520 Data Terminal C Language Programming Guide

SYNTECH INFORMATION CO., LTD.

Russian Office: Narodnogo Opolchenia, 34, 212, 123423, Moscow, Russia
Tel: +7 095 197-1871 Fax: +7 095 946-8611
e-mail: cipherlab@iname.com http:// www.syntech.ru

TABLE OF CONTENTS

Preface

Users can generate customized application programs for the 520 Data Terminal by using the C Compiler with Syntech 520 Function Library and/or the Basic Compiler with 520 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 520 Data Terminal with the C Compiler. Chapter 2, “C Compiler”, discusses some specific characteristics of the C Compiler. Chapter 3, “520 Function Library”, presents the user callable library routines specific to the features of the 520 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, μC/OS. User can generate a real time multitasking system by using the μC/OS functions. Chapter 6, “Sample Programs”, contains source codes of two sample programs. They illustrate the use of the 520 function library and also give user an idea of the 520 application in the real world.

1 Development Environment

1.1 Directory Structure

The CipherLab 520 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**{**xe "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**{**xe "ETC"**}** : 11 files, help and version information of the C compiler

3) INCLUDE**{**xe "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	

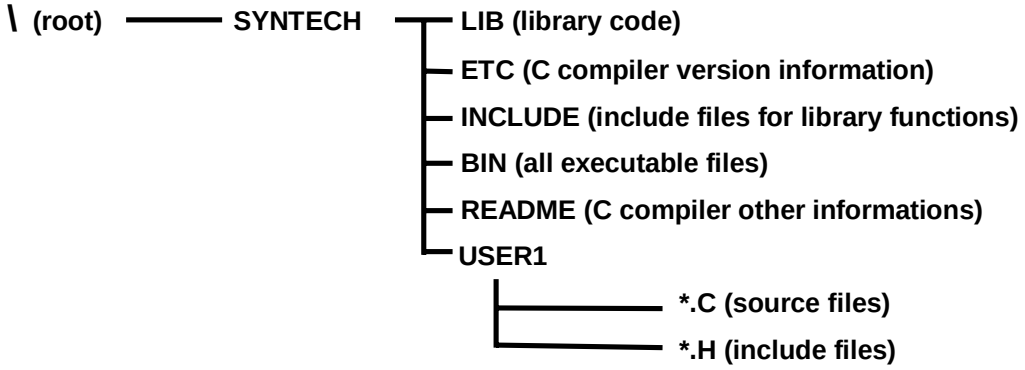
   - 1 Include file for 520 Function Library : 520lib.h
   - 1 Include file for Real Time Kernel Library : ucos.h

4) LIB**{**xe "LIB"**}** : Library object code files
   - c900ml.lib C standard library
   - 520lib.lib 520 function library

5) README**{**xe "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 520, 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).

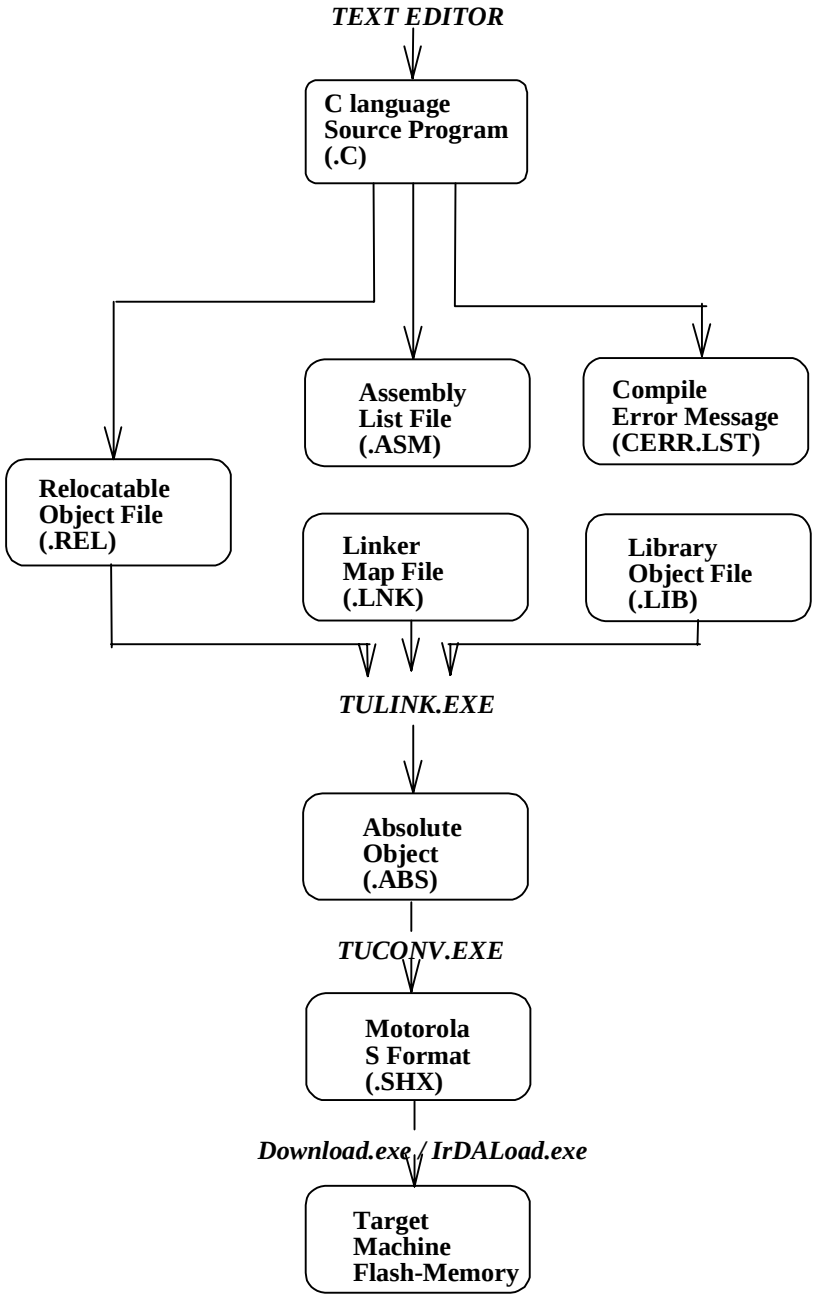
To facilitate efficiency, the compiler invokes a virtual memory manager "DOS4GW". It recognizes and supports various PCs. However, if it does not work on your PC. The program PMINFO can be used to identify the problem. (if you have difficult using the compiler, run the PMINFO, print all messages and 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 user0 and user0 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\520lib.lib       /* 520 Function library */

