIV 005B.4	phyPS-302	8
AIV 010U.4	phyPS-303	8
AIV 010B.4	phyPS-304	8
AOV 005U.4	phyPS-307	8
AOV 005B.4	phyPS-308	8
AOV 010U.4	phyPS-309	8
AOV 010B.4	phyPS-310	8
AIC 020U.4	phyPS-305	8
AIC 420U.4	phyPS-306	8
AOC 020U.4	phyPS-311	8
AOC 420U.4	phyPS-312	8
AIM 050U.4	phyPS-316	8
PT100 2.4	phyPS-313	8

Tabelle 3: Datenbreite wichtiger phyPS-Module

Für den vereinfachten Zugriff auf die im Speicher abgelegten Daten des Zählermoduls und von analogen Modulen stehen die Funktionen **SetChannel** und **GetChannel** zur Verfügung. Diese ermöglichen die Verarbeitung der intern als 2 Byte dargestellten Zähler-, AD- und DA-Wandler-Werte im int- oder WORD-Format.

2.2 Datenübernahme durch STROBE-Signal

Die phyPS-Module besitzen interne Datenpuffer (Latches), die eine Entkopplung der Modulklemmen vom Bus bewirken. Zur Übernahme der aktuellen Daten in die Latches bzw. an die Modulklemmen ist ein

Strobe-Signal notwendig. Der Anwender hat damit die Möglichkeit, zu Beginn des Lesezyklus die an den Eingangsklemmen aller Eingabemodule liegenden Informationen synchron in die internen Puffer zu übertragen. Beim nachfolgend sequentiellen Auslesen der Module werden die in den Latches gespeicherten Daten übergeben. Obwohl der Aufruf der Treiberfunktionen nur zeitlich versetz möglich ist, kann mit Hilfe des Strobe-Signales der externe Zustand als konsistente "Momentaufnahme" festgehalten und im Speicher abgebildet werden. Gleiches gilt in analoger Form für die Ausgabemodule. Hier werden ebenfalls die an ein Modul übergebenen Daten zunächst im internen Latch abgelegt. Erst die nächste Aktivierung des Strobe-Signals führt zu einer Übernahme der in den Latches enthaltenen Informationen an die Ausgabeklemmen. Zur Aktualisierung der Moduldaten ist also stets ein Strobe-Signal erforderlich.

Der Aufruf einer Treiber-Funktion ohne die Angabe weiterer Optionen im Befehls-Parameter (siehe Abschnitt 4.1) führt zu einer internen Aktivierung des Strobe-Signals, so daß stets die aktuellen Eingangsinforamtionen gelesen werden, bzw. die Ausgänge beim Verlassen der Treiber-Funktion die gewünschten Werte annehmen. Um die Vorteile der gezielten Datenübernahme zu definierten Zeitpunkten nutzen zu können, existiert ein Schalter für den Treiberaufruf (**NO_STROBE**, siehe Abschnitt 4.1), der die interne Strobe-Aktivierung unterdrückt. Für die Generierung des Strobe-Signals ist in diesem Fall der Anwender selbst verantwortlich, dem hierfür die Funktion

void PPSSetStrobe (void)

zur Verfügung steht. Für den Ablauf von Programmen, die von der Unterdrückung des intern generierten Strobe-Signals Gebrauch machen, hat die Pufferung der Moduldaten in Latches folgende Auswirkungen:

- Die Funktion **PPSSetStrobe** ist in jedem Zyklus mindestens einmal aufzurufen, damit die aktuellen Eingangsdaten übernommen werden und die Ausgangsklemmen die im vorangegangenen Zyklus ausgegebenen Werte übernehmen.

- Beim Warten auf einen bestimmten Eingabewert ("Schalter X = Ein") innerhalb einer Schleife ist die Funktion **PPSSetStrobe** mit in die Schleife einzubeziehen, da sonst fortlaufend die einmal im Latch enthaltenen Daten zurückgeliefert werden, unabhängig vom aktuellen Eingangswert.

- Die Funktion **PPSSetStrobe** ist unmittelbar nach dem Abschluß der Initialisierung aufzurufen, um die Ausgabemodule in ihren definierten Anfangswert zu setzen (Der Initialisierungszweig der Treiberfunktion setzt die Ausgänge auf den Wert 0, so daß z.B. Relais abfallen).

2.3 Modulkennung

Alle phyPS-Ein-/Ausgabemodule besitzen eine eindeutige Hardwarekennung, die als Modulkennbyte bezeichnet wird. Anhand dieser Kennung kann die Software die am phyPS-Bus angeschlossenen Module identifizieren. Die Abfrage des Modulkennbytes erfolgt mit Hilfe der Funktion **PPSGetModulID** (siehe Abschnitt 3). Die von den pyhPS-Modulen verwendeten Kennungen sind in PHYPSID.H definiert.

Durch Abfrage der Modulkennbytes kann vor der Ausführung eines Programmes zunächst geprüft werden, ob alle erforderlichen Module vorhanden und einsatzbereit sind. Weiterhin bildet die Modulkennung die Grundlage für die im Abschnitt 6 beschriebene dynamische Systemkonfiguration.

3 CPU-Treiberfunktionen

Die CPU-Treiber stellen eine Initialisierungsfunktion für die gesamte Hardware (CPU- und I/O-Module) bereit und ermöglichen den Zugriff auf die Bedienelemente der CPU-Platine. Zusätzlich enthalten die CPU-Treiber intern noch weitere Kern-Funktionen, diese werden aber nur von den Modultreibern für den Zugriff auf den phyPS-Bus benötigt. Außerdem sind diese stark hardwareabhägig, so daß sie hier nicht näher erläutert werden.

