SYSMAC ALT
ASCII Library I/F Toolkit
Operation Manual

Produced March 1999

Notice:

OMRON products are manufactured for use according to proper procedures by a qualified operator and only for the purposes described in this manual.

The following conventions are used to indicate and classify precautions in this manual. Always heed the information provided with them. Failure to heed precautions can result in injury to people or damage to the product.

⚠ DANGER! Indicates information that, if not heeded, is likely to result in loss of life or serious injury.

⚠ WARNING Indicates information that, if not heeded, could possibly result in loss of life or serious injury.

⚠ Caution Indicates information that, if not heeded, could result in relatively serious or minor injury, damage to the product, or faulty operation.

OMRON Product References

All OMRON products are capitalized in this manual. The word “Unit” is also capitalized when it refers to an OMRON product, regardless of whether or not it appears in the proper name of the product.

The abbreviation “Ch,” which appears in some displays and on some OMRON products, often means “word” and is abbreviated “Wd” in documentation in this sense.

The abbreviation “PC” means Programmable Controller and is not used as an abbreviation for anything else.

Visual Aids

The following headings appear in the left column of the manual to help you locate different types of information.

Note Indicates information of particular interest for efficient and convenient operation of the product.

1, 2, 3… 1. Indicates lists of one sort or another, such as procedures, checklists, etc.

TABLE OF CONTENTS

General overview

This section is supposed to give the reader a summary and other global information on this document; It gives an overview of the context and contents of this manual.

1 About this manual

This manual describes the use of the Library Interface of the ASCII unit types ASC11, ASC21 and ASC31. It describes how a user library is created, and how it can be used from a Basic application.

The ASCII unit is a freely programmable unit for the mid-size PLC family C200Hx. The unit's main tasks are handling serial communication and co-processing. The ASCII unit can be programmed in Basic, however, to acquire higher performances, the ASCII unit's Basic is equipped with a library interface. That library interface allows up to ten user functions -written for instance in the C language- to be used from the application in Basic.

This manual also describes how the Application Programming Interface can be used to call firmware routines from a library function. The hardware involving these firmware routines (read: 'BIOS routines') is explained and should be understood before one can use these firmware functions.

The manual is divided into four sections;

- Section one explains how library functions can be loaded and called from the Basic application. It also gives some understanding about how the user memory is affected.
- Section two describes the general way to create a user library, written in the C language. It also gives an overview of some considerations to be taken during the design of the function.
- Section three explains the benefit of using the API for writing user library functions. It gives an overview of the firmware routines that are supported, and to make the functionality of the routines more understandable, the routines are illustrated by the hard- and software involved.
- The last section is a small troubleshooting guide.

The appendices are included to give listings, examples, references and general information on creating and using library functions.

2 Intended audience

Experienced users of the OMRON ASCII units with experience in developing embedded software. To program the ASCII unit properly, special knowledge is required. If this knowledge lacks or if the functions in this manual are not applied properly, the ASCII unit or even the whole PLC system might malfunction.

WARNING: Improper use of the ASCII unit's features described in this manual can cause the PLC system to malfunction!

3 References

When reading this manual or starting writing user library functions, the following documents might be of help:

- W130:C200H Operation Manual and W322:C200Hx-CPU-Z E Operation Manual, for:
  - an explanation on memory areas.
  - ladderdiagram programming
  - PLC cycle and I/O-refresh.
- W306: ASCII unit Operation Manual, for:
  - a detailed description of communication between ASCII and PC unit.
  - description of communication with serial ports.
  - Basic language reference.
- ANSI-C programmers guide or reference guide, for:
  - datatypes, typecasting
  - (function-) pointers, prototyping.
- M68000 programmers reference manual (M68000PM/AD rev. 1), for:
  - Motorola-S format
  - Exception vector table
- M68340 users manual (M68340UM/AD rev. 1), for:
  - Register lay-out of processor-units (e.g. serial ports, timers etc.)
  - Exception vector table

4 Glossary

Acronyms:

ANSI	American National Standards Institute
API	Application Programming Interface
BCD	Binary Coded Decimal
BIOS	Basic Input/Output System
BPS	Bits Per Second
COF	Extension of so-called coff file, is output file from linker
CR	Carriage return
CTS	Clear To Send
DM	Data Memory-area
DMA	Direct Memory Access
DSR	Data Set Ready
DTR	Data Terminal Ready
FIFO	First In First Out
GCC	GNU C/C++ Compiler
HEX	Hexadecimal
HMI	Human Machine Interface
IR	Internal Relay-area
ISR	Interrupt service routine
JIS	Japanese International Standard
LF	Line feed
MAP	Extension of map-file, is output file from linker
MPU	Micro-Processing Unit. (micro-processor with built-in peripherals)
MTS	Extension of Motorola-S format file, containing machine code
OBJ	Extension of object file, is output file from compiler (also .O instead of .OBJ)
OCT	Octal
PC	Programmable controller; The main unit on the PLC system
pc	program counter
PRM	Extension of parameter file, is input file for linker
RTC	Realtime clock
RTS	Request To Send
SRAM	Static Read-Only-Memory
WDT	Watchdog Timer

Explanation of terms:

Baud	Number of bits per second, only used for serial communication.
Binary Coded Decimal	A coding technique, where a binary number of four bits represents a decimal number from 0..9 .
Cross-compiler	A tool to convert a program to object code for an other machine.
Hexadecimal	A coding technique, where a number from 0..9,A..F represents a decimal number from 0 .. 15 .
Interface	Takes care of interaction between two different devices or units
Interrupt	A request from a device to the processor, to stop the current task, and start an ISR. Normally, an interrupt is generated when immediate data processing is required.
Interrupt Service Routine	This routine is executed when the corresponding interrupt occurred. It is usually optimised for speed. Also called interrupt handler.
I/O-refresh cycle	At the end of each program cycle, the PC unit updates communication with all other units during the I/O-refresh cycle.
Octal	A coding technique, where a binary number of three bits represents a decimal number from 0..7 .
Pointer	A variable containing a vector to the actual variable plus the type of the variable
Prototyping	A programming technique, telling the compiler what functions the file contains.
Register	A group of bits (usually 8,16 or 32) inside the processor, used to store and manipulate data.
Run-time	During execution of a program.
Stack	A part of memory that is used to temporarily store data. This data can be either passed arguments during function calls, local variables or status info during function/ISR calls. The stack grows downwards by pushing something onto the stack. Data can be retrieved by popping data from the stack. The M68K uses the A7 register to point to the last pushed word on the stack.

1 Using the Library Interface in BASIC

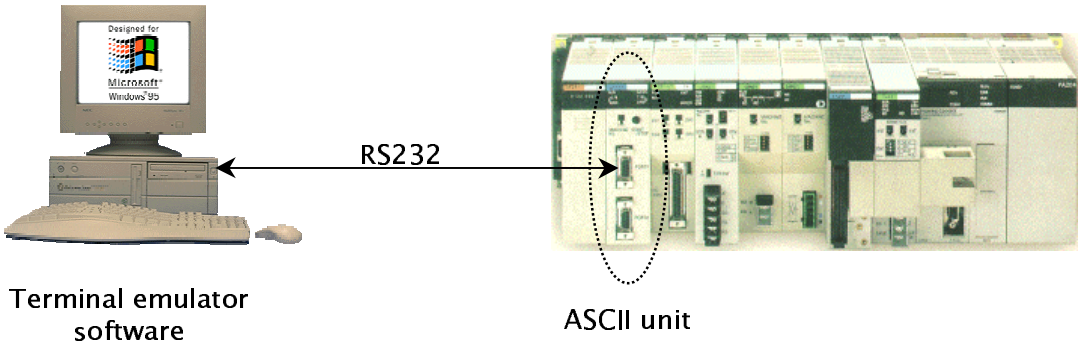
This section gives general information about library functions from a user's perspective:

- Loading a library function into user memory and the effect on memory,
- How to call a library function from a Basic prompt/application.
- What happens when a library function is called
- The way memory is built up.

1-1 The operation of the Library Interface

This section explains how a library function is loaded into user memory, how the corresponding Basic user function is defined and how the function is called from the Basic program. Section 1-1-1 explains these three steps in detail, and section 1-1-2 shows an example of using a simple library function according to the same three steps.

All Basic statements in the example of section 1-1-2 are explained in section 1-3. All that is needed for the example is a cross-compiled library function and an ASCII unit (test-)setup as shown here:



1-1-1 How to use a library function

1. *Load the library function*

Loading a library function is done with the LIB LOAD statement. The library function to be loaded can be of two types:

- compiled for a fixed address,
- compiled for a free address.

If the library function is compiled to be stored at a fixed address, this address should be passed to the LIB LOAD statement. Else, the ASCII unit will allocate a proper memory block for the library function itself. For more information on fixed or free addresses for library functions, see section 2-2-2.
If a library function is to be loaded at a fixed address, but some Basic variables reside somewhere in that part of memory, LIB LOADing will result in an 'out of memory' error. The CLEAR command can be used to remove these temporary memory blocks, assuming that Basic is operating in dynamic mode.

2. *Define a Basic user function for the library function*

Defining a Basic user function for the library function is done with the DEF LIBFN statement.
This statement tells Basic the type of the parameters and the return value. This statement is in a way similar to the normal DEF FN statement, the only difference is that in this case the functionality is described in the library function, instead of in a Basic function. Also, the programmer must specify the number of the library function to which this user function is referring.

It is possible with the DEF LIBFN statement to determine output and throughput parameters as well. This is done by means of the VARPTR function. This function returns the address of a variable, so that the library function can access the parameters by reference, allowing parameter values to be altered.

Two other options are included, STACK and WDT. If the option STACK is specified, the library interface will display the contents of passed parameters on the stack before and after calling the library function. This allows debugging of parameter passing. By specifying the WDT option, the programmer can specify whether or not Basic should refresh the watchdog timer. The watchdog timer is a counter, counting downwards and reaching zero 60ms after it was reset (refreshed). If it is not constantly refreshed in time, the PC unit assumes that the ASCII unit has crashed and stops operating due to a so-called 'special unit error'. Normally, the WDT should be refreshed by from within the library function. Only when the ASCII unit causes a special unit error on the PLC, the WDT option can be used to determine whether or not the library function has refreshed the WDT in time. The WDT option should not be used in normal situations, because the functionality of the WDT is to track if units have crashed. By specifying the WDT option, the PLC system cannot detect if the library function crashes.

3. *Calling the function*

Calling the user function is done like any other Basic function, except the name is to be preceded by 'fn' when calling the function. For instance, if a library function is declared with the DEF LIBFN statement, and the name given to the function is 'example', then the function can be called by the name of 'fnexample'. This is similar to user-defined Basic functions. Note that when a program is started (by e.g. the RUN statement) all user defined functions are cleared. This means that if a library function is to be called from a Basic program, it must be defined in this program, prior to the function call.

When a library function is called, the library interface will read the input parameters onto the system-stack, and call the function. When the function is completed, the return value is read from the appropriate register, and the output parameters are read from the stack. The library will then write these output values into the proper variables. This procedure is explained in more detail in Appendix D.
WARNING: The function header of the Basic function must correspond to the header in the source code of the library function. If this is not the case, calling the library function will probably cause the ASCII unit to crash.

BASIC	⟺	**LIBRARY FUNCTION**
(library function is loaded as nr. 3) - def libfn testfun(INT,LNG),INT,3 - print fntestfun(99,99999) -		- Library function: nr. 3, takes care of WDT - Inputs: short integer, long integer - Returns: short integer

This illustration gives an example of corresponding declarations in Basic and in the Library function.

1-1-2 Example of using a Library function

This section gives a complete example of using a library function from the Basic prompt. The statements used here to load, define and call the library function can also be executed from a Basic program. This example shows how a library function can be used from Basic on an initialised ASCII unit. The example library function 'example' returns the value of the input, and increases the input parameter by one. It is assumed that this library function is already compiled to be downloaded at fixed address &H30000. How to compile the source of a library function to the Motorola-S format is described in section 2.

1 Look at memory map [optional]

```
> map
   ****************** MAP HEAP AREA ******************
   Address    Unit Size    Bytes       Type       Link
   (Hex)      (Hex)       (Decimal)
   E000       A             40         file        0
   E028       C7F5          204756     free        E028
   ********* ds_mm_table TABLE ************
   Free chain address(Hex):E028
>> END
   Total free blocks:      1
   Total allocated blocks: 1
>> END
>
```

At start-up, there is one free memory block of approximately 200 Kbytes long and one small block of 40 bytes, used to store global data. For more information about this small block, see section 3-5.

2 Look at library entry list [optional]

```
> lib
0 - No Library defined
1 - No Library defined
2 - No Library defined
3 - No Library defined
4 - No Library defined
5 - No Library defined
6 - No Library defined
7 - No Library defined
8 - No Library defined
9 - No Library defined
>
```

No libraries have been loaded yet.

3 Load the library function

```
> LIB LOAD #1,"COMU:9600,8,n,2,CS_OFF,RS_OFF,DS_OFF,XN_ON";"example",&H30000
Library down-load checksum = 3AEC
>
```

This example will load a library and gives it the name 'example'. Once this statement is input, the ASCII unit will wait until the library is sent. When the transmission is complete, the ASCII unit will wait again, until the user presses CTRL-C. Once it is pressed, the checksum is returned and the prompt will appear again.

4 Look at memory map again[optional]

```
> map
   ******************* MAP HEAP AREA ******************
   Address    Unit Size    Bytes       Type       Link
   (Hex)      (Hex)       (Decimal)
   E000       A             40         file        0
   E028       A6            664        file        0
   E2C0       874F          138556     free        30320
   2FFFC      C9            804        file        0
   30320      3F37          64732      free        E2C0
   ********* ds_mm_table TABLE ************
   Free chain address(Hex):30320
>> END
   Total free blocks:      2
   Total allocated blocks: 3
>> END
>
```

Though the library function was stored at &H30000, the memory map shows the address &H2FFFC. At this address the header of the concerning memory block starts. Storing the library function at the fixed address caused the free memory block of 200 Kbytes to split up into two smaller blocks. When the first library function is loaded, a block of 664 bytes is created that contains the Basic Library Function entry Table.

5 Look at library entry list again[optional]

```
> lib
0 -                            example    800 bytes  28.07.98 16:49:52
1 - No Library defined
2 - No Library defined
3 - No Library defined
4 - No Library defined
5 - No Library defined
6 - No Library defined
7 - No Library defined
8 - No Library defined
9 - No Library defined
>
```

Since this is the first library loaded, it is automatically loaded as library 0.

6 Define and call the user function

```
>
> DEF LIBFN xyz(ADDR),INT,0
>
> a%=100
> b%=fnxyz(VARPTR(a%))
> print a%,b%
 101      100
>
```

Here, the user function was given the name ‘xyz’ to point out that this name is not related to the filename nor to the library function name, but only to the library function number (zero in this example).

1-2 User memory in BASIC

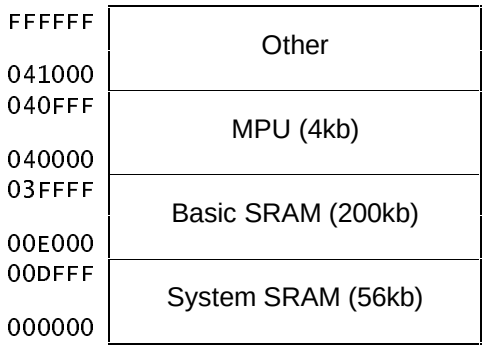
1-2-1 Memory blocks

The table below shows that the ASCII unit's RAM is 256 Kbytes big. Of this memory 56 Kbytes are used by the system, and the other 200 Kbytes is user memory. In this part of memory all data resides; all programs, library functions and variables are stored here, each in their own memory block.

There are two types of memory blocks, permanent memory blocks and non-permanent memory blocks. Basic programs and library functions are stored in permanent memory blocks. When the ASCII unit is turned off and on again, these blocks remain intact and accessible. This is not the case with variables, since they are stored in non-permanent memory blocks.

When the power is removed from the ASCII unit, the battery will backup the power to the memory, so all data remains intact. However at start-up, the memory manager deletes all non-permanent memory blocks. Basically, there is no difference between these two block types, the only difference is one bit telling the memory manager of the ASCII unit what the block type is.

To prevent permanent loss of the user memory's contents, the ROMSAVE statement can be used. It is described in the ASCII unit operation manual. Executing a ROMSAVE statement saves all Basic programs and all Library functions. The Basic programs and Library functions can be restored by means of the ROMLOAD statement.



'Memory map'

1-2-2 Block headers and the MAP statement

The management of all these memory blocks is maintained by the ASCII unit's memory manager. Each free memory block has a block-header of six bytes, containing the size of that block, and a pointer to the next free block. The last free block points to the first free block, this way all free blocks are chained in a linked-list. Consequently, all parts of memory that are not in this linked list are in use.

Each time the size of a Basic program is increased, a library function is loaded or a variable is created in Basic, a part of free memory is allocated. This means that a part of a free block is allocated and removed from the linked-list.

It is possible to take a look at the memory map with the MAP statement (see section 1-3). It shows all memory blocks, including the start and the size of the blocks. The start of an allocated block indicated by the MAP-statement points to the header of the block, not to the data in the block. The header of an allocated block is two bytes long, containing information on the size.

If this block contains a Basic variable, the address retrieved from the MAP statement, increased by two gives the address of the data. Poking to this address can change the value of the variable quickly, however, a Basic programmer should never poke. **WARNING**: Poking might corrupt the linked list causing the PLC-setup to crash! *For more information about the memory manager, see section 3-5 .*

1-3 Library Interface BASIC statements

In this paragraph the library interface statements are explained. For an example in which all these statements are used, see section 1-1-2. The functions are also used in Basic listings in Appendix C.

def libfn

Description	Declares a library function that was previously uploaded to the ASCII unit.
Syntax	DEF LIBFN <*function_name*> ([*par_type1*, … *par_type n*]), <*return_type*>, <*library expression*> [,*WDT*] [,*STACK*]
Parameters	*Function_name* is the user defined name, used to call from the BASIC application. *Par_type1 … par_type n* , determine the datatypes of the parameters sent to the function. Valid types are int, lng, sng, dbl, str and addr. WARNING: If the parameters do not correspond with the parameters of the function source file in the correct order, the ASCII unit might crash when the function is called. When the function is to be called, parameters of datatype addr can be passed by means of the VARPTR() function (passing by reference). *Return_type* determines the datatype of the return value. Valid types are int, lng, sng and dbl. *Library expression* specifies the number of the library function, it is a numeric value in the range from 0 to 9 . If the *WDT* option is specified, the system will perform the watch-dog refreshes, else it is assumed that the watch-dog timer refreshes are performed from the library function as explained in section 2-2-4. If the *STACK* option is specified, the library interface will display the contents of the stack before and after calling the library function.
Remarks	When a parameter is an output- or throughput parameter, the datatype should be specified as type addr. Note that a user defined function doesn't necessarily need a parameter, but it ALWAYS must return a value. *For more information on WDT refreshes, see section 2-2.*

Lib

Description	Gives a list of all the library entries for library functions.
Syntax	LIB
Parameters	-
Remarks	If a library entry has no corresponding library loaded then “no library defined” is displayed

lib del

Description	Deletes one, or all library functions from the library.
Syntax	LIB DEL <*lib expression* | ALL>
Parameters	*Lib expression* is a numerical expression with a value in the range from 0 to 9. It indicates the library function to be deleted. If ALL is specified, all library functions are deleted.
Remarks	Sometimes when a library function with a fixed address is to be loaded, the memory must be freed up by LIB DEL ALL and the CLEAR statement. Checking whether the concerning part of memory is free, can be done with the MAP statement. Consequently, to replace a library function with another, the existing library function must be deleted first.

lib load

Description	Uploads a library function (*filename.mts*) into user memory. Once the transfer has completed, the ASCII unit waits for the user to press CTRL-C.
Syntax	LIB LOAD #<*port expression*> [,<*com_string expression*>] [;<*lib_name*>] [,<*library address*>]
Parameters	*Port expression* is the number of the port connected to the terminal. Valid values are 1 and 2, and if the unit is a ASC31 also port 3 can be specified. *Com_string expression* is the communication definition string; if omitted, the current active communication settings are used. *Lib_name* is the name the user gives to the library function. If the name is omitted, the name "lib#*n*" is given to it. *Library_address* is the address where the user desires the library to be stored in memory. If that part of memory is (partially) in use, an "OUT OF MEMORY" error will occur. If this parameter is omitted, the memory manager of the ASCII unit will attempt to find a suitable place in memory.
Remarks	It is recommended to use the software flowcontrol option, XON, in the *com_string expression* when using the LIB LOAD statement. The library name *lib_name* doesn't need to be the same as the name used in the source of the library function, nor as the name used in the DEF LIBFN statement. *For more information on the communication definition string, see the LOAD statement, in the ASCII operation manual.*

map

Description	Shows the memory-map. From the listing of the memory-map, the location and size of a memory block can be retrieved.
Syntax	MAP
Parameters	-
Remarks	The starting address of a memory block points to the header of the block, not to the data in the block.

