

# 711/720 Portable Data Terminal “C” Language Programming Guide

Ver. 2.01

Copyright © 2000 Syntech Information Co., Ltd.



**SYNTECH INFORMATION CO., LTD.**

Head Office: 8F, No.210, Ta-Tung Rd., Sec.3, Hsi-Chih, Taipei Hsien, Taiwan

Tel: +886-2-2643-8866

Fax: +886-2-2643-8800

e-mail: [support@cipherlab.com.tw](mailto:support@cipherlab.com.tw)

<http://www.cipherlab.com.tw>

# TABLE OF CONTENTS

<b>PREFACE</b> .....	<b>V</b>
<b>1 DEVELOPMENT ENVIRONMENT</b> .....	<b>1</b>
1.1 Directory Structure .....	1
1.2 Setup.....	2
1.3 Development Flow.....	3
1.3.1 Create Your Own “C” source program .....	3
1.3.2 Compile .....	4
1.3.3 Link .....	4
1.3.4 Format Translation.....	6
1.3.5 Download Program to Flash Memory .....	6
1.4 C Compiler.....	7
1.4.1 Size of Types .....	7
1.4.2 Representation Range of Integers .....	7
1.4.3 Floating Types .....	7
1.4.4 Alignment .....	7
1.4.5 Register and Interrupt Handling .....	8
1.4.6 Reserved Words .....	8
1.4.7 Extended Reserved Words .....	8
1.4.8 Bit-Field Usage .....	9
<b>2 711 / 720 FUNCTION LIBRARY</b> .....	<b>11</b>
2.1 System .....	11
2.1.1 Power On Reset (POR) .....	11
2.1.2 System Variables .....	11
2.2 Reader .....	13
2.2.1 Barcode and Magnetic Card Decoding .....	13
2.2.2 Code Type .....	13
2.2.3 Scanner Description Table.....	14
2.2.4 Scan Modes .....	18
2.3 Keyboard Wedge Interface .....	20
2.3.1 Definition of the <i>WedgeSetting</i> array .....	20
2.3.2 KBD / Terminal Type .....	20
2.3.3 Capital Lock Status Setting .....	21
2.3.4 Capital Lock Auto-Detection .....	21
2.3.5 Alphabets Case.....	21
2.3.6 Digits Position .....	21
2.3.7 Shift / Capital Lock Keyboard.....	21
2.3.8 Digit Transmission .....	21
2.3.9 Inter-Character Delay.....	21
2.3.10 Composition of Output String .....	21
2.3.11 Special Note on DEC VT220/320/420.....	22

2.4	Buzzer.....	24
2.4.1	Beeper Sequence .....	24
2.4.2	Beep Frequency.....	24
2.4.3	Beep Duration.....	24
2.5	Calendar .....	26
2.5.1	Leap Year .....	26
2.6	File Manipulation.....	28
2.6.1	File System .....	28
2.6.2	File Name .....	28
2.6.3	File Handle (File Descriptor).....	28
2.6.4	Error Code .....	28
2.6.5	Directory.....	28
2.6.6	DAT Files .....	29
2.6.7	DBF Files and IDX Files.....	29
2.7	LED .....	50
2.8	Keypad.....	51
2.9	LCD.....	56
2.9.1	Graphic Display .....	56
2.9.2	Special Font Files.....	57
2.10	Power .....	64
2.11	Communication Ports .....	65
2.11.1	Parameters.....	65
2.11.2	Receive Buffer.....	65
2.11.3	Transmit Buffer .....	65
2.11.4	Flow Control .....	65
2.12	Memory.....	70
2.13	Smart-Media Card (for 720 only).....	72
2.14	Miscellaneous .....	84
<b>3</b>	<b>STANDARD LIBRARY ROUTINES.....</b>	<b>85</b>
3.1	Input and Output : <stdio.h>.....	85
3.2	Character Class Test : <ctype.h>.....	85
3.3	String Functions : <string.h>.....	85
3.4	Mathematical Functions : <math.h> .....	86
3.5	Utility Function : <stdlib.h>.....	87
3.6	Diagnostics : <assert.h>.....	87
3.7	Variable Argument Lists : <stdarg.h>.....	87

3.8	Non-Local Jumps : <setjmp.h> .....	87
3.9	Signals : <signal.h> .....	88
3.10	Date and Time Function : <time.h> .....	88
3.11	Implementation-defined Limits : <limits.h> and <float.h> .....	88
4	REAL TIME KERNEL.....	89

## Preface

Users can generate customized application programs for the 711/720 Data Terminal by using the “C” Compiler with CipherLab 711/720 Function Library and/or the Basic Compiler with 711/720 Basic Compiler Run-Time Engine. This programming guide describes the application development with the “C” Compiler in chapters. It starts with the general introduction about the feature and operation of the development tool, the definition of the functions/ statements, and sample programs are all included.

Chapter 1, “Development Environment”, gives a concise introduction about the “C” Compiler and provides a step by step description in developing application programs for the 711/720 Data Terminal with the “C” Compiler. Chapter 2, “C Compiler”, discusses some specific characteristics of the “C” Compiler. Chapter 3, “711/720 Function Library”, presents the user callable library routines specific to the features of the 711/720 Data Terminal. In Chapter 4, “Standard Library Routines”, the standard ANSI library routines are briefly described, as the more detailed information can be found in many ANSI C related literature. Chapter 5, “Real Time Kernel”, discusses the concepts of the real time kernel,  $\mu$ C/OS. User can generate a real time multitasking system by using the  $\mu$ C/OS functions.

# 1 Development Environment

## 1.1 Directory Structure

The CipherLab 711/720 Data Terminal “C” Language Development Kit contains six directories, namely, **BIN**, **ETC**, **INCLUDE**, **LIB**, **README**, and **USER**. The purposes/contents of each directory are listed below.

1) **BIN** : This directory contains 18 files.

- 16 execution files for compilation, linking and so on,  
asm900.exe, cc900.exe, dos4gw.exe, f\_amd4.exe,  
mac900.exe, pminfo.exe, privatxm.exe, rminfo.exe,  
thc1.exe, thc2.exe, tuapp.exe, tuconv.exe,  
tufal.exe, tulib.exe, tulink.exe, tumpl.exe
- wemu387.386 : used when DOS extender is to be run under Windows on a 386 machine
- Download.exe : download program via standard RS-232 port

Usage of these executable files will be described further in later sections.

2) **ETC** : 11 files, help and version information of the “C” compiler

3) **INCLUDE**

- 15 Include files for standard library routines  
assert.h ctype.h errno.h float.h  
limits.h locale.h math.h setjmp.h  
signal.h stdarg.h stddef.h stdio.h  
stdlib.h string.h time.h
- 1 Include file for 711(720) Function Library : 711(720)lib.h
- 1 Include file for Real Time Kernel Library : ucos.h

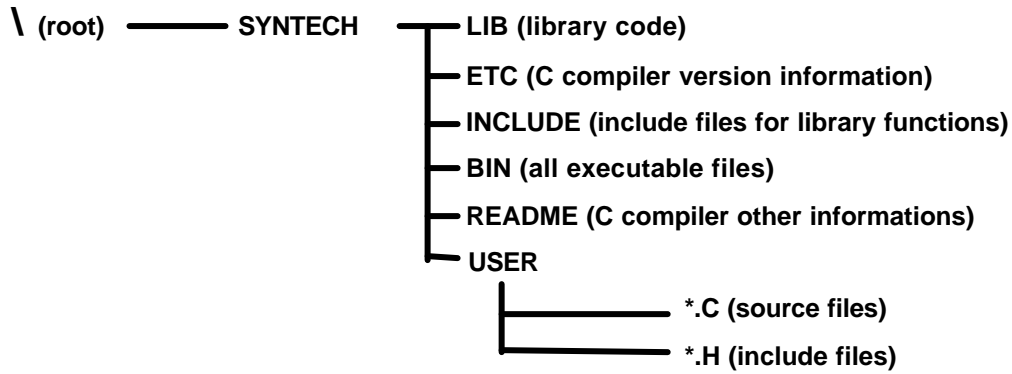
4) **LIB** : Library object code files

- c900ml.lib “C” standard library
- 711(720)lib.lib 711(720) function library

5) **README** : C compiler version update and supplemental information

6) **USER** : contains the source code for the user sample program.

To set up your C language development environment for 711/720, you can create the **\SYNTECH** sub-directory from the root directory and then copy the six mentioned directories to the **\SYNTECH** sub-directory.



## 1.2 Setup

Before using these software programs, some environmental variables must be added to the autoexec.bat.

- 1) path = (your own path);c:\SYNTECH\BIN  
So all executable files (.EXE & .BAT) can be found.
- 2) set THOME900=c:\SYNTECH  
This is a must for the C compiler to locate all necessary files
- 3) set tmp = c:\tmp  
skip this if tmp is already specified.

Step 3 can be ignored if tmp was already specified. This is the temporary working directory for compiler and linker (for memory and file swapping).

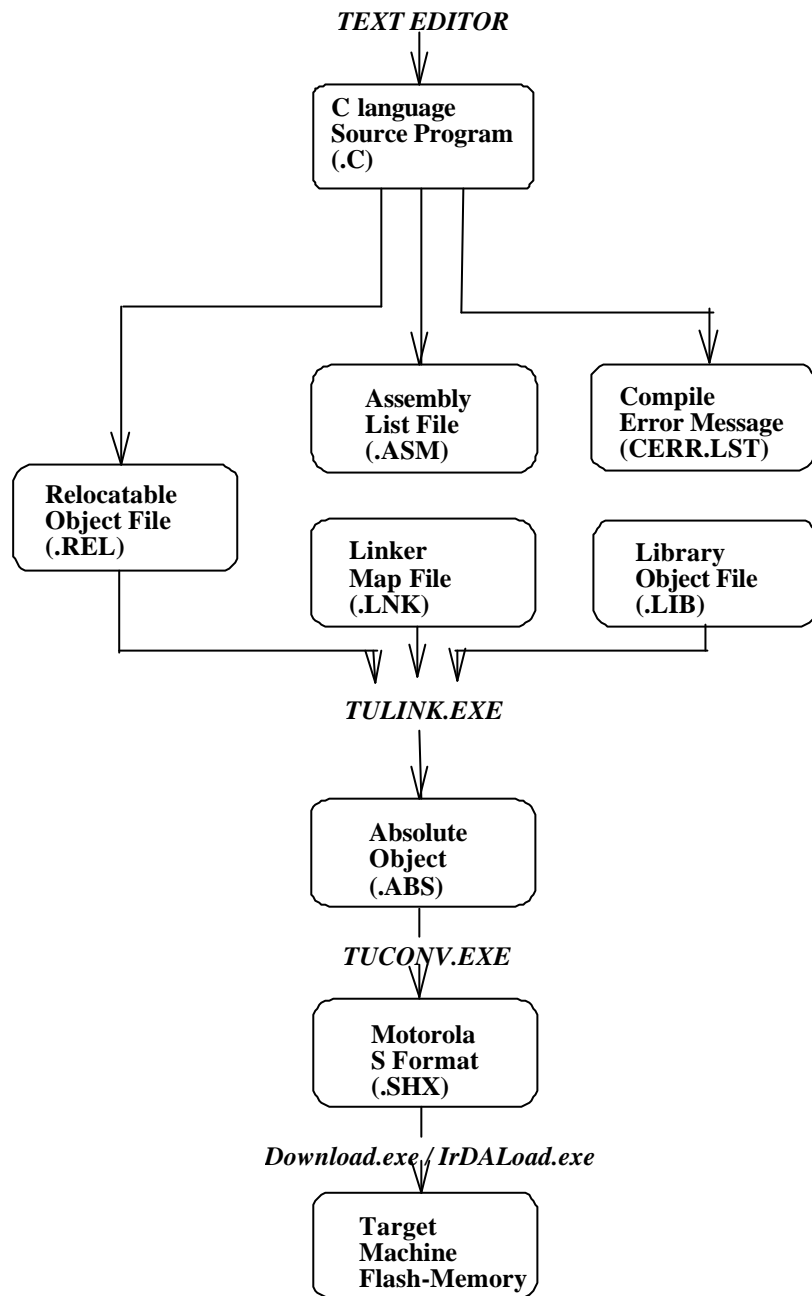
To facilitate efficiency, the compiler invokes a virtual memory manager "DOS4GW". It recognizes and supports various PCs. However, if it does not work on your PC. The program PMINFO can be used to identify the problem. (if you have difficult using the compiler, run the PMINFO, print all messages and then contact Syntech)

If you are using a 386 PC (no floating point unit) and is going to run these programs under MS-Windows compatible BOX. The module "WEMU387.386" must be installed into SYSTEM.INI.

- 1) copy the WEMU387.386 to the SYSTEM directory of the Windows
- 2) add "device=WEMU387.386" to the file SYSTEM.INI

### 1.3 Development Flow

The development process is much like writing any other "C" programs on PC. The flow is illustrated as below,



#### 1.3.1 Create Your Own "C" source program

The first step is to create or modify the desired "C" programs using any text editors. It is recommended to use ".C" as the file extension and create them under the sub-directory "User". And then use the "User" sub-directory as the working directory. Also, it is recommended to separate the whole programs into modules while retaining function integrity. And put modules into separate files to facilitate compilation time.

## 1.3.2 Compile

To compile the "C" programs, use *cc900* command in the subdirectory of the target file.

### **CC900** **–[options]** **FILENAME.C**

For the usage of the *cc900* command and the options, please refer to the *cc900.hlp* in the ETC subdirectory.

The batch file "y.bat" which can be found under the sub-directories *USER* and *USER* has been created to simplify the compiling process.

### **Y** **FILENAME.C**

This batch file invokes the "C" compiler driver which calls many other executable programs under the sub-directory *BIN*. As these programs are invoked by the driver sequentially, their individual use can be ignored. Also, many parameters are set in calling the compiler driver to accommodate target machine environments. In attempting to write your own batch file, remember to put the same parameters. These parameters are listed below,

- -XA1, -XC1, -XD1, -Xp1 : alignment setting, all 1
- -XF : no deletion of assembly file, if examination of the assembly file is not necessary, this option can be removed
- -O3 : set optimization level (can be 0 to 3, no to maximum optimization). If code size and performance is not a problem, this option can be removed which will then set to the default -O0, that is, no optimization at all. If optimization is enabled, care must be taken that some instructions might be optimized and removed. For example,

```
test()
{
    unsigned int old_msec;
    old_msec=sys_msec;
    while (old_msec == sys_msec) ;
}
```

This routine waits till *sys\_msec* changed. And *sys\_msec* is a system variable that is updated each 5 ms by background interrupt. If optimization is enabled, this whole routine is truncated as it is meaningless (which is a dead-loop). To avoid this, the type qualifier "**volatile**" can be used to suppress optimization.

- -c : create object but no link
- -e cerr.lst : create error list file "cerr.lst"

After compilation is completed, a relocatable object file named "*program\_name.rel*" is created which can be used later by the linker to create the absolute object. As the compiler compiles the program into assembler form during the process, an accompanying assembler source file "*program\_name.asm*" is also created. This file helps in debugging if necessary. If any error occurs, they will be put into the file "CERR.LST" for further examination.

## 1.3.3 Link

If the C source programs are successfully compiled into relocatable object files. The linker must be used to create the absolute objects and then can be downloaded into the target machine flash memory for execution. However, a linker map file must be created,.

### **TULINK** **FILENAME.LNK**

This map file "FILENAME.LNK" is used to instruct the linker to allocate absolute addresses of code, data, constant and so on according to the target machine environments. This is a lengthy process as it depends on the hardware architecture. Fortunately, a sample linker map file is provided and few steps are required to customize it for your own need, while leaving hardware-related stuff unchanged.

As you can see from the sample linker file listed as follows, the only parts have to be changed is the file names (under lined & bolded sections). If successfully linked, an absolute object file named "FILE1.ABS" is created. Also a file named "FILE1.MAP" lists all code, variable addresses and error messages if any.

```

-lm -lg          /* parameters for TULINK, don't change */
FILE1.REL      /* your C program name */
FILE2.REL      /* your C program name */
.....
.....
FILEN.REL     /* your C program name */
..\lib\c900ml.lib /* standard library */
..\lib\720lib.lib /* 720 Function library */

/*****
/* User could provide suitable values to
/* the following two variables
/* *****/
MainStackSize = 0x001000;
HeapSize = 0x000100;

/*****
/* Do not modify anything beyond this line
/* *****/
memory
{
    IRAM: org = 0x001100, len = 0x000e00 /* 0x1000 - 0x10ff IntVec */
                                           /* 0x1f00 - 0x1fff Stack */
    RAM : org = 0x804000, len = 0x01c000
    ROM : org = 0xf00000, len = 0x0e0000
}

sections
{
    code org = 0xf00000 : {
        *(f_head)
        *(f_code)
    } > ROM

    area org = 0x804000 : {
        . += MainStackSize;
        . += HeapSize;
        *(f_bcr)
        *(f_area)
    } > RAM

    data org=org(code)+sizeof(code) addr=org(area)+sizeof(area) : {
        *(f_data)
    } /* global variables with initial values */

    xcode org = org(data) + sizeof(data) addr = addr(data) + sizeof(data) : {
        *(f_xcode) /* code reside on RAM */
    }
    const org = org(xcode) + sizeof(xcode) : {
        *(f_const)
        *(f_tail)
    } > ROM
}

SysRamEnd = addr(xcode) + sizeof(xcode);
DataRam = addr(data);
CodeRam = addr(xcode);
HeapTop = org(area) + MainStackSize;

/* End */

```

### 1.3.4 Format Translation

The absolute object file created by TULINK is stored in TOSHIBA's own format. However, a program "TUCONV" can be used to transform it into popular Motorola S format.

```
TUCONV -Fs32 -o FILENAME.shx FILENAME.abs
```

The file extension ".shx" is a must for the code downloader.

The batch file "z.bat" which can be found under the sub-directories user0 and user0 has been created to simplify the linking and format translation process.

**Z**

### 1.3.5 Download Program to Flash Memory

Now the Motorola S format absolute object file *FILENAME.shx* is successfully created. It is ready to be downloaded into the flash memory for testing.

- *FILENAME* : the absolute object code file name, file extension must not be specified as ".shx" is automatically appended.
- *COMPORT* : select the appropriate COM port for transmission.
- *BAUDRATE* : supported baud rates are 115200, 76800, 57600, 38400, 19200, 9600, 4800, 2400.
- *PARITY* : should be no parity.
- *DATABITS* : 8

The baud rate, parity and data bits selected must match the target machine RS232 ports settings.

## 1.4 C Compiler

This C compiler is for TOSHIBA TLCS-900 family 16-bit MCUs. It is mostly ANSI compatible. However, some specific characteristics are listed below,

### 1.4.1 Size of Types

Type	Size in byte
char, unsigned char	1
short int, unsigned short int, int, unsigned int	2
long int, unsigned long int,	4
pointer	4
structure, union	4

### 1.4.2 Representation Range of Integers

Macros concerning the representation ranges of the values of integer types are defined in the header file <limits.h> as below,

Macro Name	Contents
CHAR_BIT	number of bits in a byte (the smallest object)
SCHAR_MIN	minimum value of signed char type
SCHAR_MAX	maximum value of signed char type
CHAR_MIN	minimum value of char type
CHAR_MAX	maximum value of char type
UCHAR_MAX	maximum value of unsigned char type
MB_LEN_MAX	number of bytes in a wide character constant
SHRT_MIN	minimum value of short int type
SHRT_MAX	maximum value of short int type
USHRT_MAX	maximum value of unsigned short int type
INT_MIN	minimum value of int type
INT_MAX	maximum value of int type
UINT_MAX	maximum value of unsigned int type
LONG_MIN	minimum value of long int type
LONG_MAX	maximum value of long int type
ULONG_MAX	maximum value of unsigned long int type

### 1.4.3 Floating Types

Float types are supported and conforms to IEEE standards,

Type	Size in bits
float	32
double	64
long double	64

### 1.4.4 Alignment

Alignments of different types can be adjusted. This is to facilitate CPU performance while sacrificing memory spaces. However as all target systems utilize 8-bit data bus, the alignment does not effect performance and is fixed to 1 for all types. In invoking the C compiler driver -XA1, -XD1, -XC1 and -Xp1 is specified.

## 1.4.5 Register and Interrupt Handling

These are possible through C. However, they are inhibited as all accessing to system resources should be made via Syntech library routines.

## 1.4.6 Reserved Words

Basic reserved (common to all Cs) words are listed below,

auto	double	int	struct	break
else	long	switch	case	enum
register	typedef	char	extern	return
union	const	float	short	unsigned
continue	for	signed	void	default
goto	sizeof	volatile	do	if
static	while			

## 1.4.7 Extended Reserved Words

These reserved words are specific to this C compiler and all of them start with "\_\_ \_", two underscores.

__adcel	__cdcel	__near	__far
__tiny	__asm	__io	
__XWA	__XBC	__XDE	__XHL
__XIX	__XIY	__XIZ	__XSP
__WA	__BC	__DE	__HL
__IX	__IY	__IZ	__W
__A	__B	__C	__D
__E	__H	__L	__SF
__ZF	__VF	__CF	
__DMAS0	__DMAS1	__DMAS2	__DMAS3
__DMAD0	__DMAD1	__DMAD2	__DMAD3
__DMAC0	__DMAC1	__DMAC2	__DMAC3
__DMAM0	__DMAM1	__DMAM2	__DMAM3
__NSP	__XNSP	__INTNEST	

## 1.4.8 Bit-Field Usage

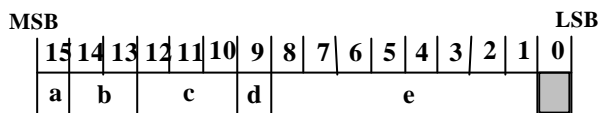
The following types can be used as the bit field base types.

