# 9. Echtzeit-Bibliotheken

## 9.1. Einführung

Nachdem in Kapitel 7 Konzept und Struktur der Echtzeitprogramme erläutert wurde, sollen in den folgenden Abschnitten die Subroutinen vorgestellt werden, die Ihnen die Bibliothek ML2RTBIB zur Verfügung stellt.

Bevor Sie dieses Kapitel "durcharbeiten", sollten Sie die vorangegangenen Kapitel sorgfältig gelesen und mit den Beispielprogrammen ein wenig experimentiert haben. Insbesondere das Kapitel 7 "Echtzeitprogrammierung" sollten Sie mit besonderer Aufmerksamkeit gelesen haben. Es enthält wichtige Informationen zur Programmierung und Compilierung.

*Die im folgenden dargestellten Funktionen und Prozeduren dienen zur Erstellung von Echtzeitprogrammen für die Multi-LAB/2 Karte! Diese Programme sind nicht auf einem PC verwendbar!* **!**

Auf der Distributionsdiskette zur Echtzeitprogrammierung finden Sie neben den abgedruckten Beispielprogrammen in C und Borland-Pascal auch die Header-Datei **ML2RTBIB.H** für C.

Lesen Sie bitte die README Dateien auf den mitgelieferten Disketten. Sie enthalten wichtige Informationen, die nicht mehr rechtzeitig ins Handbuch aufgenommen werden konnten.

## 9.2. Bibliotheksverwaltung

### ml2rt_get_bib_version Ermitteln der Bibliotheksversion

Pascal PROCEDURE ml2rt_get_bib_version (VAR version: STRING);

C void ml2rt_get_bib_version (char *version);

Funktion Die Prozedur liefert die Versionsnummer und das Erstellungsdatum der verwendeten Echtzeit-Bibliothek.

Parameter *version*: Zeiger auf eine Variable, in die die Versionsangaben in folgendem Format eingetragen werden:
" 1E.000 21/03/98"

*Zur Ermittlung der Version des aktiven Betriebssystems OsX verwenden sie die Funktion ml2rt_read_par_block. Die Version steht im Parameterbereich des Betriebssystems (Task 0, s. Anhang K).*

## 9.3. Prozeduren und Funktionen

### 9.3.1. Deklaration

Die globalen Prozeduren einer Task, also diejenigen, die von anderen Tasks, vom Betriebssystem oder vom PC aus aufgerufen werden können, werden grundsätzlich ohne Übergabewerte deklariert:

in C: **void name (void);**

in PASCAL: **PROCEDURE name;**

Einzige Ausnahme sind die Taskfunktionen. Sie haben folgenden vorgeschriebenen Aufbau:

in C: **void far pascal name (ushort task, ushort insize, void *inptr, ushort outsize, void *outptr);**

in PASCAL: **PROCEDURE name (task, insize : Word; VAR inptr; outsize: Word; VAR outptr);**

task: Nummer der eigenen Task.
insize: Anzahl der Daten, die der Funktion übergeben werden (in Byte).
inptr: Zeiger auf die übergebenen Daten.
outsize: Maximale Anzahl der Daten, die zurück erwartet werden (in Byte).
outptr: Zeiger auf den Bereich, in dem die Task ihre Daten zurückgibt.

*Alle lokalen Prozeduren und Funktionen dürfen beliebig deklariert werden.* !

## 9.3.2. Aufbau

Alle globalen Prozeduren einer Task beginnen mit einem Bibliotheksaufruf, der die Prozessor-Register sichert und für die Prozedur neu setzt. Am Ende jeder Prozedur steht ein Aufruf, der den ursprünglichen Zustand wiederherstellt. Je nachdem, ob es sich um eine globale Prozedur oder Funktion handelt und je nach Tasktyp, werden dabei unterschiedliche Funktionen verwendet.

### ml2rt_entry — Beginne globale Prozedur

Pascal PROCEDURE ml2rt_entry;

C void ml2rt_entry (void);

Funktion Diese Prozedur rettet alle Register und setzt das DS-Register auf das Datensegment des Programms. Der Aufruf dieser Prozedur sollte immer als erste Anweisung in einer Prozedur stehen, die vom Betriebssystem aufrufbar sein soll, d.h. in allen Prozeduren, die in die PDT eingetragen wurden. Prozeduren, die nur innerhalb des Programms aufgerufen werden, müssen **ml2rt_entry** nicht aufrufen.

### ml2rt_entry_task — Beginne globale Prozedur

Pascal FUNCTION ml2rt_entry_task: WORD;

C ushort ml2rt_entry_task (void);

Funktion Die Funktion arbeitet genau wie **ml2rt_entry**. In manchen Fällen, kann es erforderlich sein, in einer globalen Prozedur die eigene Tasknummer zu kennen. Das gilt vor allem dann, wenn Parameter- oder Datenbereich nicht im Programm enthalten sind und mit Betriebssystemaufrufen (z.B. ml2rt_read_par_xxx) zugegriffen werden muß. Für diesen Fall liefert diese Funktion die Tasknummer zurück. Für DI-Tasks ist diese Funktion nicht geeignet.

! *Lokale Variablen einer Taskprozedur dürfen erst nach dem Aufruf von ml2rt_entry bzw. ml2rt_entry_task initialisiert werden.*

## ml2rt_exit
## ml2rt_exit_interrupt — Beende globale Prozedur

Pascal PROCEDURE ml2rt_exit;

PROCEDURE ml2rt_exit_interrupt;

C void ml2rt_exit (void);

void ml2rt_exit_interrupt (void);

Funktion Diese Prozeduren sind das Gegenstück zu **ml2rt_entry** und **ml2rt_entry_task**. Sie restaurieren die mit **ml2rt_entry** gesicherten Register und beenden die Prozedur. Der Aufruf dieser Prozeduren muß immer als letzter Aufruf in einer Prozedur stehen. **ml2rt_exit_interrupt** muß in Hauptprozeduren (Prozedur #0) von DI-Tasks verwendet werden! Sie beendet die Prozedur mit einem IRET (Return of Interrupt).