2 Creating a user library

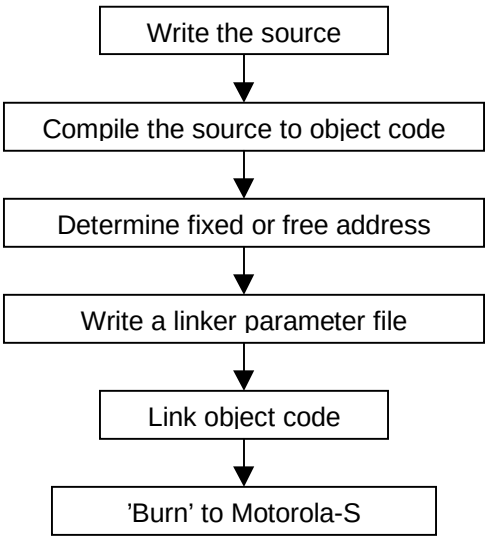
This section describes the general way to create a user library in the C language. It also gives an overview of some considerations to be taken during the design of the function. All items mentioned in this section are illustrated by the small application examples in Appendix C.

2-1 Creating a library function

When a library is to be loaded, it must consist of machine code for the M68K processor coded in the Motorola-S format. Therefore, this section describes the route from source to machine code. The source can be written in any language, as long as cross-compiler and linker can convert it to the right code. For detailed examples on creating library functions, see the C-source listings in Appendix C.

In this manual, it is assumed that the source is written in ANSI standard C language. Furthermore, the cross-compiler and linker used to write this manual is the "GNU C-language Cross Compiler for Motorola 68K". It is likely that most details are different on other cross-development systems.

The general route to create a library function consists of six steps as shown schematically:



1. **Write the source.**
   Writing the code is language dependent, and also different from programming applications on a DOS/WINDOWS machine, since the library function will become a part of embedded software. Some guidelines for designing the software are included in section 2-2.

2. **Compile the source to object code.**
   To compile, the source file and all other include files are input to the compiler. The compiler generates an object code file (.O).

3. **Determine fixed or free address.** This is actually a part of the application design, but it is included in this list to point out that it is an essential part of linking. It does not only effect the linking but also the restrictions for writing the source file. More details on free or fixed address can be found in section 2-2.

4. **Write a linker parameter file.**
   This file contains the linker options for the source file. For every library function, the programmer should write a parameter file. The parameter file (.LD) informs the linker about the concerning memory layout, and some specific options about compiling conventions.
   Furthermore, this file contains a MEMORY block with an ORIGIN option that tells the linker where the programmer wants his functions and variables to be stored in memory. The memory origin depends on the way the library function is loaded into the ASCII unit, with a free address or a fixed address:
   - *Free address*: To load the library function with a free address, the LIB LOAD statement should not contain a target address; LIB LOAD will allocate a free memory block to store the function in. The address in the ORIGIN option should be set to &H00000 to load the function at the beginning of the allocated block.
   - *Fixed address*: To load the library function with a fixed address, the LIB LOAD statement should contain a target address, specifying where to place the library function. This address must be equal to the lowest address in the ORIGIN option.

   See Appendix E for more information on the linker parameter file.

5. Link object code.
To link, the parameter file and object code files of the source are input to the linker. The parameter file will tell the linker how the files are to be linked. Also see Appendix E for more information on this topic.
The linker generates two files, the coff file and the map file.

- *coff file*: The coff file (.COF) contains the linked machine code, coded together with some debug code and references to the source file.
- *map file*: The map file (.MAP) contains the linking results and code statistics and placement as well. From the map file, the absolute addresses of the functions and variables can be retrieved.

6. 'Burn' to Motorola-S.
This step encodes the coff file to Motorola-S format. The output file, an MTS file, is the input file for LIB LOADing the library function to the ASCII unit.

Steps 2, 5 and 6 can be automated by using a makefile. The advantage is that commands for these steps do not have to be retyped each time and that the make utility automatically detects which files have to be rebuilt. It detects this by checking if changes have been made to files that the file to be built depends on. See Appendix C for examples of makefiles.

The library module creation process could look like this:

1. Create a small C file example.c:

```
C:\example>edit example.c↵
```

with the following contents:

```
/* A small example program */
long main(short arg2, short arg1)
{
  if (arg2!=0)
  {
    return (long)arg1/(long)arg2; /* arg1 / arg2 */
  }
  else
  {
    return 0;
  }
}
```

2. Compile the source file (example.c) to an object file (example.o)

```
C:\example>compile example.c↵
example.c: In function 'main':
example.c:2: warning: return type of 'main' is not 'int'
```

Before compiling, make sure that the path of the compiler is set correctly.
The compiler generates a warning because main returns a long instead of an int. This can safely be ignored.

3. Use the MAP command on the ASCII unit to check which addresses are unused

```
> map↵
   ****************** MAP HEAP AREA ******************
   Address    Unit Size    Bytes       Type       Link
   (Hex)      (Hex)       (Decimal)
   E000       A             40         file        0
   E028       C7F5          204756     free        E028
   ********* ds_mm_table TABLE ************
   Free chain address(Hex):E028
>> END
   Total free blocks:      1
   Total allocated blocks: 1
>> END
```

This example shows that there is a free memory block that starts at address &HE028. The size of this block is 204756 bytes. In this example, the lowest possible start address of the library function is &HE02C, because the memory manager uses up to four bytes when a block is allocated.

4. The default linker parameter file 'ascii.ld' is configured for address &H30000. This is within the free memory block. Therefore, the default linker parameter file can be used.

5. Link the object file (example.o) to a coff file (example.cof)
   A map file (example.map) is generated. This file shows the memory layout of the coff file.

```
C:\example>link -o example.cof example.o -Tascii.ld -Wl,-Map=example.map↵
```

6. Convert the coff file (example.cof) to a Motorola-S record file (example.mts)

```
C:\example>objcopy -O srec example.cof example.mts↵
```

To use the created library module, follow these steps:

1. Load the library module into the ASCII unit:

```
> LIB LOAD #3, "COMU:9600,8,N,2,CS_OFF,RS_OFF,DS_OFF,XN_ON";"EXAMPLE",&H30000↵
```

Now transfer the file 'example.mts' to the ASCII unit. Press Ctrl-C when loading is finished.

2. Create a BASIC program to define the library function and to perform a call to it:

```
> 10 DEF LIBFN EXAMPLE(INT,INT),LNG,0↵
> 20 PRINT FNEXAMPLE(912,5) ↵
> RUN↵
Please wait, compiling ... Finished.
 182
```

Note that the order of arguments in the BASIC function call is the opposite of the order of the arguments in the C function.

2-2 Application design considerations

Because the library function will become a part of cross-compiled embedded software, it differs from 'regular' programming on a DOS/WINDOWS platform, because:

- Debugging is not possible on the platform itself. For debugging, an M68K emulator is required.
- The Library Interface acts like a shell around the actual library function, so that the programmer must be aware of its functionality.

The rest of this section contains several points that should be kept in mind when designing an application.

2-2-1 Order of arguments passed from BASIC.

For GCC, the arguments for the primary function are received in the opposite direction as they are passed from BASIC. Care should be taken to make sure that the order of the arguments in the primary library routine is reversed. If a BASIC program calls a library function like 'FUNC(arg1, arg2)', the function should be declared as 'FUNC(arg2, arg1)' in C. Unexpected behaviour may be seen when arguments are treated in the wrong direction.

2-2-2 Fixed or free address.

The application programmer can divide up memory the way he desires by specifying the target address (fixed address) when executing the LIB LOAD statement. If the target address is omitted, a free address is used. Compiling the library function with a free address has three disadvantages:

- Global data or constants cannot be used,
- Calls to other functions (absolute jumps) are not allowed, (note that GCC uses subroutines to convert float and integer types!) It is possible to check the use of absolute jumps by looking for the keyword 'jsr' in the assembly file.
- The primary function can not be main.

Most of the time, the disadvantages of using a free address outweigh the advantages. If there is no special need for using a free address, then don't. It is also possible to store a library function compiled for a fixed address, as if it where loaded with a free address. This is done by loading the compiled function with LIB LOAD without specifying a target address. The ASCII unit will determine and display a proper address to store the library function. Instead of transferring the Motorola-S file of the function, the programmer can abort LIB LOAD and re-compile the library function at the fixed address that LIB LOAD displayed to be an appropriate location. This way the ASCII unit divides up the memory, without the programmer experiencing the disadvantages of compiling with a free address.

Normally a function in machine code should begin at an address that is a multiple of two to enable a 'Jump To Subroutine' to it. Still any address in free RAM can be specified for the fixed address. When the library function is loaded at the fixed address, a block is allocated that is larger than the library function itself since it is to be preceded by a header. The header of the block must start at an address that is a multiple of four, because the memory manager can only allocate blocks with a size that is a multiple of four. For instance, in the example in section 1-1-2 a library function is loaded at fixed address &H30000. A two byte header should precede the library function, so the starting address would be &H2FFFE. Since this is not a multiple of four, the starting address becomes &H2FFFC.

2-2-3 Data differences.

The data types in Basic and C are not completely similar. For instance, in the C language any standard variable can be signed or unsigned; This is not the case in Basic. In Basic the data type string is allowed, in C, however, strings are realised by charpointers or arrays of chars. Such a 'string' can be manipulated by means of functions from the standard library string.h .
When passing parameters between the Basic application and the library function, the programmer must be aware of differences like these.

The following table gives an overview of the standard data types in C and in Basic.

Description of data type	Type identifier in Basic	Type identifier in C
single character	-	char
String	STR , symbol $	*char* , char[]*
short integer	INT , symbol %	(short) int
long integer	LNG , symbol &	long int
single precision real	SNG , symbol !	float
double precision real	DBL , symbol #	double
Enumeration	-	enum

There are some special cases in C that must be taken into account:

- **String.** The string data type in C is not the same as in Basic. A string in the C language is actually an array of char's, that's why most compilers consider the data type char[] as char* . The last element of the array contains a zero; this is a so-called null-terminated string.
  A Basic string, however, is preceded by a header containing the length of the string. This knowledge was used in the example 'CHANGE CASE' in Appendix C where a string was input to the library function, without specifying the length. See also Appendix D in the ASCII unit operation manual for the format of Basic string variables.
- **Enum.** Basic doesn't support the enumerated data type. Most C compilers treat enumerated data as integers or have an option to do so. This makes it possible to pass the concerning data to the Basic application. In fact this is type casting from enum to int.
- **Void.** The C language supports the void data type which is actually no type at all. It is mostly used to tell the compiler either that:
  - A pointer is not pointing to anything in particular.
  - A function is not returning a value, or no parameters are passed to the function

  A library function must always return a value, but doesn't necessarily need an input parameter. In other words, the void specifier can be used to specify that the library function has no parameters, but it can not be used to specify the data type of the return value.
- **Boolean values.** A 'FALSE' expression like (1==0) is equal to zero in the C language, like the expression (0=1) is equal to zero in Basic as well. The value of a 'TRUE' expression however, is not standardised. In the ASCII unit's Basic, a 'TRUE' expression like (1=1) always returns '–1'. In the C language however, an expression like (1==1) is can be equal to any value, depending on the compiler being used. These differences are not necessarily a problem, but must be taken into account when passing expressions from or to a library function.

2-2-4 Watchdog timer.

The ASCII unit has a watchdog-timer. In fact it is just a counter, counting downwards and becomes zero, 60ms after the timer was reset. When the timer becomes zero, a special unit error is generated. In other words, the units must perform a watchdog-timer refresh that resets the watchdog timer at least once every 60ms.

When executing the LIB LOAD statement, the user can specify whether the Library Interface should refresh the watchdog timer, or not. If the programmer decides to take care of the watchdog himself, the design of the application should take into account that the watchdog should be refreshed every 10 milliseconds. Though 60ms normally would suffice, it is advised to refresh the watchdog timer every 10ms because interrupts could postpone the refresh of the watchdog timer too much. This means that the watchdog must be refreshed at several locations in the source, especially in a loop.

Refreshing the watchdog timer can be done in two ways:

1. By means of the inline assembly statement TRAP #00. Using the GNU cross-compiler, this is implemented as: ASM ("TRAP #00"); See also Appendix F.
2. Use the macro from API.H. Using the macro _wdt_refresh will refresh the watch-dog timer as well.

These methods differ since the macro will work with any compiler, but the TRAP-routine can't be used with compilers that don't support inline assembly. Another difference is that the macro only refreshes the watch-dog timer. The TRAP-routine, however, consumes more time because it also performs other checks for system integrity.

2-2-5 Use of global/static data.

Normally, the use of global data is avoided as much as possible. However, global data in a library function can be very useful. Library functions are stored in permanent memory blocks, so any global or static data will remain intact even when the power is cut off from the ASCII unit. It could even be possible to create an extra library function only containing global data (so it's a dummy function).
Using global data in a library function, from Basic or an other library function is done by passing the address of the data. Using globals in the source of the library function is only possible when the function is to be compiled at a fixed address, because globals are compiled to be stored at an absolute address.

Normally, global variables can be initialised to a certain value by adding an assignment to the declaration of the global variable. This principle can not be used in a library function!

Actually, an initialised global variable has the correct value the first time a library is loaded. Once the variable has been modified, it is not initialised again. Even cutting off the power from the ASCII unit does not re-initialise the variable. If a global variable is to be initialised, then this initialisation should be performed by a normal assignment in an initialisation routine or by deleting and re-loading the library again.

2-2-6 Programming rules.

There are four rules to keep in mind when writing the source of a library function:

1) If the library is to be compiled with a free address, the primary function can not have the name ’main’. In this a case a dummy main must be written as well. The primary function must be the first object in the object code. Therefore, the primary function must be at the start of the source file. The linker parameter file must be changed to make sure that the file containing the primary function is loaded first by adding a STARTUP(filename) command. The ‘PRINT STRING’ example in Appendix C makes use of this principle.

2) If the library is to be compiled with a fixed address, the primary function should normally have the name 'main'. Sometimes it is desired to give the primary function a name that reflects it's functionality. This is allowed. In this case rule number one should be obeyed.

3) If the library function is to be compiled at a fixed address, it is possible to use more functions in the source. If it is compiled at a free address, using (calling) more functions is not possible. It is wise to declare a prototype or an allusion for every function in the source file(s).

4) For GCC, the arguments for the primary function are received in the opposite direction as they are passed from Basic.

3 Using BIOS functions

This section explains the benefit of using the API for writing user library functions. It gives an overview of the firmware routines that are supported, and to make the functionality of the routines more understandable, the routines are illustrated by the hard- and software involved. In Appendix B, a reference of the routines is included.

3-1 General description of BIOS and API functions

The ASCII unit contains several hardware aspects: serial ports, communication with the PLC, timers and DMA controllers. These hardware units must all be controlled or supervised by embedded software in the ASCII unit's ROM. This embedded software exists as a set of functions that perform Basic input/output tasks, therefore these firmware functions are called BIOS routines.

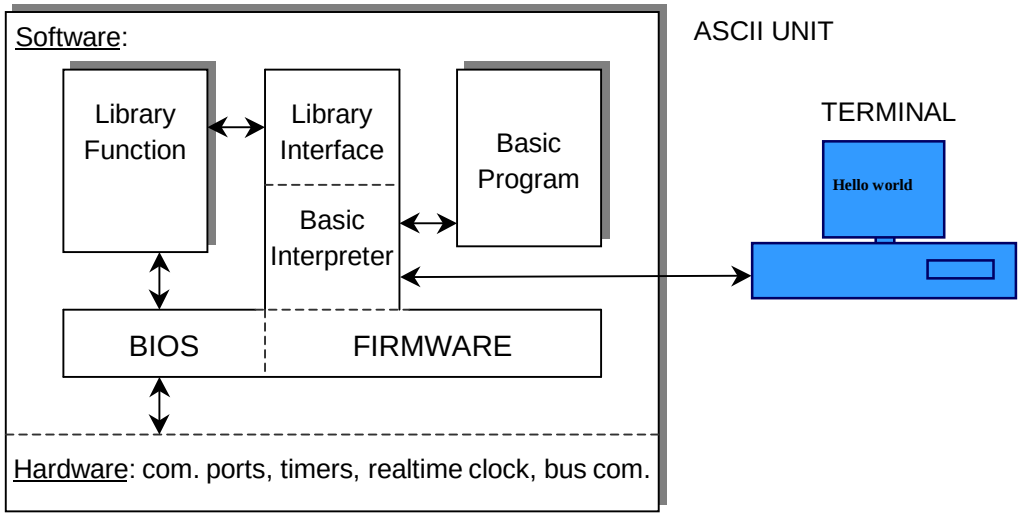
Some of the firmware routines are low-level BIOS routines, directly effecting the hardware. Others are high-level BIOS routines that are wrapping their low-level versions or make use of the low-level BIOS routines. Since most BIOS routines directly effect the hardware, it is important to understand the concerning hardware before using a function. The hardware and the concerning BIOS routines can be divided in six categories named after the software units they reside in: Interrupt handler, I/O bus handler, main executive, memory manager, port handler and the realtime clock.

See sections 3-2 to 3-7 for more details.

The BIOS routines are normally called by the system software, but can be called by the programmer of a library function as well. The BIOS routines are written in an Application Programming Interface (API). This toolkit makes it easier for the programmer to use BIOS routines.

To use the functions defined in the API, only including "api.h" is needed. Once the API is included, the firmware functions and the data types described in this manual can be used. Since this API is optimised for the compiler used (GCC), one should not copy pieces of the API into the source which makes the source compiler-dependent.

See the listing of user available data of the API in Appendix A for more information.



In this illustration, a link can be seen between the Library Function and the BIOS routines which is established by including "api.h" in the source of the library function. This link makes it possible for the user to directly use the hardware and fully exploit the possibilities of the ASCII unit. **WARNING**: Improper use of the API or improper use of the BIOS routines may cause the ASCII unit to malfunction!

3-2 Functions for interrupt handling

In the Application Programming Interface, three routines are included that are used when writing an interrupt service routine. These routines are:

- Register interrupt. To register an interrupt service routine in the exception vector table.
- Mask Basic interrupt. To mask one or all Basic interrupts.
- Unmask Basic interrupt. To unmask one or all Basic interrupts.

WARNING: Working with interrupts requires specific knowledge of both the MPU of the ASCII unit and the principles behind interrupt servicing. Make sure that both these matters are understood before developing your own interrupt service routine.

When an interrupt service routine is created, it should be optimised for speed; An interrupt service routine should be completed as fast as possible. To install a routine to be an interrupt service routine, find the corresponding (free) vector in the vector table and appoint the address of the routine to that vector, using the _register_int() routine. The following table shows some of the exceptions supported by the M68K.

No.	Offset	Description
0	000	Not used
1	004	Not used
2	008	Bus error
3	00C	Address error
4	010	Illegal instruction
5	014	Divide by zero
6	018	CHK instruction
7	01C	TRAPV instruction
8	020	Privilege violation
9	024	Trace
32	080	Trap #00 vector (WDT refresh)
33	084	Trap #01 vector not in use
:	:	:
42	0A8	Trap #10 vector not in use
43	0AC	Trap #11 vector in use
44	0B0	Trap #12 vector in use
45	0B4	Trap #13 vector in use
46	0B8	Trap #14 vector not used
47	0BC	Trap #15 vector not used
65	104	Timer1 interrupt
67	10C	Timer2 interrupt

The base address of the vector table is &H00000, therefore the offset address of a vector as shown in the table is identical to the real address of this vector. For instance, the address of vector 65 (0x41) can be calculated as follows: 0x41 $\times$ 4 = &H00104.