/*****************************************/
/*  User could provide desirable values to */
/*    the following two variables.         */
/*****************************************/
___MainStackSize__ = 0x001000;
HeapSize  = 0x000100;

/*****************************************/
/*  Do not modify anything beyond this line  */
/*****************************************/
memory
{
    RAM      : org = 0x400100, len = 0x01ff00
    ROM      : org = 0xf80000, len = 0x070000
    IO       : org = 0x100000, len = 0x100000
}

sections
{
    code  org = 0xf80000 : {
        *(f_head)
        *(f_code)
    } > ROM

    sys_area org = 0x400100 : {
        *(f_bcr)
        ..\lib\520lib.lib(f_area)
        ..\lib\c900ml.lib(f_area)
    } > RAM

    data org=org(code)+sizeof(code) addr=org(sys_area)+sizeof(sys_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

    area org = addr(xcode) + sizeof(xcode) : {
        . += ___MainStackSize__;
        . += HeapSize;
        *(f_unshare)
        *(f_area)
    } > RAM
}

SysRamEnd      = org(area) + sizeof(area);
DataRam        = addr(data);
CodeRam        = addr(xcode);
HeapTop        = org(area) + ___MainStackSize__;
___MainStack__ = org(area);
/* 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{xe "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* : A digit from 1 to 4 to specify RS232 communication port to be used for downloading. Care must be taken that in order to support high baud rate (up to 38400), the download program accesses the UART chip directly. The UART must be NSC8250 compatible and their starting I/O addresses are listed below,

Port #	Starting Address
1	0x3f8
2	0x2f8
3	0x3e8
4	0x2e8

- *BAUDRATE* : baud rate support are 38400, 19200, 9600, 4800, 2400 and 1200.
- *PARITY* : the parity can be one of "E", "O" or "N" for even, odd and no parity.
- *DATABITS* : 7 or 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{xe "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

Note that the signed and unsigned short int is 2 bytes long. This might cause trouble in calling sscanf(), for example,

```
{
        char    c,  s[20];
        int     i;
        strcpy(s, "123   456");
        sscanf(s, "%d  %hd", &i, &c);
}
```

The end result will be i=123 and c=(456-256)=200, negative for signed character. And the sscanf stores 2 bytes back to variable c's address. That is, the variable located following c is changed.

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{xe "Bit-Field"} Usage

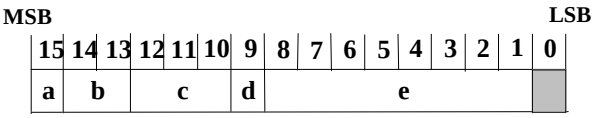
The following types can be used as the bit field base types.