## ml2rt_entry_function — Beginne globale Funktion

Pascal PROCEDURE ml2rt_entry_function;

C void ml2rt_entry_function (void);

Funktion Diese Prozedur arbeitet genauso wie **ml2rt_entry,** jedoch ist sie für eine Task-*Funktion* bestimmt. Die Task-Funktion liefert im Gegensatz zur Task-Prozedur ein Ergebnis. **ml2rt_entry_function** sorgt dafür, daß die im Funktionskopf deklarierten Variablen gesetzt werden.

*Lokale Variablen einer Taskfunktion dürfen erst nach dem Aufruf von ml2rt_entry_function initialisiert werden.*

## ml2rt_exit_function — Beende globale Funktion

Pascal: PROCEDURE ml2rt_exit_function (leftsize, outsize, error: WORD);

C: void ml2rt_exit_function (ushort leftsize, ushort outsize, ushort error);

Funktion: Diese Prozedur beendet eine Funktion einer Task. Sie arbeitet genauso wie **ml2rt_exit**. Es wird jedoch zusätzlich eine Antwort zur aufrufenden Task zurückgesendet.

Parameter:

*leftsize*: Anzahl Bytes, die nicht verwendet werden konnte.

*outsize*: Größe der Antwort (in Bytes).

*error*: Fehlermeldung für die aufrufende Task. Wenn die Funktion per Makrobefehl vom PC aus aufgerufen wird, wird nur das höherwertige Byte als Fehlermeldung zurückgegeben, das niederwertige Byte muß **e0h** sein. Wenn *error* also **xxe0h** ist, wird **xxh** als Fehlerbyte an den PC übergeben.

### 9.3.3. Aufrufe von Prozeduren und Funktionen

## ml2rt_call_func — Rufe eine Funktion einer Task auf

Pascal: PROCEDURE ml2rt_call_func (task, funcnr, VAR outsize: WORD; VAR outptr; maxinsize: WORD; VAR insize: WORD; VAR inptr; VAR error: WORD);

C: void ml2rt_call_func (ushort task, ushort funcnr, ushort far *outsize, void far *outptr; ushort maxinsize, ushort far *insize, void far *inptr, ushort far *error);

Funktion: Mit dieser Prozedur kann eine beliebige Funktion einer auf der MultiLAB/2 installierten Task aufgerufen werden. Ein möglicher Fehler wird nicht abgefangen (kein Aufruf der Fehlerbehandlung).

Parameter:

*task*: Nummer der Task, die die gewünschte Funktion enthält.

*funcnr*: Nummer der Funktion.

*outsize*: Anzahl der Bytes, die an die Funktion übergeben werden. Nach dem Aufruf der Funktion enthält *outsize* die Anzahl der nicht verwendeten Bytes.

*outptr*: Zeiger auf einen Variablenbereich, in dem die Daten stehen, die an die Funktion übergeben werden.

*maxinsize*: Maximale Anzahl an Bytes, die die aufgerufene Funktion zurückliefern darf bzw. soll.

*insize*: Zeiger auf eine Variable, in die die tatsächlich zurückgelieferte Anzahl an Bytes eingetragen wird.

*inptr*: Zeiger auf eine Variable, in die die Rückgabedaten der Funktion eingetragen werden. *inptr* kann gleich *outptr* sein.

*error*: Zeiger auf eine Variable, in die ein Fehlerstatus eingetragen wird. Ist dieser Wert = 0, ist kein Fehler bei der Ausführung der Funktion aufgetreten. Bei einem Wert größer Null handelt es sich entweder um eine Betriebssystemmeldung (siehe Anhang F) oder um eine spezielle Meldung der aufgerufenen Funktion.



## ml2rt_call_proc — Rufe eine Prozedur einer Task auf

Pascal: PROCEDURE ml2rt_call_proc (task, procnr : WORD);

C: void ml2rt_call_proc (ushort task, ushort procnr);

Funktion: Diese Prozedur ist ein Sonderfall der Prozedur **ml2rt_call_func**: Es werden keine Daten an die aufgerufene Prozedur übergeben oder von ihr übernommen.

Parameter:

*task*: Nummer der Task, die die gewünschte Prozedur enthält.

*procnr*: Nummer der Prozedur.

## 9.4. Taskbefehle

Die meisten der in diesem Abschnitt beschriebenen Funktionen enthalten den Parameter *task*. Dieser spezifiziert die Nummer, unter dem das Betriebssystem auf der Multi-LAB/2 Karte das zugehörige Echtzeitprogramm verwaltet. Er wird in den Parameterbeschreibungen nicht jedesmal aufgeführt.

### 9.4.1. Taskinformationen abfragen

#### ml2rt_get_task_status — Prüfe, ob eine Task aktiviert ist

Pascal: FUNCTION ml2rt_get_task_status (task: WORD): BYTE;

C: byte ml2rt_get_task_status (ushort task);

Funktion: Diese Funktion liefert die Anzahl der Aktivierungen einer Task. Ist das Funktionsergebnis = 0, dann ist die Task nicht aktiviert. DI- und II-Tasks können maximal einmal aktiviert sein.

#### ml2rt_get_tdt_address — Ermittle Adresse der TDT

Pascal: PROCEDURE ml2rt_get_tdt_address (task: WORD;
VAR adr: POINTER);

C: void ml2rt_get_tdt_address (ushort task, void *adr);

Funktion: Mit dieser Prozedur wird die Adresse der TDT einer Task ermittelt.