In some interrupt service routines it is needed to mask a certain Basic interrupt. When _mask_basic_int() is called it will return the previous mask level. This value must be stored in a local variable, so it can be passed to _unmask_basic_int() in order to restore the old mask level before function scope is left. The following table shows the mask numbers and the corresponding Basic interrupts (order is also order of priority).

1	on com 1
2	on com 2
3	on key 0
:	:
12	on key 9
13	on pc 1
:	:
111	on pc 99
112	on alarm
113	on timer
114	on time$
115	on error
116	wait

An Interrupt Service Routine should save the registers, that are used, on the stack and restore them again from the stack when the routine is ended. Also an ISR is terminated by a “RTE” instead of a “RTD” or “RTS”. Many compilers have an option/directive to determine which routines are ISR’s and which are not. For the GCC, the programmer must meet the restrictions mentioned above by himself. This can be done be writing a ‘wrapper’ for the actual routine.

The following code gives an example of how a new interrupt service routine can be set. The interrupt service routine of timer1 is replaced:

```
/* EXAMPLE; Interrupt service routines */
{
  ...
  DISABLE_TIMER1;
  (void)_register_int(65, (ULONG)&my_timer_isr_wrapper());
  ENABLE_TIMER1;
  ...
}


void my_timer_isr_wrapper(void)
{
  asm( "MOVEM.L %D0-%A5,-(%A7)" );  /* push registers used in my_real_timer_isr() */

  (void) my_real_timer_isr();

  asm ( "MOVEM.L (%A7)+,%D0-%A5" ); /* pop registers again */
  asm ( “UNLK %A6” ); /* Confirm matching “LINK %A6” instruction in assembly */
                      /* else remove this line (see appendix F)              */
  asm ( “RTE” );
}


void my_real_timer_isr(void)
{
  CLEAR_TIMER_INTERRUPT_FLAG; /* prevent interrupt from repeating */
  ...
}
```

3-3 Functions for exchanging data with the PC unit

For more information on data exchange methods, see ASCII unit operation manual section 6.

3-3-1 General

Communication between any special unit and the PC unit is performed during the I/O refresh cycles via the I/O bus on the backplane. Since the ASCII can address almost all memory areas in the PC unit, the ASCII unit can communicate (indirectly) with other units. Communication between the PC unit and the ASCII unit can also be done at any other moment, by means of interrupts:

- The PC unit can interrupt the ASCII unit's current task, requesting a data exchange.
- The ASCII unit can 'peek' and 'poke' directly into the memory of the PC unit, during I/O-refresh.

The BIOS routines for I/O bus communication make use of two memory areas in the PLC:

- The ASCII unit has 100 words in **DM memory**, shared with the PC unit, of which 10 words contain start-up options. During the I/O refresh cycle, the remaining 90 words are used both as input buffer and as output buffer, to exchange data between the PC unit and the ASCII unit. The programmer can divide these 90 words up himself to specify the size of the input- and output buffer.
  To calculate the starting address of the shared memory, use the formula: $m = 1000 + 100 \times$ unit no.
- The ASCII unit has 10 words in **IR memory**, containing several communication status bits. There are two versions of this part of memory: The real IR words in the PC unit, and a local copy of these words in the ASCII unit.
  When a programmer wishes to change the data in the IR memory, these changes are made in the local copy. Afterwards, an update routine must be called to update the real IR memory in the PC unit.
  When a programmer wishes to read the data in the IR memory, first an update routine must be called to update the local copy of the IR memory with the real IR memory in the PC unit.
  To calculate the starting address of the special unit memory:
  use the formula: $n = 100 + 10 \times$ unit no. , if the unit number is in the range 0..9,
  use the formula $n = 300 + 10 \times$ unit no. , if the unit number is in the range 10..15.

3-3-2 Summary of low-level routines

Many of the high level BIOS routines that perform I/O bus communication, make use of low-level routines.

- Routines to maintain the status bits;
  - _read/write_IR_word(). To read or write one of the 5 words in the IR memory
  - _set/clr_IR_bit() . To set or clear a bit in one of the 5 words on the IR memory
  - _update_IR_in/out(). To update the IR memory in the ASCII unit or in the PLC memory

  These routines can be used to realise communication between the ASCII unit and the PC unit in a way that can be compared to data exchange method no. 3 in the ASCII unit Operation manual (section 6).
- Routines for converting hexadecimal, binary coded decimal, and octal coded words;
  - _HEX2BCD()
  - _BCD2HEX()
  - _BCD2OCT()
  - _OCT2BCD()

3-3-3 Summary of high-level routines

There are three different types of communication routines:

- PC eget/eput. The ASCII unit and the PC unit can both asynchronously read or write to the ASCII unit's shared memory. These routines are used in combination with the IORD and IOWR commands in the ladder program. The IORD and IOWR areas in the DM memory are set in words DM m+6 and DM m+7.
  Corresponds to data exchange method no. 4 in the ASCII unit Operation manual (section 6).
- PC read/write. During an I/O refresh cycle, the ASCII unit reads or writes a maximum of 255 words in CPU memory. Normally this is performed on a trigger from the PLC. The words IR n+3 and IR n+4 are used to specify the memory part that is to be transferred. A special option is included, to make the routines perform like the PC@read and the PC@write. These routines independently read or write the PLC memory. In other words, calling the PC read/write function gives access to two different data exchange methods.
  Corresponds to data exchange methods 1 and 2 in the ASCII unit Operation manual (section 6).
- PC qread/qwrite. Also called quick-read and quick-write. On a request from the ASCII unit, the PC unit reads or writes a maximum of 128 words to or from the ASCII unit. These routines are used in combination with the IORD and IOWR commands in the ladder program. The handshake and settings are performed by means of words IR n+5, IR n+8 and IR n+9.
  Corresponds to data exchange method no. 5 in the ASCII unit Operation manual (section 6).

The following table gives an overview of the four communication methods mentioned.

Method	Initiating unit	Required ladder statement	Max. amount of data
PC Eget/Eput	ASCII or PC unit	IORD/IOWR (#00□ □)	180 bytes
PC read/write	PC unit	MOV	255 words
PC@ read/write	ASCII unit	-	255 words
PC qread/qwrite	ASCII unit	IORD/IOWR (#FD00)	128 words

The following code gives an example of communication with the PC unit from a library function:

```
/* EXAMPLE; Communication with PC unit */
{
  UCHAR num_of_words, area, data;
  USHORT address;

  num_of_words = 1;
  area = 0; /* DM area */
  address = 0123;

  if(_pc_read(1, 1, &num_of_words, &area, &address, &data))
  {
    /* PRINT_STRING("ERROR!"); */
  }
  else
  {
    /* Do something with the data read… */
  }
}
```

3-4 Functions for OS error messages

The ASCII unit's main executive is a single tasking operating system. It communicates with other software units and therefore handles the error messages from those software units. The Application Programming Interface supports five routines to handle error messages. These routines are:

1) _set_error().
2) _del_errors().
3) _update_errors().
4) _read_error().
5) _print_errmsg().

Error messages are stored in the so-called error FIFO. In fact it is a cyclic stack that can contain a maximum of 30 error messages. With the _set_error() routine an error can be pushed onto this stack. If already 30 errors are on the stack, pushing a new error message on the stack will result in loss of the oldest error message. With the _read_error() routine, an error can be read from the stack, and if desired printed with the _print_errmsg() routine.

After popping from the stack or pushing to the stack, the status of all remaining errors on the stack must be updated with the _update_errors() routine.

The _read_error() routine will read one error from the stack and update the pointer to the position of the next oldest error. The stack is terminated with a zero error. Calling the _read_error() routine over and over again will eventually result in a return of this zero error. Calling the _read_error() once more, will return the newest error again.

The _print_errmsg() routine is not only used to print an error message, but also to set an error message pending, or to print an old error message that was set pending before.

An error message is a 16-bit variable containing the error code (a twelve bit number) and the error type (a 4 bit symbol). For more information on the error codes and corresponding messages, see the ASCII unit manual, section 9-1.

The following code gives an example of how error messages can be used:

```
/* EXAMPLE; Error messages */
{
  USHORT errorcode;
  UCHAR* error_message;
  ...
  if (something_is_going_wrong())
  {
    errorcode = 0xB014; /* This error-code is unused by ASCII unit's Basic */
    (void) _set_error(errorcode);
    (void) determine_message(errorcode, error_message); /*make error_message point to string*/
    (void) _print_errmsg(error_message);
  }
  ...
}
```

3-5 Functions for memory managing

In some cases it is useful to create and delete variables during run-time. These variables are called dynamic variables. To use dynamic variables, dynamic memory must be allocated where these variables can be stored. The ASCII unit's memory manager allows memory blocks to be allocated by the programmer of library functions.

Like the standard C functions for memory allocating malloc(), calloc(), realloc() and free() , the Application Programming Interface supports the functions _malloc(), _calloc(), _realloc() and _free_mem(). To allocate or free dynamic memory, always use these functions instead of the functions from the standard C library. The general way of using dynamic memory, consists of the following steps:

1. **Allocate memory.**
   Before calling _malloc() or _calloc(), determine the amount of memory to be allocated, in other words, the numbers of elements and the size of the elements that are to be placed in the dynamic memory. When calling an allocation routine, an option specifies whether the memory block should be permanent or non-permanent. It is advised not to use permanent memory blocks if not really necessary.
   If it is still desired to allocate permanent memory, make use of the empty permanent memory block of 40 bytes that is stored at &H0E000. The first byte in this block contains the unit number and the ASCII unit type. The other bytes can be used to store global data in. When the user allocates a permanent memory block, the pointer to this block should be stored in this block of global data. This should be done to save the pointer to the allocated block, even when the power is switched off and on. Note that this is not a waterproof method!
   If the application allocates memory in a loop or in a recursive routine, it is wise to test the amount of free memory with _ram_available() to prevent a memory leak.
   Routines like _get_size() and _largest_block() are low-level routines used by _malloc(), _calloc() and _realloc().
2. **Use dynamic memory.**
   When _malloc() or _calloc() is called, a pointer is returned. The pointer is a void-pointer, pointing to the address where the first element of data is to be stored. The pointer returned by the allocation routines must be type-casted to a pointer to the elements to store. The value of an element that the pointer is referring to, can be accessed by means of the dereference operator or with brackets as if it where an array of elements.
   The memory manager allows altering the chosen block type of an allocated block. It is possible to change the block type from non-permanent to permanent or vice versa with the _change_blk_type() routine.
3. **Free allocated memory block.**
   When the allocated part of memory is no longer needed, it can be de-allocated with the _free_mem() routine. When a block is freed and it is located next to an other free block, they are merged to one larger free block.
   All temporary blocks are freed when the ASCII unit is powered on, when the RUN statement is executed or when the CLEAR statement is executed in dynamic mode.

The following code shows how dynamic memory can be used:

```
/* EXAMPLE; Using dynamic memory */
{
  ITEM_TYPE *items;
  items = (ITEM_TYPE*)_calloc(100, (USHORT)sizeof(ITEM_TYPE), 0);
  if (!items) PRINT_STRING("ERROR!");
  else
  {
    ...
    /* Here, the routine uses the dynamic data... */
    ...
    _free_mem(items);
  }
}
```

3-6 Functions for controlling serial ports

This section describes how the serial ports of the ASCII unit can be configured and controlled. These routines allow the ASCII to communicate at a maximum rate of 76.800 baud in a point-to-point or a multidrop configuration. For more information, see ASCII unit operation manual section 4.

3-6-1 ASCII unit serial ports

Depending on the type of ASCII unit, two different serial connections can be used, RS232 or RS422/485. The two major differences between these ports are:

RS232:

- only for point to point configurations
- full duplex

RS422/485:

- multidrop configurations possible
- half duplex!

The advantage of the RS422/485 port is that more than one device can be controlled from one port, using only one cable. Also the length of the cable is no longer restricted to 15 meter. On the other hand, the RS422/485 port can't be used when a full duplex configuration is needed; Port 2 of the ASC21 is a half-duplex port! Normally in multidrop configurations (also called *1 to n* configuration), one master communicates with *n* slaves, in which a half-duplex link suffices.

3-6-2 Low-level routines

- Set/clr leds
- Set/clr dtr
- Set/clr rts
- Get cts
- Get dsr
- Read switches

Using _set_leds() and _clr_leds(), the leds run, basic, error, err1 and err2 on the front panel can be turned on or off. Only if the used ASCII unit model is ASC31, then the led errt can be controlled as well.
All low-level routines are used to build up the high-level routines. However, to write a user hardware communication protocol, it is possible to control the exterior lines with these routines.

3-6-3 High level routines

Most of the high level routines require the parameter *port*. For ASCII unit models ASC11 and ASC21, valid values for *port* are 1 and 2. If the ASCII unit ASC31 is used, port 3 can also be specified. The following routines are supported:

- Configure port
- Close port
- Port status
- Print to port
- Rprint to port
- Read from port
- Cls

To configure a port, a structure is used to pass the port status. This port status contains all information on how the port was configured. When _conf_port() is called, new configurations can be set, but also the old configuration is returned. This feature enables the programmer to restore the port configuration as it was in Basic, before the library function is ended. Port_status is a struct that is defined in api.h containing twelve unsigned chars:

```
typedef struct
{
  UCHAR type
  UCHAR baudrate;
  UCHAR databits;
  UCHAR stopbits;
  UCHAR parity;
  UCHAR xonxoff;
  UCHAR rts;
  UCHAR cts;
  UCHAR dtrdsr;
  UCHAR overwrite;
  UCHAR print;
  UCHAR read;
} _T_PORT_STATUS;
```

type can have the following values: 0=terminal, 1=screen, 2=keyboard, 3=communication device, 4=line printer, 5=Nec Kanji printer, 6=Epson Kanji printer.
If the *xonxoff* option is set to one, the software flowcontrol is enabled. If zero, then it is disabled. The fields of overwrite, print and read are flags, used by the operating system.
See the listing of api.h in Appendix A for more about the status struct. To get more information about configuring a serial port, see Basic's *OPEN* statement in the ASCII Operation Manual. The following code gives an example of how the serial ports can be used:

```
/* EXAMPLE; Using the serial ports */
{
  _T_PORT_STATUS new_status, old_status;
  UCHAR port_nr, lines;

  port = TERMINAL_PORT;
  if (_conf_port(port, new_status, &old_status)) /* new port status */
  {
    /* PRINT_STRING("ERROR!"); */
  }
  else
  {
    _cls(port);
    _print_to_port(port, 0, "This is a small 'Hallo World' type example...", &lines);
    (void)_conf_port(port_nr, old_status, &old_status); /* restore port status */
  }
}
```

3-7 Functions for using the realtime clock

3-7-1 General description

The ASCII unit has a realtime clock on-board, that works independently from the clock inside the PC unit. The realtime clock keeps track of the current date, the current day of the week and the current time. If the power is cut off from the ASCII unit, the data inside the realtime clock remains intact. The realtime clock allows accessing the date and time separately. For each of date, time and day_of_week, there is a routine to set the value, and a routine to get the value.

The routines that can be used to get or set values can give the following error messages:

- Data out of range error. The values are incorrect.
- Time out error. The realtime clock is not responding, probably due to a hardware defect.

3-7-2 Description of data inside realtime clock

- The date:

The date consists of the year, the month and the day. The BIOS routines that are concerned with the date, use a date struct to exchange data. The date struct is also defined in “api.h” , like this:

```
typedef struct
{
  UCHAR day;
  UCHAR month;
  UCHAR year;
} _T_DATE;
```

The variables in the struct are normal unsigned chars containing the one-to-one values for day and month. The year is stored a bit different than the other two variables. If the value is in the range 70..99 then it should be interpreted by the years 1970..1999, if the value is in the range 00..69 then it should be interpreted by the years 2000..2069.
Corresponding BIOS routines: _get_date(), _set_date().

- The time:

The time consists of the hours, the minutes and the seconds. The BIOS routines that are concerned with the time, use a time struct to exchange data. The time struct is also defined in “api.h” , like this:

```
typedef struct
{
  UCHAR seconds;
  UCHAR minutes;
  UCHAR hours;
} _T_TIME;
```

All variables in the struct are unsigned chars. Valid values for minutes and seconds are 0..59 , and valid values for hours are 0..23 .
Corresponding BIOS routines: _get_time(), _set_time().

- The day of the week

The day_of_week is represented by one unsigned char enumeratively. If zero, the day is Sunday, if one, the day is Monday etc.
Corresponding BIOS routines: _get_day_of_week(), _set_day_of_week().

4 Troubleshooting

4-1 Out of memory error message after lib load

If a library function is to be loaded at a fixed address, use the map statement to check if the concerning part of memory, to load the function in, is not in use. Note that the memory manager needs up to four bytes memory preceding the library function! In other words, if there is a free memory block starting at &H30C00, specify the address &H30C04 as start address to store the library at.

If a memory block is in use by a Basic variable, use the *clear* statement to remove this variable (only works in dynamic mode). The memory might also be in use by a part of a Basic program or another library function. These blocks can be removed with the *new* and the *lib del* statement. If none of these statements can be used to clear the concerning memory block, clear the memory as described by the procedure of section 9-2-1 in the ASCII unit operation manual.

4-2 Special unit error

Special unit errors mainly occur when a library function does not properly refresh the watch-dog timer. To prove if the library function indeed caused the special unit error, the watch-dog timer option can be specified in the *def libfn* statement making the library interface refresh the watch-dog timer in time as described in section 1-3. A special unit error can also occur when the operating system of the ASCII unit is malfunctioning. The user can corrupt the operating system from within a library function by writing data to a part of the SRAM in which the system resides, as shown by the table in section 1-2. This can happen in the following situations:

- a pointer has an incorrect value or the index of an array is out of its boundaries,
- a library function which was compiled for a free address is lib loaded at a fixed address (or vice versa),
- a library function which was compiled for a free address uses global data or contains calls to subroutines.

When a library function is called and the declaration of the corresponding Basic user function's parameter declaration does not relate properly to the function header of the Library function source, the ASCII might also crash and cause a special unit error. Note that the order of Basic parameters must be in opposite order compared to the C routine.

4-3 ASCII unit malfunctions

If the ASCII unit does not respond anymore or if it behaves unexpectedly, try using the new statement or switching the power off and on again. If the ASCII unit still isn't behaving correctly (for instance, the *map* statement is in an endless loop), follow the procedure in section 9-2-1 of the ASCII unit operation manual.

4-4 Lib Load hangs

When the *lib load* statement is entered, the ASCII waits for data and will remain doing this until the user presses CTRL-C. During *lib load*, the ASCII unit scans for data and for key-presses at the port that was specified in the *lib load* statement. For instance, if an ASC31 is used, the terminal port is port 3. If in this case a *lib load #1…* is entered, the ASCII will wait endlessly for a CTRL-C character from port 1 instead of port 3. The ASCII can be stopped waiting by either switching the power on and off, or by temporarily connecting the terminal to the other port, only for sending a CTRL-C character.

4-5 Motorola-S format error

The ASCII unit gives a Motorola-S format error after lib load, if the MTS file sent was corrupted. First make sure that the correct file is sent. If the error occurs repeatedly, re-create the MTS file.

4-6 Source debugging

Debugging a library function is more difficult than debugging a stand alone program. Syntax and semantics of parts that do not call BIOS functions can be debugged on a normal PC. However, other parts can only be tested by using an in circuit emulator or by simply running the code and see what happens by printing information to the terminal or the PLC.
It can be checked if the parameter passes from and to the library function are performed correctly by specifying the *STACK* option in the *def libfn* statement as described in section 1-3. This option will cause the ASCII unit to print the pushed and popped data on the stack when a library function is called or terminated.
When a program is functioning incorrectly, it is possible to determine whether or not this is due to a compiler bug by inspecting the assembly file. See also Appendix F.