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,

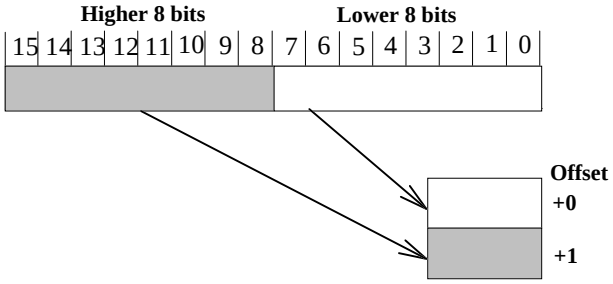
1) Fields are stored from the highest bits

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



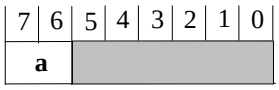
2) Little endien

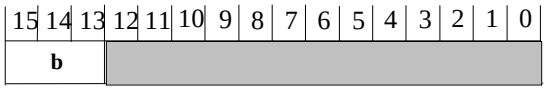
If the base type of a bit field member is a type requiring two bytes or more (e.g. unsigned int), the data is stored in memory after its bytes are turned topside down.



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

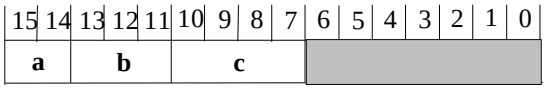
```
struct field {
        unsigned   char       a:2;
        unsigned    short     b:3;
}
```





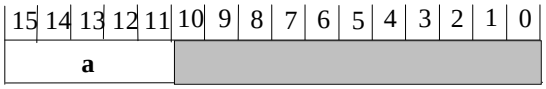
4) Different type (signed/unsigned)

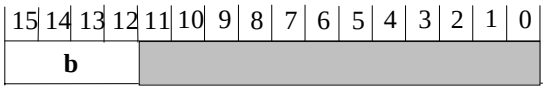
```
struct field {
        signed          short   a:2;
        unsigned short          b:3;
        signed          short   c:4;
}
```



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.

2 520 Function Library

CipherLab provides several user-callable 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 520 Library Header File, "520lib.h". Assumption was made that the reader has prior knowledge of the 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, RS485 ports : 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

Two time variables are declared by the system, which can be used for counting time-out and so on. As they are updated by timer interrupt, DO NOT write to them.

- extern volatile unsigned long sys_msec**{**xe "sys_msec"**}**; /* in unit of 5 ms */
- extern volatile unsigned long sys_sec**{**xe "sys_sec"**}**; /* in unit of 1 second */

These two variables are cleared to 0 upon power up.

shut_down	
purpose	Stop the system
syntax	void shut_down();
example call	shut_down();
description	The routine will shut down the system.
returns	none

system_restart	
purpose	Re-start the system
syntax	void system_restart();
example call	system_restart();
description	The routine jumps to the power on reset point and restarts the system. It functions the same as turn power off then on.
returns	none

2.2 Reader

The barcode decoding routines consist of 5 functions: **InitScanner1( )**, **InitScanner2( )**, **Decode( )**, **HaltScanner1( )** and **HaltScanner2( )**. The *InitScanner1( )* and *InitScanner2( )* functions are used to initialize the respective scanner port. The *Decode( )* function is used to perform decoding. And the *HaltScanner1( )* and *HaltScanner2( )* functions are used to stop the respective 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 use of *InitScanner1( )* and *InitScanner2( )* functions. After the scanner ports are initialized, the *Decode( )* function can be called in the program loops to perform barcode decoding.

It is not necessary to specify the type of scanners connected to the scanner ports. The barcode and magnetic card decoding routines will automatically recognize the scanner type whether it is a WAND, WAND/LASER emulation scanner, or an MSR scanner.

There are six variables relate to the barcode decoding routines: **ScannerDesTbl**, **CodeBuf**, **CodeLen**, **CodeType**, **MsrDirection** and **ScannerNo**. 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.

MsrDirection : This integer indicates the reading direction of the magnetic card being decoded upon successful decoding.

ScannerNo : This character indicates the scanner port 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	EAN8 with Addon 2	N
Italy Pharma-code	B	EAN8 with Addon 5	O
CIP 39	C	EAN13 no Addon	P
Industrial 25	D	EAN13 with Addon 2	Q
Interleave 25	E	EAN13 with Addon 5	R
Matrix 25	F	MSI	S
Codabar (NW7)	G	Plessey	T
Code 93	H	Code ABC	U
Code128	I	ISO Track 1	a
UPCE no Addon	J	ISO Track 2	b
UPCE with Addon 2	K	ISO Track 1 and 2	c
UPCE with Addon 5	L	ISO Track 2 and 3	d
EAN8 no Addon	M		

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

Subscriptor	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

Subscriptor	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

Subscriptor	Bit	Description
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	Industrial 25 Min Code Length / Fixed Length 2
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
20	3 - 0	Reserved
21		Scanner Time-out Duration in seconds for Scanner Port 1
22		Reserved

Decode

purpose Perform barcode decoding.

syntax int Decode( );

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.