Parameter: *adr*: Zeiger auf eine Variable, in die die Adresse (Segment: Offset) der TDT eingetragen wird.

## ml2rt_get_func_address — Ermittle Adresse einer Prozedur einer Task

Pascal PROCEDURE ml2rt_get_func_address (task: WORD; procnr: WORD; VAR adr: POINTER);

C void ml2rt_get_func_address (ushort task, ushort procnr, void *adr);

Funktion Diese Prozedur ermittelt die Adresse einer globalen Taskprozedur. Mit Hilfe des Zeigers kann die globale Prozedur direkt aufgerufen werden (Aufruf ist schneller als mit ml2rt_call_proc).

Parameter *procnr*: Nummer der gesuchten Prozedur.

*adr*: Zeiger auf eine Variable, in die die Adresse (Segment: Offset) der Prozedur eingetragen wird.

Hinweis Taskfunktionen können nicht direkt aufgerufen werden.

## ml2rt_get_task_info — Fordere Systeminformationen über eine Task an

Pascal PROCEDURE ml2rt_get_task_info (task: WORD; callnr: WORD; VAR result: LONGINT);

C void ml2rt_get_task_info (ushort task, ushort callnr, ulong *result);

Funktion Diese Prozedur liefert Systeminformationen über eine Task.

Parameter: *callnr*: Spezifikation der gewünschten Information (s. Anhang H)

*result*: Zeiger auf eine Variable, in die das Ergebnis eingetragen wird.

## ml2rt_get_int_task — Ermittle Tasktyp und -nummer

Pascal: PROCEDURE ml2rt_get_int_task (inter: WORD; VAR task: WORD; VAR typ: BYTE);

C: void ml2rt_get_int_task (ushort inter, ushort *task, byte *typ);

Funktion: Diese Prozedur ermittelt, welche Tasknummer und welchen Programmtyp die Task hat, die unter einem Interrupt installiert ist.

Parameter:

*inter*: Interruptnummer.

*task*: Zeiger auf eine Variable, in die die Nummer der Task eingetragen wird, die unter dem Interrupt *inter* installiert ist. Wenn der Interrupt unbenutzt ist, wird dieser Wert = -1 (ffffh) gesetzt.

*typ*: Zeiger auf eine Variable, in die der Tasktyp eingetragen wird. Der Wert entspricht einer der Bibliothekskonstanten **_NI_TASK, _II_TASK** oder **_DI_TASK.**

## ml2rt_get_task_number — Ermittle Task-Nummer

Pascal: FUNCTION ml2rt_get_task_number (prog_nr: WORD; VAR install_nr: WORD): WORD;

C: ushort ml2rt_get_task_number (ushort prog_nr, ushort *install_nr);

Funktion: Die Funktion liefert die Tasknummer, unter der ein Programm installiert ist.

Parameter:

*prog_nr*: In der PDT eingetragene Nummer des Programms.

*install_nr*: Wenn ein Programm mehrfach installiert ist, dann kann in diesem Parameter angegeben werden, von welcher Instanz die Tasknummer ermittelt werden soll.

| **install_nr** (vor Aufruf) | **install_nr** (nach Aufruf) | Bedeutung |
|---|---|---|
| 0 | 0 | Das Programm ist nicht installiert, der Rückgabewert ist ungültig |
| 0 | >0 | *install_nr* enthält die Anzahl der Installierungen, der Rückgabewert gibt die höchste Tasknummer an, unter der das Programm installiert ist. |
| 1 | 0 | Das Programm ist nicht installiert, der Rückgabewert ist ungültig |
| 1 | 1 | Das Programm ist installiert, der Rückgabewert gibt die niedrigste Tasknummer an, unter der das Programm installiert ist. |
| 2 bis 1024 | 0 | Das Programm ist nicht installiert, der Rückgabewert ist ungültig |
| 2 bis 1024 | = *install_nr* vor Aufruf | Der Rückgabewert enthält die Tasknummer der Task, die sich, wenn man die Tasks nach ihren Nummern ordnet, an der Stelle *install_nr* befindet. |
| 2 bis 1024 | < *install_nr* vor Aufruf | Das Programm wurde nur sooft installiert, wie in *install_nr* zurückgegeben wurde. Der Rückgabewert gibt die höchste Tasknummer an, unter der das Programm installiert ist. |

## 9.4.2. Tasks aktivieren und deaktivieren

### ml2rt_wakeup_task — Aktiviere eine Task

Pascal PROCEDURE ml2rt_wakeup_task (task: WORD);

C void ml2rt_wakeup_task (ushort task);

Funktion Diese Prozedur aktiviert eine Task. NI-Tasks können auch mehrfach aktiviert werden, wodurch sie gegenüber einfach aktivierten Tasks häufiger aufgerufen werden. Bei II- und DI-Tasks wird der zugehörige Interrupt demaskiert.

### ml2rt_sleep_task — Deaktiviere eine NI-Task

Pascal PROCEDURE ml2rt_sleep_task (task: WORD);

C void ml2rt_sleep_task (ushort task);

Funktion Durch diese Prozedur wird eine Task deaktiviert. Mehrfach aktivierte NI-Tasks müssen auch mehrfach deaktiviert werden. Bei II- und DI-Tasks wird der zugehörige Interrupt maskiert.