Appendix A
Description of API.h

```
/***************************** TYPE DEFINITIONS ******************************/
typedef unsigned char UCHAR;     /* 1 byte unsigned char                     */
typedef char CHAR;               /* 1 byte signed char                       */
typedef unsigned short USHORT;   /* 2 bytes unsigned integer                 */
typedef unsigned long ULONG;     /* 4 bytes unsigned integer                 */
typedef void VOID;               /* void                                     */

/* Date structure for _get_date and _set_date                                */
typedef struct
{
  UCHAR day;
  UCHAR month;
  UCHAR year;
} _T_DATE;         /* contains day, month and year                           */

/* Time structure for _get_time and _set_time                                */
typedef struct
{
  UCHAR seconds;
  UCHAR minutes;
  UCHAR hours;
} _T_TIME;         /* contains seconds, minutes and hours                    */

/* Port structure for _conf_port                                             */
typedef struct
{
  UCHAR type;      /* 0:TERM,1:SCRN,2:KYBD,3:COMU,4:LPRT,5:NKPRT,6:EKPRT     */
  UCHAR baudrate;  /* 0:9600, 1:300,   2:600,   3:1200, 4:2400,
                      5:4800, 6:19200, 7:38400, 8:76800                      */
  UCHAR databits;  /* 5, 6, 7, 8                                             */
  UCHAR stopbits;  /* 0:1 bit, 1:1.5 bits, 2:2 bits                          */
  UCHAR parity;    /* 0: none, 1: even, 2: odd, 3: mark or 4: space          */
  UCHAR xonxoff;   /* 0: disabled, 1: enabled                                */
  UCHAR rts;       /* 0: disabled, 1: enabled                                */
  UCHAR cts;       /* 0: disabled, 1: enabled                                */
  UCHAR dtrdsr;    /* 0: disabled, 1: enabled                                */
  UCHAR overwrite; /* 0: no, 1: yes                                          */
  UCHAR print;     /* 0: no, 1: yes                                          */
  UCHAR read;      /* 0: no, 1: yes                                          */
} _T_PORT_STATUS;  /* port 1, 2, 3 structure                                 */
/*****************************************************************************/


/************************** WATCHDOG TIMER REFRESH ***************************/
#define _wdt_refresh *(UCHAR*)0xC00007L=(UCHAR)1
/*****************************************************************************/
```

```
/*************************** FUNCTION DEFINITIONS ****************************/
UCHAR _bcd2hex(USHORT, USHORT*);                                    /*Vector  98*/
UCHAR _bcd2oct(USHORT, USHORT*);                                    /*Vector  99*/
VOID* _calloc(USHORT, USHORT, UCHAR);                               /*Vector 113*/
UCHAR _change_blk_type(UCHAR*, UCHAR);                              /*Vector 132*/
UCHAR _close_port(UCHAR);                                           /*Vector 151*/
UCHAR _clr_dtr(UCHAR);                                              /*Vector 145*/
UCHAR _clr_ir_bit(UCHAR, UCHAR);                                    /*Vector  87*/
VOID  _clr_leds(UCHAR, UCHAR, UCHAR, UCHAR, UCHAR, UCHAR);          /*Vector 136*/
UCHAR _clr_rts(UCHAR);                                              /*Vector 148*/
UCHAR _cls(UCHAR);                                                  /*Vector 152*/
UCHAR _conf_port(UCHAR, _T_PORT_STATUS, _T_PORT_STATUS*);           /*Vector 150*/
VOID  _del_errors(VOID);                                            /*Vector 102*/
UCHAR _free_mem(VOID*);                                             /*Vector 116*/
UCHAR _get_cts(UCHAR, UCHAR*);                                      /*Vector 149*/
UCHAR _get_date(_T_DATE*);                                          /*Vector 160*/
UCHAR _get_day_of_week(UCHAR*);                                     /*Vector 162*/
UCHAR _get_dsr(UCHAR, UCHAR*);                                      /*Vector 146*/
ULONG _get_size(VOID*);                                             /*Vector 128*/
UCHAR _get_time(_T_TIME*);                                          /*Vector 158*/
UCHAR _hex2bcd(USHORT, USHORT*);                                    /*Vector  97*/
ULONG _largest_block(VOID);                                         /*Vector 117*/
VOID* _malloc(USHORT, USHORT, UCHAR);                               /*Vector 112*/
UCHAR _mask_basic_int(UCHAR);                                       /*Vector  74*/
UCHAR _mots_to_str(UCHAR*, USHORT, ULONG*, USHORT*);                /*Vector 142*/
UCHAR _oct2bcd(USHORT, USHORT*);                                    /*Vector 100*/
UCHAR _pc_eget(UCHAR, UCHAR, UCHAR*);                               /*Vector  91*/
UCHAR _pc_eput(UCHAR, UCHAR, UCHAR*);                               /*Vector  92*/
UCHAR _pc_qread(UCHAR, UCHAR, USHORT, USHORT*);                     /*Vector  93*/
UCHAR _pc_qwrite(UCHAR,UCHAR, USHORT, USHORT*);                     /*Vector  94*/
UCHAR _pc_read(UCHAR, UCHAR, UCHAR*, UCHAR*, USHORT*, UCHAR*);      /*Vector  95*/
UCHAR _pc_write(UCHAR, UCHAR, UCHAR*, UCHAR*, USHORT*, UCHAR*);     /*Vector  96*/
UCHAR _port_status(UCHAR, UCHAR, USHORT*);                          /*Vector 154*/
UCHAR _print_errmsg(UCHAR*);                                        /*Vector 105*/
UCHAR _print_to_port(UCHAR, UCHAR, UCHAR*, UCHAR*);                 /*Vector 156*/
ULONG _ram_available(VOID);                                         /*Vector 118*/
UCHAR _read_error(USHORT*);                                         /*Vector 104*/
UCHAR _read_from_port(UCHAR, UCHAR, UCHAR, USHORT, UCHAR*);         /*Vector 157*/
UCHAR _read_ir_word(UCHAR, USHORT*);                                /*Vector  85*/
UCHAR _read_switches(UCHAR, UCHAR*);                                /*Vector 141*/
VOID* _realloc(VOID*, ULONG, UCHAR);                                /*Vector 115*/
UCHAR _register_int(USHORT, ULONG);                                 /*Vector  72*/
UCHAR _rprint_to_port(UCHAR, UCHAR, UCHAR*, UCHAR*, UCHAR);         /*Vector 155*/
UCHAR _set_date(_T_DATE);                                           /*Vector 161*/
UCHAR _set_day_of_week(UCHAR);                                      /*Vector 163*/
UCHAR _set_dtr(UCHAR);                                              /*Vector 138*/
VOID  _set_error(USHORT);                                           /*Vector 101*/
UCHAR _set_ir_bit(UCHAR, UCHAR);                                    /*Vector  86*/
VOID  _set_leds(UCHAR, UCHAR, UCHAR, UCHAR, UCHAR, UCHAR);          /*Vector 135*/
UCHAR _set_rts(UCHAR);                                              /*Vector 137*/
UCHAR _set_time(_T_TIME);                                           /*Vector 159*/
VOID  _sjis2jis(UCHAR, UCHAR, UCHAR*, UCHAR*);                      /*Vector 144*/
UCHAR _str_to_mots(ULONG, UCHAR*, UCHAR);                           /*Vector 143*/
VOID  _unmask_basic_int(UCHAR, UCHAR);                              /*Vector  75*/
VOID  _update_errors(VOID);                                         /*Vector 103*/
VOID  _update_irin(VOID);                                           /*Vector  89*/
VOID  _update_irout(VOID);                                          /*Vector  90*/
UCHAR _write_ir_word(UCHAR, USHORT);                                /*Vector  88*/
/*****************************************************************************/
```

Appendix B
Reference of BIOS routines

Remarks:

- Some routines require the option *port*. For ASCII unit models ASC11 and ASC21, valid values for *port* are 1 and 2. Only if the ASCII unit ASC31 is used, port 3 can be specified as well.
- In this reference, some routines are illustrated with an example. If a routine is not illustrated with an example, there might by a reference to one of the example applications in Appendix C.
- The routines in this reference are listed in alphabetical order. Cross-references in 'see also' are not. The first routine mentioned in the 'see also' is the complementary routine (if applicable), and than the other routines involved are mentioned.
- Every routine is defined in api.h in Appendix A and is described in one of the subsections 3-2 .. 3-7.

_bcd2hex

Function

Converts a 16 bit binary coded decimal integer to a 16 bit hexadecimal coded integer.

Syntax

#include "api.h"
UCHAR _bcd2hex(USHORT *bcd_val*, USHORT **hex_val*);

Parameters

Bcd_val is the input value, *hex_val* is the converted output value.

Return Value

0: OK, 1: Error, invalid bcd value.

Remarks

The value of each digit(4 bits) in the input parameter must be 9 or less. The value of the output parameter is in the range from 0 to 0x270F .

See also

_hex2bcd, _bcd2oct, _oct2bcd

Example

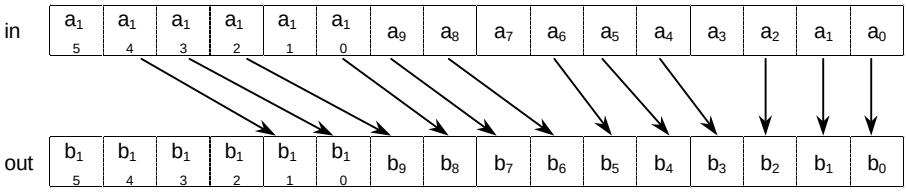
Results of calling (void) _bcd2hex(*bcd_val*, &*hex_val*);

input *bcd_val*	0x0009	0x0010	0x4096	0x789A
output *hex_val*	0x0009	0x000A	0x1000	- *ERROR* -

_bcd2oct

Function

Converts a 16 bit binary coded decimal integer to a 16 bit octal coded integer;



Syntax

#include “api.h”
UCHAR _bcd2oct(USHORT *bcd_val*, USHORT **oct_val*);

Parameters

Bcd_val is the input value, *oct_val* is the converted output value.

Return Value

0: OK, 1: Invalid bcd value.

Remarks

The value of each digit in the input parameter must be 7 or smaller (bits 3, 7, 11 and 15 must be zero).
The value of the output parameter is in the range from 0 to 0x0FFF .

See also

_oct2bcd, _bcd2hex, _hex2bcd

Example

Results of calling (void) _bcd2oct (*bcd_val*, &*oct_val*);

input *bcd_val*	0x0007	0x0010	0x7777	0x4567
output *oct_val*	0x0007	0x0008	0x0FFF	- *ERROR* -

_calloc

Function

- Allocates memory, that is to be assigned by the caller, to be used for dynamic variables.
- Clears allocated memory.

Syntax

#Include “api.h”
VOID* _calloc(USHORT *nitems*, USHORT *size*, UCHAR *blktype*);

Parameters

Nitems is the number of items, and *size* is the size of one item.
Blktype = 0: non-permanent type, *blktype* = 1: permanent type.

Return Value

0: Error, invalid input values or out of memory error.
non-zero: Address pointer to first data element in allocated memory block.

Remarks

Size × *nitems* must at least be smaller than 131068 bytes.
For more information about dynamic variables, see section 3-5.

See also

_free_mem, _malloc, _realloc

Example

-

_change_blk_type

Function

Changes block type.

Syntax

#Include “api.h”
UCHAR _change_blk_type(UCHAR **blkptr*, UCHAR *blktype*);

Parameters

Blkptr is a pointer to the block of which the type is to be changed.
Blktype = 0: non-permanent type, *blktype* = 1: permanent type.

Return Value

0: OK, 1: Error, invalid *blktype* or *blkptr*.

Remarks

This low-level function is normally called by high-level memory managing functions. Therefore this function should only be called by programmers with sufficient knowledge of the ASCII unit’s internals.
For more information about memory blocks, see section 1-2.

See also

-

Example

-

_close_port

Function	Stops transmitting and receiving data on a port, Prevents other routines from controlling the parameters of the port.
Syntax	#Include “api.h” UCHAR _close_port(UCHAR *port*);
Parameters	*Port* is the desired port number. If zero, all ports are closed.
Return Value	0: OK, 1: Error, invalid port number.
Remarks	-
See also	*_conf_port*
Example	-

_clr_dtr

Function	Clears (negates) the DTR output signal on a port. If the port is controlled automatically, an error is returned.
Syntax	#Include “api.h” UCHAR _clr_dtr(UCHAR *port*);
Parameters	*Port* is the desired port number.
Return Value	0: OK, 1: Error, invalid port number.
Remarks	-
See also	*_set_dtr, _clr_rts*
Example	-

_clr_ir_bit

Function	Clears (negates) a bit in one of the words in the local copy of the IR INPUT area.
Syntax	#Include “api.h” UCHAR _clr_ir_bit(UCHAR *wordnum*, UCHAR *bitnum*);
Parameters	*Wordnum* indicates the word of the IR INPUT area, valid values are 0 to 4. *Bitnum* indicates the bit to be cleared in the selected word.
Return Value	0: OK, 1: Error, invalid word requested, 2: Error, invalid bit requested.
Remarks	-
See also	*_set_ir_bit, _write_ir_word, _update_irin*
Example	-

_clr_leds

Function	Turns specified indicator leds off.
Syntax	#Include “api.h” VOID _clr_leds(UCHAR *led1*, UCHAR *led2*, …. , UCHAR *led6*);
Parameters	*Led1*, *led2*..*led6* correspond to indicator led’s: run, basic, error, err1, err2, errt. If the value is one, the corresponding led is turned off.
Return Value	-
Remarks	The errt led can only be controlled on an ASC31.
See also	*_set_leds*
Example	*See example 2 in Appendix C.*

_clr_rts

Function	Clears (negates) the RTS output signal on a port. If the port is controlled automatically, an error is returned.
Syntax	#Include “api.h” UCHAR _clr_rts(UCHAR *port*);
Parameters	*Port* is the desired port number.
Return Value	0: OK, 1: Error, invalid port number.
Remarks	If the ASCII unit is an ASC21, referring to port 2 is possible, although the port has no RTS output line. In this case it controls (de-)activation of receiver/transmitter mode of the RS422/485 port. When the RTS signal is cleared, the receiver is active.
See also	*_set_rts, _clr_dtr*
Example	-

_cls

Function	Clears the screen of a device, if open.
Syntax	#Include “api.h” UCHAR _cls(UCHAR *port*);
Parameters	*Port* is the desired port number.
Return Value	0: OK, 1: Error, invalid port number, 7: Error, port is not an output device.
Remarks	-
See also	*_conf_port*
Example	-

_conf_port

Function

- Initializes a port by means of a status struct,
- Clears the corresponding in- and output buffers.

Syntax

#Include “api.h”
UCHAR _conf_port(UCHAR *port*, _T_PORT_STATUS *new_p_status*,
_T_PORT_STATUS **old_p_status*);

Parameters

Port is the desired port number, *new_p_status* is the struct to configure the port with. *Old_p_status* contains the old port configuration.

Return Value

0: OK, 1: Error, invalid port number, 2: Error, port not opened, 3: Error, invalid configuration settings, 4: Error, local loop-back error.

Remarks

For more information about the port-status datatype, see section 3-6.

See also

_close_port

Example

-

_del_errors

Function

- Deletes all errors from the error stack
- Clears the error led’s ERROR, ERR1, ERR2 & ERRT on ASCII front.

Syntax

#Include “api.h”
VOID _del_errors();

Parameters

-

Return Value

-

Remarks

This routine also clears IR n+5, bits 2, 3, 4, 5 and 8 , and IR n+7.
For more about error handling, see section 3-4.

See also

_set_error, _read_error, _print_errmsg

Example

-

_free_mem

Function	Frees up memory that was assigned to variable(s).
Syntax	#Include “api.h” UCHAR _free_mem(VOID **blkptr*);
Parameters	*Blkptr* is a pointer the memory block that is to be freed.
Return Value	0: OK, 1: Error, address null or out of range, 2: Error, block already free, 3: Error, block does not exist, or contents of RAM has corrupted.
Remarks	*For more information about memory blocks, see section 1-2.*
See also	*_calloc, _malloc*
Example	-

_get_cts

Function	Reads the CTS input signal.
Syntax	#Include “api.h” UCHAR _get_cts(UCHAR *port*, UCHAR **val*);
Parameters	*Port* is the desired port number. *Val* is the output; 0: negated (clear) , 1: asserted (set)
Return Value	0: OK, 1: Error, invalid port number, 6: Error, access violation.
Remarks	It is not allowed to use _get_cts from port 2 on an ASC21 unit. This will result in an access violation error.
See also	*_get_dsr*
Example	-

_get_date

Function	Retrieves the current date from realtime clock.
Syntax	#Include “api.h” UCHAR _get_date(_T_DATE **dateptr*);
Parameters	*Dateptr* is the output, containing the date struct.
Return Value	0: OK, 1: Error, RTC reading out of time, 3: Error, read data out of range.
Remarks	*See section 3.7 for description of date struct _T_DATE.*
See also	*_set_date, _get_day_of_week, _get_time*
Example	*See example 2 in Appendix C.*

_get_day_of_week

Function	Retrieves the current day from realtime clock.
Syntax	#Include “api.h” UCHAR _get_day_of_week(UCHAR **day_of_week*);
Parameters	The output value of *day_of_week* represents the day enumeratively: 0: Sunday, 1: Monday, 2: Tuesday, … 6: Saturday
Return Value	0: OK, 1: Error, RTC reading out of time, 3: Error, read data out of range.
Remarks	-
See also	*_set_day_of_week, _get_date, _get_time*
Example	*See example 2 in Appendix C*

_get_dsr

Function	Reads the DSR input signal
Syntax	#Include “api.h” UCHAR _get_dsr(UCHAR *port*, UCHAR **val*);
Parameters	*Port* is the desired port number, *Val* is the output; 0: negated (clear), 1: asserted (set)
Return Value	0: OK, 1: Error, invalid port number, 6: Error, access violation.
Remarks	It is not allowed to use _get_dsr from port 2 on an ASC21 unit. This will result in an access violation error.
See also	*_get_cts*
Example	-

_get_size

Function	Returns the size of the block indicated by the caller.
Syntax	#Include “api.h” ULONG _get_size(VOID **blkptr*);
Parameters	*Blkptr* is a pointer to the first data element of that block, of which the size is to be returned.
Return Value	0: Error, pointer out of range , non-zero: Size of memory block.
Remarks	Normally, pointers to the first data element in the memory block are returned by functions like calloc and malloc. *For more information about memory blocks, see section 1-2.*
See also	*_ram_available*
Example	-

_get_time

Function

Retrieves the current time from realtime clock.

Syntax

#Include “api.h”
UCHAR _get_time(_T_TIME **timeptr*);

Parameters

Timeptr is the output, containing the time struct.

Return Value

0: OK, 1: Error, RTC reading out of time,
3: Error, read data out of range.

Remarks

See section 3-7 for description of date struct _T_TIME.

See also

_set_time, _get_date, _get_day_of_week

Example

See example 2 in Appendix C

_hex2bcd

Function

Converts a 16 bit hexadecimal coded integer to a 16 bit binary coded decimal integer.

Syntax

#include “api.h”
UCHAR _hex2bcd(USHORT *hex_val*, USHORT **bcd_val*);

Parameters

Hex_val is the input value, *bcd_val* is the converted output value.

Return Value

0: OK, 1: Hex value to large.

Remarks

The value of the input parameter can not be greater than 0x270F .