Type	Bits
char, unsigned char	8
short int, int, unsigned short int, unsigned int	16
long int, unsigned long int	32

The allocation is made as follows,

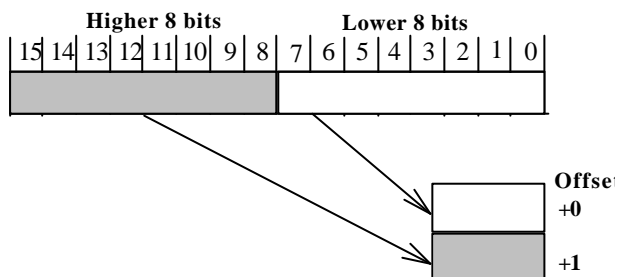
- Fields are stored from the highest bits

```
struct field1 {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int c:3;
    unsigned int d:1;
    unsigned int e:8;
}
```



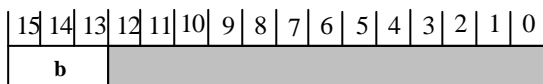
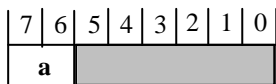
- Little endian

If the base type of a bit field member is a type requiring two bytes or more (e.g. unsigned int), the data is stored in memory after its bytes are turned topside down.



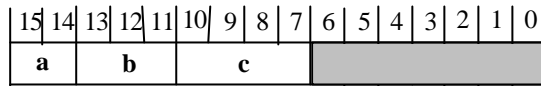
- Different types : A bit field with different type is assigned to a new area

```
struct field {
    unsigned char a:2;
    unsigned short b:3;
}
```



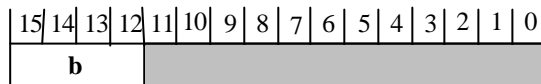
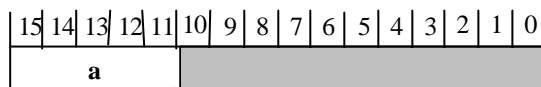
4) Different type (signed/unsigned)

```
struct field {
    signed    short  a:2;
    unsigned short  b:3;
    signed    short  c:4;
}
```



5) Different type (same size)

```
struct field {
    signed    short  a:5;
    unsigned int  b:4;
}
```



The bit-field can be very useful in some cases. However, if memory is not a concern, it is recommended not to use the bit-fields. As the code size and performance are degraded.

## 2 711 / 720 Function Library

CipherLab provides several library routines to facilitate the development of the user's application. These functions are called within the user's C programs to perform a wide variety of tasks, including communications, LCD, buzzer, scanner, file manipulation, etc. They are categorized and described in this chapter by their functions or the resources they work on. The function prototypes of the library routines and the declaration of the system variables can be found in the 711(720) Library Header File, "711(720)lib.h". Assumption was made that the programmer has prior knowledge of "C" language.

### 2.1 System

#### 2.1.1 Power On Reset (POR)

After reset, a portion of library functions called POR routine initializes the system hardware, buffers, and parameters, such as follows,

- RS232 : all disabled
- reader ports : all disabled
- keypad scanning : enabled
- LCD display : initialized and cleared to blank, cursor is on and set to the upper-left corner (0,0)
- calendar chip : initialized
- LEDs : all off
- allocate stack area and other parameters

Control is then transferred to a function called "**main**" which is the start point of the C program. There must be one and only one function in the C program that is called "main" which can then initialize the system according to application needs.

#### 2.1.2 System Variables

There are some global variables that are declared by the system, two of them are system timers that are cleared to 0 upon power up. As they are updated by timer interrupt, DO NOT write to them.

- extern volatile unsigned long sys\_msec;                   /\* in unit of 5 ms \*/
- extern volatile unsigned long sys\_sec;                    /\* in unit of 1 second \*/

Other system variables are as follows,

- extern unsigned int POWER\_ON;  
This variable can be set to either POWERON\_RESUME or POWERON\_RESTART. The default is POWERON\_RESUME, i.e., upon power up, the user program will start from the status of last power off. Note that if user removes the batteries and then reloads batteries, or by entering system menu before normal operation, the user program will always restart itself upon power up.
- extern unsigned int AUTO\_OFF;  
This variable governs the time for the system to automatically shut down the user's program whenever there is no operation during the preset period. The unit for this variable is second, and if it's set to zero, the AUTO\_OFF function will be disabled.
- char ProgVersion[16];  
This characters array can be used to store the version information of user's program. This version information can be checked from the Version submenu of the system menu. Note user must declare this variable in his "C" program to overwrite the system default setting. Following is an example,  
*char ProgVersion[16] = "Power AP 1.00";*

### ChangeSpeed

**purpose** To change the CPU running speed

**syntax** void ChangeSpeed(int speed);

**example call** ChangeSpeed(4);

**description** If high-speed operation is not required, selecting low CPU speed will save battery power. There are five speeds available: 1, 2, 3, 4, and 5, which represents 1/16, 1/8, 1/4, half and full speed respectively.

**returns** none

### \_KeepAlive\_\_

**purpose** To keep user's application program continuous running without automatic shutting down by the system.

**syntax** void \_KeepAlive\_\_ (void);

**example call** \_KeepAlive\_\_ ();

**description** Whenever this routine is called, it will reset the counter governed by the global variable AUTO\_OFF so that user's application program will keep on running without automatic shutting down by the system.

**returns** none

### shut\_down

**purpose** Shut down the system.

**syntax** void shut\_down (void);

**example call** shut\_down();

**description** This routine will shut down the system. Upon power up, the system will restart.

**returns** none

### SysSuspend

**purpose** Shut down the system.

**syntax** void SysSuspend (void);

**example call** SysSuspend();

**description** This routine will shut down the system. Upon power up, the system will resume or restart itself, depending on the system setting.

**returns** none

### system\_restart

**purpose** Re-start the system

**syntax** void system\_restart (void);

**example call** system\_restart();

**description** The routine jumps to the power on reset point and restarts the system.

**returns** none

## 2.2 Reader

The barcode decoding routines consist of 3 functions: **InitScanner1( )**, **Decode( )**, and **HaltScanner1( )**. The *InitScanner1( )* is used to initialize the scanner port. The *Decode( )* function is used to perform decoding. And the *HaltScanner1( )* is used to stop the scanner port from operating.

### 2.2.1 Barcode and Magnetic Card Decoding

To enable barcode decoding capability in the system, the scanner port must be first initialized by calling the *InitScanner1( )* function. After the scanner ports is initialized, the *Decode( )* function can be called in the program loops to perform barcode decoding.

There are four global variables relate to the barcode decoding routines: **ScannerDesTbl**, **CodeBuf**, **CodeLen**, and **CodeType**. These variables are declared by the system, the user program needs not to declare them.

**ScannerDesTbl** : This 28 bytes of unsigned character array governs the operation of the *Decode* routine.

**CodeBuf** : This string contains the decoded data upon successful decoding.

**CodeLen** : This integer indicates the length of the decoded data upon successful decoding.

**CodeType** : This character indicates the type of code (symbology) being decoded upon successful decoding.

### 2.2.2 Code Type

The following list shows the possible values of the *CodeType* variable.

Name	Type	Name	Type
Code 39	A	UPCE with Addon 2	K
Italy Pharma-code	B	UPCE with Addon 5	L
CIP 39	C	EAN8 no Addon	M
Industrial 25	D	EAN8 with Addon 2	N
Interleave 25	E	EAN8 with Addon 5	O
Matrix 25	F	EAN13 no Addon	P
Codabar (NW7)	G	EAN13 with Addon 2	Q
Code 93	H	EAN13 with Addon 5	R
Code128	I	MSI	S
UPCE no Addon	J	Plessey	T

### 2.2.3 Scanner Description Table

The unsigned character array **ScannerDesTbl** governs the Decode function operation. The following table describes the details of the **ScannerDesTbl** variable.

Subscriptor	Bit	Description
0	7	1 : Enable Code 39 0 : Disable Code 39
0	6	1 : Enable Italy Pharma-code 0 : Disable Italy Pharma-code
0	5	1 : Enable CIP 39 0 : Disable CIP 39
0	4	1 : Enable Industrial 25 0 : Disable Industrial 25
0	3	1 : Enable Interleave 25 0 : Disable Interleave 25
0	2	1 : Enable Matrix 25 0 : Disable Matrix 25
0	1	1 : Enable Codabar (NW7) 0 : Disable Codabar (NW7)
0	0	1 : Enable Code 93 0 : Disable Code 93
1	7	1 : Enable Code 128 0 : Disable Code 128
1	6	1 : Enable UPCE no Addon 0 : Disable UPCE no Addon
1	5	1 : Enable UPCE Addon 2 0 : Disable UPCE Addon 2
1	4	1 : Enable UPCE Addon 5 0 : Disable UPCE Addon 5
1	3	1 : Enable EAN8 no Addon 0 : Disable EAN8 no Addon
1	2	1 : Enable EAN8 Addon 2 0 : Disable EAN8 Addon 2
1	1	1 : Enable EAN8 Addon 5 0 : Disable EAN8 Addon 5
1	0	1 : Enable EAN13 no Addon 0 : Disable EAN13 no Addon
2	7	1 : Enable EAN13 Addon 2 0 : Disable EAN13 Addon 2
2	6	1 : Enable EAN13 Addon 5 0 : Disable EAN13 Addon 5
2	5	1 : Enable MSI 0 : Disable MSI
2	4	1 : Enable Plessey 0 : Disable Plessey
2	3	Reserved
2	2 – 0	Reserved
3	7 – 0	Reserved
4	7 – 0	Reserved

continued on next page

continued from previous page

Subscriber	Bit	Description
5	7	1 : Transmitting Code 39 Start/Stop Character 0 : No Transmitting Code 39 Start/Stop Character
5	6	1 : Verifying Code 39 Check Character 0 : No Verifying Code 39 Check Character
5	5	1 : Transmitting Code 39 Check Character 0 : No Transmitting Code 39 Check Character
5	4	1 : Full ASCII Code 39 0 : Standard Code 39
5	3	1 : Transmitting Italy Pharmacode Check Character 0 : No Transmitting Italy Pharmacode Check Character
5	2	1 : Transmitting CIP39 Check Character 0 : No Transmitting CIP39 Check Character
5	1	1 : Verifying Interleave 25 Check Digit 0 : No Verifying Interleave 25 Check Digit
5	0	1 : Transmitting Interleave 25 Check Digit 0 : No Transmitting Interleave 25 Check Digit
6	7	1 : Verifying Industrial 25 Check Digit 0 : No Verifying Industrial 25 Check Digit
6	6	1 : Transmitting Industrial 25 Check Digit 0 : No Transmitting Industrial 25 Check Digit
6	5	1 : Verifying Matrix 25 Check Digit 0 : No Verifying Matrix 25 Check Digit
6	4	1 : Transmitting Matrix 25 Check Digit 0 : No Transmitting Matrix 25 Check Digit
6	3 - 2	Select Interleave25 Start/Stop Pattern 00 : Use Industrial25 Start/Stop Pattern 01 : Use Interleave25 Start/Stop Pattern 10 : Use Matrix25 Start/Stop Pattern 11 : Undefined
6	1 - 0	Select Industrial25 Start/Stop Pattern 00 : Use Industrial25 Start/Stop Pattern 01 : Use Interleave25 Start/Stop Pattern 10 : Use Matrix25 Start/Stop Pattern 11 : Undefined
7	7 - 6	Select Matrix25 Start/Stop Pattern 00 : Use Industrial25 Start/Stop Pattern 01 : Use Interleave25 Start/Stop Pattern 10 : Use Matrix25 Start/Stop Pattern 11 : Undefined
7	5 - 4	Codabar Start/Stop Character 00 : abcd/abcd 01 : abcd/tn*e 10 : ABCD/ABCD 11 : ABCD/TN*E
7	3	1 : Transmitting Codabar Start/Stop Character 0 : No Transmitting Codabar Start/Stop Character
8	2 - 0	Reserved
7	7 - 0	Reserved

continued on next page

continued from previous page

Subscriber	Bit	Description
9	7 - 6	MSI Check Digit Verification 00 : Single Modulo 10 01 : Double Modulo 10 10 : Modulo 11 and Modulo 10 11 : Undefined
9	5 - 4	MSI Check Digit Transmission 00 : the last Check Digit is not transmitted 01 : both Check Digits are transmitted 10 : both Check Digits are not transmitted
9	3	1 : Transmitting Plessey Check Characters 0 : No Transmitting Plessey Check Characters
9	2	1 : Converting Standard Plessey to UK Plessey 0 : No Converting
9	1	1 : Converting UPCE to UPCA 0 : No Converting
9	0	1 : Converting UPCA to EAN13 0 : No Converting
10	7	1 : Enable ISBN Conversion 0 : No Conversion
10	6	1 : Enable ISSN Conversion 0 : No Conversion
10	5	1 : Transmitting UPCE Check Digit 0 : No Transmitting UPCE Check Digit
10	4	1 : Transmitting UPCA Check Digit 0 : No Transmitting UPCA Check Digit
10	3	1 : Transmitting EAN8 Check Digit 0 : No Transmitting EAN8 Check Digit
10	2	1 : Transmitting EAN13 Check Digit 0 : No Transmitting EAN13 Check Digit
10	1	1 : Transmitting UPCE System Number 0 : No Transmitting UPCE System Number
10	0	1 : Transmitting UPCA System Number 0 : No Transmitting UPCA System Number
11	7	1 : Converting EAN8 to EAN13 0 : No Converting
11	6	Reserved
11	5	Reserved
11	4	1 : Enable Negative Barcode 0 : Disable Negative Barcode
11	3 - 2	00 : No Read Redundancy for Scanner Port 1 01 : One Time Read Redundancy for Scanner Port 1 10 : Two Times Read Redundancy for Scanner Port 1 11 : Three Times Read Redundancy for Scanner Port 1
11	1 - 0	Reserved

continued on next page

continued from previous page

<b>Subscriber</b>	<b>Bit</b>	<b>Description</b>
12	7	1 : Industrial 25 Code Length Limitation in Max/Min Length Format 0 : Industrial 25 Code Length Limitation in Fix Length Format
12	6 - 0	Industrial 25 Max Code Length / Fixed Length 1
13	7 - 0	<b>Industrial 25 Min Code Length / Fixed Length 2</b>
14	7	1 : Interleave 25 Code Length Limitation in Max/Min Length Format 0 : Interleave 25 Code Length Limitation in Fix Length Format
14	6 - 0	Interleave 25 Max Code Length / Fixed Length 1
15	7 - 0	Interleave 25 Min Code Length / Fixed Length 2
16	7	1 : Matrix 25 Code Length Limitation in Max/Min Length Format 0 : Matrix 25 Code Length Limitation in Fix Length Format
16	6 - 0	Matrix 25 Max Code Length / Fixed Length 1
17	7 - 0	Matrix 25 Min Code Length / Fixed Length 2
18	7	1 : MSI Code Length Limitation in Max/Min Length Format 0 : MSI Code Length Limitation in Fix Length Format
18	6 - 0	MSI 25 Max Code Length / Fixed Length 1
19	7 - 0	MSI Min Code Length / Fixed Length 2
20	7 - 4	Scan Mode for Scanner Port 1 0000 : Auto Off Mode 0001 : Continuous Mode 0010 : Auto Power Off Mode 0011 : Alternate Mode 0100 : Momentary Mode 0101 : Repeat Mode 0110 : Laser Mode 0111 : Test Mode 1000 : Aiming Mode
20	3 - 0	Reserved
21		Scanner Time-out Duration in seconds for Auto Off and Auto Power Off scanning modes.
22		Reserved

## 2.2.4 Scan Modes

The scanner supports up to 9 scanning modes as described below.

- **Auto Off Mode** : The scanner will start scanning once the switch is triggered. The scanning continues until either a barcode is read or preset scanning period (scanner Time-Out duration) is expired.
- **Continuous Mode** : The scanner is always scanning but just decode once for the same barcode.
- **Auto Power Off Mode** : The scanner will start scanning once the switch is triggered. The scanning continues until a preset scanning period (scanner Time-Out duration) is expired. Unlike the AutoOff mode, the scanner will continue to scan and the scanning period is re-counted each time there is a successful read.
- **Alternate Mode** : The scanner will start scanning once the switch is triggered. The scanner will keep on scanning until the switch is triggered again.
- **Momentary Mode** : The scanner will be scanning as long as the switch is depressed.
- **Repeat Mode** : The scanner is always scanning just like the Continuous Mode. But now the switch acts like a "re-transmit button". If the switch is triggered within 1 second after a good read, the same data will be transmitted again without actually reading the barcode. The "re-transmit button" can be triggered as many times as user needs, so long as the time between each trigger does not exceed 1 second. This scan mode is very useful when the same barcode is to be read many times.
- **Laser Mode** : This is the scan mode used most often on laser scanners. The scanner will start scanning once the switch is pressed. The scanning goes on until either a barcode is read or the switch is released.
- **Test Mode** : The scanner is always scanning and will decode repeatedly even with the same barcode.
- **Aiming Mode** : By selecting this mode, user needs to trigger twice for a decoding. That is, the first trigger is for aiming only, while the second trigger will trully start to decode. After first trigger, the scanner will keep on scanning for one second so that user may take aim. But user must press the second trigger within this period (default to one second), otherwise it will be reset and user has to take aim again. This mode is used when two consecutive barcodes are printed too closed that users need to take aim and make sure they don't read the wrong barcode. There is a system global variable *AIMING\_TIMEOUT* that can be used to change the default one-second time-out duration. The unit for this variable is 5ms.

<b>Decode</b>
---------------

**purpose**      Perform barcode decoding.

**syntax**           int Decode (void);

**example call**   while (1) { if (Decode( )) break; }

**description**     Once the scanner port is initialized (by use of *InitScanner1* function), call this *Decode* function to perform barcode decoding. This function should be called constantly in user's program loops when barcode decoding is required.

                    If the barcode decoding is not required for a long period of time, it is recommended that the scanner port should be stopped by use of the *HaltScanner1* function.

                    If the *Decode* function decodes successfully, the decoded data will be placed in the string variable *CodeBuf* with a string terminating character appended. And the integer variable *CodeLen*, and the character variable *CodeType* will reflect the length and the code type of the decoded data respectively.

**returns**         Upon successful decoding, the *Decode* function returns an integer whose value equals to the string length of the decoded data. If decoding failed, an integer value of 0 is returned.

<b>HaltScanner1</b>
---------------------

**purpose**         Stop the scanner port from operating.

**syntax**           void HaltScanner1 (void);

**example call**   HaltScanner1( );

**description**     Use *HaltScanner1* function to stop scanner port from operating. To restart a halted scanner port, the initialization function, *InitScanner1*, must be called.

                    It is recommended that the scanner port should be stopped if the barcode decoding is not required for a long period of time.

**returns**         none

<b>InitScanner1</b>
---------------------

**purpose**         Initialize respective scanner port.

**syntax**           void InitScanner1(void);

**example call**   InitScanner1( );  
                    while (1) { if (Decode( )) break; }

**description**     Use *InitScanner1* function to initialize scanner port. The scanner port won't work unless it is initialized.

**returns**         none

## 2.3 Keyboard Wedge Interface

711/720 is able to send data to the host through the keyboard wedge interface by using **SendData** function. The *SendData* function is governed by a 3-byte unsigned character string -the **WedgeSetting**, which is a system-defined global character array. User must fill it with appropriate values before calling the *SendData* function.

### 2.3.1 Definition of the *WedgeSetting* array

Subscriptor	Bit	Description
0	7 - 0	KBD / Terminal Type
1	7	1 : enable capital lock auto-detection 0 : disable capital lock auto-detection
1	6	1 : capital lock on 0 : capital lock off
1	5	1 : ignore alphabets case 0 : alphabets are case sensitive
1	4 - 3	00 : normal 10 : digits are at lower position 11 : digits are at upper position
1	2-1	00 : normal 10 : capital lock keyboard 11 : shift lock keyboard
1	0	1 : use numeric key pad to transmit digits 0 : use alpha-numeric key to transmit digits
2	7 - 0	inter-character delay

### 2.3.2 KBD / Terminal Type

The following list shows the possible values of **WedgeSetting[0]**.