Funktion: PPSInitialize

Syntax: WORD PPSInitialize (void)

Input: ---

Output: WORD = Versionsnummer des Treibers

Verwendung: Initialisierung der Hardware für Zugriff auf Module und deaktivieren des Signals Bus-Reset

Bemerkung: Diese Funktion führt nur eine grundlegende Initialisierung der CPU durch, sie ersetzt jedoch nicht den Aufruf der Modultreiber-Funktionen mit dem Parameter (INIT_MODUL, siehe Abschnitt 4.1).

Diese Funktion muß vor der Benutzung aller anderen Treiberfunktionen (mit Ausnahme der Funktion PPSSetAddrLatch bei Verwendung ders CPUDRV51, siehe dazu Abschnitt 1.5) aufgerufen werden. Erst danach ist der Aufruf aller anderen Funktion möglich !

Funktion: PPSSetStrobe

Syntax: void PPSSetStrobe (void)

Input: ---

Output: ---

Verwendung: Aktivierung des Strobe-Signales auf dem Modul-Bus. Mit diesem Signal übernehmen alle Eingabemodule die an den Eingangsklemmen liegenden Informationen in die internen Latches, alle Ausgabemodule setzen ihre Ausgangsklemmen auf die in den internen Latches enthaltenen Werte (siehe auch Abschnitt 2.2).

Funktion: PPSGetModulID

Syntax: BYTE PPSGetModulID (BYTE ModNumber)

Input: ModNumber = Nummer des Moduls, dessen Kennung gelesen werden soll

Output: BYTE = Modul-Kennbyte

Verwendung: Abfrage des Kennbytes eines auf der angegebenen Adresse befindlichen Moduls. Das Kennbyte identifiziert den Modutyp (z.B. ID=081h -> digitales Ausgabemodul 8 Kanäle, siehe auch Abschnitt 2.3). Die zu übergebende Modul-Nummer ist die an den Hex-Codierschlaten auf dem jeweiligen Modul eingestellte Nummer bzw. Adresse.

Funktion: PPSGetSwitch

Syntax: BYTE PSGetSwitch (void)

Input: ---

Output: BYTE = Zustand des Run/Stop-Schaltes: SWITCH_RUN, SWITCH_STOP, SWITCH_MRES

Verwendung: Abfrage des Run/Stop-Schalters an der Frontplatte der CPU (bei Verwendung als SPS reserviert zur Festlegung des Betriebszustandes, ohne Programmiersystem jedoch beliebig verwendbar).
Zur Auswertung sind folgende vordefinierte Konstanten zu verwenden:

SWITCH_STOP	Schalter in Stellung "STOP" (Mittelstellung)
SWITCH_RUN	Schalter in Stellung "RUN"
SWITCH_MRES	Schalter in Stellung "MRES"

Funktion: PPSSetRunLED
PPSSetErrLED

Syntax: void PPSSetRunLED (BYTE State)
void PPSSetErrLED (BYTE State)

Input: State = Zustand der LED (ON/OFF)

Output: ---

Verwendung: Ein- bzw. Ausschalten der Run- oder Error-LED an der Frontplatte der CPU. Als Parameter sind die vordefinierten Konstanten **ON** und **OFF** verwendet werden.

Funktion: PPSGetHexNumber

Syntax: BYTE PPSGetHexNumber (void)

Input: ---

Output: BYTE = an Hex-Codierschaltern eingestellter Wert

Verwendung: Abfrage der an den Hex-Codierschaltern auf der CPU-Platine eingestellte Nummer (bei Verwendung als SPS reserviert für CPU-Adresse, ohne Programmiersystem jedoch beliebig verwendbar).

Funktion: PPSGetDipSwitch

Syntax: BYTE PPSGetDipSwitch (void)

Input: ---

Output: BYTE = am DIP-Schalter eingestellter Wert

Verwendung: Abfrage des am DIP-Schalter auf der CPU-Platine eingestellten Wertes (bei Verwendung als SPS reserviert zur Einstellung der Baudrate für CAN, ohne Programmiersystem jedoch beliebig verwendbar).

4 Modul-Treiberfunktionen

Die Modultreiber ermöglichen dem Anwender einen einfachen Zugriff auf die am phyPS-Bus angeschlossenen Module. Sie kapseln das für den Datenaustausch notwendige Signalspiel auf dem phyPS-Bus, individuelle Moduleigenschaften und interne Datenformate. Dem Anwender wird es somit möglich, den kompletten Datenaustausch mit einem Ein-/Ausgabemodul durch einen einzigen Treiberaufruf abzuwickeln.

4.1 Aufruf der Modul-Treiberfunktionen

Die pyhPS-Modultreiber-Bibliothek stellt für jedes phyPS-Modul eine eigene Treiber-Funktion zur Verfügung. Der Funktionsname besteht aus der Modulbezeichnung mit der vorangestellten Buchstabenkette 'PPS'. Die Bezeichnung der Module selbst wird nach folgendem Schlüssel gebildet:

Typ	Spannung/ Strom	Kanäle	Bezeichnung
DI (digital in)	024 (24V)	8	DI024_8
DOT (digital out transist.)	024 (24V)	8	DOT024_8
DOR (digital out relais)	250 250V	4	DOR250_4
AOV (analog out voltage)	010B (0..10V bipolar)	4	AOV010_4
AOC (analog out current)	420U (4..20mA unipolar)	4	AOC420_4

Tabelle 4: Schlüssel zur Bildung der Modulbezeichnungen

Die Treiberfunktionen haben für alle Module eine einheitliche Schnittstelle, so daß der Aufruf stets in derselben Weise erfolgt. Dies ermöglicht es, Programme zu schreiben, die erst zur Laufzeit alle am phyPS-Bus vorhandenen Module bestimmen und deren Treiberfunk-

tionen anschließend über Funktionspointer aufrufen. Die Treiber-Schnittstelle besitzt folgenden Aufbau:

PPSxxx (BYTE ModNum, WORD Cmd, BYTE* pImageData);

ModNum: Nummer des betreffenden Moduls (Die Modul-Nummer ist die an den Hex-Codierschlaten auf dem jeweiligen Modul eingestellte Nummer bzw. Adresse.)