See also

_bcd2hex, _bcd2oct, _oct2bcd

Example

Results of calling (void) _hex2bcd(*hex_val*, &*bcd_val*);

input *hex_val*	0x0009	0x000A	0x270F	0x2710
output *bcd_val*	0x0009	0x0010	0x9999	- *ERROR* -

See also example 2 in Appendix C

_largest_block

Function	Returns the size of the largest free block
Syntax	#include “api.h” ULONG _largest_block();
Parameters	-
Return Value	0: No free block available, non-zero: Size of the largest block, maximum value is 204796
Remarks	*For more information about memory blocks, see section 1-2.*
See also	*_ram_available*
Example	-

_malloc

Function	Allocates memory, that is to be assigned by the caller, to be used for dynamic variables.
Syntax	#Include “api.h” VOID* _malloc(USHORT *nitems*, USHORT *size*, UCHAR *blktype*);
Parameters	*Nitems* is the number of items, and *size* is the size of one item. *Blktype*: 0: non-permanent type, 1: permanent type.
Return Value	0: Error, invalid input values or out of memory error. non-zero: Address pointer to first data element in allocated memoryblock.
Remarks	*size* × *nitems* must at least be smaller than 131068 bytes. *For more information about dynamic variables, see section 3-5.*
See also	*_free_mem, _calloc, _realloc,*
Example	-

_mask_basic_int

Function

Masks one or all Basic interrupts.

Syntax

#Include “api.h”
UCHAR _mask_basic_int(UCHAR *type*);

Parameters

Type is the number of the interrupt type to mask (max. 116). If zero, all are masked.

Return Value

0: Previously not masked, 1: Previously already masked.

Remarks

Always use _mask_basic_int with _unmask_basic_int. Use the return value of the mask function as parameter for the unmask function.

See also

_unmask_basic_int

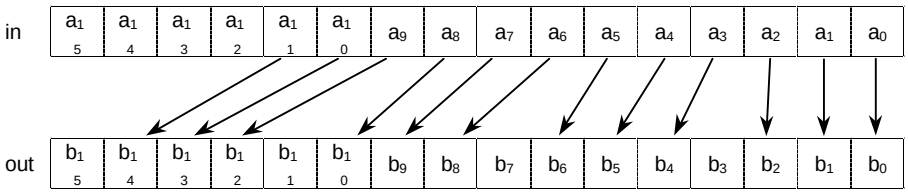
Example

-

_oct2bcd

Function

Converts a 16 bit octal coded integer to a 16 bit binary coded decimal integer.



Syntax

#include “api.h”
UCHAR _oct2bcd(USHORT *oct_val*, USHORT **bcd_val*);

Parameters

Oct_val is the input value, *bcd_val* is the converted output value.

Return Value

0: OK, 1: Error, invalid octal value.

Remarks

The value of the input parameter must be in the range from 0 to 0x0FFF .

See also

_bcd2oct, _bcd2hex, _hex2bcd

Example

Results of calling (void) _oct2bcd (*oct_val*, &*bcd_val*);

input *oct_val*	0x0007	0x0008	0x0FFF	0x1000
output *bcd_val*	0x0007	0x0010	0x7777	- *ERROR* -

_pc_eget

Function	Reads data from the IOWR buffer (shared memory).
Syntax	#include “api.h” UCHAR _pc_eget(UCHAR *num_bytes*, UCHAR *offset*, UCHAR **data*);
Parameters	*Num_bytes* is the number of bytes to read. *Offset* is added to the base-address of the IOWR buffer to determine the address of the indata buffer. *Data* is the array of data as requested (max. 180 bytes).
Return Value	0: OK, 1: Error, indata buffer is not available, 2: Error, IOWR #00xx is currently active, 3: Error, Size of data in buffer is smaller than requested.
Remarks	Data transfer-method 1. Before calling _pc_eget the share memory must be set up correctly by DM m+6 and m+7. *For more details, see ASCII unit Operation Manual, section 6*.
See also	*_pc_eput, _pc_read, _pc_qread*
Example	-

_pc_eput

Function	Writes data (characters) to the IORD buffer (shared memory).
Syntax	#include “api.h” UCHAR _pc_eput(UCHAR *num_bytes*, UCHAR *offset*, UCHAR **data*);
Parameters	*Num_bytes* is the number of bytes to write. *Offset* is added to the base-address of the IORD buffer to determine the address of the outdata buffer. *Data* is an array containing the data to be sent (max. 180 bytes).
Return Value	0: OK, 1: Error, outdata buffer is not available, 2: Error, IORD #00xx or pc_eput is currently active, 3: Error, Size of outdata buffer is to small.
Remarks	Data transfer-method 1. Before calling _pc_eput the share memory must be set up correctly by DM m+6 and m+7. *For more details, see ASCII unit Operation Manual, section 6*.
See also	*_pc_eget, _pc_write, _pc_qwrite*
Example	-

_pc_qread

Function	Reads data (words) from PLC memory.
Syntax	#include “api.h” UCHAR _pc_qread(UCHAR *num_words*, UCHAR *area*, USHORT *address*, USHORT **data*);
Parameters	*Num_words* is the number of words to read, valid values are 1..127. *Area* is the PLC area to read from: 0: DM, 1: IR, 2: LR, 3: HR, 4: AR, 5: EM, 6: TC. *Address* is the PLC memory address to read data from, *Data* is the array where the read data will be stored (max. 128 elements).
Return Value	0: OK, 1: Error, not connected to PLC, 2: Error, invalid PLC area, 3: Error, invalid address, 4: Error, invalid number of words, 5: Another PLC transfer is currently active, 6: Stopped by wait-interrupt, 7: Stopped by break-interrupt.
Remarks	Data transfer-method 4. The ASCII unit forces the PLC to write memory-data to the ASCII unit by means of an IOWR (#FD00) instruction. Before calling pc_qwrite, properly set the indices in IR n+8 and n+9. *For more details on transfer methods, see ASCII Operation Manual, section 6, for more detail on memory areas, see C200H / C200Hx-CPUxx-ZE Operation Manual, section 3*.
See also	*_pc_qwrite, _pc_eget,, _pc_read*
Example	-

_pc_qwrite

Function	Writes data (words) to PLC memory.
Syntax	#include “api.h” UCHAR _pc_qwrite(UCHAR *num_words*, UCHAR *area*, USHORT *address*, USHORT **data*);
Parameters	*Num_words* is the number of words to write, valid values are 1..127. *Area* is the PLC area to write in: 0: DM, 1: IR, 2: LR, 3: HR, 4: AR, 5: EM, 6: TC. *Address* is the PLC memory address to send data to, *Data* is an array containing the data to be sent (max. 128 elements).
Return Value	0: OK, 1: Error, not connected to PLC, 2: Error, invalid PLC area 3: Error, invalid address, 4: Error, invalid number of words, 5: Another PLC transfer is currently active, 6: Stopped by wait-interrupt, 7: Stopped by break-interrupt.
Remarks	Data transfer-method 4. The ASCII unit forces the PLC to read data from the ASCII unit into memory by means of an IORD (#FD00) instruction. Before calling pc_qwrite, properly set the indices in IR n+8 and n+9. *For more details on transfer methods, see ASCII Operation Manual, section 6, for more detail on memory areas, see C200H / C200Hx-CPUxx-ZE Operation Manual, section 3*.
See also	*_pc_qread, _pc_eput, _pc_write*
Example	-

_pc_read

Function

Reads data (words) from PLC memory, during I/O-refresh cycle.

Syntax

#include “api.h”
UCHAR _pc_read(UCHAR *at*, UCHAR *num_req*, UCHAR *num_words*,
UCHAR **area*, USHORT **address*, UCHAR **data*);

Parameters

At determines whether to use the pc_read (*at*=0) or pc_read@ (*at*=1) routine, *Num_req* is number of request, valid values are 1..5, *Num_words* is an array of 5 unsigned chars, for each request one, containing the number of words to transfer, valid values are 1..255. *Area* is an array of 5 unsigned chars, for each request one, containing the PLC area(s) to read from. Valid values are:
0: DM, 1: IR, 2: LR, 3: HR, 4: AR, 5: EM, 6: TC.
Address is an array of 5 unsigned integers, for each request one, containing the PLC memory address(es) to read data from,
Data is the array where the read data will be stored.

Return Value

0: OK, 1: Stopped by wait-interrupt, 2: Error, invalid PLC area
3: Error, invalid address, 4: Error, invalid number of words,
5: Another PLC transfer is currently active,
6: Error, invalid number of requests, 7: Stopped by break-interrupt.

Remarks

If *at*=0 ,data transfer-method 2, data is transferred on a trigger from the PCU unit. If *at*=1 ,data transfer-method 3, the ASCII unit independently reads data by interrupting the PC unit.
For more details on transfer methods, see ASCII Operation Manual, section 6, for more detail on memory areas, see C200H / C200Hx-CPUxx-ZE Operation Manual, section 3.

See also

_pc_write, _pc_eget, _pc_qread

Example

-

_pc_write

Function

Writes data (words) to PLC memory, during I/O-refresh cycle.

Syntax

#include “api.h”
UCHAR _pc_write(UCHAR *at*, UCHAR *num_req*, UCHAR *num_words*,
UCHAR **area*, USHORT **address*, UCHAR **data*);

Parameters

At determines whether to use the pc_write (*at*=0) or pc_write@ (*at*=1) routine, *Num_req* is number of request, valid values are 1..5, *Num_words* is an array of 5 unsigned chars, for each request one, containing the number of words to transfer, valid values are 1..255. *Area* is an array of 5 unsigned chars, for each request one, containing the PLC area(s) to write in. Valid values are:
0: DM, 1: IR, 2: LR, 3: HR, 4: AR, 5: EM, 6: TC.
Address is an array of 5 unsigned integers, for each request one, containing the PLC memory address(es) to write data to,
Data is the array containing the data to be sent.

Return Value

0: OK, 1: Stopped by wait-interrupt, 2: Error, invalid PLC area
3: Error, invalid address, 4: Error, invalid number of words,
5: Another PLC transfer is currently active,
6: Invalid number of requests, 7: Stopped by break-interrupt.

Remarks

If *at*=0 ,data transfer-method 2, data is transferred on a trigger from the PCU unit. If *at*=1 ,data transfer-method 3, the ASCII unit independently writes data by interrupting the PC unit.
For more details on transfer methods, see ASCII Operation Manual, section 6, for more detail on memory areas, see C200H / C200Hx-CPUxx-ZE Operation Manual, section 3.

See also

_pc_read, _pc_eput, _pc_qwrite

Example

See example 2 in Appendix C

_port_status

Function

Retrieves information on the status of a port. The caller must specify what information is to be retrieved.

Syntax

#include “api.h”
UCHAR _port_status(UCHAR *port*, UCHAR *info*, USHORT **result*);

Parameters

Port is the desired port number,
Info specifies what information about the port is to be retrieved:
1: device type, 2: number of bytes in input buffer, 3: number of bytes in output buffer, 4: RTS enabled, 5: DTR enabled.
Result contains the value of the requested data.

Return Value

0: OK, 1: Error, invalid port specified, 8: Error, invalid option requested.

Remarks

For more about the status struct, see section 3-6.

See also

_conf_port, _clr_dtr, _clr_rts, _set_dtr, _set_rts

Example

-

_print_errmsg

Function

Prints an error message -if a terminal is connected- else set message pending to be printed later. If the string is empty, the routine will print a pending message (if one is available).

Syntax

#Include “api.h”
UCHAR _print_errmsg(UCHAR **message*);

Parameters

Message is a string containing the error message.

Return Value

0: OK, 1: No terminal connected, message is set pending.
2: Error, no terminal connected and string is empty.

Remarks

-

See also

_read_error

Example

-

_print_to_port

Function	Prints a string to port.
Syntax	#include “api.h” UCHAR _print_to_port(UCHAR *port*, UCHAR *option*, UCHAR **string*, UCHAR **lines*);
Parameters	*Port* is the desired port number, possible values for *option*: 0: Print without any trailer, 1: Print with a CR trailer, 2: Print with CR-LF trailer, 3: Print cursor movement and ctrl sequence. *String* is a pointer to the string that is to be printed, *Lines* is output, value is the number of lines occupied by the string, printed on the terminal.
Return Value	0: OK, 1: Error, invalid port number, 7: Error, invalid output device, 8: Error, invalid option, 9: Port is busy.
Remarks	The string must be terminated by a NULL-character.
See also	*_read_from_port, _rprint_to_port, _conf_port*
Example	*See example 3 in Appendix C*

_ram_available

Function	Returns the total free memory in RAM.
Syntax	#include “api.h” ULONG _ram_available();
Parameters	-
Return Value	Value is in the range from 0 to 204796 .
Remarks	-
See also	*_free_mem, _get_size*
Example	-

_read_error

Function	Retrieves the next error from the ASCII units error stack, Updates corresponding bits in IR n+5 and IR n+7 .
Syntax	#Include “api.h” UCHAR _read_error(USHORT **errorcode*);
Parameters	*Errorcode* is output, containing the error code and type.
Return Value	0: OK, 1: Error stack is empty.
Remarks	The error stack is a cyclic buffer containing the errors, stacked in order of age, from oldest to newest. Therefore the oldest error is retrieved first, and the newest error is retrieved last. However if all errors have been read, the next call to read_error will return a zero. Calling read_error again, will result in starting at the oldest error again. *For more about error handling, see section 0*.
See also	*_set_error, _del_errors, _print_errmsg*
Example	-

_read_from_port

Function

Reads data from the input buffer of the specified port.

Syntax

#include “api.h”
UCHAR _read_from_port(UCHAR *port*, UCHAR *option*,
UCHAR *led_rate*, USHORT *number*, UCHAR **string*);

Parameters

Port is the desired port number.
Option must be a number from 0 to 5, or 17:

	Echo string	*Echo CR*	*End with CR*	*Special FX*
0			*Y*	*-*
1	*Y*	*Y*	*Y*	*-*
2			*Y*	*Each character is echoed as an asterisk (for passwords)*
3	*Y*		*Y*	*-*
4				*Reads fixed number of characters*
5				*Reads only one character*
17	*Y*	*Y*	*Y*	*Up/Down arrows treated as CR*

The string read out is normally terminated by a ‘\0’ character, except when option four or five are used.
When option five is chosen, the value of one byte is returned in *string*, if there is a byte in the buffer. If there is none, a zero is returned in *string*.
Led_rate is the blinking speed of the Basic indicator led; 0: slow, 1: fast.
Number specifies the number of characters to be read (max. 511) .
String is a pointer to the array where the read data will be stored.

Return Value

0: OK, 1: Error, invalid port number, 8: Error, invalid option, 9: Port is busy, 11: Error, input device, 18: Input buffer was empty,
22: Stopped by wait interrupt, 23: Stopped by break interrupt.

Remarks

Type ahead is allowed. The input string can only be echoed if the device was configured as TERM (terminal).

See also

_print_to_port, _rprint_to_port

Example

-

_read_ir_word

Function

Retrieves a word from the local copy of the IR OUPUT area.

Syntax

#include “api.h”
UCHAR _read_ir_word(UCHAR *wordnum*, USHORT **val*);

Parameters

Wordnum indicates the word of the IR INPUT area, valid values are 0 to 4. *Val* contains the value of the word selected.

Return Value

0: OK, 1: Error, invalid word requested.

Remarks

-

See also

_write_ir_word, _update_irout

Example

-

_read_switches

Function

Reads the position of the requested switch.

Syntax

#include “api.h”
UCHAR _read_switches(UCHAR *switch*, UCHAR **pos*);

Parameters

Switch is the number of which position is to be the switch to be read, valid values are 1: rotary switch, 2: default switch, 3: testmode jumper, 4: model detection, 5: start/stop switch.
Pos contains the position of the requested switch:

Requested switch	Possible values
rotary switch	*0 – 15*
default switch	*0 (no) , 1 (yes)*
testmode jumper	*0 (off) , 1 (on)*
model detection	*0x?1 : ASC31 0x?2 : ASC21 0x?3 : ASC11*
start/stop switch	*0 (stop) , 1 (start)*

Return Value

0: OK, 8: Error, invalid switch type.

Remarks

In newer ASCII models the default switch and the testmode jumper are internally connected. In such a case, the return values when requesting *switch* is 2 or 3 will always be the same.

See also

-

Example

See example 3 in Appendix C

_realloc

Function	Reallocates a memory block, and copies the contents of the block at the old location to the new location.
Syntax	#include “api.h” VOID* _realloc(VOID **blkptr*, ULONG **size*, UCHAR *type*);
Parameters	*Blkptr* is the pointer to the block to be reallocated, *size* should be smaller than 131068 . *Type* is blocktype, 0: non-permanent type, 1: permanent type.
Return Value	0: Error, not enough space, or block was freed, Non-zero: pointer to the new start-address of the replaced block.
Remarks	If the new block is smaller than the old one, the tail part is discarded, If the new block is larger than the old one, the tail part is initialized to zero.
See also	*_calloc, _malloc, _change_blk_type*
Example	-

_register_int

Function	Registers an interrupt service routine in the exception vector table.
Syntax	#include “api.h” UCHAR _register_int(USHORT *vector*, ULONG *addr*);
Parameters	*Vector* is the vector number from the vector table. *Addr* is the address of the interrupt service routine.
Return Value	0: OK, 1: Error, invalid vector, 2: Error, invalid address.
Remarks	Precondition: The involved vector should be unreserved.
See also	*_mask_basic_int, _unmask_basic_int*
Example	-

_rprint_to_port

Function	'Rawprints' a string to port.
Syntax	#include "api.h" UCHAR _rprint_to_port(UCHAR *port*, UCHAR *option*, UCHAR **string*, UCHAR **lines*, UCHAR *strlen*);
Parameters	*Port* is the desired port number, possible values for *option*: 0: Print without any trailer, 1: Print with a CR trailer, 2: Print with CR-LF trailer, 3: Print cursor movement and ctrl sequence. *String* is a pointer to the string that is to be printed, *Lines* is output, value is the number of lines occupied by the string printed on the terminal; Is related to terminal screen width. *Strlen* must be in the range from 1 to 255 .
Return Value	0: OK, 1: Error, invalid port number, 7: Error, invalid output device, 8: Error, invalid option, 9: Port is busy.
Remarks	This function is a low-level version of *_print_to_port*, allowing the string to contain NULL-characters as well.
See also	*_read_from_port, _print_to_port*
Example	-

_set_date

Function	Changes the date of the realtime clock.
Syntax	#Include "api.h" UCHAR _set_date(_T_DATE *date*);
Parameters	*Date* is the input, containing the date struct.
Return Value	0: OK, 1: Error, RTC writing out of time, 3: Error, data out of range.
Remarks	*See section 3-7 for description of date struct _T_DATE.*
See also	*_get_date, _set_day_of_week, _set_time*
Example	-

_set_day_of_week

Function	Changes the day of the realtime clock.
Syntax	#Include “api.h” UCHAR _set_day_of_week(UCHAR **day_of_week*);
Parameters	The input value *day_of_week* represents the day enumeratively: 0: Sunday, 1: Monday, 2: Tuesday, … 6: Saturday
Return Value	0: OK, 1: Error, RTC writing out of time, 3: Error, data out of range.
Remarks	-
See also	*_set_day_of_week, _get_date, _get_time*
Example	-

_set_dtr

Function	Sets (asserts) the DTR output signal on a port. If the port is controlled automatically, an error is returned.
Syntax	#Include “api.h” UCHAR _set_dtr(UCHAR *port*);
Parameters	*Port* is the desired port number.
Return Value	0: OK, 1: Error, invalid port number.
Remarks	-
See also	*_get_dtr, _set_rts*
Example	-

_set_error

Function	Adds an error to the error stack, Sets the proper indicator led's on the ASCII panel, Sets the corresponding values to IR n+5 and n+7.
Syntax	#Include "api.h" VOID _set_error(USHORT *errorcode*);
Parameters	*Errorcode* contains the errorcode and type of the error to be added to the list of errors.
Return Value	-
Remarks	The error stack can only contain 30 errors. When setting more errors, the oldest will be lost. *For more about error handling, see section 3-4.*
See also	*_del_errors, _read_error, _print_errmsg*
Example	-

_set_ir_bit

Function	Sets (asserts) a bit in one of the words in IR INPUT area.
Syntax	#Include "api.h" UCHAR _set_ir_bit(UCHAR *wordnum*, UCHAR *bitnum*);
Parameters	*Wordnum* indicates the word of the IR INPUT area, valid values are 0 to 4. *Bitnum* indicates the bit to be set in the selected word.
Return Value	0: OK, 1: Error, invalid word requested, 2: Error, invalid bit requested.
Remarks	-
See also	*_clr_ir_bit, _write_ir_word, _update_irin*
Example	-