## 9.4.3. Zugriff auf die Parameterbereiche von Tasks

Zugriffe auf Variable im Parameterbereich einer Task erfolgen immer unter Angabe ihrer Adresse (relativ zum Anfang des Parameterbereichs, gezählt in Byte). Diese wird bei allen nachfolgenden Bibliotheksfunktionen im Parameter *start* angegeben. Sollen mehrere Parameter mit einem Befehl gelesen oder geschrieben werden, spezifiziert dieser Wert den ersten zu lesenden oder zu schreibenden Parameter.

Wenn auf bestimmte Parameter oft und schnell zugegriffen werden soll, kann die Speicheradresse des Parameters mit **ml2rt_get_par_address** einmal ermittelt werden und über die Adresse (Pointer) zugegriffen werden.

### ml2rt_read_par_byte
### ml2rt_read_par_word
### ml2rt_read_par_dword — Lies Parameter einer Task

Pascal

```
FUNCTION ml2rt_read_par_byte (task, start: WORD): BYTE;
FUNCTION ml2rt_read_par_word (task, start: WORD): WORD;
PROCEDURE ml2rt_read_par_dword (task, start: WORD;
VAR data: LONGINT);
```

C

```
byte ml2rt_read_par_byte (ushort task, ushort start);
ushort ml2rt_read_par_word (ushort task, ushort start);
void ml2rt_read_par_dword (ushort task, ushort start, ulong *data);
```

Funktion: Die Funktionen lesen ein, zwei oder vier Byte aus dem Parameterbereich einer Task. Beachten Sie, daß der Wert eines Doppelwortes nicht als Funktionsergebnis sondern in *data* zurückgegeben wird.

Parameter: *data*: Zeiger auf eine Variable, die das Ergebnis von **ml2rt_read_par_dword** aufnimmt.

## ml2rt_read_par_block — Lies Parameterblock

Pascal
PROCEDURE ml2rt_read_par_block (task, start, size: WORD;
VAR data_var);

C
void ml2rt_read_par_block (ushort task, ushort start, ushort size,
void *data_var);

Funktion: Mit dieser Prozedur kann ein Block von Parametern einer Task gelesen werden.

Parameter:
*size*: Größe des Blocks (in Byte, z. B. sizeof (...))

*data_var*: Zeiger auf einen Variablenbereich, in den die gelesenen Daten eingetragen werden.

Hinweis: Das Lesen eines Blocks ist durch Interrupts unterbrechbar, falls vorher die Interrupts nicht gesperrt wurden.

## ml2rt_write_par_byte
## ml2rt_write_par_word
## ml2rt_write_par_dword — Schreibe Parameter einer Task

Pascal
PROCEDURE ml2rt_write_par_byte (task, start: WORD;
data: BYTE);

PROCEDURE ml2rt_write_par_word (task, start: WORD;
data: WORD);

PROCEDURE ml2rt_write_par_dword (task, start: WORD;
data: LONGINT);

C
void ml2rt_write_par_byte (ushort task, ushort start, byte data);

void ml2rt_write_par_word (ushort task, ushort start, ushort data);

void ml2rt_write_par_dword (ushort task, ushort start, ulong data);

Funktion: Mit diesen Prozeduren können ein, zwei oder vier Byte in den Parameterbereich einer Task geschrieben werden.

Parameter:
*data*: Wert, der geschrieben werden soll.

## ml2rt_write_par_block — Schreibe Parameterblock

Pascal: PROCEDURE ml2rt_write_par_block (task, start, size: WORD; VAR data_var);

C: void ml2rt_write_par_block (ushort task, ushort start, ushort size, void *data_var);

Funktion: Mit dieser Prozedur kann ein Block von Parametern einer Task gesetzt werden.

Parameter:
- *size*: Größe des Blocks (in Byte, z. B. sizeof (...))
- *data_var*: Zeiger auf den Variablenbereich, in dem die Daten stehen, die geschrieben werden.

Hinweis: Das Schreiben eines Blocks ist durch Interrupts unterbrechbar, falls vorher die Interrupts nicht gesperrt wurden. !

## ml2rt_get_par_address — Ermittle Adresse eines Parameters einer Task

Pascal: PROCEDURE ml2rt_get_par_address (task, start: WORD; VAR adr: LONGINT);

C: void ml2rt_get_par_address (ushort task, ushort start, ulong *adr);

Funktion: Diese Prozedur ermittelt die Adresse eines Parameters.

Parameter:
- *adr*: Zeiger auf eine Variable, in die die physikalische Adresse des Parameters eingetragen wird.

## 9.4.4. Zugriff auf die Datenbereiche von Tasks

Der Zugriff auf den Datenbereich einer Task geschieht immer über den Schreib- bzw. Lesezeiger der Task. Diese Zeiger werden vom Betriebssystem verwaltet und bei jedem Zugriff automatisch um die Anzahl der gelesenen bzw. geschriebenen Bytes inkrementiert.

### ml2rt_reset_r_pointer — Setze Daten-Lesezeiger zurück

Pascal: PROCEDURE ml2rt_reset_r_pointer (task: WORD);

C: void ml2rt_reset_r_pointer (ushort task);

Funktion: Diese Prozedur setzt den Daten-Lesezeiger einer Task an den Anfang ihres Datenbereichs.

### ml2rt_move_r_pointer — Verschiebe Daten-Lesezeiger

Pascal: PROCEDURE ml2rt_move_r_pointer (task: WORD;
offset: LONGINT);

C: void ml2rt_move_r_pointer (ushort task, long offset);

Funktion: Diese Prozedur verschiebt den Daten-Lesezeiger der Task.

Parameter: *offset*: Anzahl Byte, um die der Lesezeiger verschoben werden soll. Der Wert kann auch negativ sein.