HaltScanner1, HaltScanner2

purpose Stop respective scanner port from operating.

syntax void HaltScanner1( );
void HaltScanner2( );

example call HaltScanner1( );
HaltScanner2( );

description Use *HaltScanner1* function to stop scanner port 1 from operating and use *HaltScanner2* function to stop scanner port 2 from operating. To restart a halted scanner port, the initialization function (*InitScanner1* and *InitScanner2*) must be called.

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

returns none

InitScanner1, InitScanner2

purpose Initialize respective scanner port.

syntax void InitScanner1( );
void InitScanner2( );

example call InitScanner1( );
InitScanner2( );
while (1) { if (Decode( )) break; }

description Use *InitScanner1* function to initialize scanner port 1 and use *InitScanner2* function to initialize scanner port 2 . The scanner ports won't work unless they are initialized.

returns none

2.3 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 according to it is automatically handled by the background program. There is no need for the application program waiting for the beeper stops.

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

2.3.1 Beeper Sequence**{**xe "Beeper Sequence"**}**

A beeper sequence is an integer array which is used to instruct how the beeper activates. It is comprised of **beep frequency{xe "beep frequency"}** / **beep duration{xe "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.

2.3.2 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.

Beep Frequency = 76000 / 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.

2.3.3 Beep Duration

Beep duration is an integer used to specify how long the beeper activates with a specified beep frequency. Beep duration is specified in units of 0.05 second. To get a beep of 1 second, the beep duration should be 20. Beep duration with value of 0 will terminate the beeper sequence.

beeper_status

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

off_beeper

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

on_beeper

purpose	Assign a beeper sequence to instruct beeper action.
syntax	void on_beeper(int* sequence); int* sequence; /* pointer to integer array where beeper sequence resides */
example call	int two_beeps[]= { 19, 10, 0, 10, 19, 10, 0, 0 }; on_beeper(two_beeps);
description	The *on_beeper* 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.
returns	The *on_beeper* function has no return value.

2.4 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 NiCd battery keeps the calendar chip running even when power is turned off.

Note that the system time variable *sys_msec***{**xe "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.

2.4.1 Timer Adjustment

The calendar chip can be fine tuned to compensate for a fast or slow clock. This is an outstanding feature for those applications which need punctual system time such as a time/clock application. The tuning of the calendar chip is done by modifying the value of the **trimming register** of the calendar chip. The **adjust_timer** function can be used to modify the value of the trimming register.

2.4.2 Trimming Register

The frequency of the calendar chip can be tuned in units of ppm via a digital trimming register. The trimming range is from 0 to 255 ppm. The bigger the value of the trimming register the slower the calendar chip runs. For instance, if the calendar chip is 1 second **slow** in one day then the value of the trimming register should **decrease** 12 to correctly adjust the calendar chip. During system initialization, this register is set to 186.

1 sec/ 1 day = 1000000 / (24 hours X 60 min X 60 sec) = 11.57 ppm ~= 12 ppm

2.4.3 Leap Year

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

adjust_timer

purpose Modify the value of the trimming register of the calendar chip.

syntax int adjust_timer(int offset);

int offset; /* the amount of modification made to the trimming register */

example call adjust_timer(12);

description The *adjust_timer* function modifies the value of the trimming register of the calendar chip with the amount specified in the argument *offset*. If *offset* is positive, the *adjust_timer* function increases the trimming register by this value and thus slows down the calendar chip. If *offset* is negative, the *adjust_timer* function decreases the trimming register by this value and thus makes the calendar chip runs faster. If *offset* is 0, no modification is made to the trimming register.

returns The *adjust_timer* function returns the value of the trimming register after the operation. If the calendar chip malfunctions, the return value will be 0 to indicate error.

comments Since the value allowed for the trimming register is from 0 to 255. Decreasing the value of trimming register down to 0 is possible but should be avoided because a trimming register with a value of 0 also indicates error in the return value of the *adjust_timer* function.

DayOfWeek

purpose Get the day of the week information.

syntax int DayOfWeek( );

example call day = DayOfWeek( );

description The *DayOfWeek* function returns the day of week information based on current date.

returns The *DayOfWeek* 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

purpose Get current date and time.

syntax int get_time(char*cur_time);
char* cur_time; /*pointer of character array where the date and time will be copied to */

example call get_time(system_time);

description The *get_time* function reads current date and time from the calendar chip and copies them to a character array specified in the argument *cur_time.* The character array *cur_time* 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 **YYYY** : year, 4 digits
MM : month, 2 digits
DD : day, 2 digits
hh : hour, 2 digits

mm : minute, 2 digits
ss : second, 2 digits

returns Normally the *get_time* function always returns an integer value of 1. If the calendar chip malfunctions, the *get_time* function will then return 0 to indicate error.

get_time_ms

purpose Get current time to tens of millisecond precision.

syntax
```
int get_time_ms(char*cur_time);
char* cur_time;          /*pointer of character array where the date
                           and time will be copied to */
```

example call
```
get_time_ms(now_time);
```

description The *get_time_ms* function reads current time down to tens of millisecond precision from the calendar chip and copies them to a character array specified in the argument *cur_time.* The character array *cur_time* allocated must have a minimum of 9 bytes to accommodate the time and the string terminator. The format of the system date and time is listed below.