_set_leds

Function	Turns specified indicator led's on.
Syntax	#Include "api.h" VOID _set_leds(UCHAR *led1*, UCHAR *led2*, …. , UCHAR *led6*);
Parameters	*Led1*, *led2*..*led6* correspond to indicator led's: run, basic, error, err1, err2, errt. If the value is one, the corresponding led is turned on.
Return Value	-
Remarks	The errt led can only be controlled on an ASC31.
See also	*_clr_leds*
Example	*See example 2 in Appendix C*

_set_rts

Function	Sets (asserts) the RTS output signal on a port. If the port is controlled automatically, an error is returned.
Syntax	#Include "api.h" UCHAR _set_rts(UCHAR *port*);
Parameters	*Port* is the desired port number.
Return Value	0: OK, 1: Error, invalid port number.
Remarks	If the ASCII unit is an ASC21, referring to port 2 is possible, although the port has no RTS output line. In this case it controls (de-)activation of receiver/transmitter mode of the RS422/485 port. When the RTS signal is set, the transmitter is active.
See also	*_clr_rts, _set_dtr*
Example	-

_set_time

Function	Changes the time of the realtime clock.
Syntax	#Include “api.h” UCHAR _set_time(_T_TIME *time*);
Parameters	*Time* is the input, containing the time struct.
Return Value	0: OK, 1: Error, RTC writing out of time, 3: Error, data out of range.
Remarks	*See section 3-7 for description of date struct _T_TIME.*
See also	*_get_time, _set_date, _set_day_of_week*
Example	-

_sjis2jis

Function	Converts a two-byte shifted JIS character to a two-byte JIS character.
Syntax	#include “api.h” VOID _sjis2jis(UCHAR *in1*, UCHAR *in2*, UCHAR **out1,* UCHAR **out2*);
Parameters	*In1* and *in2* together form the shifted JIS input-character. *Out1* and *out2* together form the JIS output-character.
Return Value	-
Remarks	-
See also	-
Example	-

_str_to_mots

Function	Converts a struct containing an address and a string to a Motorola-S format string.
Syntax	#Include “api.h” UCHAR _str_to_mots(ULONG *addr*, UCHAR **ptr*, UCHAR *num_bytes*);
Parameters	*Addr* is the address where the data should reside, *ptr* is pointer to the string that is to be converted. *Num_bytes* contains the number of unsigned chars of the string that is to be converted, valid values are even numbers from 2 to 120 .
Return Value	0: OK, 1: Error, invalid number of bytes.
Remarks	*For more information on the Motorola-S format, see section 3-6.*
See also	*_mots_to_str*
Example	-

_unmask_basic_int

Function	Unmasks one or all Basic interrupts.
Syntax	#Include “api.h” VOID _unmask_basic_int(UCHAR *type*, UCHAR *level*));
Parameters	*Type* is the number of the interrupt type to unmask (max. 116). If zero, all are unmasked. *Level* contains the mask-level at moment of masking, 0: unmasked, 1: masked.
Return Value	-
Remarks	Always use _unmask_basic_int with _mask_basic_int. Use the return value of the mask function as parameter for the unmask function.
See also	*_mask_basic_int*
Example	-

_update_errors

Function	Updates the error status of all errors in the error stack.
Syntax	#Include “api.h” VOID _update_errors();
Parameters	-
Return Value	-
Remarks	*For more about error handling, see section 3-4*.
See also	*_set_error, _read_error, _del_errors*
Example	-

_update_irin

Function	Copies words of the local copy of the IR OUTPUT area to the IR OUTPUT area.
Syntax	#Include “api.h” VOID _update_irin();
Parameters	-
Return Value	-
Remarks	-
See also	*_update_irout, _write_ir_word*
Example	-

_update_irout

Function	Copies words of the IR INPUT area to the local copy of the IR INPUT area.
Syntax	#Include “api.h” VOID _update_irout();
Parameters	-
Return Value	-
Remarks	-
See also	*_update_irin, _read_ir_word*
Example	-

_write_ir_word

Function	Changes a word of the local copy in the IR OUTPUT area.
Syntax	#include “api.h” UCHAR _write_ir_word(UCHAR *wordnum*, USHORT **val*);
Parameters	*Wordnum* indicates the word of the IR INPUT area, valid values are 0 to 4. *Val* contains the value of the word selected.
Return Value	0: OK, 1: Error, invalid word requested.
Remarks	-
See also	*_read_ir_word, _update_irin*
Example	-

Appendix C
Examples of small applications

- This appendix contains four examples of small applications, CHANGE CASE, DATE AND TIME, PRINT STRING and ALGORITHM. These examples can show how:
  - library functions can be written
  - an application is executed from Basic
  - to use several BIOS routines and how to execute a WDT refresh.

- The included examples:

Example No.	No. of BIOS routines used	Example of
1	-	Writing a library function in general.
2	7	RT clock, data conversion I/O bus comm's, WDT refreshing.
3	2	Port communication, use of a free address.
4	-	Use of multiple source files

- Contents of following pages:
  - Example 1: CHANGE CASE
    1a) Basic listing
    1b) Screen dump
    1c) Library function source code
    1d) Linker file
    1e) Make file
  - Example 2: DATE AND TIME
    2a) Basic listing
    2b) Library function source code
    2c) Linker file
    2d) Make file
  - Example 3: PRINT STRING
    3a) Basic listing
    3b) Library function source code
    3c) Linker file
    3d) Make file
  - Example 4: ALGORITHM
    4a) Basic listing
    4b) Library function source code: ALGORITH.C
    4c) Library function source code: SQUARED.C
    4d) Library function header file: SQUARED.H
    4e) Linker file
    4f) Make file

EXAMPLE 1: CHANGE CASE

From Basic, a function can be called that changes the case of a string to either lowercase, uppercase or sentence case.

1a) The listing of CHNGCASE.BAS, using the library function in an application:

```
10 ’ CHNGCASE.BAS
20 ’ DEMONSTRATES THE USE OF A LIBRARY FUNCTION.
30 ’
40 ’ The library function can convert an input string to lowercase,
50 ’ uppercase, or sentence case (only first character in uppercase).
60 ’
70 ’ (Make sure that the library chngcase.mts is loaded as library 0 !)
80 ’
90 ’
100 DEF LIBFN CASE(STR,INT),INT,0
110 PRINT
120 INPUT "Please, enter a string: ",A$
130 PRINT
140 B%=FNCASE(A$,0)
150 PRINT "String in uppercase:    ";A$
160 B%=FNCASE(A$,1)
170 PRINT "String in lowercase:    ";A$
180 B%=FNCASE(A$,2)
190 PRINT "String in sentencecase: ";A$
200 PRINT
210 END
```

1b) A possible screen-dump from the application CHANGE CASE:

```
> run
Please wait, compiling ... Finished.

Please, enter a string: ascii rules!

String in uppercase:    ASCII RULES!
String in lowercase:    ascii rules!
String in sentencecase: Ascii rules!

>
```

1c) The source code of the library function CHNGCASE.C :

```
/* CHNGCASE.C
   EXAMPLE OF A LIBRARY FUNCTION
   Can convert an input string to uppercase, lowercase or sentence case. When
   it is called from BASIC, the caller specifies how to convert, by means of
   parameter ’option’. The length of the string is retrieved from the byte
   preceding the string...
   Valid values for option are 0, 1 and 2.
   If the function returns a zero, all went fine. If the return value is 1, the
   Value for option was invalid.                                              */
```

```
/* PROTOTYPING:                                                          */
int main(int, char*);
void upcase(char*);
void locase(char*);

/* PRIMARY ROUTINE:                                                      */
int main(int option, char *bas_str)
{
  int teller, len;
  len=*(bas_str-1); /* determine string length */

  if (option==0)
  {
    for (teller=0; teller<len; teller++) /* convert to uppercase */
      upcase(bas_str+teller);
    return 0; /* OK */
  }
  else if ((option==1) || (option==2))   /* convert to lower/sentence case */
  {
    for (teller=0; teller<len; teller++) /* convert to lowercase */
      locase(bas_str+teller);
    if (option==2) upcase(bas_str);      /* convert first char to uppercase. */
    return 0; /* OK */
  }

  else return 1; /* invalid option requested */
}

/* SUB-ROUTINES:                                                         */
void upcase(char* kar)
{
  if ((*kar>96)&&(*kar<123)) /* if character is lowercase */
    *kar=*kar-32;            /*   then convert to uppercase */
}

void locase(char* kar)
{
  if ((*kar>64)&&(*kar<91))  /* if character is uppercase */
    *kar=*kar+32;            /*   then convert to lowercase */
}
```

1d) The linker file CHNGCASE.LD :

```
/* Output architecture: Motorola 68k */
OUTPUT_ARCH(m68k)

/* Uncomment this if you want Motorola-S record format output instead of a COFF format */
/*OUTPUT_FORMAT(srec)*/

/* Search directory */
SEARCH_DIR(.)

/* Link these libraries
 * (Select from: libascii.a, libc.a, libgcc.a, libm.a, ...) */
/*GROUP(-lascii -lc -lgcc -lm)*/

/* The library must be loaded into memory at a fixed address i.e. 0x30000
 * The amount of memory reserved is 64k. This can be increased if this if this
 * is too little. */
MEMORY
{
  ram     : ORIGIN = 0x30000, LENGTH = 0xffff
}

/* stick everything in ram (of course) */
SECTIONS
```

```
{
  .text :
  {
    *(.text)
    . = ALIGN(0x4);

     __CTOR_LIST__ = .;
    LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
    *(.ctors)
    LONG(0)
    __CTOR_END__ = .;
    __DTOR_LIST__ = .;
    LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
    *(.dtors)
     LONG(0)
    __DTOR_END__ = .;

    *(.rodata)
    *(.gcc_except_table)
    __INIT_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.init)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */
    __FINI_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.fini)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */
    _etext = .;
    *(.lit)
  } > ram

  .data BLOCK (0x4) :
  {
    *(.shdata)
    *(.data)
    _edata = .;
  } > ram

  .bss BLOCK (0x4) :
  {
    __bss_start = . ;
    *(.shbss)
    *(.bss)
    *(COMMON)
    *(.eh_fram)
    _end =  ALIGN (0x8);
    __end = _end;
  } > ram

/* Write some data at the end of the memory area to make sure the ASCII
 * unit allocates enough memory */
  .endofmem :
  {
    LONG (0x12345678)
  } > ram

  .stab 0 (NOLOAD) :
  {
    *(.stab)
  }

  .stabstr 0 (NOLOAD) :
  {
    *(.stabstr)
  }
}
```

1e) The make file:

```
# Project name; used for project filenames
PROJNAME = chngcase


# Program names
COMPILE  = compile
LINK     = link
OBJCOPY  = objcopy
REMOVE   = rm -f


# Compile flags
# -On      Set optimisation level n to 0..3
# -Wall    Turn all warnings on
CFLAGS      =  -O0 -Wall

# Link flags
# -Wl,-Map=  Write linker results to a .MAP file
# -T...      Linker file
LDFLAGS     = -Wl,-Map=$(PROJNAME).map -T$(PROJNAME).ld

# Output conversion flags
OBJCFLAGS   = -O srec


INCFILES =
OBJFILES = chngcase.o


all: $(PROJNAME).mts

clean:
       $(REMOVE) $(OBJFILES)
       $(REMOVE) $(PROJNAME).cof
       $(REMOVE) $(PROJNAME).map
       $(REMOVE) $(PROJNAME).mts

chngcase.o: chngcase.c $(INCFILES) makefile
       @ECHO .
       @ECHO Compiling $<
       $(COMPILE) $(CFLAGS) $< -o $@

$(PROJNAME).cof: $(OBJFILES) $(PROJNAME).ld makefile
       @ECHO .
       @ECHO Linking
       $(LINK) $(LDFLAGS) -o $(PROJNAME).cof $(OBJFILES)

$(PROJNAME).mts: $(PROJNAME).cof makefile
       @ECHO .
       @ECHO Burning
       $(OBJCOPY) $(OBJCFLAGS) $< $@
```

EXAMPLE 2: DATE AND TIME

This function continuously writes the date and time to a DM address specified by the caller. By means of a programming console the date and time can be read from the concerning memory addresses.

2a) The listing of DATETIME.BAS, using the library function in an application:

```
10 ’ DATETIME.BAS
20 ’ DEMONSTRATES THE USE OF A LIBRARY FUNCTION THAT USES SOME BIOSROUTINES.
30 ’
40 ’ The library function writes the date and time to an address in DM area.
50 ’ The user specifies the address to write to.
60 ’
70 ’ (Make sure that the library datetime.mts is loaded as library 0 !)
80 ’
90 ’
100 DEF LIBFN DT(INT),INT,0
110 PRINT
120 INPUT"Address ";A%
130 PRINT"Use a programming console or Syswin to look at address";A%
140 PRINT "You can stop this routine by pressing ctrl-x."
150 E=FNDT(A%)
160 IF E=1 THEN PRINT:PRINT"Invalid address requested!":PRINT:GOTO 120
170 IF E=2 THEN PRINT:PRINT"PLC bus transfer busy... Try again later."
180 END
```

2b) The source code of the library function DATETIME.C :

```
/* DATETIME.C
   EXAMPLE OF A LIBRARY FUNCTION USING SOME BIOS ROUTINES, THAT ARE DEFINED IN
   API.H .

   The function continuously read the date, the time and the day of the week.
   These data are converted to readable Binary Coded Decimal, and written to DM
   memory area.
   Every second, the data is updated, and two indicator leds flash.

   The caller determines the DM address to write to.
   Possible return values:   0: OK
                             1: Invalid address requested
                             2: Other PLC-bus transfer is active
                             3: Stopped by wait- or break-interrupt          */



#include "API.H"
#define MAX_NR_CHARS 8
#define MAX_NR_WORDS (MAX_NR_CHARS/2)
#define AT 1
#define NUM_REQ 1


/* Prototyping:                                                              */
USHORT main(USHORT);
UCHAR hex2bcd(UCHAR);


/* MAIN-ROUTINE:                                                             */
USHORT main(USHORT address)
{
  /* variables for temporary storing of time/date/day: */
  _T_TIME tijd;
  _T_DATE datum;
  UCHAR day_ot_week;

  /* parameters for the _pc_write function: */
  UCHAR nr_of_words = MAX_NR_WORDS;
  UCHAR area[MAX_NR_WORDS];
  UCHAR data[MAX_NR_CHARS];

  /* variables used to control program-flow */
  UCHAR old_seconds = 0;
  UCHAR led1on = 0;
  UCHAR problem = 0; /* 0=No problem */

  do
  {
    /* retrieve data */
    (void)_get_time(&tijd);
    (void)_get_date(&datum);
    (void)_get_day_of_week(&day_ot_week);

    data[0] = hex2bcd(tijd.hours);
    data[1] = hex2bcd(tijd.minutes);
    data[2] = hex2bcd(tijd.seconds);
    data[3] = hex2bcd(day_ot_week);
    data[4] = hex2bcd(datum.day);
    data[5] = hex2bcd(datum.month);
    data[7] = hex2bcd(datum.year);

    if (tijd.seconds!=old_seconds) /* Update data every second */
    {
```

```
      old_seconds=tijd.seconds;
      if (data[7]>0x69) data[6]=0x19; else data[6]=0x20; /* determine century */
      problem=_pc_write(AT,NUM_REQ,&nr_of_words,&area[0],&address,(UCHAR*)data);
      if (led1on) /* if led1 is on, then turn it off, and turn on led2 */
      {
         led1on=0;
         _clr_leds(1,0,0,0,0,0);
         _set_leds(0,1,0,0,0,0);
      }
      else /* if led1 is off, then turn it on, and turn led2 off */
      {
         led1on=1;
         _set_leds(1,0,0,0,0,0);
         _clr_leds(0,1,0,0,0,0);
      }
    }

    _wdt_refresh; /* Instead of asm("TRAP #00"); */
  } while (!problem);

  /* end of routine: */
  if (problem==3) return 1;
  else if (problem==5) return 2;
  else if ((problem==1)||(problem==7)) return 3;

  return 0; /* OK */
}


UCHAR hex2bcd(UCHAR hex)
{
  /* This routine is the 8-bit version of _hex2bcd, casting USHORTs<-->UCHARs */
  USHORT result;
  (void)_hex2bcd((USHORT)hex, &result);
  return (UCHAR)result;
}
```

2c) The linker file DATETIME.LD :

```
/* Output architecture: Motorola 68k */
OUTPUT_ARCH(m68k)

/* Uncomment this if you want Motorola-S record format output instead of a COFF format */
/*OUTPUT_FORMAT(srec)*/

/* Search directory */
SEARCH_DIR(.)

/* Link these libraries
 * (Select from: libascii.a, libc.a, libgcc.a, libm.a, ...) */
/*GROUP(-lascii -lc -lgcc -lm)*/

/* The library must be loaded into memory at a fixed address i.e. 0x30000
 * The amount of memory reserved is 64k. This can be increased if this if this
 * is too little. */
MEMORY
{
  ram     : ORIGIN = 0x30000, LENGTH = 0xffff
}

/* stick everything in ram (of course) */
SECTIONS
{
  .text :
```

```
  {
    *(.text)
    . = ALIGN(0x4);

     __CTOR_LIST__ = .;
    LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
    *(.ctors)
    LONG(0)
    __CTOR_END__ = .;
    __DTOR_LIST__ = .;
    LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
    *(.dtors)
     LONG(0)
    __DTOR_END__ = .;

    *(.rodata)
    *(.gcc_except_table)
    __INIT_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.init)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */
    __FINI_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.fini)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */
    _etext = .;
    *(.lit)
  } > ram

  .data BLOCK (0x4) :
  {
    *(.shdata)
    *(.data)
    _edata = .;
  } > ram

  .bss BLOCK (0x4) :
  {
    __bss_start = . ;
    *(.shbss)
    *(.bss)
    *(COMMON)
    *(.eh_fram)
    _end =  ALIGN (0x8);
    __end = _end;
  } > ram

/* Write some data at the end of the memory area to make sure the ASCII
 * unit allocates enough memory */
  .endofmem :
  {
    LONG (0x12345678)
  } > ram

  .stab 0 (NOLOAD) :
  {
    *(.stab)
  }

  .stabstr 0 (NOLOAD) :
  {
    *(.stabstr)
  }
}
```

2d) The make file:

```
# Project name; used for project filenames
PROJNAME = datetime


# Program names
COMPILE  = compile
LINK     = link
OBJCOPY  = objcopy
REMOVE   = rm -f


# Compile flags
# -On      Set optimisation level n to 0..3
# -Wall    Turn all warnings on
CFLAGS      =  -O0 -Wall -I../inc

# Link flags
# -Wl,-Map=  Write linker results to a .MAP file
# -T...      Linker file
LDFLAGS     = -Wl,-Map=$(PROJNAME).map -T$(PROJNAME).ld

# Output conversion flags
OBJCFLAGS   = -O srec


INCFILES = ../inc/api.h
OBJFILES = datetime.o


all: $(PROJNAME).mts

clean:
       $(REMOVE) $(OBJFILES)
       $(REMOVE) $(PROJNAME).cof
       $(REMOVE) $(PROJNAME).map
       $(REMOVE) $(PROJNAME).mts

datetime.o: datetime.c $(INCFILES) makefile
       @ECHO .
       @ECHO Compiling $<
       $(COMPILE) $(CFLAGS) $< -o $@

$(PROJNAME).cof: $(OBJFILES) $(PROJNAME).ld makefile
       @ECHO .
       @ECHO Linking
       $(LINK) $(LDFLAGS) -o $(PROJNAME).cof $(OBJFILES)

$(PROJNAME).mts: $(PROJNAME).cof makefile
       @ECHO .
       @ECHO Burning
       $(OBJCOPY) $(OBJCFLAGS) $< $@
```

EXAMPLE 3: PRINT STRING

This application demonstrates how a library function for a free address can be made. Because of the free address, no function calls can be made.