## ml2rt_read_data_byte
## ml2rt_read_data_word — Lies Einzelwert aus dem
## ml2rt_read_data_dword — Datenbereich einer Task

Pascal
FUNCTION ml2rt_read_data_byte (task: WORD): BYTE;

FUNCTION ml2rt_read_data_word (task: WORD): WORD;

PROCEDURE ml2rt_read_data_dword (task: WORD;
VAR data: LONGINT);

C
byte ml2rt_read_data_byte (ushort task);

ushort ml2rt_read_data_word (ushort task);

void ml2rt_read_data_dword (ushort task, ulong *data);

Funktion
Die Prozeduren lesen ein, zwei oder vier Bytes aus dem Datenbereich einer Task. Es wird ab dem Lesezeiger der Task gelesen. Danach wird der Lesezeiger der Task um die entsprechende Anzahl Byte erhöht. Beachten Sie, daß der Wert eines Doppelwortes (vier Byte) nicht als Funktionsergebnis, sondern in *data* zurückgegeben wird.

Parameter
*data*: Zeiger auf eine Variable, die das Ergebnis von **ml2rt_read_data_dword** aufnimmt.

## ml2rt_read_data_block — Lies Datenblock



Pascal
PROCEDURE ml2rt_read_data_block (task, size: WORD;
VAR data_var);

C
void ml2rt_read_data_block (ushort task, ushort size, void *data_var);

Funktion
Diese Prozedur liest einen Datenblock aus dem Datenbereich einer Task. Es wird ab dem Lesezeiger der Task gelesen. Der Lesezeiger der Task wird anschließend um die gelesene Anzahl Bytes erhöht.

Parameter
*size*: Größe des Blocks (in Byte, z. B. sizeof (...)).

*data_var*: Zeiger auf einen Variablenbereich, in den die gelesenen Daten eingetragen werden.

## ml2rt_reset_w_pointer — Setze Daten-Schreibzeiger zurück

Pascal PROCEDURE ml2rt_reset_w_pointer (task: WORD);

C void ml2rt_reset_w_pointer (ushort task);

Funktion Diese Prozedur setzt den Daten-Schreibzeiger einer Task an den Anfang ihres Datenbereichs.

## ml2rt_move_w_pointer — Verschiebe Daten-Schreibzeiger

Pascal PROCEDURE ml2rt_move_w_pointer (task: WORD;
offset: LONGINT);

C void ml2rt_move_w_pointer (ushort task, long offset);

Funktion Diese Prozedur verschiebt den Daten-Schreibzeiger einer Task.

Parameter *offset*: Anzahl Byte, um die der Schreibzeiger verschoben werden soll. Der Wert kann auch negativ sein.

## ml2rt_write_data_byte<br>ml2rt_write_data_word<br>ml2rt_write_data_dword — Schreibe Daten in den Datenbereich einer Task

Pascal PROCEDURE ml2rt_write_data_byte (task: WORD; data: BYTE);

PROCEDURE ml2rt_write_data_word (task: WORD; data: WORD);

PROCEDURE ml2rt_write_data_dword (task: WORD;
data: LONGINT);

C void ml2rt_write_data_byte (ushort task, byte data);

void ml2rt_write_data_word (ushort task, ushort data);

void ml2rt_write_data_dword (ushort task, ulong data);

Funktion Diese Prozeduren schreiben ein, zwei oder vier Byte ab dem Schreibzeiger einer Task in ihren Datenbereich. Der Schreibzeiger wird danach automatisch um die entsprechende Anzahl von Byte erhöht.

Parameter *data*: Wert, der geschrieben werden soll.

## ml2rt_write_data_block Schreibe Datenblock

Pascal PROCEDURE ml2rt_write_data_block (task, size: WORD;
VAR datavar);

C void ml2rt_write_data_block (ushort task, ushort size, void *datavar);

Funktion: Diese Prozedur schreibt einen Block von Daten ab dem Schreibzeiger einer Task in ihren Datenbereich. Der Schreibzeiger der Task wird danach automatisch um die entsprechende Anzahl von Byte erhöht.

Parameter: *size*: Größe des Blocks (in Byte, z. B. sizeof (...))

*datavar*: Zeiger auf den Variablenbereich, in dem die Daten stehen, die geschrieben werden.

## 9.5. Funktionen zur Hardwarekontrolle

### 9.5.1. Interrupt-Controller

Alle Funktionen in diesem Abschnitt beziehen sich jeweils auf einen Interrupt (übergeben in der Variablen *hrdirq).* Dort ist die Interruptnummer des gewünschten Interrupts anzugeben. Dafür sind in ML2RTBIB.H folgende Konstanten definiert:

| Bezeichnung | Wert | Bedeutung |
|---|---|---|
| NMI | 02h | Nicht maskierbarer Interrupt |
| IRQ_IP_0 | 80h | externer Eingang IP-0 |
| IRQ_IP_1 | 81h | externer Eingang IP-1 |
| IRQ_RBF | 82h | Receive-Buffer Full der PC-Schnittstelle |
| IRQ_TIMER_A | 83h | Timer A |
| IRQ_TIMER_B | 84h | Timer/Counter B |
| IRQ_TIMER_B_UP | 84h | Aufwärtszähler für Inkrementalgeber-Überlauf |
| IRQ_TIMER_B_DOWN | 85h | Abwärtszähler für Inkrementalgeber-Überlauf |
| IRQ_SIO_RBF | 86h | Zeichen über serielle Schnittstelle empfangen |

### ml2rt_unmask_int — Gib Hardware-Interrupt frei

Pascal PROCEDURE ml2rt_unmask_int (hrdirq: BYTE);

C void ml2rt_unmask_int (byte hrdirq);

Funktion Diese Prozedur gibt einen Interrupt frei.