Setting Value	Terminal Type	Setting Value	Terminal Type
0	Null (Data not Transmitted)	21	PS55 002-81, 003-81
1	PCAT (US)	22	PS55 002-2, 003-2
2	PCAT (FR)	23	PS55 002-82, 003-82
3	PCAT (GR)	24	PS55 002-3, 003-3
4	PCAT (IT)	25	PS55 002-8A, 003-8A
5	PCAT (SV)	26	IBM 3477 TYPE 4
6	PCAT (NO)	27	PS2-30
7	PCAT (UK)	28	Memorex Telex 122 Keys
8	PCAT (BE)	29	PCXT
9	PCAT (SP)	30	IBM 5550
10	PCAT (PO)	31	NEC 5200
11	PS55 A01-1	32	NEC 9800
12	PS55 A01-2	33	DEC VT220,320,420
13	PS55 A01-3	34	Macintosh (ADB)
14	PS55 001-1	35	Hitachi Elles
15	PS55 001-81	36	Wyse Enhance KBD (US)
16	PS55 001-2	37	NEC Astra
17	PS55 001-82	38	Unisys TO-300
18	PS55 001-3	39	Televideo 965
19	PS55 001-8A	40	ADDS 1010
20	PS55 002-1, 003-1		

### 2.3.3 Capital Lock Status Setting

To send alphabets with correct case (upper or lower case), the *SendData* routine must know the capital lock status of keyboard when transmitting data. Incorrect capital lock setting will result in different letter case ('A' becomes 'a', and 'a' becomes 'A').

### 2.3.4 Capital Lock Auto-Detection

When the keyboard type selected is either PCAT (all available languages), PS2-30, PS55, or Memorex Telex, *SendData* routine can automatically detect the capital lock status of keyboard when transmitting data, if this setting is enabled. If this is the case, the *SendData* routine will ignore the capital lock status setting and perform auto-detection when transmitting data. If the auto-detection setting is disabled, the *SendData* routine will transmit alphabets according to the setting of the capital lock status.

If the keyboard type selected is neither PCAT, PS2-30, PS55, nor Memorex Telex, the *SendData* routine will transmit the alphabets according to setting of the capital lock status even though the auto-detection setting is enabled.

### 2.3.5 Alphabets Case

The setting of this bit affect the way *SendData* routine transmits alphabets. The *SendData* routine can transmit alphabets according to their original case (case sensitive) or just ignore it. If ignoring case is selected, the *SendData* routine will always transmit alphabets without adding shift key.

### 2.3.6 Digits Position

This setting can force the *SendData* routine to treat the position of the digit keys on the keyboard differently. If this setting is set to upper, the *SendData* routine will add shift key when transmitting digits. Please configure this setting to **Normal** unless the user are absolutely sure what he is doing. This setting will be effective only when the keyboard type selected is either PCAT (all available language), PS2-30, PS55, or Memorex Telex. Also if the user choose to send digits using numeric keypad, then this setting is meaningless.

### 2.3.7 Shift / Capital Lock Keyboard

This setting can force the *SendData* routine to treat the keyboard type to be a shift lock keyboard or a capital lock keyboard. Please configure this setting to **Normal** unless the user are absolutely sure what he is doing. This setting will be effective only when the keyboard type selected is either PCAT (all available language), PS2-30, PS55, or Memorex Telex.

### 2.3.8 Digit Transmission

This setting instructs the *SendData* routine which group of keys are used to transmit digits, whether to use the digit keys on top of the alphabet keys or use the digit keys on the numeric key pad.

### 2.3.9 Inter-Character Delay

A 0 to 255 ms inter-character delay can be added before transmit each character. This is used to provide some response time for PC to process keyboard input.

### 2.3.10 Composition of Output String

The keyboard wedge character map is shown below. Each character in the output string is translated by this table when *SendData* routine transmits data.

	00	10	20	30	40	50	60	70	80
0		F2	SP	0	@	P	`	p	⓪
1	INS	F3	!	1	A	Q	a	q	①
2	DLT	F4	"	2	B	R	b	r	②
3	Home	F5	#	3	C	S	c	s	③
4	End	F6	\$	4	D	T	d	t	④
5	Up	F7	%	5	E	U	e	u	⑤
6	Down	F8	&	6	F	V	f	v	⑥
7	Left	F9	'	7	G	W	g	w	⑦
8	BS	F10	(	8	H	X	h	x	⑧
9	HT	F11	)	9	I	Y	i	y	⑨
A	LF	F12	*	:	J	Z	j	z	
B	Right	ESC	+	;	K	[	k	{	
C	PgUp	Exec	,	<	L	\			
D	CR	CR*	-	=	M	]	m	}	
E	PgDn		.	>	N	^	n	~	
F	F1		/	?	O	_	o	Dly	

**Dly :**  
 Delay 100 ms  
 ⓪...⑨ :  
 Digits of Numeric Key Pad  
**CR\*** : Enter key on the  
 numeric key pad

The *SendData* routine can not only transmit simple characters as above, but also provide a way to transmit combination key status, or even direct scan code. This is done by inserting some special command code in the output string. A command code is a character whose value is between 0xC0 and 0xFF.

0xC0 : Indicates that the next character is to be treated as scan code. Transmit it as it is, no translation required.

0xC0 | 0x01 : Send next character with Shift key.

0xC0 | 0x02 : Send next character with left Ctrl key.

0xC0 | 0x04 : Send next character with left Alt key.

0xC0 | 0x08 : Send next character with right Ctrl key.

0xC0 | 0x10 : Send next character with right Alt key.

0xC0 | 0x20 : Clear all combination status key after sending the next character.

For example, to send **[A] [Ctrl-Insert] [5] [scan code 0x29] [Tab] [2] [Shift-Ctrl-A] [B] [Alt-1] [Alt-2-Break] [Alt-1] [Alt-3]**, the following characters are fill into the string supplied to the *SendData* routine when calling. Please note that, the scan code 0x29 is actually a space for PCAT, Alt-12 is a form feed character, and Alt-13 is an enter. The **break** after Alt-12 is necessary, if omitted the characters will be treated as Alt-1213 instead of Alt-12 and Alt-13.

0x41, 0xC2, 0x01, 0x35, 0xC0, 0x29, 0x09, 0x32, 0xC3, 0x41, 0x42, 0xC4, 0x31  
 0xE4, 0x32, 0xC4, 0x31, 0xC4, 0x33, 0x00

### 2.3.11 Special Note on DEC VT220/320/420

Because the DEC VT220/320/420 is using RS232 to communicate between DEC and its keyboard, the following two lines must be included into the program, if this keyboard type is selected.

```
open_com(1,0x0b);
open_com(2,0x0b);
```

These 2 lines must be executed immediately after the **WedgeSetting** is initialized.

## SendData

<b>purpose</b>	Send a string to keyboard interface.
<b>syntax</b>	void SendData (char* out_str);
<b>example call</b>	SendData (CodeBuf);
<b>description</b>	<i>SendData</i> routine transmits a string pointed by <i>out_str</i> to the keyboard interface.
<b>returns</b>	None.

## 2.4 Buzzer

This section describes the beeper manipulation routines. The activating of beeper is directed by specifying a **beeper sequence**, which is a series of **beep frequency / beep duration** pairs. Once a beeper sequence is specified, the activation of the beeper is automatically handled by the background operating system. There is no need for the application program to wait for the stop of beeper.

Also there are routines for determining whether a beeper sequence is under going, or to terminate a beeper sequence immediately.

### 2.4.1 Beeper Sequence

A beeper sequence is an integer array that used to instruct how the beeper activates. It is comprised of **beep frequency / beep duration** pairs. Each pair represents one beep. A beep with beep duration value of 0 represents end of beeper sequence, the beeper will then terminate activation.

### 2.4.2 Beep Frequency

A beep frequency is an integer used to specify the frequency (tone) when the beeper activates. The actual frequency that the beeper activates is not the value specified to the beep frequency. It is calculated by the following formula.

$$\text{Beep Frequency} = 76000 / \text{Actual Frequency Desired}$$

For instance, to get a frequency of 4KHz, the value of beep frequency should be 19. If no sound is desired (pause), the beep frequency should be set to 0. A beep with frequency 0 does not terminate the beeper sequence. Suitable frequency for the beeper ranges from 1 to 6 KHz, where peak at 4 KHz.

### 2.4.3 Beep Duration

Beeper duration is an integer used to specify how long the beeper activates with a specified beep frequency. Beep duration is specified in units of 0.01 second. To get a beep of 1 second, the beep duration should be 100. Beep duration with value of 0 will terminate the beeper sequence.

### beeper\_status

<b>purpose</b>	To see whether a beeper sequence is under going or not.
<b>syntax</b>	int beeper_status (void);
<b>example call</b>	while (beeper_status( ));      /* wait till beeper sequence complete */
<b>description</b>	The <i>beeper_status</i> function checks if there is a beeper sequence in progress.
<b>returns</b>	1 if beeper sequence still in progress, 0 otherwise

### off\_beeper

<b>purpose</b>	Terminate beeper sequence.
<b>syntax</b>	void off_beeper (void);
<b>example call</b>	off_beeper ( );
<b>description</b>	The <i>off_beeper</i> function terminates beeper sequence immediately if there is a beeper sequence in progress.
<b>returns</b>	The <i>off_beeper</i> function has no return value.

### on\_beeper

<b>purpose</b>	Assign a beeper sequence to instruct beeper action.
<b>syntax</b>	void on_beeper(int* sequence); int* sequence; /* pointer to integer array where beeper sequence resides */
<b>example call</b>	int two_beeps[]= { 19, 10, 0, 10, 19, 10, 0, 0 }; on_beeper(two_beeps);
<b>description</b>	The <i>on_beeper</i> function assigns a beeper sequence to instruct how the beeper activates. If there is a beeper sequence already in progress, the newly assigned beeper sequence will override the old one.
<b>returns</b>	The <i>on_beeper</i> function has no return value.

## 2.5 Calendar

This section describes the calendar manipulation routines. The system date and time are kept by the calendar chip, and they can be retrieved from or set to the calendar chip by the **get\_time** and **set\_time** functions. A backup rechargeable Lithium battery keeps the calendar chip running even when power is turned off.

Note that the system time variable *sys\_msec*, and *sys\_sec* is maintained by CPU timers and has nothing to do with this calendar chip. Accuracy of these two time variables depends on the CPU clock and is not suitable for precise time manipulation. Also, they are reset to 0 upon power up.

### 2.5.1 Leap Year

The calendar chip automatically handles the leap year. The **year** field set to the calendar chip must be in four-digit year.

#### DayOfWeek

<b>purpose</b>	Get the day of the week information.
<b>syntax</b>	int DayOfWeek (void);
<b>example call</b>	day = DayOfWeek ( );
<b>description</b>	The <i>DayOfWeek</i> function returns the day of week information based on current date.
<b>returns</b>	The <i>DayOfWeek</i> function returns an integer indicating the day of week information. A value of 1 to 6 represents Monday to Saturday accordingly. And a value of 7 indicates Sunday.

#### get\_time

<b>purpose</b>	Get current date and time.
<b>syntax</b>	int get_time (char*cur_time); char* cur_time; /*pointer of character array where the date and time will be copied to */
<b>example call</b>	get_time (system_time);
<b>description</b>	The <i>get_time</i> function reads current date and time from the calendar chip and copies them to a character array specified in the argument <i>cur_time</i> . The character array <i>cur_time</i> allocated must have a minimum of 15 bytes to accommodate the date, time, and the string terminator. The format of the system date and time is listed below.  "YYYYMMDDhhmmss" where <b>YYYY</b> : year, 4 digits <b>MM</b> : month, 2 digits <b>DD</b> : day, 2 digits <b>hh</b> : hour, 2 digits <b>mm</b> : minute, 2 digits <b>ss</b> : second, 2 digits
<b>returns</b>	Normally the <i>get_time</i> function always returns an integer value of 1. If the calendar chip malfunctions, the <i>get_time</i> function will then return 0 to indicate error.

<b>set_time</b>
-----------------

**purpose** Set new date and time to the calendar chip.

**syntax** int set\_time (char\* new\_time);  
char\* new\_time;

**example call** set\_time ("19980105125800"); /\* JAN 5, 1998 12:58:00 \*/

**description** The *set\_time* function set a new system date and time specified in the argument *new\_time* to the calendar chip. The character string *new\_time* must have the following format,

**"YYYYMMDDhhmmss"**

where **YYYY** : year, 4 digits  
**MM** : month, 2 digits, 1-12  
**DD** : day, 2 digits, 1-31  
**hh** : hour, 2 digits, 0-23  
**mm** : minute, 2 digits, 0-59  
**ss** : second, 2 digits, 0-59

**returns** Normally the *set\_time* function always returns an integer value of 1. If the calendar chip malfunctions, the *set\_time* function will then return 0 to indicate error. Also, if the format is illegal (e.g. set hour to 25), the operation is simply denied and the time is not changed.

## 2.6 File Manipulation

This section describes the file manipulation routines provided. These routines can help to make the manipulation of the transaction data and the implementation of data base system very easy. Although the programmer can device his / her own ways of manipulating the data by declaring some huge arrays, the resulting program will become bigger and harder to be debugged, and will also be less efficient in execution speed and memory usage.

There are two different types of file structures supported. The first one is a sequential file structure, which is much like the ordinary sequential file but is modified to support FIFO structure. We call this type of files as DAT files. The DAT files are usually used to store transaction data.

Another file structure supported is an index sequential file structure. Table look-up and report generation is easily done by use of the index sequential file routines. There are actually two types of files in this file structure. One is the file that stores the data records (data members), and the other is the associate key (index) file. These two types of files are called DBF files and IDX files respectively. We will talk about these two file structures in detail later in this section.

Please note that, not all of the routines described in this section apply to both types of files. In the paragraph of each routine description, we have listed the target file types that the routine under description applies.

### 2.6.1 File System

On 711/720 terminal, there is an on-board 1MB/256K bytes base memory (SRAM). This is the place where all the system parameters, program variables, program stack, and file system reside.

### 2.6.2 File Name

A file name is a null terminated character string of at least 1 and up to 8 characters (not including the terminating null), which is used to identify each file in the system. There is no file extension as in MS-DOS operation system. The file name is case sensitive in identifying files in the system. It is given to each file when a file is created. If a file name specified is more than 8 characters, it will be truncated to 8 characters. The file name can be changed later by the *rename* function.

### 2.6.3 File Handle (File Descriptor)

File handles are used to identify files after files are opened. Most of the file manipulation functions needs file handles instead of file names when specifying target files. A file handle is a positive integer (excludes 0) returned from system when a file is created or opened. Subsequent file operation can then use the file handle to identify the file.

### 2.6.4 Error Code

There is a system parameter "**fErrorCode**", which indicates the result of the last file manipulation routine executed. A value other than 0 indicates error. This error code can be fetched by referencing the variable *fErrorCode*, or by calling the *read\_error\_code* function.

### 2.6.5 Directory

The file system of 711/720 does not support tree-like directory structure. That is, no sub-directory can be created. The maximum number of files can exist in the system is limited to 32 files (includes all DAT files, DBF files, and their associate IDX files). The file directory information can be fetched by calling *filelist* routine.

## 2.6.6 DAT Files

The DAT files have a sequential file structure. All the functions that are needed to manipulate sequential files are included in this library. Besides the ordinary sequential file manipulation routines, there are some special routines that can support FIFO data structure.

The data at the beginning of a DAT file can be removed from the DAT file by calling the *delete\_top* or *delete\_topln* function. The new file top (beginning) position, the file pointer position, and the size of the DAT file will be adjusted accordingly after calling either of these functions. The *append* and *appendln* functions can write data directly to the EOF (end of file) position, no matter where the file pointer points to. That is, the file pointer position is not changed after calling these functions.

By using the four functions mentioned above, the FIFO data structure can be easily implemented and this is usually the way to handle the transaction data in real time system (the host computer keeps reading and removing data from top of file, and new data are written to the bottom at the same time).

## 2.6.7 DBF Files and IDX Files

The DBF files and the IDX files form the platform of the data base system. A DBF file has a fixed record length structure. This is the file that stores the data records (members). Whereas, the associate IDX files are the files that keep the information of the position of each record stored in the DBF file, but they are re-arranged (sorted) according to some specific key values.

A library would be a good example to illustrate how DBF and IDX file work. When you are trying to find a specific book in a library, you always start from looking into indexes. The book can be found by looking into the index of **book title, writer, publisher, ISBN number, ...**etc. All these indexes are sorted in ascending order for easy lookup according to some specific information of books (book title, writer, publisher, ISBN number, ...). When the book is found in the index, it will tell you where the book is actually kept.

As you can see, the books kept in the library are analogous to the data records stored in the DBF file, and the various indexes are just its associate IDX files. Some information in the data records (the book title, writer, publisher, and ISBN number) is used to create the IDX files.

Each DBF file can have at most 8 associate IDX files, and each of them is identified by its key (index) number. The key number is assigned by user program when the IDX file is created. The valid key numbers are from 1 to 8.

Data records are not fetched directly from the DBF file but rather through associate IDX files. The value of file pointers of the IDX files (index pointers) does not represent the address of the data records stored in the DBF file. It indicates the sequence number of the specific data record in the IDX file.

## access

**target file type** DAT DBF

**purpose** Check for file existence.

**syntax** int access (char\* filename);  
char\* filename; /\* file name of the file being checked \*/

**example call** if (access("data1")) puts("data1 exist!\n");

**description** Check if the file specified by *filename* exists. If *filename* exceeds 8 characters, it will be truncated to 8 characters.

**returns** If the file specified by *filename* exist, *access* returns an integer value of 1, 0 otherwise. In case of error, *access* will return an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
1	<i>filename</i> is a NULL string.

## add\_member

**target file type** DBF

**purpose** Add a data record (member) to a DBF file.

**syntax** int add\_member (int DBF\_fd, char\* member);  
int DBF\_fd; /\* file handle of target DBF file \*/  
char\* member; /\* pointer to a character array from where the added member is copied \*/

**example call** add\_member(DBF\_fd, member);

**description** The *add\_member* function adds a member specified by the argument *member* to a DBF file whose file handle is *DBF\_fd* and add index entries to all the IDX file associated to it. If the length of the added member is greater than the length defined for the DBF file (*member\_len* in *create\_DBF* function), the member will be truncated to that length.

**returns** If *add\_member* successfully adds the member, it returns an integer value of 1. In case of error, *add\_member* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>DBF_fd</i> does not exist.
4	File specified by <i>DBF_fd</i> is not a DBF file.
7	Invalid file handle
8	File not opened
10	No free file space for adding member.

## append

**target file type** DAT

**purpose** Write a specified number of bytes to bottom (end-of-file position) of a DAT file.

**syntax** int append (int fd, char\* buffer, int count);  
int fd; /\* file handle of the target DAT file \*/  
char\* buffer; /\* pointer to array of characters representing data to be written \*/

```
int count;      /* number of bytes to be written */
```

**example call** `append(fd, "1234567890", 10);`

**description** The *append* function writes the number of bytes specified in the argument *count* from the character array *buffer* to the bottom of a DAT file whose file handle is *fd*. Writing of data starts at the end-of-file position of the file, and the file pointer position is unaffected by the operation. The *append* function will automatically extend the file size of the file to hold the data written.

**returns** The *append* function returns the number of bytes actually written to the file. In case of error, *append* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>fd</i> does not exist.
4	File specified by <i>fd</i> is not a DAT file.
7	Invalid file handle
8	File not opened
9	The value of <i>count</i> is negative.
10	No more free file space for file extension.

**comments** The maximum number of characters can be written is limited to 32767.

<b>appendln</b>
-----------------

**target file type** DAT

**purpose** Write a null terminated character string to the bottom (end-of-file position) of a DAT file.

**syntax**

```
int appendln (int fd, char* buffer);
int fd;      /* file handle of the target DAT file */
char* buffer; /* pointer to array of characters representing data to be
              written */
```

**example call** `appendln (fd, data_buffer);`

**description** The *appendln* function writes a null terminated character string from the character array *buffer* to a DAT file whose file handle is *fd*. Characters are written to the file until a null character (0) is encountered. The null character is also written to the file. Writing of data starts at the end-of-file position. The file pointer position is unaffected by the operation. The *appendln* function will automatically extend the file size of the file to hold the data written.

**returns** The *appendln* function returns the number of bytes actually written to the file (includes the null character). In case of error, *appendln* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>fd</i> does not exist.
4	File specified by <i>fd</i> is not a DAT file.
7	Invalid file handle
8	File not opened
10	No more free file space for file extension.
11	Can not find string terminator in <i>buf</i> .

**comments** The maximum number of characters can be written is limited to 32767.

## chsize

**target file type** DAT

**purpose** Extends or truncates a DAT file.

**syntax** int chsize (int fd, long new\_size);  
int fd; /\* file handle of the target DAT file \*/  
**long new\_size;** /\* new length of file in bytes \*/

**example call** if (chsize(fd,0L)) puts("file truncated!\n");

**description** The *chsize* function truncates or extends the file specified by the argument *fd* to match the new file length in bytes given in the argument *new\_size*. If the file is truncated, all data beyond the new file size will be lost. If the file is extended, no initial value is filled to the newly extended area.

**returns** If *chsize* successfully changes the file size of the specified DAT file, it returns an integer value of 1. In case of error, *chsize* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>fd</i> does not exist.
4	File specified by <i>fd</i> is not a DAT file.
7	Invalid file handle
8	File not opened
10	No more free file space for file extension.

## close

**target file type** DAT

**purpose** Close a DAT file.

**syntax** int close(int fd);  
int fd; /\* file handle of the target DAT file \*/

**example call** if (close(fd)) puts("file closed!\n");

**description** Close a previously opened or created DAT file whose file handle is *fd*.

**returns** *close* returns an integer value of 1 to indicate success. In case of error, *close* returns an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>fd</i> does not exist.
4	File specified by <i>fd</i> is not a DAT file.
7	Invalid file handle
8	File not opened

## close\_DBF

**target file type** DBF

**purpose** Close DBF and its associated IDX file.

**syntax** int close\_DBF (int DBF\_fd);  
int DBF\_fd; /\* file handle of the target DBF file \*/

**example call** if (close\_DBF(DBF\_fd)) send\_lcds("DBF file closed!\n");

**description** Close a previously opened or created DBF file whose file handle is *DBF\_fd*. The *close\_DBF* function not only closes the specified DBF file but also closes all the IDX files associated to it.