"hhmmssnn"

where **hh** : hour, 2 digits
mm : minute, 2 digits
ss : second, 2 digits
nn : tens of millisecond, 2 digits

returns Normally the *get_time_ms* function always returns an integer value of 1. If the calendar chip malfunctions, the *get_time_ms* function will then return 0 to indicate error.

set_time

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.

2.5 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.

2.5.1 File System

On 520 terminal, there is an on-board 128K bytes base memory (SRAM). This is the place where all the system parameters, program variables, and program stack reside. User may as well install an optional memory board inside 520. The memory size of this optional memory board can be 128K bytes up to 2M bytes. The file system (which includes file allocation table, directory, and file data) resides on the memory board, if a memory board is installed. But if the memory board does not exist, the file system will be on the base memory.

When 520 is started, the 520 Kernel will try to find the file system on the optional memory board. If a memory board is installed and the 520 Kernel fails to find the file system, the 520 Kernel will initialize the memory board and setup a file system on it. If the 520 Kernel detects that there is no memory board installed, then it will try to find the file system on the base memory. A new file system will be setup on the base memory, if 520 Kernel can not find it.

For the file system, the memory board always has a higher preference over the base memory. The 520 Kernel will always assume the file system is on the memory board (if it exist), even if there is already a file system on the base memory. Under this case, however, the file system on the base memory is remained untouched.

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.5.5 Directory

The file system of 520 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.

2.5.6 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 which 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 use of 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).

2.5.7 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	*filename* 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 *DBF_fd* does not exist.
4	File specified by *DBF_fd* 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 *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened
9	The value of *count* is negative.
10	No more free file space for file extension.

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

appendln

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 *fd* does not exist.
4	File specified by *fd* 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 treminator in *buf*.

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 *fd* does not exist.
4	File specified by *fd* 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 *fd* does not exist.
4	File specified by *fd* 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 *DBF_fd* does not exist.
4	File specified by *DBF_fd* 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	*filename* is a NULL string.
6	Can't create file. Because the maximum number of files allowed in the system is exceeded.
9	Illegal argument : *member_len*
12	File specified by *filename* 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 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 *DBF_fd* does not exist.
4	File specified by *DBF_fd* 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 *key_number.*
17	Illegal value in argument *key_offset*,and/or *key_len*.
18	DBF file specified by *DBF_fd* is not empty.
19	IDX file specified by *key_number* already exists.

delete_member

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 *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument *key_number.*
14	The IDX file specified by *key_number* 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 *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened
9	The value of *count* 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 *fd* does not exist.
4	File specified by *fd* 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 *fd* does not exist.
4	File specified by *fd* 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 *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened

filelist

purpose Get file directory information.

syntax
```
int filelist(char* dir);
char* dir;      /* pointer to a character array where the file directory
                   information is copied to */
```

example call
```
total_file = filelist(dir);
```

description The *filelist* 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 *dir*.

returns The *filelist* function returns the number of files currently exist in the system.

get_member

target file type DBF

purpose Read the member pointed by the index pointer.

syntax
```
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 */
```

example call
```
if (get_member(DBF_fd,1,buffer) == 0) puts(buffer);
```

description The *get_member* function copies the member pointed to by a index pointer to a character array specified in the argument *buffer*. The IDX file concerned is specified in the argument *key_number* which is associated to a DBF file whose file handle is *DBF_fd*.

Returns The *get_member* function returns an integer value of 1 to indicate success. In case of error, *get_member* 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 *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument *key_number.*
14	The IDX file specified by *key_number* does not exist.
16	There are no members in the DBF file.

has_member

target file type DBF

purpose Check if a specific member exist in an IDX file.