### ml2rt_mask_int — Sperre Hardware-Interrupt

Pascal PROCEDURE ml2rt_mask_int (hrdirq: BYTE);

C void ml2rt_mask_int (byte hrdirq);

Funktion Diese Prozedur sperrt (maskiert) einen Interrupt.

### ml2rt_set_int_edge — Stelle Interrupt-Flanke ein

Pascal PROCEDURE ml2rt_set_int_edge (hrdirq: BYTE; edge: BYTE);

C void ml2rt_set_int_edge (byte hrdirq, byte edge);

Funktion Diese Prozedur stellt die aktive Flanke eines Interrupts ein.

Parameter *edge*: Spezifiziert, ob der Interrupt bei steigender (POS_EDGE) oder fallender (NEG_EDGE) Flanke ausgelöst werden soll.

## ml2rt_end_of_int — Beende Interrupt-Prozedur

Pascal PROCEDURE ml2rt_end_of_int (hrdirq: BYTE);

C void ml2rt_end_of_int (byte hrdirq);

Funktion Diese Prozedur sendet für den Interrupt ein "End of Interrupt" (EOI) zum Master- und zum Slave-Interrupt-Controller.

## ml2rt_clear_int — Lösche Pending-Interrupt

Pascal PROCEDURE ml2rt_clear_int (hrdirq: BYTE);

C void ml2rt_clear_int (byte hrdirq);

Funktion Diese Prozedur löscht den Interrupt im Interrupt-Controller. Sie wird in der Regel vor **ml2rt_unmask_int** verwendet, um sicherzustellen, daß ein vorher aufgetretener Interrupt nicht zur Ausführung kommt.

## 9.5.2. Speicherverwaltung

### ml2rt_allocate_ram Reserviere Speicher auf der Multi-LAB/2

Pascal
PROCEDURE ml2rt_allocate_ram (task, usage: WORD;
strategy, align: BYTE; VAR size: LONGINT; VAR adr: LONGINT);

C
void ml2rt_allocate_ram (ushort task, ushort usage, byte strategy,
byte align, ulong *size, ulong *adr);

Funktion
Diese Prozedur reserviert Speicherplatz auf der Multi-LAB/2 Karte.

Parameter

*task*: Task, für die Speicher reserviert werden soll. Die Angabe dient Protokollzwecken auf der Multi-LAB/2 Karte.

*usage*: Reserviert, immer = 0 setzen.

*strategy*: Reservierungsstrategie. Folgende Werte sind zulässig:

1 = Es wird die in *size* angegebene Speichermenge (in Byte), von <u>unten</u> im freien Speicher beginnend, reserviert. Reicht der Speicher nicht, dann soll nichts reserviert und ein Fehler ausgelöst werden. In diesem Fall wird in *size* eine Null zurückgegeben.

2 = wie 1, jedoch soll, falls die in *size* angegebene Speichermenge (in Byte) nicht zur Verfügung steht, so viel wie möglich reserviert werden. Die tatsächlich reservierte Speichermenge wird in *size* zurückgegeben.

3 = wie 1, jedoch soll der Speicher von <u>oben</u> im freien Speicher beginnend reserviert werden.

4 = wie 2, jedoch soll der Speicher von <u>oben</u> im freien Speicher beginnend reserviert werden.

*align*: Ausrichtung des Anfangs des zu reservierenden Speicherbereichs in Potenzen zur Basis 2: 0 = Byte, 1 = Wort, 2 = Doppelwort, 3 = 8 Byte, ...

*size*: Größe des zu reservierenden Speicherbereichs (in Byte). Der Rückgabewert von *size* enthält die *tatsächlich* reservierte Speichergröße.

*adr*: Zeiger auf eine Variable, in die die physikalische Anfangsadresse des reservierten Speichers eingetragen wird. Wenn *size* = 0 ist, dann ist die Adresse nicht gültig.

## ml2rt_get_free_ram_size — Ermittle freien Speicher

Pascal
```
PROCEDURE ml2rt_get_free_ram_size (align: BYTE;
VAR size: LONGINT);
```

C
```
void ml2rt_get_free_ram_size (byte align, long *size);
```

Funktion: Diese Prozedur ermittelt die Größe des freien Speichers auf der MultiLAB/2 Karte.

Parameter:

*align*: Gewünschte Ausrichtung des Speichers in Potenzen zur Basis 2: 0 = Byte, 1 = Wort, 2 = Doppelwort, 3 = 8 Byte, ...

*size*: Zeiger auf eine Variable, in die die Größe des freien Speichers (in Byte) eingetragen wird.

## 9.5.3. Timer-Kontrolle

Für die Auswahl der Timer stehen die Konstanten TIMER_A (12 Bit) und TIMER_B (16 Bit) in ML2RTBIB.H zur Verfügung.

### ml2rt_set_timer — Setze und starte einen Timer

Pascal: PROCEDURE ml2rt_set_timer (counter: BYTE; count: WORD; mode: BYTE);

C: void ml2rt_set_timer (byte counter, ushort count, byte mode);

Funktion: Diese Prozedur setzt einen Timer und startet ihn.

Parameter:

*counter*: Gibt an, welcher Timer gesetzt werden soll.

*count*: Initialisierungswert des Timers. Die Timerfrequenz beträgt standardmäßig 500 kHz (Auflösung 2 •s, kann in Betriebssystem-Parameter 150 verändert werden), die maximale Laufzeit damit 65535 * 2 •s = 131 ms für Timer B bzw. 4095 * 2 •s = 8 ms für Timer A.