**returns** The *close\_DBF* function returns an integer value of 1 to indicate success. In case of error, *close\_DBF* returns an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>DBF_fd</i> does not exist.
4	File specified by <i>DBF_fd</i> is not a DBF file.
7	Invalid file handle
8	File not opened

### create\_DBF

**target file type** DBF

**purpose** Create a DBF file and get the file handle of the file for further processing.

**syntax**  
int create\_DBF (char\* filename, unsigned member\_len);  
char\* filename; /\* file name of the DBF file being created \*/  
unsigned member\_len; /\* member (record) length of the DBF file \*/

**example call** if (fd = create\_DBF("data1",64) > 0) puts("data1 created!\n");

**description** The *create\_DBF* function creates a DBF file specified by *filename* and gets the file handle of the file. A file handle is a positive integer (excludes 0) used to identify the file for subsequent file manipulations on the file. The argument *member\_len* supplied in the function call specifies the maximum member length for the DBF file. Any members subsequently added to this DBF file with length greater than *member\_len* will be truncated to this length. If *filename* exceeds 8 characters, it will be truncated to 8 characters.

**returns** If *create\_DBF* successfully creates the DBF file, it returns the file handle of the file being created. In case of error, *create\_DBF* will return an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
1	<i>filename</i> is a NULL string.
6	Can't create file. Because the maximum number of files allowed in the system is exceeded.
9	Illegal argument : <i>member_len</i>
12	File specified by <i>filename</i> already exists.

### create\_index

**target file type** DBF

**purpose** Create an IDX file of a DBF file.

**Syntax**  
int create\_index (int DBF\_fd, int key\_number, int key\_offset, int key\_len);  
int DBF\_fd; /\* file handle of a DBF file which the target index  
file associated to \*/  
int key\_number; /\*key number of the index file to be created \*/  
int key\_offset; /\* the byte offset address in member where the key

```

                                value begins */
int key_len;          /* the length (size of) of key value for the index */

```

**example call** `create_index (DBF_fd,1,0,10);`

**description** The *create\_index* function creates an IDX file specified by the argument *key\_number* which is associated to a DBF file whose file handle is *DBF\_fd*. The key value field for the index is specified by the argument *key\_offset* and *key\_len*. The argument *key\_offset* specifies the byte offset address where the key value in a member begins. And *key\_len* specifies the length of the key value. The key field defined by *key\_offset* and *key\_len* should be within the member as defined by *member\_len* in *create\_DBF* function. That is, *key\_offset* plus *key\_len* should not be greater than *member\_len*. The *create\_index* function can only be called before any members are added to the DBF file. That is, when the DBF file is empty (no members exist). If any member should exist in the DBF file, *rebuild\_index* should be used instead.

**returns** If *create\_index* successfully creates an IDX file, it returns an integer value of 1. In case of error, *create\_index* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>DBF_fd</i> does not exist.
4	File specified by <i>DBF_fd</i> is not a DBF file.
6	Can't create file. Because the maximum number of files allowed in the system is exceeded.
7	Invalid file handle
8	File not opened
13	Illegal value in argument <i>key_number</i> .
17	Illegal value in argument <i>key_offset</i> , and/or <i>key_len</i> .
18	DBF file specified by <i>DBF_fd</i> is not empty.
19	IDX file specified by <i>key_number</i> already exists.

<b>delete_member</b>
----------------------

**target file type** DBF

**purpose** Delete a member of a DBF file.

**syntax** `int delete_member (int DBF_fd, int key_number);`  
`int DBF_fd; /* file handle of target DBF file */`  
**int key\_number;** */\* key number of the index file whose index pointer pints to the target member \*/*

**example call** `delete_member (DBF_fd, 1);`

**description** The *delete\_member* function deletes the member pointed by the index pointer of an IDX file whose key number is specified in the argument *key\_number*. The DBF file which the IDX file associates to is specified in the argument *DBF\_fd*.

**returns** If *delete\_member* successfully deletes the member, it returns an integer value of 1. In case of error, *delete\_member* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>DBF_fd</i> does not exist.
4	File specified by <i>DBF_fd</i> is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument <i>key_number</i> .
14	The IDX file specified by <i>key_number</i> does not exist.
16	There are no members in the DBF file.

## delete\_top

**target file type** DAT

**purpose** Remove a specified number of bytes from top (beginning-of-file position) of a DAT file.

**syntax**

```
int delete_top (int fd, int count);
int fd;          /* file handle of the target DAT file */
int count;      /* number of bytes to be removed */
```

**example call** `delete_top (fd, 80);`

**description** The *delete\_top* function removes the number of bytes specified in the argument *count* from a DAT file whose file handle is *fd*. Removing of data starts at the beginning-of-file position of the file. The file pointer position is adjusted accordingly by the operation. For instance, if initially the file pointer points to the tenth character, after removing 8 character from the file, the new file pointer will points to the second character of the file. The *delete\_top* function will resize the file size automatically.

**returns** The *delete\_top* function returns the number of bytes actually removed from the file. In case of error, *delete\_top* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>fd</i> does not exist.
4	File specified by <i>fd</i> is not a DAT file.
7	Invalid file handle
8	File not opened
9	The value of <i>count</i> is negative.

## delete\_topln

**target file type** DAT

**purpose** Remove a null terminated character string from the top (beginning-of-file position) of a DAT file.

**syntax**

```
int delete_topln (int fd);
int fd;          /* file handle of the target DAT file */
```

**example call** `delete_topln (fd);`

**description** The *delete\_topln* function removes a line terminated by a null character from a DAT file whose file handle is *fd*. Characters are removed from the file until a null character (\0) or end-of-file is encountered. The null character is also removed from the file. Removing of data starts at the top (beginning-of-file position) of the file, and the file pointer position is adjusted accordingly. The *delete\_topln* function will resize the file size automatically.

**returns** The *delete\_topln* function returns the number of bytes actually removed from the file (includes the null character). In case of error, *delete\_topln* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>fd</i> does not exist.
4	File specified by <i>fd</i> is not a DAT file.
7	Invalid file handle
8	File not opened

## eof

**target file type** DAT

**purpose** Check if file pointer of a DAT file reaches end of file.

**syntax** int eof (int fd);  
int fd; /\* file handle of the target DAT file \*/

**example call** if (eof(fd)) puts("end of file reached!\n");

**description** The *eof* function checks if the file pointer of the DAT file whose file handle is specified in the argument *fd*, points to end-of-file.

**returns** The *eof* function returns an integer value of 1 to indicate an end-of-file and a 0 when not. In case of error, *eof* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>fd</i> does not exist.
4	File specified by <i>fd</i> is not a DAT file.
7	Invalid file handle
8	File not opened

## filelength

**target file type** DAT

**purpose** Get file length information of a DAT file.

**syntax** long filelength (int fd);  
int fd; /\* file handle of the target DAT file \*/

**example call** data\_size = filelength (fd);

**description** The *filelength* function returns the size in number of bytes of the DAT file whose file handle is specified in the argument *fd*.

**returns** The long integer value returned by *filelength* is the size of the DAT file in number of bytes. In case of error, *filelength* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>fd</i> does not exist.
4	File specified by <i>fd</i> is not a DAT file.
7	Invalid file handle
8	File not opened

## filelist

<b>purpose</b>	Get file directory information.
<b>syntax</b>	<pre>int filelist (char* dir); char* dir;    /* pointer to a character array where the file directory               information is copied to */</pre>
<b>example call</b>	<pre>total_file = filelist (dir);</pre>
<b>description</b>	The <i>filelist</i> function copies the file name, file type, and file size information (separated by a blank character) of all files in existence into a character array specified in the argument <i>dir</i> .
<b>returns</b>	The <i>filelist</i> function returns the number of files currently exist in the system.

## get\_member

**target file type** DBF

<b>purpose</b>	Read the member pointed by the index pointer.
<b>syntax</b>	<pre>int get_member (int DBF_fd, int key_number, char* buffer); int DBF_fd;    /* file handle of a DBF file which the target index               file associated to */ int key_number; /* key number of the target index file char* buffer;   /* pointer to a character array where the member is               copied to */</pre>
<b>example call</b>	<pre>if (get_member(DBF_fd,1,buffer) == 0) puts(buffer);</pre>
<b>description</b>	The <i>get_member</i> function copies the member pointed to by a index pointer to a character array specified in the argument <i>buffer</i> . The IDX file concerned is specified in the argument <i>key_number</i> which is associated to a DBF file whose file handle is <i>DBF_fd</i> .
<b>Returns</b>	The <i>get_member</i> function returns an integer value of 1 to indicate success. In case of error, <i>get_member</i> returns an integer value of 0 and an error code is set to the global variable <i>fErrorCode</i> to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>DBF_fd</i> does not exist.
4	File specified by <i>DBF_fd</i> is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument <i>key_number</i> .
14	The IDX file specified by <i>key_number</i> does not exist.
16	There are no members in the DBF file.

## has\_member

**target file type** DBF

<b>purpose</b>	Check if a specific member exist in an IDX file.
<b>syntax</b>	<pre>int has_member (int DBF_fd, int key_number, char* key_value); int DBF_fd;    /* file handle of a DBF file which the target index               file associated to */ int key_number; /* key number of the target index file */</pre>

```
char* key_value; /* pointer of a character array which is used to
identify a specific member */
```

**example call** if (has\_member(DBF\_fd,1,"WANG"))  
puts("WANG is on the name list!\n");

**description** The *has\_member* function tries to locate a member which matches the key value specified in the argument *key\_value* in an IDX file *key\_number*. The IDX file is associated to a DBF file whose file handle is specified in the argument *DBF\_fd*. If there is a complete match to the *key\_value*, the index pointer will point to the first of all matches. In case there are several members with the same key value, the user can then check each member sequentially from the member pointed by the index pointer to find the desired member. If *has\_member* does not find a complete match in the index, the index pointer will still point to the first member with key value greater than *key\_value* specified.

**returns** The *has\_member* function returns an integer value of 1 to indicate a complete match in key value has been found, 0 if not. In case of error, *has\_member* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>DBF_fd</i> does not exist.
4	File specified by <i>DBF_fd</i> is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument <i>key_number</i> .
14	The IDX file specified by <i>key_number</i> does not exist.

<b>lseek</b>
--------------

**target file type** DAT

**purpose** Move file pointer of a DAT file to a new position.

**syntax** long lseek (int fd, long offset, int origin);  
int fd; /\* file handle of the target DAT file \*/  
long offset; /\* offset of new position (in bytes) from origin \*/  
int origin; /\* constant indicating the position from where to offset \*/

**example call** lseek(fd, 512L, 0); /\* skip 512 bytes \*/

**description** The *lseek* function moves the file pointer of a DAT file whose file handle is specified in the argument *fd* to a new position within the file. The new position is specified with an offset byte address to a specific origin. The offset byte address is specified in the argument *offset* which is a long integer. There are 3 possible values for the argument *origin*. The values and their interpretations are listed below.

Value of origin	Interpretation
1	beginning of file
0	current file pointer position
-1	end of file

**returns** When successful, *lseek* returns the new byte offset address of the file pointer from the beginning of file. In case of error, *lseek* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>fd</i> does not exist.
4	File specified by <i>fd</i> is not a DAT file.
7	Invalid file handle
8	File not opened
9	Illegal <i>origin</i> value.
15	New position is beyond end-of-file.

## Iseek\_DBF

**target file type** DBF

**purpose** Move index pointer of an IDX file to a new position.

**syntax** long Iseek\_DBF (int DBF\_fd, int key\_number, long offset, int origin);  
int DBF\_fd; /\* file handle of a DBF file which the target index file  
associated to \*/  
int key\_number; /\* key number of the target index file \*/  
long offset; /\* offset of new position, sequence number from origin \*/  
int origin; /\* constant indicating the position from where to offset \*/

**example call** Iseek\_DBF(DBF\_fd, 1, 1L, 0); /\* move to next member \*/

**description** The *Iseek\_DBF* function moves the index pointer of a INDEX file which is specified in the argument *key\_number* to a new position. The index file is associated to a DBF file whose file handle is in the argument *DBF\_fd*. The new position is specified with an offset sequence address to a specific origin. The offset rank address is specified in the argument *offset* which is a long integer. There are 3 possible values for the argument *origin*. The values and their interpretations are listed below.

Value of origin	Interpretation
1	first index of index file
0	current index pointer position
-1	last index of index file

**returns** When successful, *Iseek\_DBF* returns the new sequence position that the index pointer points to. In case of error, *Iseek\_DBF* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>DBF_fd</i> does not exist.
4	File specified by <i>DBF_fd</i> is not a DBF file.
7	Invalid file handle
8	File not opened
9	Illegal <i>origin</i> value.
13	Illegal value in argument <i>key_number</i> .
14	The IDX file specified by <i>key_number</i> does not exist.
15	New position is beyond end-of-file.

## member\_in\_DBF

**target file type** DBF

**purpose** Determine how many members exist in a DBF file.

**syntax** long member\_in\_DBF (int DBF\_fd);  
int DBF\_fd; /\* file handle of the target DBF file \*/

**example call** total\_member = member\_in\_DBF(DBF\_fd);

**description** The *member\_in\_DBF* function returns the number of member in a DBF file whose file handle is specified in the argument *DBF\_fd*.

**returns** The long integer value returned by *member\_in\_DBF* is the number of members exist in the DBF file. In case of error, *member\_in\_DBF* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>DBF_fd</i> does not exist.
4	File specified by <i>DBF_fd</i> is not a DBF file.
7	Invalid file handle
8	File not opened

## open

**target file type** DAT

**purpose** Open a DAT file and get the file handle of the file for further processing.

**Syntax** int open (char\* filename);  
char\* filename; /\* file name of file to be opened \*/

**example call** if (fd = open("data1") > 0) puts("data1 opened!\n");

**description** The *open* function opens a DAT file specified by *filename* and gets the file handle of the file. A file handle is a positive integer (excludes 0) used to identify the file for subsequent file manipulations on the file. If the file specified by *filename* does not exist, it will be created first. If *filename* exceeds 8 characters, it will be truncated to 8 characters long. After the file is opened, the file pointer points to the beginning of file.

**returns** If *open* successfully opens the file, it returns the file handle of the file being opened. In case of error, *open* will return an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
1	<i>filename</i> is a NULL string.
4	File specified by <i>filename</i> is not a DAT file.
5	File specified by <i>filename</i> is already opened.
6	Can't create file. Because the maximum number of files allowed in the system is exceeded.

## open\_DBF

**target file type** DBF

**purpose** Open a DBF file and get the file handle of the file for further processing.

**syntax** int open\_DBF (char\* filename);  
char\* filename; /\* file name of file to be opened \*/

**example call** if (fd = open\_DBF("data1") > 0) puts("data1 opened!\n");

**description** The *open\_DBF* function opens a DBF file specified by *filename* and gets the file handle of the file. A file handle is a positive integer (excludes 0) used to identify the file for subsequent file manipulations on the file. The *open\_DBF* function will also open all the index (key) files associated to the DBF file being opened simultaneously. If *filename* exceeds 8 characters, it will be truncated to 8 characters long. After the DBF file is

opened, the index pointers of all the associated index (key) files point to the beginning of the respective index.

**returns** If *open\_DBF* successfully opens the DBF file, it returns the file handle of the file being opened. In case of error, *open\_DBF* will return an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
1	<i>filename</i> is a NULL string.
2	File specified by <i>filename</i> does not exist.
4	File specified by <i>filename</i> is not a DBF file.
5	File specified by <i>filename</i> is already opened.

## read

**target file type** DAT

**purpose** Read a specified number of bytes from a DAT file.

**syntax**

```
int read (int fd, char* buffer, unsigned count);
int fd;          /* file handle of the target DAT file */
char* buffer;   /* pointer to array of characters where the read data
                will be placed */
unsigned count; /* number of bytes to be read */
```

**example call** if ((bytes\_read = read(fd, buffer,80)) == -1)  
puts("read error!\n");

**description** The *read* function copies the number of bytes specified in the argument *count* from the DAT file whose file handle is *fd* to the array of characters *buffer*. Reading starts at the current position of the file pointer, which is incremented accordingly when the operation is completed.

**returns** The *read* function returns the number of bytes actually read from the file. In case of error, *read* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by <i>fd</i> is not a DAT file.
7	<i>fd</i> is not a file handle of a previously opened file.

**comments** Since *read* returns an signed integer, the return value should be converted to *unsigned int* when reading more than 32,767 bytes of data from a file or the return value will be negative. Because the number of bytes to be read is specified in an unsigned integer argument, you could theoretically read 65,535 bytes at a time. But 65,535 (or FFFFh) also means -1 in signed representation, so when reading 65,535 bytes the return value indicates an error. The practical maximum then is 65,534.

## read\_error\_code

**purpose** Get the value of the global variable *fErrorCode*.

**syntax** int read\_error\_code( );

**example call** if (read\_error\_code() == 2) puts("File not exist!\n");

**description** The *read\_error\_code* function gets the value of the global variable *fErrorCode* and returns the value to the calling program. The programmer can use this function to get the error code of the file manipulation routine previously called. However, the global variable *fErrorCode* can be directly accessed without making a call to this function.

**returns** The *read\_error\_code* function returns the value of the global variable *fErrorCode*.

## readln

**target file type** DAT

**purpose** Read a line terminated by a null character from a DAT file.

**syntax**

```
int readln(int fd, char* buffer, unsigned max_count);
int fd;          /* file handle of the target DAT file */
char* buffer;    /* pointer to array of characters where the read line will
                  will be placed */
unsigned max_count; /* maximum number of bytes to be read before
                  null character encountered */
```

**example call** `readln(fd, buffer,80);`

**description** The *readln* function reads a line from the DAT file whose file handle is *fd* and stores the characters in the character array *buffer*. Characters are read until end-of-file encountered, a null character ( ) encountered, or the total number of characters read equals the number specified in *max\_count*. The *readln* function then returns the number of bytes actually read from the file. The null character ( ) is also counted if read. If the *readln* function completes its operation not because a null character is read, there will be no null character stored in *buffer*. Reading starts at the current position of the file pointer, which is incremented accordingly when the operation is completed.

**returns** The *readln* function returns the number of bytes actually read from the file (includes the null character if read). In case of error, *readln* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by <i>fd</i> is not a DAT file.
7	<i>fd</i> is not a file handle of a previously opened file.

**comments** Since *readln* returns an signed integer, the return value should be converted to *unsigned int* when reading more than 32,767 bytes of data from a file or the return value will be negative. Because the number of bytes to be read is specified in an unsigned integer argument, you could theoretically read 65,535 bytes at a time. But 65,535 (or FFFFh) also means -1 in signed representation, so when reading 65,535 bytes the return value indicates an error. The practical maximum then is 65,534. The argument *max\_count* is usually set to a value which equals the size of the character array *buffer* to avoid string overflow.

**cautions** Under some situations (end-of-file encountered or *max\_count* reached), there might not be a null character exist in *buffer*.

## rebuild\_index

**target file type** DBF

**purpose** Rebuild an IDX file of a DBF file.

**syntax**

```
int rebuild_index (int DBF_fd, int key_number, int preference_index,
                  int key_offset, int key_len);
int DBF_fd;          /* file handle of a DBF file which the target
                    index file associated to */
int key_number;     /* key number of the index file to be created */
int preference_index; /* key number of the preference index file, see
                    description below */
int key_offset;     /* the byte offset address in member where the
                    key value begins */
int key_len;        /* the length (size of) of key value for the index */
```

**example call** `rebuild_index(DBF_fd,1,0,10);`

**description** The *rebuild\_index* function rebuilds or creates an index file specified by the argument *key\_number* which is associated to a DBF file whose file handle is *DBF\_fd*. If the index file specified by *key\_number* exists, it will be overwritten, otherwise, the *rebuild\_index* function will create and build a new IDX file. The key-value field of the index is specified by the *key\_offset* and *key\_len* arguments. The argument *key\_offset* specifies the byte offset where the key value in a member begins. And *key\_len* specifies the length of the key value. The key field defined by *key\_offset* and *key\_len* should be within the member as defined by *member\_len* in *create\_DBF* function. That is, *key\_offset* plus *key\_len* should not greater than *member\_len*.

The *rebuild\_index* function can be used whenever an index file has same key values in a key field. The argument *preference\_index* specifies the index file from which the *rebuild\_index* function takes as the input sequence for building the index file. For instance, if a report is to be generated by the sequence of date, department, and ID number. And the date data and department data may be duplicated. Then this can be done by rebuilds the ID number index first and then rebuilds the department index with ID number as the preference index, and finally rebuilds the date index with department index as the preference index. The resulting member sequence in the date index will be in date, department, and ID number. If there is no preferred index desired, the *preference\_index* should be 0. The preferred sequence will be the original member sequence in the DBF file.