syntax
```
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 */
```

```
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 *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened
13	Illegal value in argument *key_number.*
14	The IDX file specified by *key_number* does not exist.

lseek

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 *fd* does not exist.
4	File specified by *fd* is not a DAT file.
7	Invalid file handle
8	File not opened
9	Illegal *origin* value.
15	New position is beyond end-of-file.

lseek_DBF

target file type DBF

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

syntax
```
long lseek_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
```
lseek_DBF(DBF_fd, 1, 1L, 0);  /* move to next member */
```

description The *lseek_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, *lseek_DBF* returns the new sequence position that the index pointer points to. In case of error, *lseek_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 *DBF_fd* does not exist.
4	File specified by *DBF_fd* is not a DBF file.
7	Invalid file handle
8	File not opened
9	Illegal *origin* value.
13	Illegal value in argument *key_number.*
14	The IDX file specified by *key_number* 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 *DBF_fd* does not exist.
4	File specified by *DBF_fd* 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	*filename* is a NULL string.
4	File specified by *filename* is not a DAT file.
5	File specified by *filename* 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	*filename* is a NULL string.
2	File specified by *filename* does not exist.
4	File specified by *filename* is not a DBF file.
5	File specified by *filename* 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 *fd* is not a DAT file.
7	*fd* 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 (\0) 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 (\0) 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 *fd* is not a DAT file.
7	*fd* 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 IDX 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, the *rebuild_index* function will first delete it. If the index does not exist, *rebuild_index* will directly create and rebuild the index. 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 greater than *member_len*.

The argument *preference_index* specifies an index file from which the *rebuild_index* function takes as the input sequence for building index. This function is quite useful when generating reports. For instance, if a report is to be generated by the sequence of date, department, and ID number, this is easily done by first rebuilds the ID number index 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. The *preference_index* has no effect on the following added members. It takes effect only when rebuilding index. When there is no preferred index desired, *preference_index* should have the value of 0. The preferred sequence will be the original member sequence in the DBF file so done.

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

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 f *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	*filename* is a NULL string.
2	File specified by *filename* does not exist.

remove_index

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 *fd* is not a DBF file.
8	*fd* is not a file handle of a previously opened file.
11	Index file specified by *key_number* does not exist.

rename

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 *old_filename* or *new_filename* is a NULL string.
2	File specified by *old_filename* does not exist.
3	A file with file name *new_filename* already exists.

tell

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 *fd* is not a DAT file.
7	*fd* is not a file handle of a previously opened file.

tell_DBF

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 *DBF_fd* is not a DAT file.
8	*DBF_fd* is not a file handle of a previously opened file.
11	Index file specified by *key_number* does not exist.

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 *fd* is not a DAT file.
7	*fd* 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 *fd* is not a DAT file.
7	*fd* is not a file handle of a previously opened file.
9	no null character found in *buffer*
10	No more free file space for file extension.

2.6 Digital Input / Output

This section describes the digital Input / Output manipulation routines. There are four digital input and four digital output on 520 (if the optional DIO board is installed). The four digital input are numbered from 0 to 3, and the four digital output are numbered from 4 to 7.

get_di

purpose Read digital Input.

syntax
```
int get_di(int di);
int di;        /* digital input number from 0 to 3, depends on I/O board */
```

example call
```
if (get_di(0)) send_lcds(DI0 is ON!\n");
```

returns 1, if photo-coupler is turned on, that is current flows through the LED.
0, otherwise.

set_dout

purpose Set the digital output

syntax
```
void set_dout(int do, int mode, int duration);
int do;          /* digital output number starts from 4 to 7 */
int mode;        /* output mode */
int duration;    /* duration */
```

example call
```
set_dout(4,DOUT_ON,200);    /* on digital output for 1 second then off */
```

description The *set_dout* sets the digital output points specified by do.
The duration specified in the argument *duration* is in units of 5 milisecond. That is, if a duration of 1 second is desired, a value of 200 should be assigned to the argument *duration*. A value of 0 in the argument *duration* will keep the output stay in the specific state indefinitely.

There are 3 possible output modes can be assigned to the argument *mode*. Their values and interpretation are listed below.

output mode	interpretation
DOUT_OFF	Turn off the DO immediately for specific duration and then go back on.
DOUT_ON	Turn on the DO immediately for specific duration and then go back off.
DOUT_FLASH	Flash the DO with a specific period indefinitely. The flashing period equals to 2 * *duration*.

returns none

2.7 LED

Number of LEDs on 520 can be used to indicate the system status. They are listed as follows,

Name	Number
LED_F1	12
LED_F2	8
LED_F3	4
LED_F4	0
LED_F5	13
LED_F6	9
LED_F7	5
LED_F8	1
LED_SHT	14
LED_GDRD1	16
LED_GDRD2	17
LED_RDY	18