*mode*: Betriebsart des Timers. Derzeit ist dieser Parameter immer mit der Konstanten **TIMER_MODE** zu beschreiben**.**

### ml2rt_get_timer — Lies Anfangsstand eines Timers

Pascal: PROCEDURE ml2rt_get_timer (counter: BYTE; VAR count: WORD; VAR status: WORD);

C: void ml2rt_get_timer (byte counter, ushort *count, ushort *status);

Funktion: Diese Prozedur liest den Anfangswert und den Status eines Timers. Die Funktion darf nur verwendet werden, wenn der Timer in der Betriebsart TIMER_MODE betrieben wird. Der aktuelle Zählerstand von Timer B kann nur direkt vom I/O Port 4 gelesen werden.

Parameter:

*counter*: Gibt an, welcher Timer gelesen werden soll.

*count*: Zeiger auf eine Variable, in die der Anfangswert eingetragen wird.

*status*: Zeiger auf eine Variable, in deren Bits 0..6 die eingestellte Betriebsart der Timer entsprechend dem Timer-Konfigurations-Register (s. Anhang C, I/O Adresse 7) eingetragen wird, Bit 7 bleibt unbenutzt. In den Bits 8 bis 15 wird der von beiden Timern gemeinsam genutzte Basistakt eingetragen (s. Anhang C, I/O Adresse 82h).

### 9.1.4. LED

**ml2rt_led_on**
**ml2rt_led_off** **Schalte LED ein/aus**

Pascal PROCEDURE ml2rt_led_on;

PROCEDURE ml2rt_led_off;

C void ml2rt_led_on (void);

void ml2rt_led_off (void);

Funktion Die Prozeduren schalten die LED ein bzw. aus.

## 9.1.5. EEPROM-Zugriffe

### ml2rt_read_eeprom_copy
### ml2rt_read_eeprom_direct — Lies EEPROM-Wort

Pascal
FUNCTION ml2rt_read_eeprom_copy (reladr: BYTE): WORD;
FUNCTION ml2rt_read_eeprom_direct(reladr: BYTE): WORD;

C
ushort ml2rt_read_eeprom_copy (byte reladr);
ushort ml2rt_read_eeprom_direct (byte reladr);

Funktion
Diese Funktionen lesen ein Wort direkt aus dem EEPROM bzw. aus dessen im RAM angelegter Kopie. Die Kopie wird beim Aktivieren eines Betriebssystems automatisch erzeugt. Direkte Zugriffe sind langsam und in einer Echtzeitumgebung nicht zu empfehlen.

Parameter
*reladr*: relative Adresse des zu lesenden Wortes (0 - 63).

### ml2rt_write_eeprom_copy
### ml2rt_write_eeprom_direct — Schreibe EEPROM-Wort

Pascal
PROCEDURE ml2rt_write_eeprom_copy (reladr: BYTE; data: WORD);
PROCEDURE ml2rt_write_eeprom_direct (reladr: BYTE; data: WORD);

C
void ml2rt_write_eeprom_copy (byte reladr, ushort data);
void ml2rt_write_eeprom_direct (byte reladr, ushort data);

Funktion:
Diese Funktionen schreiben ein Wort direkt ins EEPROM bzw. in dessen im RAM angelegte Kopie. In die Kopie geschriebene Daten gehen beim Abschalten der Karte verloren.

Parameter:
*reladr*: relative Adresse des zu lesenden Wortes (0 - 63).
*data*: Daten, die geschrieben werden sollen.

## 9.1.6. Service-Requests

### ml2rt_send_host_srq — Sende direkten Service-Request

Pascal: FUNCTION ml2rt_send_host_srq (data: WORD): WORD;

C: ushort ml2rt_send_host_srq (ushort data);

Funktion: Diese Funktion darf nur in der Hauptprozedur einer NI-Task angewendet werden! Sie sendet einen Service-Request zum PC, indem sie auf dem PC einen Interrupt auslöst und ein Wort in die Schnittstelle des PC schreibt. Wenn der Request nicht abgesetzt werden kann, wird als Funktionsergebnis einer der folgenden Fehlercodes zurückgegeben.

| Bit-Nummer | Bedeutung |
|---|---|
| Bit-8 = 1 | RBF (PC-Schnittstelle) |
| Bit-11 = 1 | Interruptleitung nicht angeschlossen |
| Bit-12 = 1 | Angewählte Interruptleitung ungültig |
| Bit-13 = 1 | Softlock der Schnittstelle durch eine andere Task |
| Bit-14 = 1 | DLP war gesetzt |
| Bit-15 = 1 | TBF war gesetzt (Schnittstelle voll) |

Parameter: *data*: Datenwort, das an den Host-PC gesendet wird. Das Low-Byte darf nur Werte zwischen 80h und BFh (SRQ-Codes für User, s. Anhang F) annehmen.

Hinweis: Wenn der Versuch, einen Service-Request zum PC zu senden, mißglückt ist, muß die Kontrolle wieder an das Betriebssystem zurückgegeben werden. Der nächste Versuch sollte erst beim nächsten Aufruf der NI-Task unternommen werden. !

## 9.6. Fehlerbehandlung

Wenn beim Aufruf einer Bibliotheksfunktion ein Fehler auftritt, springt die Bibliothek in eine Fehlerbehandlungsprozedur, die vom Benutzer programmiert werden kann. Innerhalb der Fehlerbehandlungsprozedur kann die Fehlerursache mit **ml2rt_get_error_info** bestimmt werden. Wenn keine Fehlerbehandlungsprozedur installiert ist, geschieht beim Auftreten eines Fehlers nichts.