Cmd: Steuerkommando für den Modultreiber. Das Steuerkommando setzt sich aus einem Befehl im Lowbyte (auszuführende Aktion) und einer Option im Highbyte (Art der Datenbehandlung) zusammen, wobei die Option auch entfallen kann (Highbyte dann Null)

Befehle: **INIT_MODUL**
READ_MODUL
WRITE_MODUL

Optionen: **NO_STROBE**
NO_CHECK
RAW_CURRENT
RAW_VALUE

pImageData: Pointer auf Speicherbereich zur Ablage der Prozeßdaten

Für die Befehle stehen die vordefinierten Konstanten INIT_MODUL, READ_MODUL und WRITE_MODUL mit folgender Bedeutung zur Verfügung:

INIT_MODUL: Der Aufruf der Treiberfunktion mit dem Befehl INIT_MODUL dient zur Initialisierung von Modul und Treiber. **Es ist daher zwingend notwendig, den Treiber vor dem ersten Modulzugriff (READ_MODUL bzw. WRITE_MODUL) mit diesem Befehl zurück zu setzen.** Bei Eingabemodulen wird der Speicherbereich für die Prozeßdaten definiert gelöscht, Ausgabemodule werden zudem so initialisiert, daß sich an ihrem Ausgang ein unkritischer Zustand einstellt (Relais abgefallen, Transistoren gesperrt, 0V bei analogen Ausgängen).

READ_MODUL: Auslesen der Latchdaten (siehe Strobe-Signal im Abschnitt 2.2) und Übertragung in den angegebenen Speicherbereich. Falls erforderlich (z.B. analoge Module) werden die Daten von der internen Darstellung auf Byte- bzw. Integerwerte konvertiert.

WRITE_MODUL: Beschreiben der Latches mit aktuellen Daten (siehe Strobe-Signal im Abschnitt 2.2) aus dem angegebenen Speicherbereich. Falls erforderlich (z.B. analoge Module) werden die Daten von Byte- bzw. Integerwerten in die interne Darstellung konvertiert.

Zusätzlich enthält PPSDRVxx.H noch eine Definition für das Kommando **MODUL_INFO.** Dieses Kommando besitzt jedoch eine Sonderfunktion und dient nicht zum Datenaustasch mit dem Modul, sondern veranlaßt den Treiber zur Ausgabe von Information über Moduleigenschaften. Eine Anwendung für dieses Kommando beschreibt Abschnitt 6.

Für die Optionen stehen die vordefinierten Konstanten NO_STROBE, NO_CHECK, RAW_CURRENT und RAW_VALUE mit folgender Bedeutung zur Verfügung:

NO_STROBE: Diese Option führt zur Unterdrückung des intern generierten Strobe-Signals beim Treiber-Aufruf (siehe Strobe-Signal im Abschnitt 2.2).

NO_CHECK: Diese Option übergeht die Überprüfung der Ausgabewerte für analoge Module. Ohne Angabe dieser Option werden die im RAM abgelegten Werte vor der Ausgabe an das analoge Modul auf Einhaltung des zulässigen Wertebereiches überprüft. Bei einer Bereichsüberschreitung wird der Ausgabewert im Prozeßabbild auf seine maximal zulässige Größe begrenzt. Dadurch werden "Treppenspannungen", die beim "Umkippen" der Ausgangsspannung entstehen, vermieden (DA-Wandler verarbeitet nur 12-Bit !). Die Angabe dieser Option verkürzt die Laufzeit der Treiberfunktion und verhindert eine Modifikation der Moduldaten im Prozeßabbild.

RAW_CURRENT: Diese Option besitzt nur für die analogen 4..20mA Ein- und Ausgabemodule Bedeutung. Ohne Angabe dieser Option wird der Grundstrom von 4mA intern von den Eingabedaten subtrahiert bzw. zu den Ausgabedaten addiert, so daß der Anwender nur mit dem verbleibenden Signal-Strom arbeitet. Bei Angabe dieser Option wird die interne Korrektur des Stromes mit dem Wert 4mA unterdrückt.

RAW_VALUE: Diese Option besitzt nur für PT100-Module Bedeutung. Bei Angabe dieser Option wird die Umrechnung des gelesenen AD-Wandlerwertes in die entsprechende Temperatur unterdrückt. Im Prozeßabbild wird dafür der unmittelbare Ausgabewert des AD-Wandlers abgelegt. Dieser Wert entspricht der Spannung, die modulintern über dem PT100 abfällt. Die Angabe dieser Option verkürzt die Laufzeit der Treiberfunktion und ermöglicht die Verwendung eigener Routinen zur Konvertierung (siehe Abschnitt 4.5).