set_led

purpose Set LED

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 50 miliseconds */
```

example call
```
set_led(LED_RDY, LED_FLASH, 20);  /* set LED_RDY to flash for each
                                     1 second */
```

description 3 modes are supported,

- LED_OFF : off for (duration X 0.05) seconds then on
- LED_ON : on for (duration X 0.05) seconds then off
- LED_FLASH : flash, on then off each for (duration X 0.05) seconds then repeat

returns none

2.8 Keypad

A scanning circuitry of 4 by 8 matrix is utilized on the 520 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	
purpose	Clear the keyboard buffer.
syntax	void clr_kb( );
example call	clr_kb( );
description	The *clr_kb* function clears the keyboard buffer. This function is automatically called by the system program upon power up.
returns	none

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

dis_alpha	
purpose	Disable alphabet key stroke processing.
syntax	void dis_alpha( );
example call	dis_alpha( );
description	The *dis_alpha* 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.
returns	none

get_alpha_enable_state	
purpose	Get the status of the alphabet key stroke processing.
syntax	int get_alpha_enable_state( );
example call	state = get_alpha_enable_state( );
description	This routine gets the current status, enable/disable, of the alphabet key stroke processing.
returns	ALPHA_ENABLE, if the alphabet key stroke processing is enabled. ALPHA_DISABLE, if disabled.

get_alpha_lock_state

purpose	Get alpha lock state information.
syntax	int get_alpha_lock_state( );
example call	state = get_alpha_lock_state( );
description	This function returns an integer indicates the alpha lock status.
returns	ALPHA_LOCK_ON, if alpha lock is on. ALPHA_LOCK_OFF, if alpha lock is off.

getchar

purpose	Get one key stroke from the keyboard buffer.
syntax	char getchar( );

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.

kbhit

purpose	Check whether the keyboard buffer is empty.
syntax	int kbhit( );
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.

peek_kb

purpose	Get multiple key combination from the keypad.
syntax	unsigned long peek_kb( );

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.

set_alpha_lock	
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

2.9 External AT Keyboard

The external AT keyboard is supported on 520 for full alphanumeric keyboard entry and is processed as following.

- The keys which have a corresponding ASCII code value are stored with the ASCII code value when they are pressed.
- The **Caps Lock** key and the **Shift** keys are recognized and are automatically processed.
- The **Num Lock** is always set to the **on** state.
- The **Ctrl** key and the **Alt** key are not supported.
- The function keys (F1 to F12) are mapped to the value 0x80 to 0x8b respectively.
- Up, down, right, and left keys are stored as 0x8c, 0x8d, 0x8e, and 0x8f respectively.
- Other keys that are not mentioned above are not supported.

Scancodes are sent from the keyboard to the machine and stored into a 32-byte FIFO (first-in first-out) buffer. If this buffer is full, keys followed will be ignored. A keyboard handling routines process the code translation (from scan code to ASCII), shift, and capslock keys.

en_extkb	
purpose	Enable external AT keyboard.
syntax	void en_extkb( );
example call	en_extkb( );
description	The en_*extkb* function enables the external AT keyboard. The external keyboard is disabled upon power on. This routine must be called prior to use of the external keyboard. It starts all related background routines and also clears the keyboard buffer.
returns	none

dis_extkb	
purpose	Disable external AT keyboard.
syntax	void dis_extkb( );
example call	dis_extkb();
description	The dis_*extkb* function disables the external AT keyboard. All related background routines are stopped.
returns	none

ext_getchar	
purpose	Get one character from the keyboard buffer of the external AT keyboard.
syntax	char ext_getchar( );
example call	c = ext_getchar( );
description	The *ext_getchar* function reads one character from the keyboard buffer of the external AT keyboard and then removes the character from the keyboard buffer.
returns	The *ext_getchar* function returns the character read from the keyboard buffer. If the keyboard buffer is empty, a null character (0x00) is returned.

capital_lock

purpose	set external keyboard capslock status	
syntax	void capital_lock(int capslock);	
	int capslock;	/* capslock to be set , 1/0 to turn on/off */
example call	capital_lock(1);	/* on capital lock */
description	This routine forces to turn on or off the capslock and is usually used during system initialization.	
returns	none	

num_lock

purpose	set external keyboard numlock status	
syntax	void num_lock(int numlock);	
	int numlock;	/* numlock to be set , 1/0 to turn on/off */
example call	num_lock(1);	/* on num lock */
description	This routine forces to turn on or off the numlock.	
returns	none	

2.10 LCD

This section describes the output routines and the control routines concerning the LCD display.

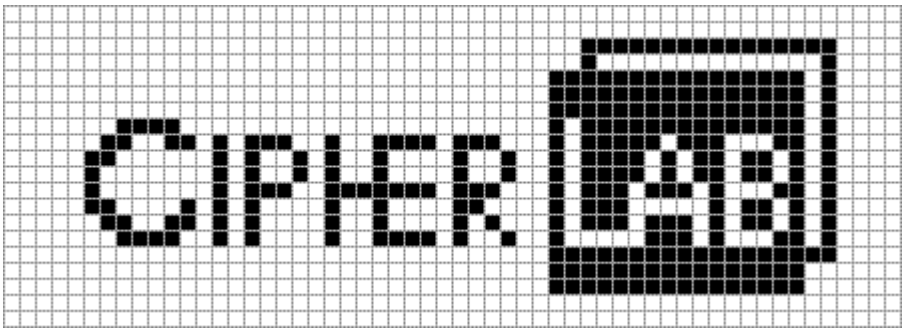
2.10.1 Graphic Display