### ml2rt_set_error_handling Lege Fehlerbehandlungsprozedur fest

Pascal PROCEDURE ml2rt_set_error_handling (errhandle: POINTER);

C void ml2rt_set_error_handling (long errhandle);

Funktion Mit dieser Prozedur wird der Bibliothek mitgeteilt, welche Prozedur nach dem Auftreten eines Fehlers aufgerufen werden soll. Die Prozedur muß als **FAR** compiliert worden sein (d.h. **retf** als Rücksprung). Die Adresse ist als Segment:Offset zu übergeben, z.B. in C: ml2rt_set_error_handling ((long)my_error_service);

Parameter *errhandle*: Adresse der Fehlerbehandlungsprozedur.

### ml2rt_get_error_info Fordere Fehlerinformation an

Pascal PROCEDURE ml2rt_get_error_info
(VAR err_rec: ml2rt_error_info_type);

C void ml2rt_get_error_info (ml2rt_error_info_type *err_rec);

Funktion Diese Prozedur findet vor allem in einer Fehlerbehandlungsprozedur Verwendung. Sie liefert folgende Datenstruktur mit Informationen über einen aufgetretenen Fehler zurück.

| Feldbezeichner | Typ | Bedeutung |
|---|---|---|
| errorcode | Word | Fehlercode des Betriebssystems |
| subnum | Word | Nummer der Betriebssystem-Subroutine, die den Fehler ausgelöst hat. |
| mark | Word | Vom Benutzer gesetzte Fehlermarkierung. |
| erroverflow | Word | Wird auf 1 gesetzt, wenn ein weiterer Fehler auftritt, bevor der letzte Fehler mit **ml2rt_reset_error** zurückgesetzt wurde. |

Nach dem Auftreten eines Fehlers wird der Fehlerrecord durch nachfolgende Fehler nicht verändert. Das heißt, daß nur der erste Fehler im Fehlerrecord eingetragen wird. Nachfolgende Fehler setzen *erroverflow* = 1. Erst nach einem Aufruf von **ml2rt_reset_error** wird der Fehlerrecord wieder neu beschrieben.

## ml2rt_reset_error — Setze Fehlerrecord zurück

Pascal PROCEDURE ml2rt_reset_error;

C void ml2rt_reset_error (void);

Funktion Diese Prozedur setzt den Fehlerrecord zurück.

## ml2rt_set_mark — Markiere eine Programmstelle für die Fehlerbehandlung

Pascal PROCEDURE ml2rt_set_mark (mark: WORD);

C void ml2rt_set_mark (ushort mark);

Funktion Mit dieser Prozedur wird eine Bibliotheksvariable gesetzt. Tritt danach ein Fehler auf, so kann in der Fehlerbehandlungsprozedur anhand der Markierung auf den Programmteil geschlossen werden, in dem der Fehler auftrat.

Parameter: *mark*: Wert, auf den die Variable gesetzt werden soll.



## ml2rt_force_error — Erzeuge eine Fehlermeldung

Pascal PROCEDURE ml2rt_force_error;

C void ml2rt_force_error (void);

Funktion Diese Prozedur löst einen Fehler mit der Nummer 0FE0h aus. Sie kann dazu verwendet werden, aus anderen Bibliotheken die Fehlerbehandlungsprozedur der **ML2RTBIB** aufzurufen.

## 9.7. Sonstige Funktionen

### ml2rt_set_pdt_adr — Setze die PDT-Adresse

Pascal PROCEDURE ml2rt_set_pdt_adr (headadr: Pointer);

C void ml2rt_set_pdt_adr (void *headadr);

Funktion Diese Prozedur übergibt die Adresse der Programm-Deskriptor-Tabelle an das Betriebssystem und initialisiert die Bibliothek. Die PDT muß zu diesem Zeitpunkt vom Programm angelegt und ausgefüllt worden sein. Das Betriebssystem entnimmt der PDT Daten, die zur Installierung des Programmes nötig sind (Tasktyp, Interruptnummer ...). Deshalb muß diese Funktion immer im **Prepare**-Teil eines Echtzeitprogrammes aufgerufen werden. (Siehe Kapitel 7)

Parameter *headadr*: PDT-Adresse (Segment:Offset).

## ml2rt_real_adr — Rechne Adresse um: Physikalisch ⇒ SEG:OFFS

Pascal PROCEDURE ml2rt_real_adr (physadr: LONGINT; VAR realptr);

C void ml2rt_real_adr (long physadr, void *realptr);

Funktion Diese Prozedur berechnet aus einer physikalischen Adresse eine Adresse vom Typ Segment:Offset. Die umgewandelte Adresse hat immer einen Offset kleiner 16 (normalisiert).

Parameter *physadr*: Physikalische Adresse.

*realptr*: Zeiger auf eine Variable, in die das Ergebnis vom Typ Segment:Offset eingetragen wird. Diese Variable ist normalerweise ein Zeiger.

Beispiel:

```
VAR   BytePtr : ^BYTE;          { Pointer auf ein Byte }
...
ml2rt_real_adr($12345, BytePtr);      {BytePtr = $1234(SEG):$0005(OFFS)}
```

Derselbe Aufruf muß in C wie folgt aussehen:

```
char    *byteptr;                     /* Pointer auf ein Byte */
...
ml2rt_real_adr(0x12345, &byteptr);    /* byteptr = 0x1234(SEG):0x0005(OFFS)*/
```

9

## ml2rt_phys_adr — Rechne Adresse um: SEG:OFFS ⇒ Physikalisch

Pascal PROCEDURE ml2rt_phys_adr (realadr: POINTER;
VAR physadr: LONGINT);

C void ml2rt_phys_adr (void *realadr, ulong *physadr);

Funktion Diese Prozedur berechnet aus einer Adresse vom Typ Segment:Offset eine physikalische Adresse.

Parameter *realadr*: Umzuwandelnde Zeigervariable (Segment:Offset).

*physadr*: Zeiger auf eine Variable, in die das Ergebnis der Konvertierung eingetragen wird.