Die Befehle und Optionen stehen als vordefinierte Konstanten zur Verfügung, wobei jeweils ein Befehl bei Bedarf mit ein oder zwei Optionen durch eine Oder-Verknüpfung ergänzt werden kann:

```
INIT_MODUL
WRITE_MODUL | NO_CHECK
WRITE_MODUL | NO_STROBE
READ_MODUL  | NO_STROBE | RAW_CURRENT
```

Beispiele für den Aufruf von Modultreibern:

```
BYTE ImgIn  [IMAGESIZE];  // Prozeßd. Eingabe
BYTE ImgOut [IMAGESIZE]; // Prozeßd. Ausgabe
BYTE Res;                 // ReturnCode Treiber

  // Digitales Eingabemodul auf
  // Adresse 3 initialisieren,
  // STROBE wird intern aktiviert
  Res = PPSDI024_8 (3, INIT_MODUL, ImgIn);

// Analoges Ausgabemodul auf Adresse 5 setzen
  // STROBE wird intern aktiviert,
  // Bereichsüberprüfung der Ausgabewerte wird
  // unterdrückt
  Res = PPSAOV005U_4 (5, WRITE_MODUL | NO_CHECK,
                ImgOut);
```

```
// Analoges Eingabemodul auf Adresse 7 lesen
  // STROBE wird beim Lesen nicht aktiviert
  // Ausgabe-Wert = realer Strom ohne
  // Berücksichtigng des Grundwertes 4mA
  Res = PPSAIC420U_4 (7, READ_MODUL | NO_STROBE |
                RAW_CURRENT, ImgIn);
```

4.2 Rückgabewerte der Modul-Treiberfunktionen

Im File PPSDRVxx.H werden einige Treiberfunktionen in Abhängigkeit des Symbols **PPS_ENABLE_WARNING** mit verschiedenen Rückgabewerten vereinbart. Im Grundzustand der phyPS-Modultreiber ist das Symbol PPS_ENABLE_WARNING nicht definiert. Damit gilt die Standard-Deklaration für die Prototypen der phyPS-Modultreiber, so daß diese ein BYTE als Returnwert zurückgeben. Dadurch sind alle kritische Fehler wie ungültige Befehle oder Hardwarefehler erkennbar, da diese einen Returncode kleiner 0x100 besitzen. Ein solcher Fehler bedeutet, daß auf das Modul nicht zugegriffen werden konnte.

Wir das Symbol PPS_ENABLE_WARNING vom Anwender definiert, erweitert sich der Returncode bestimmter Treiber (siehe Prototypen) auf ein WORD. Dadurch werden auch Warnungen an das aufrufende Programm übergeben. Warnungen besitzen einen Returncode größer als 0x100 und zeigen z.B. einen Überlauf des Ausgabewertes bei analogen Modulen an. Der Treiber begrenzt in einem solchen Fall den Ausgabewert auf seine maximal zulässige Größe, informiert jedoch das aufrufende Programm durch die Warnung von der Modifizierung der Ausgabedaten (Hinweis: Die Begrenzung ist unabhängig von der Deklaration des Symbols PPS_ENABLE_WARNING, für das aufrufende Programm wird sie aber nur bei definiertem Symbol erkennbar).

Der Treiber übergibt in den Prozessorregistern stets einen 16-Bit-Wert als Returncode, wobei kritische Fehler im Lowbyte abgelegt werden, Warnungen dagegen im Highbyte. Bei einer Vereinbarung des Returnwertes im Prototypen als BYTE (Grundzustand), wertet der C-Compiler nur den 8-Bit-Wert im Lowbyte aus, das Highbyte (also

Warnungen) wird ignoriert. Damit ist es möglich, weiterhin folgende Konstruktion für den Aufruf der Treiberfunktionen zu verwenden:

```
// wegen Vereinbarung als BYTE wird nur das
// Lowbyte ausgewertet
BYTE PPSxxx (BYTE ModNr, WORD Command,
        BYTE* ImageData);

Res = PPSxxx(ModNr,Command,&ImageData);
if (Res == PPS_SUCCESSFUL)
  {
  // Zugriff auf das Modul OK, Funktion
  // gewährleistet, evtl. Ausgabewert auf 10.00V
  // begrenzt obwohl Berechnung durch
  // Rundungsfehler 10.01V ergeben hat
  }

else
  {
  // Fehler bei Zugriff auf das Modul, Funktion
  // beeinträchtigt !
  }
```

Die Treiberfunktionen liefern bei ihrer Rückkehr folgende Werte als Error-Codes an das aufrufende Programm zurück:

PPS_SUCCESSFUL	Modulzugriff erfolgreich
PPS_UNKNOWN_COMMAND	unbekannter Befehl (nur INIT_MODUL, READ_MODUL, WRITE_MODUL und MODUL_INFO zulässig)
PPS_INVALID_COMMAND	angegebener Befehl wird von Modul nicht unterstützt (z.B. WRITE_MODUL bei Eingabe-Modulen)

PPS_INVALID_MOD_ID	auf der angegebenen Moduladresse befindet sich ein Modul mit einer anderen als vom Treiber erwarteten Modulkennung
PPS_MOD_ACCESS_ERROR	Hardware-Fehler bei Zugriff auf Modul

Bei definiertem Symbol PPS_ENABLE_WARNING liefert die Treiberfunktionen bei ihrer Rückkehr zusätzlich folgende Werte als Warnings an das aufrufende Programm:

PPS_SUPPRESS_OVERFLOW	der für ein analoges Ausgabemodul übergebene Wert lag außerhalb des zulässigen Bereiches und wurde auf die maximal zulässige Größe begrenzt (Prozeßabbild modifiziert!)