**returns** If *rebuild\_index* successfully creates / rebuilds an IDX file, it returns an integer value of 0. In case of error, *rebuild\_index* will return an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

### Error Code Interpretation

4	File specified by <i>DBF_fd</i> is not a DBF file.
6	Can't create file. Because the maximum number of files allowed in the system is exceeded.
8	<i>DBF_fd</i> is not a file handle of a previously opened file.
9	Illegal value in argument <i>key_offset</i> , and/or <i>key_len</i> .
10	No more free file space for rebuilding index.
11	Illegal value in argument <i>key_number</i> .
18	Illegal value in argument <i>preference_index</i> .

**target file type** DAT DBF

**purpose** To receive files from host PC and then store the received files on Smart-Media card.

**syntax** int receive\_file (int com\_port, const char \*file\_name);  
int com\_port; /\* the COM port from which files will be received \*/  
char \*file\_name; /\* file name to be used when saving the file \*/

**example call** open\_com (1, 0x08);  
rtn\_val = receive\_file (1, "");

**description** This routine can be used to receive files from host PC or decides that support Zmodem transmission protocol, and the received files will be saved on Smart-Media card. If the file name is not given, the original file name will be used; otherwise, the given file name will replace the original file name.

**returns** If no error found, it returns 0; otherwise, it returns -1.

## remove

**target file type** DAT DBF

**purpose** Delete file.

**syntax** int remove(char\* filename);  
char\* filename; /\* file name of file to be deleted \*/

**example call** if (remove("data1")) puts("data1 deleted!\n");

**description** Delete the file specified by *filename*. If *filename* exceeds 8 characters, it will be truncated to 8 characters long. If the file to be deleted is a DBF file, the DBF file and all the index (key) files associated to it will be deleted altogether.

**returns** If *remove* deletes the file successfully, it returns an integer value of 1. In case of error, *remove* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretations are listed below.

Error Code	Interpretation
1	<i>filename</i> is a NULL string.
2	File specified by <i>filename</i> does not exist.

## remove\_DBF

**target file type** DBF on SMC

**purpose** Delete an DBF file.

**syntax** remove\_DBF(const char\* file\_name);  
const char \*filename; /\* name of DBF file to be deleted \*/

**example call** if (remove\_DBF("dbf1")) puts ("dbf1 deleted!\n");

**description** Delete the DBF file specified by *filename* and the DBF file and all the index (key) files associated to it will be deleted altogether.. If *filename* exceeds 8 characters, it will be truncated to 8 characters long.

**returns** If *remove\_DBF* deletes the file successfully, it returns an integer value of 1. In case of error, *remove\_DBF* will return an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error

condition encountered. Possible error codes and their interpretations are listed below.

Error Code	Interpretation
1	<i>filename</i> is a NULL string.
2	File specified by <i>filename</i> does not exist.
4	The <i>filename</i> is not a DBF file.

<b>remove_index</b>
---------------------

**target file type** DBF

**purpose** Delete an index file.

**syntax** int remove\_index(int DBF\_fd, int key\_number);  
 int DBF\_fd; /\* file handle of a DBF file which the target index  
 file associated to \*/  
 int key\_number; /\* key number of the target index file \*/

**example call** if (remove\_index(DBF\_fd, 1)) puts("index removed!\n");

**description** The *remove\_index* function deletes the index file specified in the argument *key\_number* which is associated to a DBF file whose file handle is *DBF\_fd*.

**returns** The *remove\_index* function returns an integer value of 1 if it successfully deletes the index file. In case of error, *remove\_index* returns an integer value of 0 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by <i>fd</i> is not a DBF file.
8	<i>fd</i> is not a file handle of a previously opened file.
11	Index file specified by <i>key_number</i> does not exist.

<b>rename</b>
---------------

**target file type** DAT DBF

**purpose** Change file name of an existing file.

**syntax** int rename(char\* old\_filename, char\* new\_filename);  
 char\* old\_filename; /\* file name of file to be renamed \*/  
 char\* new\_filename; /\* new file name desired \*/

**example call** if (rename("data1", "text1")) puts("data1 renamed!\n");

**description** Change the file name of the file specified by *old\_filename* to *new\_filename*. If either *old\_filename* or *new\_filename* exceeds 8 characters, it will be truncated to 8 characters long. If the file specified by *old\_filename* is a DBF file, the file name of the DBF file and all the index (key) files associated to it will be changed to *new\_filename* altogether.

**returns** If *rename* successfully changes the file name, it returns an integer value of 1. In case of error, *rename* will return an integer value of 0, and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
1	Either <i>old_filename</i> or <i>new_filename</i> is a NULL string.
2	File specified by <i>old_filename</i> does not exist.
3	A file with file name <i>new_filename</i> already exists.

<b>send_file</b>	<b>720</b>
------------------	------------

**target file type** DAT DBF

**purpose** Send data files to host PC or devices that support Z-modem transmission protocol.

**syntax** int send\_file (int com\_port, const char \*file\_name);  
int com\_port; /\* the COM port to which files will be sent through\*/  
char \*file\_name; /\* name of the to-be-sent file \*/

**example call** open\_com (1, 0x08);  
rtn\_val = send\_file (1, "A:MyData.dat");

**description** This routine can be used to send files to host PC or devices that support Z-modem transmission protocol. Note that the files to be sent must exist on Smart-Media card.

**returns** If no error found, it returns 0; otherwise, it returns -1.

<b>tell</b>
-------------

**target file type** DAT

**purpose** Get file pointer position of a DAT file.

**syntax** long tell (int fd);  
int fd; /\* file handle of the target DAT file \*/

**example call** current\_position = tell (fd);

**description** The *tell* function returns the current file pointer position of the DAT file whose file handle is specified in the argument *fd*. The file pointer position is expressed in number of bytes from the beginning of file. For instance, if the file pointer points to the beginning of file, the file pointer position will be 0L.

**returns** The long integer value returned by *tell* is the current file pointer position in file. In case of error, *tell* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by <i>fd</i> is not a DAT file.
7	<i>fd</i> is not a file handle of a previously opened file.

<b>tell_DBF</b>
-----------------

**target file type** IDX

**purpose** Get index pointer position of an IDX file.

**syntax**        long tell\_DBF (int DBF\_fd, int key\_number);  
                   int DBF\_fd;        /\* file handle of the target DAT file \*/  
                   int key\_number; /\* key number of the target index file \*/

**example call** rank\_number = tell\_DBF(DBF\_fd, 1);

**description**    The *tell\_DBF* function returns the current index pointer position of the IDX file which is specified in the argument *key\_number*. The IDX file is associated to a DBF file whose file handle is specified in the argument *DBF\_fd*. The index pointer position is expressed in rank number in the IDX file. For instance, if the index pointer points to the first index, the index pointer position will be 1L.

**returns**        The long integer value returned by *tell\_DBF* is the current index pointer position in ranks in file. In case of error, *tell\_DBF* returns a long value of -1L and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by <i>DBF_fd</i> is not a DAT file.
8	<i>DBF_fd</i> is not a file handle of a previously opened file.
11	Index file specified by <i>key_number</i> does not exist.

## update\_member

**target file type** DBF

**purpose**        Update a member of a DBF file.

**syntax**        int update\_member(int fd, int key\_number, char\* buffer);  
                   int fd;                /\* file handle of target DBF file \*/  
                   int key\_number; /\* key number of the index file whose index pointer  
   pints to the target member \*/  
                   char\* buffer;     /\* pointer to array of characters representing data to be  
   written \*/

**example call** update\_member (DBF\_fd, 1,1);

**description**    The *update\_member* function updates the member pointed by the index pointer of an IDX file whose key number is specified in the argument *key\_number*. The DBF file which the IDX file associates to is specified in the argument *DBF\_fd*.

**returns**        If *update\_member* successfully updates the member, it returns an integer value of 1. In case of error, *update\_member* will return an integer value of 0. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
2	File specified by <i>DBF_fd</i> does not exist.
4	File specified by <i>DBF_fd</i> is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument <i>key_number</i> .
14	The IDX file specified by <i>key_number</i> does not exist.
16	There are no members in the DBF file.

## write

**target file type** DAT

**purpose** Write a specified number of bytes to a DAT file.

**syntax**

```
int write(int fd, char* buffer, unsigned count);
int fd;          /* file handle of the target DAT file */
char* buffer;   /* pointer to array of characters representing data to be
                written */
unsigned count; number of bytes to be written
```

**example call** `write(fd, data_buffer, 1024);`

**description** The *write* function writes the number of bytes specified in the argument *count* from the character array *buffer* to a DAT file whose file handle is *fd*. Writing of data starts at the current position of the file pointer, which is incremented accordingly when the operation is completed. If the end-of-file condition is encountered during the operation, the file will be extended automatically to complete the operation.

**returns** The *write* function returns the number of bytes actually written to the file. In case of error, *write* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by <i>fd</i> is not a DAT file.
7	<i>fd</i> is not a file handle of a previously opened file.
10	No more free file space for file extension.

## writeln

**target file type** DAT

**purpose** Write a line terminated by a null character (\0) to a DAT file.

**syntax**

```
int writeln(int fd, char* buffer);
int fd;          /* file handle of the target DAT file */
char* buffer;   /* pointer to array of characters representing data to be
                written */
```

**example call** `writeln(fd, data_buffer);`

**description** The *writeln* function writes a line terminated by a null character from the character array *buffer* to a DAT file whose file handle is *fd*. Characters are written to the file until a null character (\0) is encountered. The null character is also written to the file. Writing of data starts at the current position of the file pointer, which is incremented accordingly when the operation is completed. If the end-of-file condition is encountered during the operation, the file will be extended automatically to complete the operation.

**returns** The *writeln* function returns the number of bytes actually written to the file (includes the null character). In case of error, *writeln* returns an integer value of -1 and an error code is set to the global variable *fErrorCode* to indicate the error condition encountered. Possible error codes and their interpretation are listed below.

Error Code	Interpretation
4	File specified by <i>fd</i> is not a DAT file.
7	<i>fd</i> is not a file handle of a previously opened

9 file.  
no null character found in *buffer*  
10 No more free file space for file extension.

## 2.7 LED

Number of LEDs on 711/720 can be used to indicate the system status. They are listed as follows,

Name	Number
LED_RED	0
LED_GREEN	1

### set\_led

**purpose** To set the LED indicators

**syntax** int set\_led(int led, int mode, int duration);  
int led; /\* number of LED to be accessed \*/  
int mode; /\* activation mode \*/  
int duration; /\* duration in unit of 10 milliseconds \*/

**example call** set\_led (LED\_RED, LED\_FLASH, 20); /\* set Red LED to flash for each  
1 second \*/

**description** 3 modes are supported,  
LED\_OFF : off for (duration X 0.01) seconds then on  
LED\_ON : on for (duration X 0.01) seconds then off  
LED\_FLASH : flash, on then off each for (duration X 0.01) seconds  
then repeat

**returns** none

## 2.8 Keypad

A scanning circuitry of 4 by 8 matrix is utilized on the 711/720 keypad. The background routine constantly scans the keypad to see if any key was pressed. There is a keyboard buffer of size 32 bytes. However, if the buffer is full, the keys followed will be ignored. The C program must constantly checks to see if any keystroke is available in the buffer.

### clr\_kb

<b>purpose</b>	To clear the keyboard buffer.
<b>syntax</b>	void clr_kb (void);
<b>example call</b>	clr_kb ( );
<b>description</b>	The <i>clr_kb</i> function clears the keyboard buffer. This function is automatically called by the system program upon power up.
<b>returns</b>	none

### dis\_alpha

<b>purpose</b>	Disable alphabet key stroke processing.
<b>syntax</b>	void dis_alpha (void);
<b>example call</b>	dis_alpha ( );
<b>description</b>	The <i>dis_alpha</i> function disables the alphabet key stroke processing. If the alpha lock status is on prior to calling this function, it will become off after calling this function.
<b>returns</b>	none

### dis\_shift

720

<b>purpose</b>	Disable shift key stroke processing.
<b>syntax</b>	void dis_shift (void);
<b>example call</b>	dis_shift ( );
<b>description</b>	The <i>dis_shift</i> function disables the shift key stroke processing. If the shift lock status is on prior to calling this function, it will become off after calling this function.
<b>returns</b>	none

### en\_alpha

<b>purpose</b>	Enable alphabet key stroke processing.
<b>syntax</b>	void en_alpha (void);
<b>example call</b>	en_alpha ( );
<b>description</b>	The <i>en_alpha</i> function enables the alphabet key stroke processing. It is disabled upon power on.
<b>returns</b>	none

<b>en_shift</b>	<b>720</b>
-----------------	------------

**purpose** Enable shift key stroke processing.

**syntax** void en\_shift (void);

**example call** en\_shift ( );

**description** The *en\_shift* function enables the shift key stroke processing. It is disabled upon power on.

**returns** none

<b>get_alpha_enable_state</b>
-------------------------------

**purpose** Get the status of the alphabet key stroke processing.

**syntax** int get\_alpha\_enable\_state (void);

**example call** state = get\_alpha\_enable\_state ( );

**description** This routine gets the current status, enable/disable, of the alphabet key stroke processing. The default is enabled.

**returns** 1, if the alphabet key stroke processing is enabled.  
0, if disabled.

<b>get_alpha_lock_state</b>
-----------------------------

**purpose** Get alpha lock state information.

**syntax** int get\_alpha\_lock\_state (void);

**example call** state = get\_alpha\_lock\_state( );

**description** This function returns an integer indicates the alpha lock status. The default is unlocked.

**returns** 1, if alpha key is locked.  
0, if alpha key is not locked.

<b>getchar</b>
----------------

**purpose** Get one key stroke from the keyboard buffer.

**syntax** char getchar (void);

**example call** c = getchar( );  
if (c >0 ) printf("Key %d pressed", c);  
else printf("No key pressed");

**description** The *getchar* function reads one key stroke from the keyboard buffer and then removes the key stroke from the keyboard buffer.

**returns** The *getchar* function returns the key stroke read from the keyboard buffer. If the keyboard buffer is empty, a null character (0x00) is returned. The keystroke returned is the ASCII code of the key being pressed.

<b>get_shift_enable_state</b>	<b>720</b>
-------------------------------	------------

**purpose** Get the status of the shift key stroke processing.

**syntax** int get\_shift\_enable\_state (void);

**example call** `state = get_shift_enable_state ( );`

**description** This routine gets the current status, enable/disable, of the shift key stroke processing. The default is enabled.

**returns** 1, if the shift key stroke processing is enabled.  
0, if disabled.

<b>get_shift_lock_state</b>	<b>720</b>
-----------------------------	------------

**purpose** Get shift lock state information.

**syntax** `int get_shift_lock_state (void);`

**example call** `state = get_shift_lock_state( );`

**description** This function returns an integer indicates the shift lock status. The default is unlocked.

**returns** 1, if shift key is locked.  
0, if shift key is not locked.

<b>GetKeyClick</b>	
--------------------	--

**purpose** Get current key click status

**syntax** `int GetKeyClick(void);`

**example call** `state = GetKeyClick( );`

**description** The function returns an integer indicates the key click status. The default is enabled.

**returns** 1, if key click sound is enabled.  
0, if key click sound is disabled.

<b>kbhit</b>	
--------------	--

**purpose** Check whether the keyboard buffer is empty.

**syntax** `int kbhit (void);`

**example call** `for ( ;!kbhit( ); ); /* wait till key pressed */`

**description** The *kbhit* function checks if there is any character waiting to be read from the keyboard buffer.

**returns** If the keyboard buffer is empty, the *kbhit* function returns an integer value of 0, 1 if not.

<b>peek_kb</b>	
----------------	--

**purpose** Get multiple key combination from the keypad.

**syntax** `unsigned long peek_kb (void);`

**example call** `unsigned long keycode;  
keycode = peek_kb( );  
printf("Keys %ld pressed", keycode);`

**description** The *peek\_kb* function disables the background scanning routines and directly scans the keypad. An unsigned long integer is returned to show up to 4 keys that are pressed at the same time. These scan codes are stored in ascending order (higher byte with smaller scan codes). This is

used to get the special power-on code for diagnostic and/ or special function and should not be used for normal operation.

**returns** An unsigned long integer is returned and each byte represents a scan code. That is, up to 4 keys can be read simultaneously.

<b>set_alpha_lock</b>
-----------------------

**purpose** Set alpha lock state.  
**syntax** void set\_alpha\_lock (int state);  
int state; /\* alpha lock state to be set , 1/0 to turn on/off \*/  
**example call** set\_alpha\_lock (1); /\* on alpha lock \*/  
**description** This routine turns on or off the alpha lock.  
**returns** none

<b>set_shift_lock</b>	<b>720</b>
-----------------------	------------

**purpose** Set shift lock state.  
**syntax** void set\_shift\_lock (int state);  
int state; /\* shift lock state to be set , 1/0 to turn on/off \*/  
**example call** set\_shift\_lock (1); /\* on shift lock \*/  
**description** This routine turns on or off the shift lock.  
**returns** none

<b>SetKeyClick</b>
--------------------

**purpose** to Enable/Disable the key click sound  
**syntax** void SetKeyClick(int status);  
int state; /\* key click status to be set,1/0 to trun on/off \*/  
**example call** SetKeyClick(1); /\* Enable key click sound \*/  
**description** The routine truns on or off the key click sound  
**returns** none

<b>shift_arrow</b>	<b>720</b>
--------------------	------------

**purpose** To enable key combination of the shift and arrow keys.  
**syntax** void shift\_arrow (int state);  
int state; /\* 1/0 to enable/disable \*/  
**example call** shift\_arrow (1); /\* enable shift+arrow key combination\*/  
**description** This routine can be used to enable the shift+arrow key combination, i.e., if user hits the arrow key when the shift key state is on, it will generate KEY\_SLEFT, instead of KEY\_LEFT, etc.  
**returns** none

<b>purpose</b>	To enable key combination of the shift and func keys.
<b>syntax</b>	<pre>int shift_func(int state); int state;          /* 1/0 to enable/disable */</pre>
<b>example call</b>	<pre>shift_func(1);      /* enable shift+func key combination*/</pre>
<b>description</b>	This routine can be used to enable the shift+func key combination, i.e., if user hits the func key when the shift key state is on, it will generate other function, instead of func1, etc.
<b>returns</b>	none

**TriggerStatus**

<b>purpose</b>	To check if the scan key has been pressed.
<b>syntax</b>	<pre>int TriggerStatus (void);</pre>
<b>example call</b>	<pre>if ( TriggerStatus( ) )     printf ("Scan key is pressed");</pre>
<b>description</b>	This function is used to check if the scan key has been pressed or not.
<b>returns</b>	If scan key is pressed, it returns 1, else it returns 0.



## 2.9.2 Special Font Files

Besides the standard font, 711/720 can display special characters such as the foreign language characters providing that those font files have been downloaded to 711/720. CipherLab provides users three special font files to display Japanese, Simplified Chinese, and Traditional Chinese characters. And also the specific library needs to be included if the related functions are called in user's C program.

Font files:

- Font-jp.shx : Japanese kanji Font File
- Font-sc.shx : Simplified Chinese Font File
- Font-tc.shx : Traditional Chinese Font File

Libraries for special fonts:

- 711/720jplib.lib : including *jpprintf*, *jpputchar*, and *jpputs* functions
- 711/720sclib.lib : including *scprintf*, *scputchar*, and *scputs* functions
- 711/720tclib.lib : including *tcprintf*, *tcputchar*, and *tcputs* functions

### clr\_eol

<b>purpose</b>	Clear from where the cursor is to the end of the line.
<b>syntax</b>	void clr_eol (void);
<b>example call</b>	clr_eol( );
<b>description</b>	The <i>clr_eol</i> function clears from where the cursor is to the end of the line, and then moves the cursor to the original place.
<b>returns</b>	none

### clr\_rect

<b>purpose</b>	Clear a rectangular area on the LCD display.
<b>syntax</b>	void clr_rect (int left, int top, int width, int height); int left; /* x coordinate of the upper left corner of the rectangle */ int top; /* y coordinate of the upper left corner of the rectangle */ int width; /* the width in dots of the rectangle to be cleared */ int height; /* the height in dots of the rectangle to be cleared */
<b>example call</b>	clr_rect (12, 8, 40, 8 );
<b>description</b>	The <i>clr_rect</i> function clears an rectangular area on the LCD display whose top left position and size are specified by <i>left</i> , <i>top</i> , <i>width</i> , and <i>height</i> . The cursor position is not affected after the operation.
<b>returns</b>	none

### clr\_scr

<b>purpose</b>	Clear LCD display.
<b>syntax</b>	void clr_scr (void);
<b>example call</b>	clr_scr( );
<b>description</b>	The <i>clr_scr</i> function clears the LCD display and places the cursor at the first column of the first line, that is (0,0) as expressed with the coordinate system.

**returns** none

### DecContrast

**purpose** Decrease the LCD contrast

**syntax** void DecContrast (void);

**example call** DecContrast();

**description** The *DecContrast* function will decrease the LCD contrast by one level whenever it is being called. However, the lowest contrast is 0.

**returns** none.

**See also** IncContrast, SetContrast.

### fill\_rect

**purpose** Fill a rectangular area on the LCD display.

**syntax** void fill\_rect (int left, int top, int width, int height);  
int left; /\* x coordinate of the upper left corner of the rectangle \*/  
int top; /\* y coordinate of the upper left corner of the rectangle \*/  
int width; /\* the width in dots of the rectangle to be filled \*/  
int height; /\* the height in dots of the rectangle to be filled \*/

**example call** fill\_rect (12, 8, 40, 8);

**description** The *fill\_rect* function fills a rectangular area on the LCD display whose top left position and size are specified by *left*, *top*, *width*, and *height*. The cursor position is not affected after the operation.