3a) The listing of PRINTSTR.BAS, using the library function in an application:

```
10 ’ PRINTSTR.BAS
20 ’ DEMONSTRATES THE USE OF A LIBRARY FUNCTION.
30 ’
40 ’ The library function prints a string to the terminal
50 ’
60 ’ This library can be loaded at a free address
70 ’
80 ’ (Make sure that the library printstr.mts is loaded as library 0 !)
90 ’
100 ’
110 DEF LIBFN PRNTSTR(STR),INT,0
120 PRINT
130 INPUT "Please, enter a string: ",A$
140 PRINT
150 B%=FNPRNTSTR(A$)
160 IF B%<>0 THEN PRINT "Could not print the string"
170 END
```

3b) The source code of the library function PRINTSTR.C :

```
/* PRINTSTR.C
   EXAMPLE OF A LIBRARY FUNCTION
   Prints a string.                                                    */

/* INCLUDES                                                            */
#include "api.h"

/* PROTOTYPING:                                                        */
USHORT printstr(UCHAR *str);
USHORT main(VOID);


/* Return values:                                                      */
#define OK 0
#define ERROR 1

/* Constants:                                                          */
#define MODEL_DETECTION 4 /* Tells read_switches to detect unit model  */
#define PRINT_NORMAL 0    /* Print without any trailer                 */
#define PRINT_CRLF 2      /* Print with cr and lf trailer              */
#define MAX_STR_LEN 256   /* Maximum string length                     */


/* PRIMARY ROUTINE:                                                    */
USHORT printstr(UCHAR *str)
{
  UCHAR port;
  UCHAR l; /* dummy */
  UCHAR length;
  UCHAR count;
  UCHAR ch[8];
  UCHAR strcopy[MAX_STR_LEN];

  ch[0] = ’P’;
```

```
  ch[1] = 'r';
  ch[2] = 'i';
  ch[3] = 'n';
  ch[4] = 't';
  ch[5] = ':';
  ch[6] = ' ';
  ch[7] = '\0';

  /* Copy the string and add a '\0' to terminate the string */
  length = *(str-1);
  for (count=0; count<length; count++)
  {
    strcopy[count] = *str;
    str++; /* Increment pointer */
  }
  strcopy[count] = '\0';

  if (_read_switches(MODEL_DETECTION,&port))
  {
    return ERROR;
  }
  else
  {
    /* Determine terminal port, by checking ASCII unit model. */
    port=(port&0x02)+3; /* Now, port contains 3 if ASC31 else port contains 5 */
    if (port==5) port=1;

    (VOID)_print_to_port(port, PRINT_NORMAL, &ch[0], &l);
    (VOID)_print_to_port(port, PRINT_CRLF, &strcopy[0], &l);
  }
  return OK;
}

/* Dummy main routine                                                    */
USHORT main(VOID)
{
  return 0;
}
```

3c) The linker file PRINTSTR.LD :

```
/* Startup module must be specified to make sure crt0.o is not loaded first */
STARTUP(printstr.o)

/* Output architecture: Motorola 68k */
OUTPUT_ARCH(m68k)

/* Uncomment this if you want Motorola-S record format output instead of
 * a COFF format */
/*OUTPUT_FORMAT(srec)*/

/* Search directory */
SEARCH_DIR(.)

/* Link these libraries
 * (Select from: libascii.a, libc.a, libgcc.a, libm.a, ...) */
/*GROUP(-lascii -lc -lgcc -lm)*/

/* Set the memory origin to 0x00000, because the library will be loaded at a
 * free addres */
MEMORY
{
  ram     : ORIGIN = 0x00000, LENGTH = 0xffff
}
```

```
/* stick everything in ram (of course) */
SECTIONS
{
  .text :
  {
    *(.text)
    . = ALIGN(0x4);

     __CTOR_LIST__ = .;
    LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
    *(.ctors)
    LONG(0)
    __CTOR_END__ = .;
    __DTOR_LIST__ = .;
    LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
    *(.dtors)
     LONG(0)
    __DTOR_END__ = .;

    *(.rodata)
    *(.gcc_except_table)
    __INIT_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.init)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */
    __FINI_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.fini)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */
    _etext = .;
    *(.lit)
  } > ram

  .data BLOCK (0x4) :
  {
    *(.shdata)
    *(.data)
    _edata = .;
  } > ram

  .bss BLOCK (0x4) :
  {
    __bss_start = . ;
    *(.shbss)
    *(.bss)
    *(COMMON)
    *(.eh_fram)
    _end =  ALIGN (0x8);
    __end = _end;
  } > ram

/* Write some data at the end of the memory area to make sure the ASCII
 * unit allocates enough memory */
  .endofmem :
  {
    LONG (0x12345678)
  } > ram

  .stab 0 (NOLOAD) :
  {
    *(.stab)
  }


  .stabstr 0 (NOLOAD) :
  {
```

```
    *(.stabstr)
  }
}
```

3d) The make file:

```
# Project name; used for project filenames
PROJNAME = printstr


# Program names
COMPILE  = compile
LINK     = link
OBJCOPY  = objcopy
REMOVE   = rm -f


# Compile flags
# -On      Set optimisation level n to 0..3
# -Wall    Turn all warnings on
CFLAGS      =  -O0 -Wall -I../inc

# Link flags
# -Wl,-Map=  Write linker results to a .MAP file
# -T...      Linker file
LDFLAGS     = -Wl,-Map=$(PROJNAME).map -T$(PROJNAME).ld

# Output conversion flags
OBJCFLAGS   = -O srec


INCFILES = ../inc/api.h
OBJFILES = printstr.o


all: $(PROJNAME).mts

clean:
       $(REMOVE) $(OBJFILES)
       $(REMOVE) $(PROJNAME).cof
       $(REMOVE) $(PROJNAME).map
       $(REMOVE) $(PROJNAME).mts

printstr.o: printstr.c $(INCFILES) makefile
       @ECHO .
       @ECHO Compiling $<
       $(COMPILE) $(CFLAGS) $< -o $@

$(PROJNAME).cof: $(OBJFILES) $(PROJNAME).ld makefile
       @ECHO .
       @ECHO Linking
       $(LINK) $(LDFLAGS) -o $(PROJNAME).cof

$(PROJNAME).mts: $(PROJNAME).cof makefile
       @ECHO .
       @ECHO Burning
       $(OBJCOPY) $(OBJCFLAGS) $< $@
```

EXAMPLE 4: ALGORITHM

This application demonstrates the use of multiple functions that are in separate files. The application returns a value y that is computed according to the formula: $y = x^2 - x$.

4a) The listing of ALGORITH.BAS, using the library function in an application:

```
10 ’ ALGORITH.BAS
20 ’ DEMONSTRATES THE USE OF A LIBRARY FUNCTION.
30 ’
40 ' The library function computes y = x^2 – x and squares the input parameter
50 '
60 ' (Make sure that the library algorith.mts is loaded as library 0 !)
70 '
80 '
90 DEF LIBFN ALGORTH(ADDR),LNG,0
100 PRINT
110 INPUT "Please, enter a number: ",A&
120 PRINT
130 B&=FNALGORTH(VARPTR(A&))
140 PRINT "The results: "; A&; B&
150 END
```

4b) The source code of ALGORITH.C :

```
/* TWOFILES.C
   EXAMPLE OF A LIBRARY FUNCTION
   This example calls function Squared from squared.c
   Return value: y = x^2 - x                                          */


/* Include header file                                                */
#include "squared.h"


/* PROTOTYPING                                                        */
long main(long*);


/* PRIMARY ROUTINE                                                    */
long main(long* arg)
{
  long result;
  long save;

  save = *arg;                /* Save value of input        */
  *arg = Squared((short)*arg); /* Square input value         */
  result = *arg - save;       /* Subtract saved input value */

  return result; /* Return the computed value */
}
```

4c) The source code of SQUARED.C :

```
/* SQUARED.C
   EXAMPLE OF A LIBRARY FUNCTION
   The function Squared is called by main from algorith.c
   Return value: y = x^2                                                    */


/* Include header file */
#include "squared.h"


/* Calculate arg squared */
long Squared(short arg)
{
  return (long)arg * (long)arg;
}
```

4d) The header file SQUARED.H :

```
/* SQUARED.H
   EXAMPLE OF A LIBRARY FUNCTION                                            */


/* Public function Squared */
extern long Squared(short);
```

4e) The linker file ALGORITH.LD :

```
/* Output architecture: Motorola 68k */
OUTPUT_ARCH(m68k)

/* Uncomment this if you want Motorola-S record format output instead of
 * a COFF format */
/*OUTPUT_FORMAT(srec)*/

/* Search directory */
SEARCH_DIR(.)

/* Link these libraries
 * (Select from: libascii.a, libc.a, libgcc.a, libm.a, ...) */
/*GROUP(-lascii -lc -lgcc -lm)*/

/* The library must be loaded into memory at a fixed address i.e. 0x30000
 * The amount of memory reserved is 64k. This can be increased if this if this
 * is too little. */
MEMORY
{
  ram     : ORIGIN = 0x30000, LENGTH = 0xffff
}

/* stick everything in ram (of course) */
SECTIONS
{
  .text :
```

```
  {
    *(.text)
    . = ALIGN(0x4);

     __CTOR_LIST__ = .;
    LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
    *(.ctors)
    LONG(0)
    __CTOR_END__ = .;
    __DTOR_LIST__ = .;
    LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
    *(.dtors)
     LONG(0)
    __DTOR_END__ = .;

    *(.rodata)
    *(.gcc_except_table)
    __INIT_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.init)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */
    __FINI_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.fini)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */
    _etext = .;
    *(.lit)
  } > ram

  .data BLOCK (0x4) :
  {
    *(.shdata)
    *(.data)
    _edata = .;
  } > ram

  .bss BLOCK (0x4) :
  {
    __bss_start = . ;
    *(.shbss)
    *(.bss)
    *(COMMON)
    *(.eh_fram)
    _end =  ALIGN (0x8);
    __end = _end;
  } > ram

/* Write some data at the end of the memory area to make sure the ASCII
 * unit allocates enough memory */
  .endofmem :
  {
    LONG (0x12345678)
  } > ram

  .stab 0 (NOLOAD) :
  {
    *(.stab)
  }

  .stabstr 0 (NOLOAD) :
  {
    *(.stabstr)
  }
}
```

4f) The make file:

```
# Project name; used for project filenames
PROJNAME = algorith


# Program names
COMPILE  = compile
LINK     = link
OBJCOPY  = objcopy
REMOVE   = rm -f


# Compile flags
# -On      Set optimisation level n to 0..3
# -Wall    Turn all warnings on
CFLAGS      =  -O0 -Wall

# Link flags
# -Wl,-Map=  Write linker results to a .MAP file
# -T...      Linker file
LDFLAGS     = -Wl,-Map=$(PROJNAME).map -T$(PROJNAME).ld

# Output conversion flags
OBJCFLAGS   = -O srec


INCFILES = squared.h
OBJFILES = algorith.o squared.o


all: $(PROJNAME).mts

clean:
       $(REMOVE) $(OBJFILES)
       $(REMOVE) $(PROJNAME).cof
       $(REMOVE) $(PROJNAME).map
       $(REMOVE) $(PROJNAME).mts

algorith.o: algorith.c $(INCFILES) makefile
       @ECHO .
       @ECHO Compiling $<
       $(COMPILE) $(CFLAGS) $< -o $@

squared.o: squared.c $(INCFILES) makefile
       @ECHO .
       @ECHO Compiling $<
       $(COMPILE) $(CFLAGS) $< -o $@

$(PROJNAME).cof: $(OBJFILES) $(PROJNAME).ld makefile
       @ECHO .
       @ECHO Linking
       $(LINK) $(LDFLAGS) -o $(PROJNAME).cof $(OBJFILES)

$(PROJNAME).mts: $(PROJNAME).cof makefile
       @ECHO .
       @ECHO Burning
       $(OBJCOPY) $(OBJCFLAGS) $< $@
```

Appendix D
HMI of Library Interface

Use a cross compiler (and simulator) for writing “other” language routines to execute operations that cannot be processed with BASIC programs. The ASCII unit incorporates the Motorola 68340 CPU.

Library routines can be written for the ASCII unit and called from the BASIC program just like any other BASIC function. A library routine cannot be saved to the personal computer but can only be loaded from the personal computer with the `lib load` command. A total of 10 library functions can be stored in the ASCII unit. Library routines are stored in the Motorola-S format.

The PASCAL calling convention, arguments are pushed from left to right, is used for functions (only a fixed number of arguments is allowed): the caller pushes the arguments, as they appear, from left to right on stack (register A7). The callee (the routine being called) removes the parameters from the stack again. Parameters can be passed by reference by using the VARPTR function in BASIC to specify the address of the parameter. This parameter is then passed as an output parameter on the stack.

Function results are to be returned in registers. Depending on the type, different registers are used, if the return value of a function is assigned to a variable then the Expression Parser/Executor ensures that the values are taken from the correct registers and assigned to the variable. If the type of the return value is a string some temporary storage is required to store the return value. This is allocated before the function is called and freed after assignment.

The table below shows the register assignment for different types of return value.

Return type	Register
char	D0
short int	D0
long int	D0
single float	D0
double float	D0/D1
string pointer	A0

Returning string pointers is a risk since the memory associated with the string is not allocated and may be overwritten at any time before assignment. In this case it is better to pass the string variable as a reference parameter and allocate memory before the call to the library function.

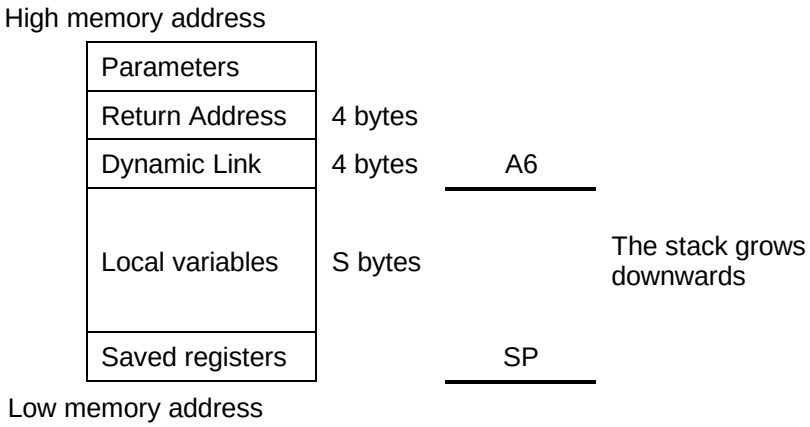
Functions should have a stack frame containing all their local data set up by a LINK/UNLK pair using A6 as the frame pointer. Normal entry code is just a link instruction reserving space for local variables and setting up the frame pointer:

```
LINK.W A6, #(-S)
```

where `S` is the size (in bytes) of the local data of the function. Exit code has to remove local variables from the stack, reset the frame pointer to its previous value, remove the parameters and finally return to the caller:

```
UNLK A6
RTD
```

The following figure shows the layout of the stack frame described above. The “Dynamic Link” is actually the A6 register pointing to the address in the stack of the previous calling routine.



Detailed calling procedure

The following procedure is to be used for calling library routines. Items 6 to 14 are executed by the library routine, the other items are executed by the BASIC executor. If required the available space for the System Stack can be displayed using the PRINT or WATCH statements in BASIC to display the SSTACK System Variable.

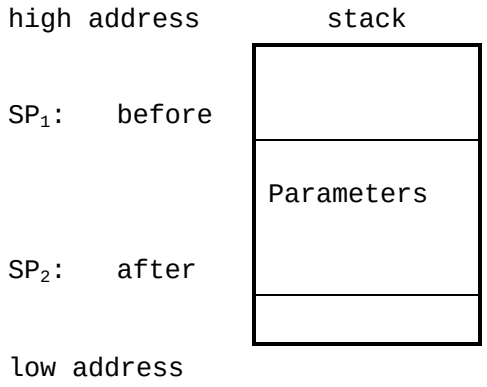
1. Push the routine’s IO parameters on the stack;
2. If required, display top of stack on terminal, showing the input parameters; [optional]
3. Push the return address on the stack;
4. Jump to callee;
    5. Create a new stack frame;
    6. Reserve space for local variables on the stack;
    7. Save all registers on the stack; [optional]
    8. Read input parameters from the stack;
    9. Execute the routine;
    10. Write return parameters on the stack;
    11. Restore saved registers to the stack; [optional]
    12. Set the return value in the appropriate register(s);
    13. Free space for the local variables from the stack;
    14. Restore the previous stack frame;
    15. Set the Program Counter to the return address;
16. Read return value from the appropriate register(s);
17. If required, display top of stack on terminal, showing the output parameter value; [optional]
18. Read output parameters from the stack and pop them from the stack;

Transforming the above items into Motorola assembly instructions and their corresponding action on the stack and the registers is described below.

Item 1:

This can be done in several ways, e.g.:

```
MOVE.W    #0x0002,-(A7)   ; Put 2 on the stack and move the Stack
                          ; Pointer two bytes (one word) down
PEA.L     (-2,A3)         ; Put the address calculated by adding the
                          ; address stored in A3 and -2 on stack and
                          ; move the Stack Pointer four bytes down: SP2
```



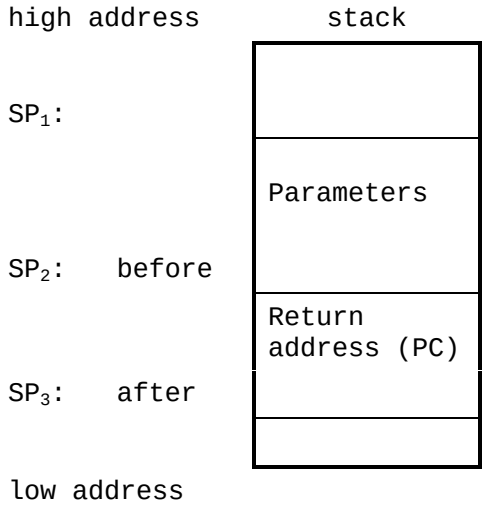
Item 2:

This item does not change the stack pointer.

Item 3 and 4:

These two items can be combined by using the following statement:

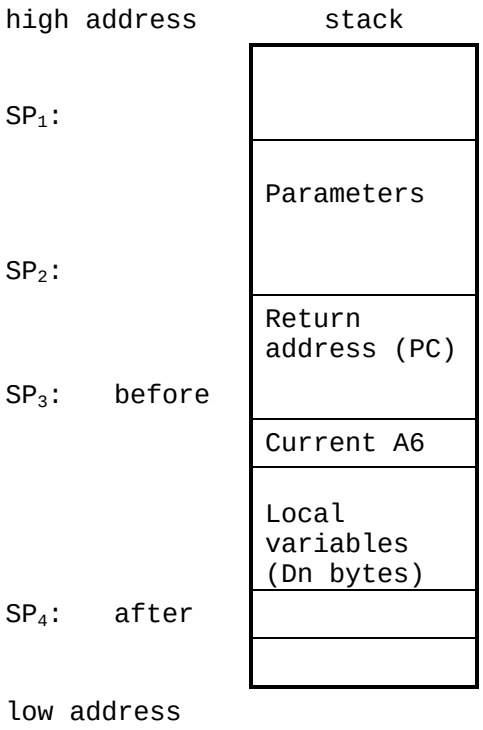
```
JSR <effective address>   ; Move the Stack Pointer four bytes down, SP4,
                          ; copy the current Program Counter to the stack
                          ; and put the PC at the <effective address>
```



Item 5 and 6:

These two items are handled by the following instruction:

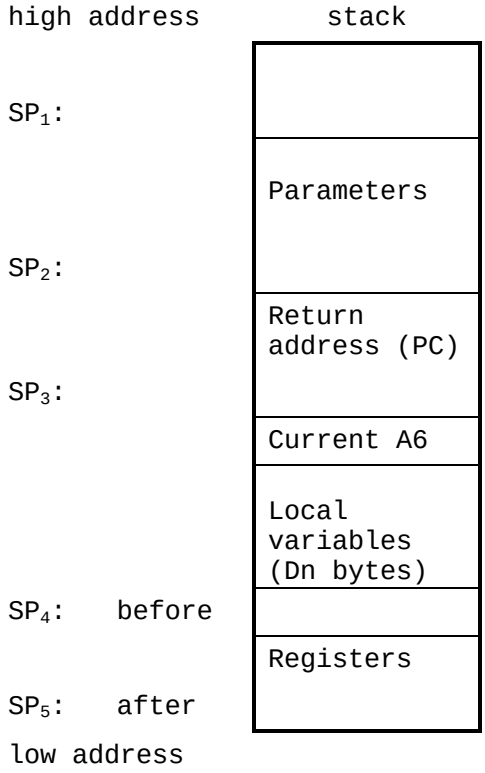
```
LINK A6, -Dn            ; Move the stack pointer four bytes down,
                        ; copy the current A6 register to the stack,
                        ; copy the current stack pointer to A6,
                        ; reserve space on the stack for local
                        ; variables (Dn bytes) by moving the stack
                        ; down: SP5
```



Item 7:

This is done by executing e.g. the following instruction:

```
MOVEM.L   D0/D1/D2/D3/D4/D5/D6/D7/A0/A1/A2/A3/A4/A5/A6,-(A7)
                          ; For each register the stack pointer (A7) is
                          ; moved 4 bytes down: SP3
```



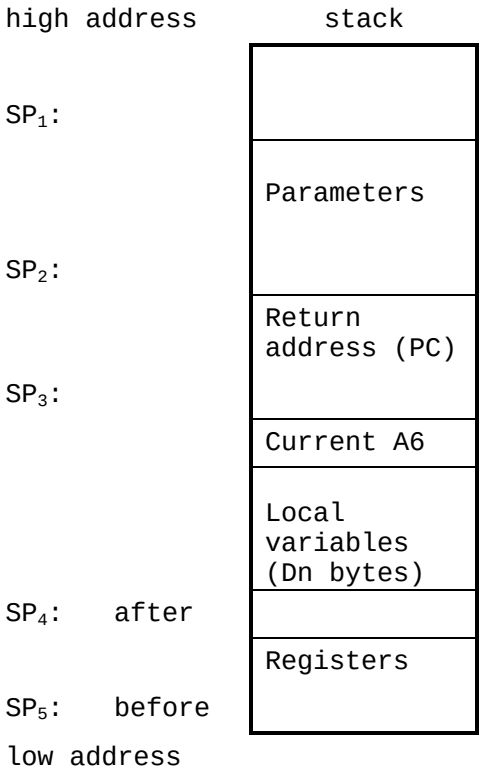
Item 8 to 10:

These items do not change the stack pointer.