4.3 Anwendung der Treiberfunktionen für Zählermodule

Die Darstellung der Werte von Zählermodulen erfolgt im Word-Format (16Bit ohne Vorzeichen). Ein Zählermodul benötigt somit im Prozeßabbild 4 Bytes zur Übergabe der zwei Kanalwerte. Der Wertebereich der Zählermodule umfaßt 0..65535. Ein Verändern des Zählerstandes ist nur durch ein entsprechendes Signalspiel an den Eingangsklemmen des Zählermoduls möglich, ein softwaremäßiges "Preload" ist nicht vorgesehen.

Zur Vereinfachung des Zugriffs auf die im Prozeßabbild abgelegten Zählerwerte kann die Funktion **GetChannel** verwendet werden (siehe Abschnitt 5).

4.4 Anwendung der Treiberfunktionen für analoge Module

Die Darstellung der Werte analoger Module erfolgt im Interger-Format (16Bit mit Vorzeichen). Ein analoges Modul benötigt somit im Prozeßabbild 8 Bytes zur Übergabe der vier Kanalwerte. Für analoge Module gelten folgende Wertebereiche:

unipolar: 0 ... +4095 = 0x0000 ... 0x0FFF
bipolar: -2048 ... +2047 = 0xF800 ... 0x07FF

Die Treiberfunktion übernimmt die Umwandlung zwischen Integerwert und interner Darstellung des AD- bzw. DA-Wandlers. Die sich aus den jeweiligen Werten ergebende Größe für Strom oder Spannung (Skalierung) hängt vom jeweiligen Modul ab (Endwert, Auflösung). Die Formeln zur Skalierung der Werte können dem jeweiligen Modulhandbuch entnommen werden. Beispielsweise ergibt der Ausgabewert 07FF bei verschiedenen Modulen folgende Werte:

Ausgabemodul -10...+10V -> +10V
Ausgabemodul 0...+10V -> +5V
Ausgabemodul 0...+20mA -> +10mA

Zur Umrechnung der AD-Wandlerwerte in die zugehörige Größe Spannung/Strom bzw. zur Berechnung des erforderlichen DA-Wandlerwertes zum Erreichen einer vorgegebenen Größe Spannung/Strom sind im File PPSDRVxx.H entsprechende Konstanten definiert. Sie stellen den Umrechnungsfaktor in Abhängigkeit von Endwert und Auflösung für das jeweilige Modul dar:

RES_AV005U = Unrechnungsfaktor für Module 0.. +5V
RES_AV005B = Unrechnungsfaktor für Module -5.. +5V
RES_AV010U = Unrechnungsfaktor für Module 0..+10V
RES_AV010B = Unrechnungsfaktor für Module -10..+10V
RES_AC020U = Unrechnungsfaktor für Module 0.. 20mA
RES_AC420U = Unrechnungsfaktor für Module 4.. 20mA
RES_AVM050U = Unrechnungsfaktor für Module 0.. 50mV

Die vordefinierten Konstanten sind wie folgt anzuwenden:

- Umrechnung des gelesenen AD-Wandlerwertes ADC_In in zugehörige Größe Spannung/Strom:

```
float UI;          // Spannung oder Strom (je
                   // nach Modul)
int   ADCIn;       // Wert des AD-Wandlers im
                   // Prozeßabbild

// Umrechnung des gelesenen AD-Wandlerwertes
// ADCIn in Spannung/Strom
UI = (float) (ADCIn * RES_Axxx);
```

- Umrechnung der gegebenen Größe Spannung/Strom in auszugebenden DA-Wandlerwert DAC_Out:

```
float UI;          // Spannung oder Strom (je
                   // nach Modul)
int   DACOut;      // Wert des AD-Wandlers im
                   // Prozeßabbild

// Umrechnung Spannung/Strom in auszugebenden
// DA-Wandlerwert DACOut
DACOut = (int) (UI / RES_Axxx);
```

Die im Prozeßabbild übergebenen Integer-Daten sind immer die gelesenen Werte eines AD-Wandlers bzw. die an einen DA-Wandler auszugebenden Werte. Die Umrechnung in die zugehörige reale (skalierte) Größe Spannung oder Strom ist Aufgabe des Anwenders. Folgende Beispiele sollen den Aufruf der Treiberfunktionen und die Vorgehensweise bei der Umrechnung der Größen verdeutlichen:

```
float Channel0, Channel1;
BYTE  ModData[8];
```



```
// --- Eingabemodul 0..20mA auf Adr. 1 lesen
PPSAIC020U_4 (1, READ_MODUL, &ModData);

Channel0  = (float) GetChannel(&ModData, 0);
Channel0 *= RES_AC020U;   // Normierung
Channel1  = (float) GetChannel(&ModData, 1);
Channel1 *= RES_AC020U;   // Normierung

printf ("Strom Kanal0 [mA]: %.3f\n", Channel0);
printf ("Strom Kanal1 [mA]: %.3f\n", Channel1);


// --- Ausgabemodul -5..+5V auf Adr. 2 schreiben

Channel0 =  3.7;    // auf Kanal0 +3.7V ausgeben
Channel1 = -2.1;    // auf Kanal1 -2.1V ausgeben

Channel0 /= RES_AV005B;   // Normierung
SetChannel (&ModData, 0, (int)Channel0);
Channel1 /= RES_AV005B;   // Normierung
SetChannel (&ModData, 1, (int)Channel1);

PPSAOV005B_4 (2, WRITE_MODUL, &ModData);
```

Beim Aufruf einer Treiberfunktion für ein analoges Ausgabemodul wird bei nicht angegebener Option **NO_CHECK** (siehe Abschnitt 4.1) eine Überprüfung der auszugebenden Werte auf Einhaltung des zulässigen Bereiches vorgenommen. Wird dabei eine Bereichsüberschreitung festgestellt, wird der Ausgabewert auf seine maximal zulässige Größe begrenzt. Diese Begrenzung erfolgt direkt im Prozeßabbild des jeweiligen Moduls, so daß nach dem Treiberaufruf die begrenzten Werte im Prozeßabbild verbleiben. Diese Modifikation ist bei der weiteren Bearbeitung dieser Werte zu beachten !