**returns** none

### GetCursor

**purpose** Get current cursor status.

**syntax** int GetCursor (void);

**example call** if (GetCursor( ) == 0) puts ("Cursor Off");

**description** The *GetCursor* function check if the cursor is visible or not.

**returns** The *GetCursor* function returns an integer of 1 if the cursor is visible (turned on), 0 if not.

### GetFont

**purpose** Get current font information.

**syntax** int GetFont (void);

**example call** if (GetFont( ) == FONT8X16) puts ("Font : 8X16");

**description** The *GetFont* function returns the information about the current font type.

**returns** The return value depends on the current font being used.  
FONT6X8 : if 6X8 font is used  
FONT8X16 : if 8X16 font is used

## get\_image

<b>purpose</b>	Read the bitmap pattern of a rectangular area on the LCD display.
<b>syntax</b>	<pre>void get_image(int left, int top, int width, int height, unsigned char *pat); int left; /* x coordinate of the upper left corner of the rectangle */ int top; /* y coordinate of the upper left corner of the rectangle */ int width; /* the width in dots of the rectangle */ int height; /* the height in dots of the rectangle */ unsigned char *pat; /* the buffer where bitmap data will be copied to */</pre>
<b>example call</b>	<pre>get_image(12, 32, 60, 16, buf);</pre>
<b>description</b>	The <i>get_image</i> function copies the bitmap pattern of a rectangular area on the LCD display whose top left position and size are specified by <i>left</i> , <i>top</i> , <i>width</i> , and <i>height</i> to the buffer specified by <i>pat</i> . The cursor position is not affected after the operation.
<b>returns</b>	none

## GetVideoMode

<b>purpose</b>	Get current display mode information.
<b>syntax</b>	<pre>int GetVideoMode (void);</pre>
<b>example call</b>	<pre>if (GetVideoMode( ) == VIDEO_NORMAL) puts("Normal Mode");</pre>
<b>description</b>	The <i>GetVideoMode</i> function returns the information about the display mode.
<b>returns</b>	The return value depends on the current display mode being used. VIDEO_NORMAL : if normal mode is selected VIDEO_REVERSE : if reverse mode is selected

## gotoxy

<b>purpose</b>	Move cursor to new position.
<b>syntax</b>	<pre>int gotoxy (int x_position, int y_position); int x_position; /* x coordinate of the new cursor position desired */ int y_position; /* y coordinate of the new cursor position desired */</pre>
<b>example call</b>	<pre>gotoxy(10,0); /* move to the 11th column of the first line */</pre>
<b>description</b>	The <i>gotoxy</i> function moves the cursor to a new position whose coordinate is specified in the argument <i>x_position</i> and <i>y_position</i> .
<b>returns</b>	Normally the <i>gotoxy</i> function will return an integer value of 1 when operation completes. In case of LCD fault, 0 is returned to indicate error.

## IncContrast

<b>purpose</b>	Increase the LCD contrast
<b>syntax</b>	<pre>void IncContrast (void);</pre>
<b>example call</b>	<pre>IncContrast();</pre>
<b>description</b>	The <i>IncContrast</i> function will increase the LCD contrast by one level whenever it is being called. However, the highest contrast level is 7.
<b>returns</b>	none.
<b>See also</b>	IncContrast, SetContrast.

## lcd\_backlit

<b>purpose</b>	Set LCD backlight
<b>syntax</b>	<pre>void lcd_backlit (int state);  int state;          /* LCD backlight state 0 / 1 (off / on) */</pre>
<b>example call</b>	<pre>lcd_backlit (1); /* turn on LCD backlight */</pre>
<b>description</b>	The <i>lcd_backlit</i> turns the LCD backlight on or off depending on the value of <i>state</i> . The backlight will be on if <i>state</i> is 1, off if 0. The system global variable <b>BKLIT_TIMEOUT</b> can be used to specify the backlight duration in unit of second. But if this value is set to zero, the backlight will be on until the backlight state is set to off or user turn off it manually.
<b>returns</b>	none.

## printf

<b>purpose</b>	Write character strings and values of C variables in a specified format to the LCD display.
<b>syntax</b>	<pre>int printf (char* format, var); char* format; /* character string that describes the format to be used variable number of arguments whose values are being printed on the LCD display */</pre>

**example call**

```
printf("ID : %s", id_buffer);
```

**description** The *printf* function accepts a variable number of arguments and prints them to the LCD display. The value of each argument is formatted according to the codes embedded in the format specification *format*.

To print values of C variables, a format specification must be embedded in *format* for each variable to be printed. The format specification for each variable has the following form :

`%[flags][width].[precision][size][type]`

Field	Explanation
% (required)	Indicates the beginning of a format specification. Use %% to print a percentage sign.
flags (optional)	One or more of the '-', '+', '#' characters or a blank space specifies justification, and the appearance of plus / minus signs in the values printed (see table below).
width (optional)	A number that indicates how many characters, at a minimum, must be used to print the value
precision (optional)	A number that specifies how many characters, at maximum, can be used to print the value. When printing integer variables, this is the minimum number of digits used.
size (optional)	A character that modifies the type field which comes next. One of the characters 'h', 'l', 'L' can appear in this field to differentiate between short and long integers. 'h' is for short integers, and 'l' or 'L' for long integers.
type (required)	A letter that indicates the type of variable being printed (see table below)

### Flags Meaning

- Left justify output value. Default is right justification.
- + If the output value is a numerical one, print a '+' or '-'

character according to the sign of the value. A '-' character is always printed for a negative value no matter this flag is specified or not.

- blank Positive numerical values are prefixed with blank spaces. This flag is ignored if the + flag also appears.
- # When used in printing variables of type o, x, or X, none zero output values are prefixed with 0, 0x, or 0X, respectively.

**Type Expected Input**

- c Single character.
- d Signed decimal integer.
- i Signed decimal integer.
- o Octal digits without sign.
- u Unsigned decimal integer.
- x Hexadecimal digits using lower case letter.
- X Hexadecimal digits using upper case letter.
- s A null terminated character string.

The *jpprint*, *scprintf*, and *tcprintf* functions are special *printf* functions to display a string that consists of the Japanese, simplified Chinese and/ or traditional Chinese characters and the other variables.

**returns** The *printf* function returns the number characters sent to the LCD display

**putchar**

- purpose** Display a character on the LCD display.
- syntax** int putchar (char c);  
char c; character sent to the LCD display
- example call** putchar('A');
- description** The *putchar* function sends the character specified in the argument *c* to the LCD display at the current cursor position and moves the cursor accordingly.  
The *jpputchar*, *scputchar*, and *tcputchar* functions are special *putchar* functions to display a single Japanese, simplified Chinese and/ or traditional Chinese character.
- returns** none

**puts**

- purpose** Display a string on the LCD display.
- syntax** char puts (char\* string);  
char\* string; /\* string to be displayed \*/
- example call** puts ("Password : ");
- description** The *puts* function sends a character string whose address is specified in the argument *string* to the LCD display starting from the current cursor position. The cursor is moved accordingly as each character of *string* is sent to the LCD display. The operation continues until a terminating null character is encountered.  
The *jpputs*, *scputs*, and *tcputs* functions are special *puts* functions to display a string which consists of the Japanese, simplified Chinese and/ or traditional Chinese characters.

**returns** The *puts* function returns the number characters sent to the LCD display

### SetContrast

**purpose** To set contrast level for the LCD

**syntax** void SetContrast (int level);

**example call** SetContrast (4);

**description** The *SetContrast* function is used to set the contrast level for LCD. The valid level is ranging from 0 to 7. The higher value, the higher contrast.

**returns** none.

**See also** IncContrast, DecContrast.

### SetCursor

**purpose** Turn on or off the cursor of the LCD display.

**syntax** void SetCursor (int status);  
int status; /\* integer representing cursor status to be set \*/

**example call** SetCursor (0); /\* invisible the cursor \*/

**description** The *SetCursor* function displays or hides the cursor of the LCD display according to the value of *status* specified. If *status* equals 1, the cursor will be turned on to show the current cursor position. If *status* equals 0, the cursor will be invisible.

**returns** none.

### SetFont

**purpose** Select the font to be used afterwards.

**syntax** int SetFont (int font);  
int font; /\* integer representing font to be used \*/

**example call** SetFont (FONT\_8X16);

**description** The *SetFont* function selects the font specified by *font* to be used following this call. The valid values are as follow

FONT\_6X8: 6x8 font  
FONT\_8X16: 8x16 font

**returns** none

### SetVideoMode

**purpose** Select video mode for the display.

**syntax** void SetVideoMode (int mode);  
int mode; /\* integer representing video mode to be set \*/

**example call** SetVideoMode(VIDEO\_REVERSE); /\* select reverse video mode \*/

**description** The *SetVideoMode* function set the display mode for the following LCD operation. The available modes are VIDEO\_NORMAL and VIDEO\_REVERSE.

**returns** none

## show\_image

<b>purpose</b>	Put a rectangular bitmap to the LCD display.
<b>Syntax</b>	<pre>void show_image (int left, int top, int width, int height, unsigned char *pat); int left;      /* x coordinate of the upper left corner of the rectangle */ int top;       /* y coordinate of the upper left corner of the rectangle */ int width;     /* the width in dots of the rectangle */ int height;    /* the height in dots of the rectangle */ unsigned char *pat; /* the buffer that hold the bitmap to be displayed */</pre>
<b>example call</b>	<code>show_image (35, 5, 52, 24, CipherLab_logo[]);</code>
<b>description</b>	The <i>showet_image</i> function displays a rectangular bitmap specified by <i>pat</i> to the LCD display. The rectangular's top left position and size are specified by <i>left</i> , <i>top</i> , <i>width</i> , and <i>height</i> . The cursor position is not affected after the operation.
<b>returns</b>	none

## wherex

<b>purpose</b>	Get x-coordinate of the cursor location.
<b>syntax</b>	<code>int wherex (void);</code>
<b>example call</b>	<code>x_position = wherex ( );</code>
<b>description</b>	The <i>wherex</i> function determines the current x-coordinate location of the cursor.
<b>returns</b>	The <i>wherex</i> function returns the x-coordinate of the cursor location.

## wherexy

<b>purpose</b>	Get x-coordinate and y-coordinate of the cursor location
<b>syntax</b>	<pre>int wherexy (int* column, int* row); int* column; /* pointer to integer where x-coordinate is stored */ int* row;    /* pointer to integer where y-coordinate is stored */</pre>
<b>example call</b>	<code>wherexy (&amp;x_position, &amp;y_position);</code>
<b>description</b>	The <i>wherexy</i> function copies the value of x-coordinate and y-coordinate of the cursor location to the variables whose address is specified in the arguments <i>column</i> and <i>row</i> .
<b>returns</b>	none

## wherey

<b>purpose</b>	Get y-coordinate of the cursor location.
<b>syntax</b>	<code>int wherey (void);</code>
<b>example call</b>	<code>y_position = wherey( );</code>
<b>description</b>	The <i>wherey</i> function determines the current y-coordinate location of the cursor.
<b>returns</b>	The <i>wherey</i> function returns the y-coordinate of the cursor location.

## 2.10 Power

This section describes the power management functions for 711/720. The **get\_vmain**, and **get\_vbackup** function is used to monitor the voltage level of the main power supply and the backup battery.

### get\_vmain

<b>purpose</b>	Get voltage level of the main power supply.
<b>syntax</b>	unsigned get_vmain (void);
<b>example call</b>	if ( get_vmain( ) < 2000) puts("Lose Power");
<b>description</b>	This function reads the voltage level of the main power in units of mV.
<b>returns</b>	The voltage level of the main power in units of mV (mili-volt).

### get\_vbackup

<b>purpose</b>	Get voltage level of the backup battery.
<b>syntax</b>	unsigned get_vbackup (void);
<b>example call</b>	bat1 = get_vbackup( );
<b>description</b>	This function reads the voltage level of the backup battery in units of mV.
<b>returns</b>	The voltage level of the backup battery in units of mV (mili-volt).

## 2.11 Communication Ports

There are totally two communication ports on 711/720, namely COM1, and COM2. User has to set up the communication type for the COM ports before they can be used. The COM1 could be used for docking and/ or direct RS232 port. COM2 could be used for either direct RS232, docking, IR, IrDA, or RF port. Besides the data signals (transmit & receive), 2 handshake signals (RTS & CTS) are also provided for data flow control. Features provided are described in detail below,

### 2.11.1 Parameters

- Baud rate : One out of 8 baud rates can be selected (115200, 76800, 57600, 38400, 19200, 9600, 4800, 2400)
- Data Bits : 7 or 8
- Parity : Even, Odd or none
- Stop bit : 1

### 2.11.2 Receive Buffer

A 256 bytes FIFO buffer is allocated for each port. The data successfully received is stored into this buffer sequentially (if any error such as framing, parity error and so on occurs, the data is simply discarded). However if the buffer is full, the data followed will be discarded and an overrun flag is set to indicate this error.

### 2.11.3 Transmit Buffer

The system does not allocate any transmit buffer, it simply records the pointer to the string to be sent. The transmission stops when a null (0x00) character was encountered. The application program must allocate its own transmit buffer and not to modify it during transmission.

### 2.11.4 Flow Control

To avoid data loss, 3 kinds of flow control are supported and is done by background routines.

- 1) None : no flow control is performed
- 2) CTS : RTS and CTS signals are used for flow control.
  - Transmission : The transmission is allowed only when CTS signal is at the active level (mark). If the CTS is dropped and later become active again, the transmission is automatically resumed by background routines. However, due to the UART design (on-chip temporary transmission buffer), up to 2 characters might be sent after the CTS was dropped.
  - Receive : The RTS signal is used to indicate that the receiving buffer is or is going to be full and instruct the transmitting side to halt transmission. If there are less than 5 character spaces available in the receiving buffer, the RTS is dropped. Then the RTS is activated again when there are no less than 10 character spaces available in the receiving buffer. If there are sufficient spaces in the buffer, the received data is stored even when RTS is dropped.
- 3) XON/XOFF : instead of RTS/CTS signals, 2 special characters are used for flow control. That is, XON (hex 11) and XOFF (hex 13). XON is used to enable transmission while XOFF to disable transmission.
  - Transmission : when the port is opened, the transmission is enabled. Then every character received is examined to see if it is a normal data or flow control codes. If XOFF is received, transmission is halted. It is resumed later when a XON is received. Just like RTS/CTS control, up to 2 characters might be sent after the XOFF was received.

- Receive : The received characters are examined to see if it is normal data (stored into receive buffer) or flow control codes (set/reset transmission flag but not stored). If there are less than 5 character spaces available in the receiving buffer, the XOFF is sent. Then the XON is sent when there are no less than 10 character spaces available in the receiving buffer. If there are sufficient spaces in the buffer, the received data is stored even when in XOFF state. **Note** that if receiving/transmission are concurrently in operation, XON/XOFF control codes might be inserted into normal transmit data string. In using this method, make sure the respective side features the same control methodology or dead lock might happen.

Regardless of the flow control methodology selected, the RTS is activated when the port is *opened* and dropped when the port is *closed* (the power on default status).

#### clear\_com

<b>purpose</b>	Clear receive buffer
<b>syntax</b>	void clear_com (int port); int port; /* port to be opened, from 1 to 2 */
<b>example call</b>	clear_com(1); /* clear COM1 receive buffer */
<b>description</b>	This routine is used to clear all data stored in the receive buffer. This can be used to avoid mis-interpretation when overrun or other error occurred.
<b>returns</b>	none

#### close\_com

<b>purpose</b>	To close specified communication port
<b>syntax</b>	void close_com (int port); int port; /* port to be opened, from 1 to 2 */
<b>example call</b>	close_com(1); /* close com1 */
<b>description</b>	The <i>close_com</i> disables the communication port specified.
<b>returns</b>	none

#### com\_cts

<b>purpose</b>	Get CTS level
<b>syntax</b>	int com_cts (int port); int port; /* port to be opened, from 1 to 2 */
<b>example call</b>	if (com_cts(1) == 0) printf ("COM1 CTS is space"); else printf("COM1 CTS is mark");
<b>description</b>	This routine is used to check current CTS level.
<b>returns</b>	1, if CTS is in mark state 0, if CTS is in space state

#### com\_eot

<b>purpose</b>	To see if any COM port transmission in process (End Of Transmission)
<b>syntax</b>	int com_eot(int port); int port; /* port to be opened, from 1 to 2 */

**example call** `while (com_eot(1) != 0x00); /* wait till prior transmission completed */  
write_com (1, "NEXT STRING");`

**description** This routine is used to check if prior transmission is still in process or not.

**returns** 0, prior transmission still in course  
1, transmission completed

#### com\_overn

**purpose** See if overrun error occurred

**syntax** `int com_overn (int port);  
int port; /* port to be opened, from 1 to 2 */`

**example call** `if (overn(1) > 0) clear_com(1);  
/* if overrun, data stored in the buffer is not complete, clear them */`

**description** This routine is used to see if overrun met. The overrun flag is automatically cleared after examined.

**returns** 1, overrun error met  
0, OK

#### com\_rts

**purpose** Set RTS signal

**syntax** `void com_rts (int port, int i);  
int port; /* port to be opened, from 1 to 2 */  
int i; /* RTS state, 1/0, mark/space */`

**example call** `com_rts (1, 1); /* set COM1 RTS to mark */`

**description** This routine is used to control the RTS signal. It works even when the CTS flow control is selected. However, RTS might be changed by the background routine according to receiving buffer status. It is strongly recommended not to use this routine if CTS control is utilized.

**returns** none

#### nwrite\_com

**purpose** Send a specific number of characters out through RS232 port

**syntax** `void nwrite_com(int port, char *s, int count);  
int port; /* port to be opened, from 1 to 2 */  
char *s; /* string to be sent */  
int count; /* number of character to be sent */`

**example call** `char s[] = { "Hello\n" };  
nwrite_com(1, s, 2); /* send two characters "He" through COM1 */`

**description** This routine is used to send a specific number of characters specified by *count* through RS232 ports. If any prior transmission is still in process, it is terminated then the current transmission resumes. The character string is transmitted one by one until the specified number of character is sent.

**returns** none

## open\_com

**purpose** Initialize and enable specified RS232 port

**syntax** void open\_com(int port, int parameter);  
int port; /\* port to be opened, from 1 to 2 \*/  
int parameter; /\* port parameters as below \*/

D0-D2	baud rate	0 to 7 = 115200/76800/57600/ 38400/19200/9600/4800/2400
D3	data bits	0 : 7bits 1 : 8 bits
D4	Parity enable	0 : disable 1 : enable
D5	even/odd	0 : odd 1 : even
D6	flow control	0 : disable 1 : enable
D7	flow control method	0 : CTS, 1 : XON/XOFF

**example call** open\_com(1, 0x0b);  
/\* open com1 to 38400, 8 data bits, no parity and no handshake \*/

**description** The open\_com function initializes the specified RS-232 port. It clears the receive buffer, stops any data transmission under going, reset the status of the port, and set the RS-232 specification according to parameters set.

**returns** none

## read\_com

**purpose** Read 1 byte from the RS232 receive buffer

**syntax** int read\_com (int port, char \*c);  
int port; /\* port to be opened, from 1 to 2 \*/  
char \*c; /\* pointer to character returned \*/

**example call** char c;  
i=read\_com(1, c);  
if (i) printf\_us("char %c received from COM1", \*c);

**description** This routine is used to read one byte from the receive buffer and then remove it from the buffer. However, if the buffer is empty, no action is taken and 0 is returned.

**returns** 1, available or 0 if buffer is empty

## SetCommType

**purpose** Set the communication type of the port specified.

**syntax** int SetCommType (int port, int type);  
int port; /\* port to be set, can be either 1 or 2 \*/  
int type; /\* communication types, following types are available

Port 1:

- 0 direct RS-232
- 1 docking (via communication cradle)

Port 2:

- 2 high speed IR (via IR transceiver)
- 3 standard IrDA communication

#### 4 RF communication \*/

**example call** SetCommType (2, 2); /\* set COM2 to high speed communication \*/

**description** This routine is used to set the communication types for the COM ports. Before opening the COM port, please call this function to assign communication type.

**returns** 1 for valid setting (successful), 0 for invalid setting (failed).

#### **write\_com**

**purpose** Send a string out through RS232 port

**syntax** void write\_com (int port, char \*s);  
int port; /\* port to be opened, from 1 to 2 \*/  
char \*s; /\* string to be sent \*/

**example call** char s[] = { "Hello\n" };  
write\_com (1, s); /\* send String "Hello\n" through COM1 \*/

**description** This routine is used to send a string through RS232 ports. If any prior transmission is still in process, it is terminated then the current transmission resumes. The character string is transmitted one by one until a NULL character is met. A null string can be used to terminate prior transmission.

**returns** none

## 2.12 Memory

Flash and SRAM manipulation routines are described in this section.

### EraseSector

<b>purpose</b>	To erase a whole sector of flash memory.
<b>syntax</b>	int EraseSector (void* sector_start_addr);
<b>example call</b>	EraseSector (0xf60000);
<b>description</b>	Before calling <i>WriteFlash</i> to write data into flash memory, the flash memory should be erased first by calling this function.
<b>returns</b>	Number of bytes that has been erased.