Item 11:

This is done by executing e.g. the following instruction:

```
MOVEM.L   (A7)+,D0/D1/D2/D3/D4/D5/D6/D7/A0/A1/A2/A3/A4/A5/A6
                          ; For each register the stack pointer (A7) is
                          ; moved 4 bytes up: SP2
```



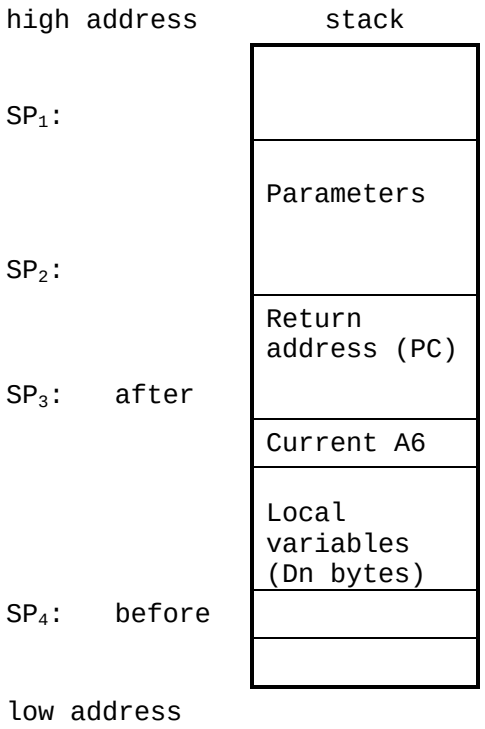
Item 12:

This item does not change the stack pointer.

Item 13 and 14:

These two items are handled by the following instruction:

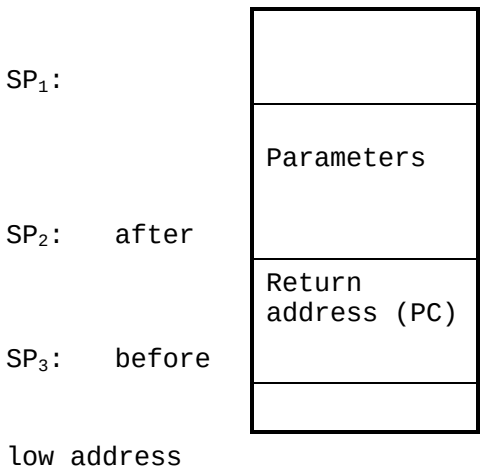
```
UNLK A6                 ; Put the Stack Pointer at the address stored
                        ; in A6 (see item "4 and 5"); this removes all
                        ; local variables from the stack. Restore A6
                        ; from the stack and move the stack pointer 4
                        ; bytes up: SP4
```



Item 15:

This is done by executing e.g. the following instruction:

```
RTS                      ; Write the address pointed at by the Stack
                        ; Pointer to the Program Counter and move the
                        ; Stack Pointer 4 bytes up: SP3
```



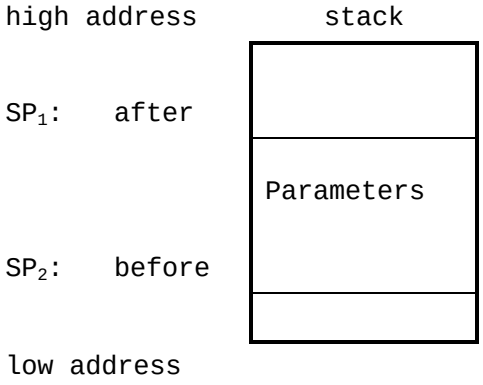
Item 16 and 17:

These items do not change the stack pointer.

Item 18:

This can be done in several ways, e.g.:

```
MOVE.W    (A7)+, D0       ; Put 2 what is on the stack in D0 and move the
                          ; Stack Pointer two bytes (one word) up: SP1
```



Appendix E
Description of Linker and Map files

Linker parameter files

A linker parameter file contains settings that are needed for linking the object files to one coff file. A file will look something like this:

```
/* Startup module can be specified here, because the linker loads crt0.o
 * first by default */
/*STARTUP(crt0.o)*/

/* Output architecture: Motorola 68k */
OUTPUT_ARCH(m68k)

/* Uncomment this if you want Motorola-S record format output instead of
 * a COFF format */
/*OUTPUT_FORMAT(srec)*/

/* Search directory */
SEARCH_DIR(.)

/* Link these libraries and search them until no
 * unresolved are present between them */
GROUP(-lascii -lgcc –lc)

/* Memory origin specified at 0x30000 */
MEMORY
{
  ram     : ORIGIN = 0x30000, LENGTH = 0xffff
}

/* stick everything in ram (of course) */
SECTIONS
{
  .text :
  {
    *(.text)
    . = ALIGN(0x4);

     __CTOR_LIST__ = .;
    LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
    *(.ctors)
    LONG(0)
    __CTOR_END__ = .;
    __DTOR_LIST__ = .;
    LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
    *(.dtors)
     LONG(0)
    __DTOR_END__ = .;

    *(.rodata)
    *(.gcc_except_table)
    __INIT_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.init)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */
    __FINI_SECTION__ = . ;
    LONG (0x4e560000)   /* linkw %fp,#0 */
    *(.fini)
    SHORT (0x4e5e)      /* unlk %fp */
    SHORT (0x4e75)      /* rts */
    _etext = .;
    *(.lit)
  } > ram

  .data BLOCK (0x4) :
  {
    *(.shdata)
    *(.data)
    _edata = .;
```

```
  } > ram

  .bss BLOCK (0x4) :
  {
    __bss_start = . ;
    *(.shbss)
    *(.bss)
    *(COMMON)
    *(.eh_fram)
    _end =  ALIGN (0x8);
    __end = _end;
  } > ram

/* Write some data at the end of the memory area to make sure the ASCII
 * unit allocates enough memory */
  .endofmem :
  {
    LONG (0x12345678)
  } > ram

  .stab 0 (NOLOAD) :
  {
    *(.stab)
  }

  .stabstr 0 (NOLOAD) :
  {
    *(.stabstr)
  }
}
```

The entry ’STARTUP(crt0.o)’ specifies which object module should be placed at the start of the output file. If this entry is omitted, crt0.o is used by default. This entry must be specified, if the primary function does not have the name ’main’.

The entry ’OUTPUT_ARCH(m68k)’ specifies the output architecture. Since the ASCII unit has a Motorola 68340 processor, this should always be ’m68k’.

The entry ’OUTPUT_FORMAT(srec)’ specifies the output format of the generated file. If this entry is omitted, the output format used is ’coff’. The ASCII unit can only load Motorola-S record files. By specifying ’srec’ here, Motorola-S records can be created directly without having to create coff files first. However, because coff files can be used for debugging it’s advised to create coff files first.

The entry ’SEARCH_DIR(.)’ specifies the search directory that is searched for files. ’.’ means that the current directory is searched.

The entry 'GROUP(-lascii -lgcc –lc)' specifies which libraries (libascii.a, libgcc.a, libc.a) should be included. The linker keeps searching these libraries until all unresolved references between functions from these libraries are resolved.

The entry 'MEMORY ' specifies the used memory block(s). For each block a name is declared. The keyword 'ORIGIN' specifies at which memory address the block should start, and 'LENGHT' specifies the size of the memory block.

The entry 'SECTIONS' specifies the memory layout. Inside this section are sub sections.

Functions are placed in the .text section

Initialised global data are placed in the .data section

Uninitialised global data are placed in the .bss section

At the end of most sections is specified in what memory block a section should be placed. The settings as shown in this example should be OK for most situations.

Map files

A map file shows which object files are loaded and it shows a memory map of the generated coff file. A file will look something like this:

```
Memory Configuration

Name             Origin             Length             Attributes
ram              0x00030000         0x0000ffff
*default*        0x00000000         0xffffffff

Linker script and memory map

LOAD c:/xgcc/68k/2_8_1/mshort/mrtd/mcpu32/crt0.o
LOAD example.o
LOAD c:/xgcc/68k/2_8_1/mshort/mrtd/mcpu32/libgcc.a
LOAD c:/xgcc/68k/2_8_1/mshort/mrtd/mcpu32/libgcc.a
START GROUP
LOAD c:/xgcc/68k/2_8_1/mshort/mrtd/mcpu32/libbcc.a
LOAD c:/xgcc/68k/2_8_1/mshort/mrtd/mcpu32/libc.a
LOAD c:/xgcc/68k/2_8_1/mshort/mrtd/mcpu32/libgcc.a
LOAD c:/xgcc/68k/2_8_1/mshort/mrtd/mcpu32/libm.a
END GROUP
                0x00000000                __DYNAMIC=0x0
                0x00000000                PROVIDE (__stack, 0x0)

.text           0x00030000       0x54
 *(.text)
 .text          0x00030000        0xe c:/xgcc/68k/2_8_1/mshort/mrtd/mcpu32/crt0.o
                0x00030006                __main
                0x00030000                start
 *fill*         0x0003000e        0x2
 .text          0x00030010       0x24 example.o
                0x00030010                main
                0x00030034                .=ALIGN(0x4)
                0x00030034                __CTOR_LIST__=.
                0x00030034        0x4 LONG 0x0 (((__CTOR_END__-__CTOR_LIST__)/0x4)-0x2)
 *(.ctors)
                0x00030038        0x4 LONG 0x0
                0x0003003c                __CTOR_END__=.
                0x0003003c                __DTOR_LIST__=.
                0x0003003c        0x4 LONG 0x0 (((__DTOR_END__-__DTOR_LIST__)/0x4)-0x2)
 *(.dtors)
                0x00030040        0x4 LONG 0x0
                0x00030044                __DTOR_END__=.
 *(.rodata)
 *(.gcc_except_table)
                0x00030044                __INIT_SECTION__=.
                0x00030044        0x4 LONG 0x4e560000
 *(.init)
                0x00030048        0x2 SHORT 0x4e5e
                0x0003004a        0x2 SHORT 0x4e75
                0x0003004c                __FINI_SECTION__=.
                0x0003004c        0x4 LONG 0x4e560000
 *(.fini)
                0x00030050        0x2 SHORT 0x4e5e
                0x00030052        0x2 SHORT 0x4e75
                0x00030054                _etext=.
 *(.lit)

.data           0x00030054        0x4
 *(.shdata)
 *(.data)
 .data          0x00030054        0x4 c:/xgcc/68k/2_8_1/mshort/mrtd/mcpu32/crt0.o
                0x00030058                _edata=.

.bss            0x00030058        0x0
                0x00030058                __bss_start=.
 *(.shbss)
 *(.bss)
 *(COMMON)
 *(.eh_fram)
                0x00030058                _end=ALIGN(0x8)
                0x00030058                __end=_end

.endofmem       0x00030058        0x4
                0x00030058        0x4 LONG 0x12345678

.stab           0x00000000      0x1b0
```

```
 *(.stab)
 .stab          0x00000000      0x1b0 example.o

.stabstr        0x00000000      0x35c
 *(.stabstr)
 .stabstr       0x00000000      0x35c example.o
                                  0x0 (size before relaxing)
OUTPUT(example.cof coff-m68k)
```

The output is placed in four columns:

- Name of the memory block
- Start address of the memory block
- Length of the memory block
- Name of the elements inside the memory block

Appendix F
Assembly

GNU GCC can handle assembly files. As a matter of fact, all C source code is converted to intermediate assembly files during compilation. This appendix describes how C files can be converted to assembly files and how coff and MTS files can be disassembled.

The source file that is used for the examples in this chapter looks like this:

```
int main(int argument)
{
  unsigned char val;

  val = (unsigned char)argument;
  val += 1;

  return (int)val;
}
```

Inline assembly

GCC supports inline assembly. The keyword 'asm' can be used for this. The syntax is:

```
asm( "assembly code");
```

Where 'assembly code' represents the inline assembly code. All code between quotation marks is copied exactly the same to the assembly intermediate files. The assembly code between quotation marks can even span multiple lines.

To perform a watchdog timer refresh, the following inline assembly code can be used:

```
asm( "trap #00" );
```

Creating assembly output files from C input files

GCC can convert C files (.c) to assembly files (.s). The assembly file could also be manually adjusted to your needs. Whenever possible use C instead of assembly. Code can be re-used and ported more easily when using C.

To create an assembly output file, add the option '-S' when compiling:

```
compile -O0 -S -o example.s example.c
```

The resulting file will look something like this:

```
        .file   "example.c"
gcc2_compiled.:
__gnu_compiled_c:
.text
        .even
.globl main
main:
        link.w %a6,#-4
        jsr __main
        move.b 9(%a6),-1(%a6)
        addq.b #1,-1(%a6)
        clr.w %d0
        move.b -1(%a6),%d0
        jbra .L1
        .even
.L1:
        unlk %a6
        rtd #2
```

The –O**0** option can be used to disable optimisation. This will make the assembly output more readable, but it will be less efficient. To get highly optimised assembly output, use the option -O**3** instead.

The assembly file can be compiled by supplying the assembly file instead of the C file to the compiler:

```
compile -O0 -c -o example.o example.s
```

Disassembling COFF files and Motorola-S files

With the utility objdump.exe, coff files (.cof) and Motorola-S files (.mts) can be converted to assembly:

```
objdump --disassemble example.cof > outfile
objdump -b srec --architecture=m68k --disassemble-all example.mts > outfile
```

Motorola-S files, and coff files that have been linked with the -Wl,-s option, do not contain symbol information. As a result, the assembly output is not entirely correct:

```
00030000 <.text>:
   30000:       4ef9 0003 0010          jmp 0x30010
   30006:       4e56 0000               linkw %fp,#0
   3000a:       4e5e                    unlk %fp
   3000c:       4e75                    rts
   3000e:       0000 4e56               orib #86,%d0
   30012:       fffc                    0177774
   30014:       4eb9 0003 0006          jsr 0x30006
   3001a:       1d6e 0009 ffff          moveb %fp@(9),%fp@(-1)
   30020:       522e ffff               addqb #1,%fp@(-1)
   30024:       4240                    clrw %d0
   30026:       102e ffff               moveb %fp@(-1),%d0
   3002a:       6000 0002               braw 0x3002e
   3002e:       4e5e                    unlk %fp
   30030:       4e74 0002               rtd #2
```

This problem is caused, because there are some zeroes between functions. A way to solve this is by specifying the start address of the function to be disassembled. The starting address can be obtained from the map file, or it can be obtained by looking at destination addresses for function calls and jumps in the disassembled file. In this case, at address 30000 a jump is made to address 0x30010, so an instruction should start at address 0x30010:

```
objdump --disassemble --start-address=0x30010 example.cof > outfile
```

```
00030010 <.text+0x10>:
   30010:       4e56 fffc               linkw %fp,#-4
   30014:       4eb9 0003 0006          jsr 0x30006
   3001a:       1d6e 0009 ffff          moveb %fp@(9),%fp@(-1)
   30020:       522e ffff               addqb #1,%fp@(-1)
   30024:       4240                    clrw %d0
   30026:       102e ffff               moveb %fp@(-1),%d0
   3002a:       6000 0002               braw 0x3002e
   3002e:       4e5e                    unlk %fp
   30030:       4e74 0002               rtd #2
```

Coff files with symbol information produce a more readable result:

```
00030000 <start>:
   30000:       4ef9 0003 0010          jmp 30010 <main>

00030006 <__main>:
   30006:       4e56 0000               linkw %fp,#0
   3000a:       4e5e                    unlk %fp
   3000c:       4e75                    rts
        ...

00030010 <main>:
   30010:       4e56 fffc               linkw %fp,#-4
   30014:       4eb9 0003 0006          jsr 30006 <__main>
   3001a:       1d6e 0009 ffff          moveb %fp@(9),%fp@(-1)
   30020:       522e ffff               addqb #1,%fp@(-1)
   30024:       4240                    clrw %d0
   30026:       102e ffff               moveb %fp@(-1),%d0
   3002a:       6000 0002               braw 3002e <main+0x1e>
   3002e:       4e5e                    unlk %fp
   30030:       4e74 0002               rtd #2
```

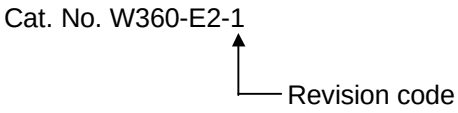
To get debug information in the disassembled output of the coff file, compile and link with the command line option -g. A number can be placed after the -g option to specify the debug level. The default level is 2. If more debug information is required, this can be changed to 3 by specifying -g3.

If the output file contains symbol information, source code and line numbers can be included in the disassembled output:

```
objdump --source --line-numbers example.cof > outfile


00030000 <start>:
   30000:      4ef9 0003 0010        jmp 30010 <main>

00030006 <__main>:
   30006:      4e56 0000             linkw %fp,#0
   3000a:      4e5e                  unlk %fp
   3000c:      4e75                  rts
        ...

00030010 <main>:
main():
c:/ascii/example.c:2
{
   30010:      4e56 fffc             linkw %fp,#-4
   30014:      4eb9 0003 0006        jsr 30006 <__main>
c:/ascii/example.c:4
  unsigned char val1;
  val1 = (unsigned char)argument1;
   3001a:      1d6e 0009 ffff        moveb %fp@(9),%fp@(-1)
c:/ascii/example.c:5
  val1 += 1;
   30020:      522e ffff             addqb #1,%fp@(-1)
c:/ascii/example.c:6
  return (int)val1;
   30024:      4240                  clrw %d0
   30026:      102e ffff             moveb %fp@(-1),%d0
   3002a:      6000 0002             braw 3002e <main+0x1e>
c:/ascii/example.c:7
}
   3002e:      4e5e                  unlk %fp
   30030:      4e74 0002             rtd #2
```

Revision History

A manual revision code appears as a suffix to the catalog number on the front cover of the manual.



The following table outlines the changes made to the manual during each revision. Page numbers refer to the previous version.

Revision code	Date	Revised content
1	March 1999	Initial version