Für die gültigen Bereiche der Ausgabewerte an den DAC und die verwendeten Maximalwerte gilt Tabelle 5:

Modulart	gültiger Bereich	verwendeter Maximalwert
unipolar	(+) 0000...0FFF	1000...7FFF -> 0FFF (max. pos.) 8000...FFFF -> 0000 (min.)
bipolar	(+) 0000...07FF (-) F800...FFFF	0800...7FFF -> 07FF (max. pos.) 8000...F7FF -> F800 (max. neg.)

Tabelle 5: Begrenzung der Ausgabewerte

Die in Tabelle 5 angegebenen Werte beziehen sich auf die direkten Ausgabewerte an den DA-Wandler.

Zur Vereinfachung des Zugriffs auf die im Prozeßabbild abgelegten Integerwerte kann die Funktion **GetChannel** verwendet werden (siehe Abschnitt 5).

4.5 Anwendung der Treiberfunktionen für PT100-Module

Beim Aufruf der Treiberfunktion für das PT100-Modul wird im Standardmodus (keine Option angegeben) der gelesene Ausgabewert des AD-Wandlers intern in die zugehörige Temperatur konvertiert und im Prozeßabbild übergeben. Die Temperaturangabe erfolgt in Grad Celsius. Die Darstellung der Werte analoger Module erfolgt im Interger-Format und läßt damit nur ganzzahlige Werte zu. Um jedoch eine höhere Auflösung als 1-Grad-Schritte zu ermöglichen, wird die Temperatur vom Treiber mit dem Faktor 10 multipliziert und als 1/10 °C im Prozeßabbild übergeben. Der Integer-Wert 238 im Prozeßabbild entspricht damit einer Temperatur von 23.8°C. Im File PPSDRVxx.H ist für die Umrechnung die Konstante **RES_PT100_2** definiert. Ihre Verwendung sichert korrekte Ergebnisse auch im Fall einer Änderung in späteren Treiberversionen.

```
BYTE  TempData[8];
float Temp;

// --- PT100-Modul auf Adresse 10 lesen
PPSPT1002_4 (10, READ_MODUL, &TempData);

Temp  = (float) GetChannel(&TempData,0);
Temp /= RES_PT100_2;

printf ("Temperatur Kanal0: %.3f°C\n", Temp);
```


Zur Messung der Temperatur wird die modulintern über dem PT100 abfallende Spannung gemessen. Der Ausgabewert des AD-Wandlers ist ein Maß für diese interne Meßspannung (sieh Handbuch zum PT100-Modul). Die Angabe der Option **RAW_VALUE** beim Aufruf der Treiberfunktion für das PT100-Modul (siehe Abschnitt 4.1) unterdrückt die Umrechnung des gelesenen AD-Wandlerwertes durch die Treiberfunktion in die entsprechende Temperatur. Dafür wird der direkte Ausgabewert des AD-Wandlers abgelegt. Die Verwendung der Option **RAW_VALUE** verkürzt die Laufzeit der Treiberfunktion und ermöglicht die Einbindung eigener Konvertierungsroutinen. Dies ist z.B. erforderlich, wenn vom Standard (-50...+600°C) abweichende Temperaturbereiche verarbeitet werden müssen.

Soll von einem PT100-Modul z.B. nur ein Kanal weiterverarbeitet werden, läßt sich durch Angabe der Option **RAW_VALUE** die Laufzeit der Treiberfunktion verkürzen, da die interne Wandlung aller vier Kanäle unterdrückt wird. Außerhalb der Treiberfunktion kann nun durch Aufruf der Funktion **PT1002_Temp** gezielt der Wert des einen zu bearbeitenden Kanals konvertiert werden (siehe Abschnitt 5).

```
BYTE  TempData[8];
WORD  ADCOut;
float Temp;


// --- PT100-Modul auf Adresse 10 lesen

PPSPT1002_4 (10, READ_MODUL | RAW_VALUE,
        &TempData);

// Kanalwert lesen...
ADCOut  = GetChannel (&TempData, 0);
// ...in Temperatur wandeln...
Temp = (float) PT1002_Temp (ADCOut);
// ...und normieren
Temp /= RES_PT100_2;

printf ("Temperatur Kanal0: %.3f°C\n", Temp);
```

4.6 Abfrage der Treiberadresse

Für den Aufruf der Treiberfunktionen gibt es zwei Möglichkeiten:

1. Aufruf über den Funktionsnamen (üblicher Weg des Funktionsaufrufes, in allen Beispielen bisher verendet)
2. Aufruf über Funktionspointer

Der Aufruf einer Treiberfunktion über einen Funktionspointer ermöglicht zum einen eine Vereinfachung der Software, da so z.B. eine Funktion mit verschiedenen Modulen operieren kann, andererseits wird aber auch eine dynamische Systemkonfiguration unterstützt (sie Abschnitt 6).