### FlashSize

<b>purpose</b>	To check the size of flash memory
<b>syntax</b>	int FlashSize (void);
<b>example call</b>	FlashSize ( );
<b>description</b>	The <i>FlashSize</i> function allows you to check the flash memory that available for the user program.
<b>returns</b>	Flash memory size in K bytes.

### free\_memory

<b>purpose</b>	Get free memory size information.
<b>syntax</b>	long free_memory (void);
<b>example call</b>	available_memory = free_memory ( );
<b>description</b>	The <i>free_memory</i> function gets the information of the amount of free (unused) memory of the file space.
<b>returns</b>	The <i>free_memory</i> function returns a long integer indicating the amount of free memory in bytes.

### GetFlashID

<b>purpose</b>	Get flash memory size information.
<b>syntax</b>	int GetFlashID (void);
<b>example call</b>	printf ("Flash ID : %d", GetFlashID( ));
<b>description</b>	The <i>GetFlashID</i> function gets the information of the size of the flash memory.
<b>returns</b>	181 for 1M flash memory 62 for 512K flash memory

## init\_free\_memory

<b>purpose</b>	Initialize file space.
<b>syntax</b>	void init_free_memory (void);
<b>example call</b>	init_free_memory ( );
<b>description</b>	The <i>init_free_memory</i> function will first try to identify how many SRAMs are installed, and then initialize the contents of the file space (total SRAM installed excludes memory of system space and user space). The original contents of the file space will be wiped out after this function is called. Whenever the amount of the SRAM installed is changed, this function must be called to recognize the changes.
<b>returns</b>	This function has no return values.

## RamSize

<b>purpose</b>	To check the size of data memory (SRAM).
<b>syntax</b>	int RamSize (void);
<b>example call</b>	RamSize( );
<b>description</b>	The <i>RamSize</i> function allows you to check the SRAM size that could be used for storing data.
<b>returns</b>	RAM size in K bytes

## WriteFlash

<b>purpose</b>	To write data to the flash memory.
<b>syntax</b>	int WriteFlash (void *target_addr, void *source_addr, unsigned long size);
<b>example call</b>	char szData[100]; EraseSector (0xf60000); WriteFlash (0xf60000, szData, 100);
<b>description</b>	The flash memory can also be used to store data if it's not fully used by user program. The possible available flash memory is 32 Kbytes and start from 0xF60000.
<b>returns</b>	Number of bytes that has been written to flash.



## fclose

<b>purpose</b>	To close a file opened earlier for buffered input and output using <i>fopen</i> .
<b>syntax</b>	<pre>int fclose (FILE *file_pointer); FILE *file_pointer;          /* pointer to file to be closed */</pre>
<b>example</b>	<pre>FILE *fp; fp = fopen ("A:\\file1", "r+");  /* opened for read &amp; write */ /* processing */ if ( fclose (fp) )     printf ("file close error);</pre>
<b>description</b>	The <i>fclose</i> function closes the file specified by the argument <i>file_pointer</i> . If the file is open for writing, the contents of the buffer associated with the file are flushed before the file is closed.
<b>returns</b>	This function returns zero if the file was successfully closed, or EOF if any errors were detected. The contents of the buffer associated with the file will be flushed before the file is closed
<b>see also</b>	<i>fopen</i>

## feof

<b>purpose</b>	To determine whether the end of a file has been reached.
<b>syntax</b>	<pre>int feof (FILE *file_pointer); FILE *file_pointer;  /* pointer to the FILE data structure associated with the file whose status is being checked */</pre>
<b>example</b>	<pre>FILE *fp; int c; fp = fopen ("A:\\file1", "r+");  /* opened for read &amp; write */ while ( !feof (fp) ) {     c = fgetc (fp); }</pre>
<b>returns</b>	This function returns a non-zero value if the end of the file is reached.

## ferror

<b>purpose</b>	To determine if an error has occurred during a previous read or write operation on a file.
<b>syntax</b>	<pre>int ferror (FILE *file_pointer); FILE *file_pointer;  /* pointer to the FILE data structure associated with the file whose status is being checked */</pre>
<b>example</b>	<pre>FILE *fp; int c; fp = fopen ("A:\\file1", "r+");  /* opened for read &amp; write */ while ( !feof (fp) ) {     c = fgetc (fp);     if (ferror (fp)) {</pre>

```

        printf ("Error detected\n");
        clearerr (fp);
        printf ("Error cleared\n");
    }
}

```

**returns** This function returns a non-zero value if an error has occurred during a read or a write operation. Otherwise, it returns a 0.

**see also** clearerr

### fflush

**purpose** To flush the output buffer associated with a file opened for buffered I/O. This will cause any remaining data in the output buffer written to the file.

**syntax** int fflush (FILE \*file\_pointer);  
 FILE \*file\_pointer; /\* pointer to the FILE data structure associated with the file whose buffer is being flushed \*/

**example** FILE \*fp;  
 if (fflush (fp)) {  
     /\* file flush error \*/  
 }

**returns** If the buffer is successfully flushed, *fflush* returns a 0. In case of an error, the return value is the constant EOF defined in *stdio.h*.

### fgetc

**purpose** To read a single character from a file opened for buffered input.

**syntax** int fgetc (FILE \*file\_pointer);  
 FILE \*file\_pointer; /\* pointer to the FILE data structure associated with the file from which a character is to be read \*/

**description** The *fgetc* function reads a character from the current position of the file pointed to by the file\_pointer and then increments this position. The character is returned as an integer.

**example** FILE \*fp;  
 char buffer[81];  
 int i, c;  
 if ((fp = fopen ("A:\\file1", "r")) == NULL)  
 {  
     printf ("fopen failed.\n");  
     exit (0);  
 }  
 c = fgetc (fp);  
 for (i=0; (i < 80) && (feof (fp) == 0) && (c != '\n'); i++)  
 {  
     buffer [i] = c;  
     c = fgetc (fp);  
 }

```

    }
    buffer [i] = '\0';
    printf ("First line of A:file1 : %s\n", buffer);

```

**returns** If there are no errors, *fgetc* returns the character read. Otherwiae, it returns the constant EOF. Call *error* and *feof* to determine if there was an error or the file simply reached its end.

**see also** *fgets*, *fputc*, *fputs*

## fgetpos

**purpose** To get and save the current read or write position of a file.

**syntax**

```

int fgetpos (FILE *file_pointer, unsigned long *position);
FILE *file_pointer; /* pointer to the FILE data structure associated with
                    the file whose current position is requested */
unsigned long *position; /* pointer to location where file's current
                        position is returned */

```

**description** The *fgetpos( )* function fills *position* with a value representing the current position of the file pointed to by the file pointer. This is usually the byte number from the beginning of the file. In the case of a file open in text mode this may not be the same as the actual number of bytes you have read from the file. The *position* returned by *fgetpos( )* should be used as an argument to *fsetpos( )* to reposition a file to a former location..

**example**

```

FILE *fp;
int c;
unsigned long position;
if ((fp = fopen ("A:\\file1", "r")) == NULL)
{
    printf ("fopen failed.\n");
    exit (0);
}
c = fgetc (fp);
if (fgetpos (fp, &position) != 0)
    printf ("fgetpos failed");

```

**returns** The *fgetpos* returns a zero when successful. In case of error, the return value is nonzero and the global variable *errno* is set to the constant EBADF if the *file\_pointer* does not point to a file or if it points to an inaccessible file.

**see also** *fsetpos*

## fgets

**purpose** To read a line from a file opened for buffered input. The line is read until a newline (\n) character is encountered or until the number of characters reaches the specified maximum.

**syntax**

```

char *fgets (char *string, int max_char, FILE *file_pointer);
char *string; /* pointer to buffer where characters are stored */

```

```

int max_char;    /* maximum number of characters that can be stored */
FILE *file_pointer; /* pointer to FILE data structure associated with
                    the file from which a line is read */

```

**description** The *fgets( )* function reads at most one less than the number of characters specified by *max\_char* from the file pointed to by *file\_pointer* into the buffer pointed to by *string*. No additional characters are read after the new-line character (which is retained). A null character is written immediately after the last character read into the buffer.

**Example**

```

FILE *fp;
char string[81];
if ((fp = fopen ("A:\\file1", "r")) == NULL)
    {
    printf ("fopen failed.\n");
    exit (0);
    }
while (fgets (string, 80, fp) != NULL)
    printf ("%s\n", string);

```

**returns** If there are no errors, *fgets* returns the argument string. Otherwiae, it returns a NULL. Call *feof* and *ferror* to determine if there was an error or the file simply reached its end.

**see also** fgetc, fputc, fputs

<b>fopen</b>
--------------

**purpose** To open a file for buffered input and output operations.

**syntax**

```

FILE *fopen (const char *filename, const char *mode);
const char *filename;    /* name of file to be opened including
                        drive and directory specification */
const char *mode;    /* type of access permitted */

```

**description** The *fopen()* function opens the file specified in the argument *filename*. The *filename* must include the drive, which is "A:". If the operation fails, a null pointer is returned. The mode string specifies the type of access requested as follows:

- "r" Open for reading in text mode
- "w" Create for writing in text mode
- "a" Append (open/create for writing at EOF)
- "rb" Open for reading in binary mode
- "wb" Create or truncate for writing in binary mode
- "ab" Append in binary mode (open/create for writing at EOF)
- "r+" Open for reading and writing in text mode
- "w+" Truncate or create for reading and writing in text mode
- "a+" Open / create for reading and appending.
- "r+b" Open for reading and writing in binary mode
- "w+b" Truncate or create for reading and writing in binary mode
- "a+b" Open/create for reading and appending in binary mode
- "d" Open directory

**example** FILE \*fp;  
if ((fp = fopen ("A:\\file1", "r+")) == NULL) /\* opened for read & write \*/  
{  
printf ("fopen failed.\n");  
exit (0);  
}

**returns** If the file is opened successfully, *fopen* returns a pointer to the file. Actually, this is a pointer to a structure of type FILE, which is defined in the header file **smc.h**. In case of an error, *fopen* returns a NULL. The value of the global *errno* may contain additional error status. See **smc.h** for the error codes returned in *errno*

**see also** fclose

<b>fputc</b>
--------------

**purpose** To write a single character to a file opened for buffered output.

**syntax** int fputc (int c, FILE \*file\_pointer);  
int c; /\* character to be written \*/  
FILE \*file\_pointer; /\* pointer to the FILE data structure associated with the file to which the character is to be written \*/

**description** The *fputc()* function writes a character given in the argument *c* to the file specified by the *file\_pointer* in the current position and then increments this position after writing the character.

**example** FILE \*fp;  
char buffer[81] = "Testing the function fputc";  
int i;  
if ((fp = fopen ("A:\\file1", "w")) == NULL)  
{  
printf ("fopen failed.\n");  
exit (0);  
}  
for (i=0; (i < 80) && (fputc (buffer[i], fp) != EOF); i++)  
;

**returns** If there are no errors, *fputc* returns the character written. Otherwiae, it returns the constant EOF. Call *error* to determine if there was an error or the integer argument *c* just happened to be equal to EOF.

**see also** fgetc, fgets, fputs

<b>fputs</b>
--------------

**purpose** To write a null-terminated string to a file opened for buffered output.

**syntax** int fputs (char \*string, FILE \*file\_pointer);  
char \*string; /\* null-terminated character string to be output \*/  
FILE \*file\_pointer; /\* pointer to the FILE data structure associated with the file to which the string is output \*/

<b>description</b>	The <i>fputs</i> function writes a string given in the argument <i>string</i> to the file specified by the <i>file_pointer</i> .
<b>Example</b>	<pre>FILE *fp; char string[81] = "Testing the function fputs"; if ((fp = fopen ("A:\\file1", "w")) == NULL)     {         printf ("fopen failed.\n");         exit (0);     } fputs (string, fp);</pre>
<b>returns</b>	If there are no errors, <i>fputs()</i> returns the number of characters written. Otherwise, it returns the constant EOF. Call <i>error()</i> to find out the error.
<b>see also</b>	fgetc, fgets, fputc

<b>fread</b>
--------------

<b>purpose</b>	To read a specified number of data items, each of a given size, from the current position in a file opened for buffered input.
<b>syntax</b>	<pre>int fread (void *buffer, int size, int count, FILE *file_pointer); void *buffer;      /* pointer to memory where fread stores the bytes it                     reads */ int size;          /* size in bytes of each data item */ int count;        /* maximum number of items to be read */ FILE *file_pointer; /* pointer to FILE data structure associated with                     the file from which data items are read */</pre>
<b>description</b>	The <i>fread()</i> function reads <i>count</i> data items, each of <i>size</i> bytes, starting at the current read position of the file specified by <i>file_pointer</i> . After the read is complete, the current position is updated.
<b>Example</b>	<pre>FILE *fp; char buffer[81]; int count; if ((fp = fopen ("A:\\file1", "r")) == NULL)     {         printf ("fopen failed.\n");         exit (0);     } count = fread (buffer, 1, 80, fp); printf ("Read these %d characters:\n %s\n", count, buffer);</pre>
<b>returns</b>	The actual number of items read is returned. Note that the number of items returned will be equal to <i>count</i> unless the EOF is reached or some error occurs.
<b>see also</b>	fwrite

## fseek

<b>purpose</b>	To reposition a file pointer.
<b>syntax</b>	<pre>int fseek (FILE *file_pointer, long offset, int origin); FILE *file_pointer; /* pointer to FILE data structure associated with                     the file whose position is to be set */ long offset;      /* offset of new position (in bytes) from origin */ int origin;       /* file position from which to add offset, there are three                     values available :                     SEEK_SET (1) - Beginning of file                     SEEK_CUR (0) - Current file pointer position                     SEEK_END (-1) - End of file */</pre>
<b>description</b>	The <i>fseek( )</i> function repositions the file specified by <i>file_pointer</i> by <i>offset</i> bytes from <i>origin</i> . If the file is opened in text mode, the <i>offset</i> should be 0 or the value returned by <i>ftell( )</i> . The value in <i>origin</i> should be <i>SEEK_SET</i> for beginning of file, <i>SEEK_CUR</i> for current file pointer position, or <i>SEEK_END</i> for end of file.
<b>Example</b>	<pre>FILE *fp; if (fseek(fp, 30L, SEEK_SET) != 0)     printf ("fseek failed!");</pre>
<b>returns</b>	If successful, <i>fseek</i> returns a zero, otherwise, it returns a nonzero value.
<b>see also</b>	<i>ftell</i>

## fsetpos

<b>purpose</b>	To set the position where reading or writing can take place in a file opened for buffered I/O.
<b>syntax</b>	<pre>int fsetpos (FILE *file_pointer, const unsigned long *pos); FILE *file_pointer; /* pointer to FILE data structure associated with                     the file whose position is to be set */ const unsigned long * pos; /* pointer to location containing new value of                     file position */</pre>
<b>description</b>	The <i>fsetpos( )</i> function sets the file pointer associated with opened file to the new position <i>pos</i> . The new position is the value obtained by a previous call to <i>fgetpos( )</i> on that stream. The reason for the existence of <i>fgetpos</i> and <i>fsetpos</i> (in addition to <i>fseek</i> ) is that if you want to position to a file in text mode, you cannot necessarily find a position by counting the characters you have written out, since text mode translation may change that number. In this case you can only use <i>fgetpos</i> to find a current position and then return there later with <i>fsetpos</i> .
<b>Example</b>	<pre>FILE *fp; unsigned long curpos; char buffer [80]; if (fgetpos (fp, &amp;curpos) != 0)          /* save current position */     printf ("fgetpos failed!"); if (fgets(buffer, 20, fp) == NULL)      /* read 20 characters */     printf ("fgets failed!");</pre>

```

if (fsetpos (fp, &curpos) != 0)          /* reset to previous position */
    printf ("fsetpos failed!");

```

**returns** If successful, *fsetpos* returns a zero, otherwise, it returns a nonzero value with the global variable *errno* set to a nonzero error code.

**see also** *fgetpos*

<b>ftell</b>
--------------

**purpose** To get current file position.

**syntax** long ftell (FILE \*file\_pointer);  
FILE \*file\_pointer; /\* pointer to FILE data structure associated with the file whose current position is to be returned \*/

**description** The *ftell()* function returns the current read and write position of the file specified by argument *file\_pointer*.

**Example** FILE \*fp;  
long curpos;  
if ((curpos = ftell (fp)) == -1L)  
 printf ("ftell failed!");

**returns** If successful, *ftell* returns a long integer containing the number of bytes the current position is offset from the beginning of the file. In case of error, *ftell* returns -1L with the global variable *errno* set to a positive error code.

**see also** *fseek*

<b>fwrite</b>
---------------

**purpose** To write a specified number of data items, each of a given size, from a buffer to the current position in a file opened for buffered output.

**syntax** int fwrite (const void \*buffer, int size, int count, FILE \*file\_pointer);  
const void \*buffer; /\* pointer to buffer from which fwrite will get the bytes it writes \*/  
int size; /\* size in bytes of each data item \*/  
int count; /\* maximum number of items to be written \*/  
FILE \*file\_pointer; /\* pointer to FILE data structure associated with the file from to which data items are to be written \*/

**description** The *fwrite()* function writes *count* data items, each of *size* bytes, to the file specified by the argument *file\_pointer*, starting at the current position. After the write operation is complete, the current position is updated.

**Example** FILE \*fp;  
char buffer[81] = "Testing the fwrite function";  
int count;  
if ((fp = fopen ("A:\\file1", "r")) == NULL)  
{  
 printf ("fopen failed.\n");  
 exit (0);  
}

```

    }
    count = fwrite (buffer, 1, 20, fp);
    printf ("%d characters written to a file", count);

```

**returns** The actual number of items written is returned. Note that the number of items returned will be equal to *count* except an error occurred.

**see also** fread

### fremove

**purpose** To delete a file.

**syntax** int fremove (const char \*filename);  
const char \*filename; /\* the complete pathname of the file to delete \*/

**description** The *fremove( )* function deletes a file specified by *filename*. The complete filename should include the device name.

**Example** if ( fremove ("a:\\subdir\\thisfile.txt") )  
printf ("errno = %dn", errno);

**returns** If successful, *fremove* returns a zero, otherwise, it returns a nonzero value with the global variable *errno* set to a nonzero error code (refer to *smc.h*).

**see also** frename, rmdir

### frename

**purpose** To rename (or move) a file or a subdirectory.

**syntax** int frename (const char \*oldname, const char \*newname);  
const char \*oldname; /\* the complete pathname of an existing file \*/  
const char \*newname; /\* the complete pathname of the target file \*/

**description** The *frename( )* function changes the name of the file *oldname* to *newname*. A complete pathname must be given for both, which must be on the same device (drive). Subdirectories can also be renamed. The *newname* does not need to be in the same directory as *oldname*. The effect in this case is that of moving the file to the new directory (and possibly renaming it during the process).

**Example** if ( frename("a:\\file1.txt", "a:\\file2.txt") )  
printf ("errno = %dn", errno);

**returns** If successful, *frename* returns a zero, otherwise, it returns a nonzero value with the global variable *errno* set to a nonzero error code (refer to *smc.h*).

**see also** fremove, rmdir

### mkdir

**purpose** To create a new directory.

**syntax** int mkdir (const char \* path);  
const char \*path; /\* the complete pathname of the directory to create \*/

**description** The *mkdir( )* function creates a new directory from the given pathname *path*.

**Example**      `if (mkdir ("A:\\thisdir\\thatdir\\newdir") != 0)  
                  printf ("Fail to create a directory");`

**returns**        If successful, *mkdir* returns a zero, otherwise, it returns a nonzero value with the global variable *errno* set to a nonzero error code (refer to *smc.h*).

**see also**        `rmdir`

<b>rmdir</b>
--------------

**purpose**        To remove (delete) a directory.

**syntax**         `int rmdir (const char * path);`  
                  `const char *path; /* the complete pathname of the directory to delete */`

**description**    The *rmdir( )* function removes the directory specified by the argument *path* from the file system. The directory must be empty or an error is returned. An attempt to remove the root directory also returns an error.

**Example**        `if (rmdir ("a:\\thisdir\\thatdir") != 0)  
                  printf ("Fail to delete the directory");`

**returns**        If successful, *rmdir* returns a zero, otherwise, it returns a nonzero value with the global variable *errno* set to a nonzero error code (refer to *smc.h*).