Eine Bestimmung der Treiberadresse zur Laufzeit ermöglicht die Funktion **PPSGetDrvAddress:**

Funktion: PPSGetDrvAddress

Syntax: tModDrv PPSGetDrvAddress (BYTE ModID)

Input: ModID = Kennbyte des Moduls (repräsentiert den Modultyp), für das die Treiberadresse bestimmt werden soll (die Modulkennbytes sind in PHYPSID.H definiert)

Output: Einsprung-Adresse des Treibers oder NULL bei Fehler

Das folgende Beispiel zeigt das Lesen eines digitalen Eingabemoduls durch eine Treiberaufruf via Pointer:

```
tModDrv fpModDrv;
BYTE   ModData;

// Treiberadresse ermitteln
fpModDrv = PPSGetDrvAddress (ID_DI024_8);

if (fpModDrv != NULL)
  {
  // --- Digitales Eingabemodul auf Adr. 1 lesen
  fpModDrv (1, READ_MODUL, &ModData);
  }
```


5 Treiberfunktionen zur Datenmanipulation

Die Treiberfunktionen zur Datenmanipulation operieren stets mit den im Prozeßabbild hinterlegten Moduldaten. Sie ersetzen daher nicht den Aufruf der Treiberfunktion für das entsprechende Modul.

Funktion: SetChannel

Syntax: void SetChannel (BYTE* pImage-
Data, int Channel, int OutVal)

Input: pImageData = Pointer auf Prozeßdaten des Kanal0
Channel = Kanalnummer (0...3)
OutVal = Int-Wert für DA-Wandler oder Zähler

Output: ---

Verwendung: Ablegen eines Int-Wertes für einen DA-Wandler- oder Zählerkanal im Speicher. Diese Funktion ermöglicht einen einfachen Zugriff auf die Prozeßdaten mit Datentypen der Programmiersprache C (WORD, int). Der Funktionsaufruf bearbeitet nur das im Speicher verwaltete Prozeßabbild und ersetzt daher nicht den Aufruf der Treiberfunktion für das jeweilige Modul !

Funktion: GetChannel

Syntax: int GetChannel (BYTE* pImageData, int Channel)

Input: pImageData = Pointer auf Prozeßdaten des Kanal0
Channel = Kanalnummer (0...3)

Output: int = Int-Wert eines AD-Wandlers oder Zählers

Verwendung: Lesen eines AD-Wandler- oder Zählerkanal als Int aus dem Speicher. Diese Funktion ermöglicht einen einfachen Zugriff auf die Prozeßdaten mit Datentypen der Programmiersprache C (WORD, int). Der Funktionsaufruf bearbeitet nur das im Speicher verwaltete Prozeßabbild und ersetzt daher nicht den Aufruf der Treiberfunktion für das jeweilige Modul !

Funktion: ConvertPT1002

Syntax: void ConvertPT1002 (BYTE* pImageData)

Input: pImageData = Pointer auf Prozeßdaten des Kanal0

Output: Die Prozeßdaten werden durch die gewandelten Temperatur-Werte überschrieben

Verwendung: Konvertierung der Ausgabe-Werte des AD-Wandlers für alle vier Kanäle eines PT100-Moduls (2-Leiter-Technik) in die entsprechenden Temperaturwerte. Die konvertierten Werte überschreiben dabei die Ausgabe-Werte des AD-Wandlers im Speicher. Die Ausgabe erfolgt in 1/10 °C, so daß der Integer-Wert 238 einer Temperatur von 23.8°C entspricht. Zur Normierung kann die in PPSDRVxx.H definierte Konstante **RES_PT100_2** verwendet werden.

Funktion: PT1002_Temp

Syntax: int PT1002_Temp (WORD ADCIn)

Input: ADCIn = vom AD-Wandler gelesener Wert

Output: int = Temperatur in 1/10 °C

Verwendung: Konvertierung des Ausgabe-Wertes eines PT100-Kanals (2-Leiter-Technik) die zugehörige, linearisierte Temperatur. Die Ausgabe erfolgt in 1/10 °C, so daß der Integer-Wert 238 einer Temperatur von 23.8°C entspricht. Zur Normierung kann die in PPSDRVxx.H definierte Konstante **RES_PT100_2** verwendet werden.

6 Dynamische Systemkonfiguration

In der Praxis ergeben sich häufig Einsatzfälle, in denen eine dynamische Systemkonfiguration gewünscht wird. Dies kann zum einen für eine Vereinheitlichung und Modularisierung der Software von Vorteil sein, andererseits kann aber damit auch eine flexible Reaktion auf zur Laufzeit vorgefundenen Module erreicht werden. Als Grundlage für eine dynamische Systemkonfiguration dient die im Abschnitt 2.3 beschriebene Hardwarekennung, die eine eindeutige Identifizierung der am phyPS-Bus befindlichen Module ermöglicht.

Zum Abfragen der Hardwarekennung steht die Funktion **PPSGetModulID** (siehe Abschnitt 3) zur Verfügung. Werden dieser Funktion innerhalb einer Schleife als Aufrufparameter alle Moduladresse von 1 bis 254 übergeben (0 und 255 sind reserviert), lassen sich alle über den phyPS-Bus ansprechbaren Module bestimmen. Eine Moduladresse ist dann besetzt, wenn **PPSGetModulID** ein Modulkennbyte ungleich 255 (0xFF) zurückliefert. Damit läßt sich in einem ersten Schritt eine Tabelle aufbauen, in der zu jeder benutzten Moduladressen das zugehörige Modulkennbyte abgelegt wird. In einem zweiten Schritt können mit Hilfe der Funktion **PPSGetDrvAddress** (siehe Abschnitt 4.6) die Adressen der benötigten Treiberfunktionen ermittelt werden, indem diese Funktion für jeden vorhandenen Modultyp (für jedes gefundene Modulkennbyte) aufgerufen wird.

```
typedef strct
{
  BYTE     ModAddr;      // Moduladresse
  BYTE     ModID;        // Modulkennbyte
  tModDrv  fpModDrv;     // Treiberadresse

} tModInf;

tModInf  ModTab[16];      // Modultabelle
BYTE     i,j;

// Modultabelle initialisieren
for (i=0; i<16; i++)
```

```
   {
   ModTab[i].ModAddr  = 0;
   ModTab[i].ModID    = 0;
   ModTab[i].fpModDrv = NULL;
   }

// alle verfügaben Module bestimmen
j = 0;
for (i=0x01; i<0xFF; i++)
   {
   if ( PPSGetModulID(i) != 0xFF )
      {
      // Modul in Tabelle eintragen
      ModTab[i].ModAddr  = i;
      ModTab[i].ModID    = PPSGetModulID(i);
      }
   }

// Aufrufadressen der benötigen Treiber bestimmen
for (i=0; i<16; i++)
   {
   if (ModTab[i].ModAddr != 0)
      {
      fpMod = PPSGetDrvAddress(ModTab[i].ModID);
      ModTab[i].fpModDrv = fpMod;
      }
   }
```

Wie bereits im Abschnitt 2.3 erläutert, dient das Modulkennbyte zur eindeutigen Identifizierung eines Ein-/Ausgabemoduls. Das Modulkennbyte selbst enthält jedoch keine direkt verwertbaren Angaben über das Modul, das es repräsentiert. Dafür lassen sich aber alle relevanten Informationen vom Treiber des Moduls erfragen. Ist also nur das Kennbyte eines Moduls bekannt, muß zunächst mit Hilfe der Funktion **PPSGetDrvAddress** (siehe Abschnitt 4.6) die Adresse des zugehörigen Treibers ermittelt werden.

Zur Abfrage von Moduleigenschaften wird die in PPSDRVxx.H definierte Struktur **tModInf** verwendet:

```
typedef struct
{
  BYTE   m_bModID;
  BYTE   m_bImageSize;
  BYTE   m_bChannelAccess;
  BYTE   m_bModType;
  WORD   m_wCmdMask;
  float  m_rScalar;

} tModInf;
```

Um die Moduleigenschaften vom Treiber zu erfragen, ist der Modultreiber mit dem Kommando **MODUL_INFO** aufzurufen (siehe Abschnitt 4.1). Als Parameter *pImageData* ist dabei statt des sonst anzugebenden Pointers auf die Prozeßdaten ein Pointer auf die Struktur *tModInf* zu übergeben. Für das Anlegen einer Variablen vom Typ der Struktur *tModInf* ist der Anwender verantwortlich. Die einzelnen Elemente der Struktur *tModInf* haben folgende Bedeutung:

m_bModID	Modulkennbyte
m_bImageSize	Anzahl der zur Ablage der Prozeßdaten benötigten Bytes im Prozeßabbild
m_bChannelAccess	Bit0..4 = Anzahl der vom Modul zur Verfügung gestellten Kanäle ("Anzahl Kanäle an Modulfrontplatte") Bit5..7 = Anzahl Bits für konsistente Informationseinheit (Bit, Byte, Word), kodiert als Zugriffsbreite [bit] = $2^{Bit5..7}$
m_bModType	Bitmaske aus den Konstanten: DIGITAL_MODUL, ANALOG_MODUL, INPUT_MODUL, OUTPUT_MODUL

m_wCmdMask	Bitmaske der vom Modul akzeptierten Kommandos und Optionen: READ_MODUL, WRITE_MODUL, INIT_MODUL, NO_STROBE, RAW_CURRENT, GET_TEMP, NO_CHECK
m_rScalar	Skalierungsfakor analoger Module: RES_AV005U, RES_AV005B, RES_AV010U, RES_AV010B, RES_AC020U, RES_AC420U, RES_AVM050U

Alle angegebenen Konstanten sind in PPSDRVxx.H definiert.

Das folgende Beispiel verdeutlicht den Aufruf des Modultreibers mit dem Parameter **MODUL_INFO**:

```
tModDrv  fpModDrv;
tModInf  ModInf;
BYTE     ModID;

for (ModID=1; ModID<255; ModID++)
   {
   fpModDrv = PPSGetDrvAddress(ModID);

   if (fpModDrv == NULL) continue;

   // Auskunftsfunktion des Treibers aufrufen
   fpModDrv (0, MODUL_INFO, (BYTE*)&ModInf);

   printf("\n%02X: ", (WORD)ModID);
   printf("ID=%02X, ", (WORD)ModInf.m_bModID);
   printf("Size=%d, ", (WORD)ModInf.m_bImageSize);
   printf("Ch&Access=%02X, ",
       (WORD)ModInf.m_bChannelAccess);
   printf("TypeMask=%02X, ",
       (WORD)ModInf.m_bModType);
   printf("CmdMask=%04X, ",
       (WORD)ModInf.m_wCmdMask);
   printf("Scalar=%9.6f", ModInf.m_rScalar);
   }
```


Dokument: **Modultreiber für CPU8051 und CPU167**
Dokumentnummer: **L-342-01, Juni 1998**

Wie würden Sie dieses Handbuch verbessern?

Haben Sie in diesem Handbuch Fehler entdeckt? Seite

Eingesandt von

Name:

Firma:

Adresse:

Kundennummer:

Manual gekauft am:

Rechnungsnummer:

Einsenden an: **PHYTEC** Elektronik GmbH
August-Bebel-Str. 29
D-07973 Greiz

Published by

PHYTEC

Ordering No. L-342-01
Printed in Germany