**see also**        `mkdir`

<b>chmod</b>
--------------

**purpose**        To change the attributes of the given pathname file

**syntax**         `int chmod (const char * pathname, int attribute);`  
                  `const char *pathname; /* the complete pathname to the file */`  
                  `int attribute; /* new attribute value for the file */`

**description**    The *chmod( )* function will change the *attribute* associated with the file specified by *pathname*. The attributes must be one or more of the following:

<code>FA_NORMAL</code>	Normal file (no attributes)
<code>FA_RDONLY</code>	Read-only file
<code>FA_HIDDEN</code>	Hidden file (does not affect accessibility)
<code>FA_SYSTEM</code>	System file
<code>FA_ARCH</code>	Archive bit (file changed since bit cleared)

**Example**        `int att;  
att = chmod ("a:\\myfile.bin", FA_SYSTEM | FA_RDONLY)  
if (att == EOF)  
    printf ("Chmod error, a:\\myfile.bin\\n");`

**returns**        If successful, *chmod* returns the new attributes, otherwise, it returns the constant EOF.

**see also**        `chmodfp`

## chmodfp

<b>purpose</b>	To changes the attributes of a file by using pointer												
<b>syntax</b>	<pre>int chmodfp (FILE * file_pointer, int function, int attribute); FILE *file_pointer; /* pointer to FILE data structure associated with the file whose attribute is to be changed */ int function; /* 0 = return current, 1 = set new attribute */ int attribute; /* new attribute value for the file */</pre>												
<b>description</b>	<p>The <i>chmodfp</i>( ) function will either return, or change the attributes of the opened file specified by <i>file_pointer</i>. If <i>function</i> = 0, then the current file attributes are returned. If <i>function</i> = 1, then the file attributes are set to new <i>attribute</i>. The FA_DIR attribute cannot be changed by this function. The new attributes will have no effect until the file is closed and reopened (e.g., if the file is currently open for writing, and is made read-only by this function, writes to the file are still permitted until the file is closed and reopened). The attributes must be one or more of the following:</p> <table><tr><td>FA_NORMAL</td><td>Normal file (no attributes)</td></tr><tr><td>FA_RDONLY</td><td>Read-only file</td></tr><tr><td>FA_HIDDEN</td><td>Hidden file (does not affect accessibility)</td></tr><tr><td>FA_SYSTEM</td><td>System file</td></tr><tr><td>FA_ARCH</td><td>Archive bit (file changed since bit cleared)</td></tr><tr><td>FA_DIR</td><td>File is a subdirectory</td></tr></table>	FA_NORMAL	Normal file (no attributes)	FA_RDONLY	Read-only file	FA_HIDDEN	Hidden file (does not affect accessibility)	FA_SYSTEM	System file	FA_ARCH	Archive bit (file changed since bit cleared)	FA_DIR	File is a subdirectory
FA_NORMAL	Normal file (no attributes)												
FA_RDONLY	Read-only file												
FA_HIDDEN	Hidden file (does not affect accessibility)												
FA_SYSTEM	System file												
FA_ARCH	Archive bit (file changed since bit cleared)												
FA_DIR	File is a subdirectory												
<b>Example</b>	<pre>FILE *fp; Int att; fp = fopen ("A:\\MYFILE.BIN", "r+b") att = chmodfp (fp, 1, FA_SYSTEM   FA_RDONLY);</pre>												
<b>returns</b>	If successful, <i>chmodfp</i> returns the current attributes of the file, otherwise, it returns the constant EOF.												
<b>see also</b>	chmod												

## chvlabel

<b>purpose</b>	To changes an existing volume label
<b>syntax</b>	<pre>int chvlabel(const char *drivename, char *oldlabel, const char *newlabel); const char *drivename; /* name of drive to alter label on (e.g. "A:") */ char *oldlabel; /* pointer to where to return old label */ const char *newlabel; /* the new label string to set */</pre>
<b>description</b>	The <i>chvlabel</i> ( ) function returns the existing volume label of the specified drive in <i>oldlabel</i> . If no volume label currently exists, <i>oldlabel</i> will be set to an empty string. If <i>newlabel</i> does not equal NULL, then the <i>newlabel</i> string is made the current volume label.
<b>Example</b>	<pre>char old_label [12]; if (chvlabel ("A:",old_label ,NULL)) printf ("chvlabel failed");</pre>
<b>returns</b>	If successful, <i>chmodfp</i> returns a zero, otherwise, it returns a nonzero value.

## 2.14 Miscellaneous

### DownLoadPage

<b>purpose</b>	Enter the 'Download' mode
<b>syntax</b>	<code>void DownLoadPage();</code>
<b>example call</b>	<code>open_com (1, 0x08); /* 38400, N, 8 */ DownLoadPage(); /* enter download mode */</code>
<b>description</b>	The <i>DownLoadPage</i> function is used to set 711/720 to the download mode. The Download page will show up and user can select the communication port and the baud rate for program download.
<b>returns</b>	none

### prc\_menu

<b>purpose</b>	Create a menu-driven interface.
<b>syntax</b>	<code>void prc_menu (MENU *menu);</code>
<b>example call</b>	<code>MENU MyMenu = {3, 1, 0, "My menu", {&amp;Collect, &amp;Upload, &amp;Download}}; MENU_ENTRY Collect = {0, 1, "1. Collect", FuncCollect, 0}; MENU_ENTRY Upload = {0, 2, "2. Upload", FuncUpload, 0}; MENU_ENTRY Download = {0, 3, "3. Download", FuncDownload, 0}; Void FuncCollect (void) { /* to do: add your own program code here */ } Void FuncUpload (void) { /* to do: add your own program code here */ } Void FuncDownload (void) { /* to do: add your own program code here */ } prc_menu (&amp;MyMenu); /* process MyMenu menu*/</code>
<b>description</b>	The <i>prc_menu</i> function is used to create a user-defined menu. SMENU and MENU structures are defined in "711lib.h" and "720lib.h". Users can just fill the MENU structure and call the <i>prc_menu</i> function to build a hierarchy menu-driven user interface.
<b>returns</b>	none

## 3 Standard Library Routines

The standard library routines supported are categorized and listed below,

### 3.1 Input and Output : <stdio.h>

- File Operations: Not supported, please use Syntech Library routines.
- Formatted Output: Only printf is supported, for formatted output to display, please refer to Syntech Library "LCD".
- Formatted Input: Only scanf is supported.
- Character Input and Output: Not supported, please refer to Syntech Library "External AT Keyboard" and "Membrane Keypad"
- Direct Input and Output: Not supported.

### 3.2 Character Class Test : <ctype.h>

For each function, the argument is an int, whose value must be EOF or representable as an unsigned char, and the return value is an int. The functions return non-zero (true) if the argument c satisfies the condition described, and zero if not.

- isalnum(c) isalpha(c) or isdigit(c) is true
- isalpha(c) isupper(c) or islower(c) is true
- iscntrl(c) control character
- isdigit(c) decimal digit
- isgraph(c) printing character except space
- islower(c) lower-case letter
- isprint(c) printing character including space
- ispunct(c) printing character except space or letter or digit
- isspace(c) space, formfeed, newline, carriage return, tab, vertical tab
- isupper(c) upper-case letter
- isxdigit(c) hexadecimal digit

In addition, there are two functions that convert the case of letters,

- int tolower(c) convert c to lower case
- int toupper(c) convert c to upper case

### 3.3 String Functions : <string.h>

#### **Functions start with "str"**

In the routine list, the type of variables used are as below,

- ```
char *s, t;  
const char * cs, ct;  
size_t n;  
int c;
```
- char \*strcpy(s, ct) copy string ct to string s, including 0x00, return s
  - char \*strncpy(s, ct, n) copy at most n characters of string ct to s, return s, pad with 0x00s if ct has fewer than n characters
  - char \*strcat(s, ct) concatenate string ct to end of string s, return s
  - char \*strncat(s, ct, n) concatenate at most n characters of ct to s, return s
  - int strcmp(cs, ct) compare string cs and ct, return value < 0 if cs<ct, = 0 if cs = ct, > 0 if cs>ct
  - int strncmp(cs, ct, n) compare at most n characters of string cs and ct, return value < 0 if cs < ct, = 0 if cs = ct, > 0 if cs>ct

- `char *strchr(cs, c)` return pointer to first occurrence of `c` in `cs` or `NULL` if not present
- `char *strrchr(cs, c)` return pointer to last occurrence of `c` in `cs` or `NULL` if not present
- `size_t strspn(cs, ct)` return length of prefix of `cs` consisting of characters in `ct`
- `size_t strcspn(cs, ct)` return length of prefix of `cs` consisting of characters not in `ct`
- `char *strpbrk(cs, ct)` return pointer to first occurrence in string `cs` of any character of string `ct`, or `NULL` if none are present
- `char *strstr(cs, ct)` return pointer to first occurrence of string `ct` in `cs`, or `NULL` if not present
- `size_t strlen(cs)` return length of string `cs`
- `char *strtok(s, ct)` searches `s` for tokens delimited by characters from `ct`
- `strcoll` Not supported
- `strerror` Not supported

### Functions start with "mem"

In the list, types of variables are as below,

```
void *s, *t;
const void *cs, *ct;
size_t n;
int c;
```

- `void *memcpy(s, ct, n)` copy `n` characters from `ct` to `s`, return `s`
- `void *memmove(s, ct, n)` same as `memcpy` except that it works fine even if the objects overlap
- `int memcmp(cs, ct, n)` compare the first `n` characters of `cs` with `ct`; return as `strcmp`
- `void *memchr(cs, c, n)` return pointer to first occurrence of character `c` in `cs` or `NULL` if not present among the first `n` characters
- `void *memset(s, c, n)` place character `c` into first `n` characters of `s`, return `s`

## 3.4 Mathematical Functions : <math.h>

Mathematical functions are listed below and all of them return a double.

In the list, types of variables are as below,

```
double x, y;
int n;
```

- `sin(x)` sine of `x`
- `cos(x)` cosine of `x`
- `tan(x)` tangent of `x`
- `asin(x)`  $\sin^{-1}(x)$  in range  $[-\pi/2, \pi/2]$ ,  $x \in [-1, 1]$
- `acos(x)`  $\cos^{-1}(x)$  in range  $[0, \pi]$ ,  $x \in [-1, 1]$
- `atan(x)`  $\tan^{-1}(x)$  in range  $[-\pi/2, \pi/2]$
- `atan2(y, x)`  $\tan^{-1}(y/x)$  in range  $[-\pi, \pi]$
- `sinh(x)` hyperbolic sine of `x`
- `cosh(x)` hyperbolic cosine of `x`
- `tanh(x)` hyperbolic tangent of `x`
- `exp(x)` exponential function  $e^x$
- `log(x)` natural logarithm  $\ln(x)$ ,  $x > 0$
- `log10(x)` base 10 logarithm  $\log_{10}(x)$ ,  $x > 0$
- `pow(x, y)`  $x^y$ . A domain error occurs if  $x=0$  and  $y \leq 0$ , or if  $x < 0$  and  $y$  is not an integer
- `sqrt(x)`  $x$ ,  $x \geq 0$

- `ceil(x)` smallest integer not less than x, as a double
- `floor(x)` largest integer not greater than x, as a double
- `fabs(x)` absolute value x
- `ldexp(x, n)`  $x * 2^n$
- `frexp(x, int *exp)` splits x into a normalized fraction in the interval [1/2, 1], which is returned, and a power of 2, which is stored in \*exp. If x is zero, both parts of the result are zero.
- `modf(x, double *ip)` splits x into integral and fractional parts, each with the same sign as x. It stores the integral part in \*ip, and returns the fractional part.
- `fmod(x, y)` floating point remainder of x/y, with the same sign as x. If y is 0, the result is implementation-defined.

### 3.5 Utility Function : <stdlib.h>

#### Number Conversion

- `double atof( const char *s)` convert s to double, equivalent to `strtod(s, (char **)NULL)`
- `int atoi(const char *s)` convert s to integer, equivalent to `strtol(s, (char**)NULL, 10)`
- `int atol(const char *s)` convert s to long, equivalent to `strtol(s, (char**)NULL, 10)`
- `double strtod(const char *s, char **endp)` converts the prefix of s to double
- `long strtol(const char *s, char **endp, int base)` converts the prefix of s to long
- `unsigned long strtoul(const char *s, char **endp, int base)` converts the prefix of s to unsigned long
- `int rand(void)` returns a random integer from 0 to 32767
- `void srand(unsigned int seed)` seed for new pseudo-random generation
- `void *bsearch()` binary search
- `void qsort()` ascending sorts
- `int abs(int n)` integer absolute
- `long labs(long n)` long absolute
- `div_t div(int num, int denom)` integer division
- `ldiv_t ldiv(long num, long denom)` long division

#### Storage Allocation

Not supported. Please use Syntech library routines instead.

### 3.6 Diagnostics : <assert.h>

Not supported.

### 3.7 Variable Argument Lists : <stdarg.h>

Functions for processing variable arguments are listed below.

```
va_start(va_list ap, lastarg)
type va_arg(va_list ap, type)
void va_end(va_list ap)
```

### 3.8 Non-Local Jumps : <setjmp.h>

Not supported.

### **3.9 Signals : <signal.h>**

Not supported.

### **3.10 Date and Time Function : <time.h>**

Not supported.

### **3.11 Implementation-defined Limits : <limits.h> and <float.h>**

Please refer to limit.h and float.h.

## 4 Real Time Kernel

711/720 Data Terminal comes with a real-time kernel ( $\mu$ C/OS) that allows user to generate a preemptive multitasking application. User can apply the real time kernel functions to split the application into multiple tasks that each task takes turns to gain the access to the system resource by a priority-based schedule.

$\mu$ C/OS applies the semaphore mechanism to control the access to the shared resource for the multiple tasks. There are generally only three operations that can be performed on a semaphore: CREATE, PEND, and POST. A semaphore is a key that the task requires in order to continue execution. If the semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner.

A task is an infinite loop function or a function which deletes itself when it is done executing. Each task is assigned with an appropriate priority. The more important the task, the higher the priority given to it.  $\mu$ C/OS can manage up to 32 tasks (with priority 0 to 31, the lower number, the higher priority) for the user's program of the 711/720 Data Terminal. The main task, `main()`, takes priority 16.

A task desiring the semaphore will perform a PEND operation. A task releases a semaphore by performing a POST operation. If there are several tasks on the pending list, the highest priority task waiting for the semaphore will receive the semaphore when the semaphore is posted. The pending list of tasks is always initially empty.

The  $\mu$ C/OS related functions are discussed as follows.

## OS\_ENTER\_CRITICAL

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>purpose</b>      | Disable the processor's interrupt                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>syntax</b>       | void OS_ENTER_CRITICAL(void);                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>example call</b> | OS_ENTER_CRITICAL();<br>... /* user code */<br>OS_EXIT_CRITICAL();                                                                                                                                                                                                                                                                                                                                                                                                |
| <b>description</b>  | A critical section of code is code that needs to be treated indivisibly. Once the section of code starts executing, it must not be interrupted. To ensure this, user can call <i>OS_ENTER_CRITICAL</i> function to disable interrupts prior to executing the critical code and enable the interrupts when the critical code is done. The function executes in about 5 CPU clock cycles. This function and <i>OS_EXIT_CRITICAL</i> function must be used in pairs. |
| <b>returns</b>      | none                                                                                                                                                                                                                                                                                                                                                                                                                                                              |

## OS\_EXIT\_CRITICAL

|                     |                                                                                                                               |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <b>purpose</b>      | Enable the processor's interrupt                                                                                              |
| <b>syntax</b>       | void OS_EXIT_CRITICAL(void);                                                                                                  |
| <b>example call</b> | OS_ENTER_CRITICAL();<br>... /* user code */<br>OS_EXIT_CRITICAL();                                                            |
| <b>description</b>  | The function executes in about 5 CPU clock cycles. This function and <i>OS_ENTER_CRITICAL</i> function must be used in pairs. |
| <b>returns</b>      | none                                                                                                                          |

## OSSemCreate

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>purpose</b>      | Create and initialize a semaphore                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>syntax</b>       | OS_EVENT OSSemCreate(unsigned <i>value</i> );<br><br>where, OS_EVENT, a data structure to maintain the state of an event called Event Control Block (ECB), is defined as below,<br><br>typedef struct os_event {<br>unsigned char OSEventTb[8]; /* Group corresponding to tasks waiting for event to occur */<br>unsigned char OSEventGrp; /* List of tasks waiting for event to occur */<br>long OSEventCnt; /* Count of used when event is a semaphore */<br>void *OSEventPtr; /* Pointer to message or queue structure */<br>} OS_EVENT;<br><br><i>value</i> is the initial value of the semaphore. The initial <i>value</i> of the semaphore is allowed to be between 0 and 32767. |
| <b>example call</b> | sem_time = OSSemCreate(1); /* create a semaphore sem_time and the initial value of sem_time is set to 1. */                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>description</b>  | This function is used to create and initialize a semaphore. Semaphores must be created before they are used.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

**returns** A pointer to the event control block allocated to the semaphore. If no event control block is available, a NULL pointer will be returned.  
OS\_NO\_ERR, if the function was successful.

## OSSemPend

**purpose** List a task on the pending list for the semaphore

**syntax** unsigned char OSSEmPend (OS\_EVENT \*pevent, unsigned long timeout, unsigned char \*err);

where, *pevent* is a pointer to the semaphore. This pointer is returned to your application when the semaphore is created.

*timeout* is used to allow the task to resume execution if the semaphore is not acquired within the specified number of clock ticks. A *timeout* value of 0 indicates that the task desires to wait forever for the semaphore. The maximum *timeout* is 65535 clock ticks.

*err* is a pointer to a variable which will be used to hold an error code. *OSSEmPend* sets \*err to either:

- (1) OS\_NO\_ERR, if the semaphore is available
- (2) OS\_TIMEOUT, if a timeout occurred

**example call** OSSEmPend (sem\_time, 0, &err);

**description** This function is used when a task desires to get exclusive access to a resource, synchronize its activities with an Interrupt Service Routine (ISR) or wait until an event occurs. If a task calls *OSSEmPend* function and the value of the semaphore is greater than 0, then *OSSEmPend* function will decrement the semaphore and return to its caller. However, if the value of the semaphore is less than or equal to zero, *OSSEmPend* function decrements the semaphore value and places the calling task in the waiting list for the semaphore. The task will thus wait until a task or an ISR releases the semaphore or signals the occurrence of the event. In this case, rescheduling occurs and the next highest priority task ready to run is given control of the CPU. An optional timeout may be specified when pending for a semaphore.

**returns** none

## OSSEmPost

**purpose** Signal the semaphore

**syntax** unsigned char OSSEmPost (OS\_EVENT \*pevent);

where, *pevent* is a pointer to the semaphore. This pointer is returned to your application when the semaphore is created.

**example call** OSSEmPost (sem\_time);

**description** A semaphore is signaled by calling *OSSEmPost* function. If the semaphore value is greater than or equal to zero, the semaphore is incremented and *OSSEmPost* function returns to its caller. If the semaphore value is negative then tasks are waiting for the semaphore to be signaled. In this case, *OSSEmPost* function removes the highest priority task pending for the semaphore from the waiting list and makes this task ready to run. The schedule is then called to determine if the awakened task is now the highest priority task ready to run

**returns** (1) OS\_NO\_ERR, if the semaphore is available

(2) OS\_TIMEOUT, if a timeout occurred

### OSTaskCreate

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>purpose</b>      | Create a task                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <b>syntax</b>       | <pre>unsigned char OSTaskCreate (void (*task)(void *pd), void *pdata, unsigned char *pstk, unsigned long stk_size, unsigned char prio);</pre> <p>where, <i>task</i> is a pointer to the task's code.</p> <p><i>pdata</i> is a pointer to an optional data area which can be used to pass parameters to the task when it is created.</p> <p><i>pstk</i> is a pointer to the task's top of stack. The stack is used to store local variables, function parameters and return addresses and CPU registers during an interrupt. The size of this stack is defined by the task requirements and the anticipated interrupt nesting. Determining the size of the stack involves knowing how many bytes are required for storage of local variables for the task itself, all nested functions, as well as requirements for interrupts (accounting for nesting).</p> <p><i>prio</i> is the task priority. A unique priority number must be assigned to each task and the lower the number, the higher the priority.</p> |
| <b>example call</b> | <pre>OSTaskCreate(beep_task, (void *)0, beep_stk, 256, 10); /* create a beep_task with priority 10 */ static unsigned char beep_stk[256]; void beep_task(void*);</pre>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>description</b>  | This function allows an application to create a task. The task is managed by $\mu$ C/OS. Tasks can be created prior to the start of multitasking or by a running task.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>returns</b>      | OS_PRIO_EXIST, if the requested priority already exist.<br>OS_NO_ERR, if the function was successful.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

### OSTaskDel

|                     |                                                                                                                                                                                                                                                                                                                    |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>purpose</b>      | To delete a task                                                                                                                                                                                                                                                                                                   |
| <b>syntax</b>       | <pre>unsigned char OSTaskDel (unsigned char prio);</pre> <p>where, <i>prio</i> is the task priority. A unique priority number must be assigned to each task and the lower the number, the higher the priority.</p>                                                                                                 |
| <b>example call</b> | <pre>OSTaskDel (10); /* delete a task with priority number 10 */</pre>                                                                                                                                                                                                                                             |
| <b>description</b>  | This function allows user's application to delete a task by specifying the priority number of the task to delete. The calling task can be deleted by specifying its own priority number. The deleted task is returned to the dormant state. The deleted task may be created to make the deleted task active again. |
| <b>returns</b>      | OS_TASK_DEL_IDLE           if the task to delete is an idle task.<br>OS_TASK_DEL_ERR           if the task to delete does not exist.<br>OS_NO_ERR                 if the task was deleted.                                                                                                                         |

## OSTimeDly

- purpose** Allow a task to delay itself for a number of clock ticks.
- syntax** void OSTimeDly (unsigned long *ticks*);  
where, *ticks* is the delay time in units of 5 ms.
- example call** OSTimeDly(10); /\* delay the task for 10 X 5 ms \*/
- description** This function allows a task to delay itself for a number of clock ticks. Rescheduling always occurs when the number of clock ticks is greater than zero. Valid delays range from 1 to 65535 ticks. Note that calling this function with a delay of 0 results in no delay and thus the function returns to the caller.
- returns** none