A 240 X 64 graphics display is used on the 520. There are three different sizes of character fonts supported on 520, i.e., 6X8, 8X16, and 16X24. If 6X8 font is used, there are at most 40 characters by 8 lines. If 8X16 font is used, there are at most 30 characters by 4 lines. And there are at most 15 characters by 2.5 lines if the font 16X24 is used. Different fonts can co-exist on the display. And user can change the font to be used at will.

A coordinate system is used in the cursor movement routines to determine the position of the cursor. The coordinate of the top left position is (0,0) and the bottom left position is assigned with coordinate (0,7). That is, for row positions, it is always from 0 to 7 (each row occupies 8 dots) in regardless of the font being used. Whereas, for the column positions, it will depend on the size of the font being used. For example, if an 8X16 font is used, the bottom right position will be (29,7).

For some graphics display routines (*clr_rect*, *fill_rect*, *show_image*, and *get_image*), the coordinate system used is on dot (pixel) basis. The top left position is (0,0), and the bottom right position is (239,63).

The *show_image* function can be used to display user's logo or other images on the 520 LCD. User needs to allocate an unsigned char array to store the bitmap data for the image. This array begins with the top row of pixels. Each row begins with the leftmore pixels. Each bit of the bitmap represents a single pixel of the image. If the bit is set to 1, the pixel is marked, and if it is 0, the pixel is unmarked. The first pixel in each row is represented by the least significant bit of the first byte in each row. If the image is wider than 8 pixels, the ninth pixel in each row is represented by the least significant bit of the second byte in each row. The following is an example of the company logo and the static unsigned char array to store its bitmap data.



static unsigned char CipherLab_logo[] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xf0, 0xff, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x10, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x0b, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x0b, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x0b, 0x80, 0x07, 0x00, 0x00, 0xf4, 0xff, 0x0b, 0xc0, 0xac, 0x93, 0x77, 0xf4, 0x1d, 0x0b, 0x60, 0xa0, 0x94, 0x90, 0xf4, 0xda, 0x0a, 0x20, 0xa0, 0x94, 0x90, 0xf4, 0xda, 0x0a, 0x20, 0xa0, 0xf3, 0x77, 0x74, 0x17, 0x0b, 0x60, 0xa8, 0x90, 0x30, 0x74, 0xd0, 0x0a, 0xc0, 0xac, 0x90, 0x50, 0x74, 0xd7, 0x0a, 0x80, 0xa7, 0x90, 0x97, 0x04, 0x17, 0x0b, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x0f, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x03, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

2.10.2 Special Font Files

Besides the standard font, 520 can display special characters such as the foreign language characters providing that those font files have been downloaded to 520. 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:

- 520Font-jp.shx : Japanese Characters Font File
- 520Font-sc.shx : Simplified Chinese Characters Font File
- 520Font-tc.shx : Traditional Chinese Characters Font File

Libraries for special fonts:

- 520jplib.lib : including *jpprintf*, *jpputchar*, and *jpputs* functions
- 520sclib.lib : including *scprintf*, *scputchar*, and *scputs* functions
- 520tclib.lib : including *tcprintf*, *tcputchar*, and *tcputs* functions

clr_eol

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

backlit_status

purpose	Get current backlit status.
syntax	int backlit_status( );
example call	backlit_status( );
description	The *backlit_status* function returns the current setting of backlit.
returns	BKLIT_ON, backlit is on. BKLIT_OFF, backlit is off.

clr_rect

purpose	Clear a rectangular area on the LCD display.
syntax	void clr_rect(int left, int top, int width, int height); int left; /* x coordinate of the left most dot of the rectangular to be cleared */ int top; /* y coordinate of the top most dot of the rectangular to be cleared */ int width; /* the width in dots of the rectangular to be cleared */ int height;/* the height in dots of the rectangular to be cleared */
example call	clr_rect(12,8,40,8);
description	The *clr_rect* function clears an 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

clr_scr

purpose Clear LCD display.

syntax void clr_scr( );

example call clr_scr( );

description The *clr_scr* 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

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 left most dot of the rectangular to
               be filled */
int top;    /* y coordinate of the top most dot of the rectangular to
               be filled */
int width; /* the width in dots of the rectangular to be filled */
int height; /* the height in dots of the rectangular 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( );

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( );

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

FONT16X24 : if 16X24 font is used

get_image

purpose Read the bit map pattern of a rectangular area on the LCD display.

syntax
```
void get_image(int left, int top, int width, int height, unsigned char *pat);
int left;   /* x coordinate of the left most dot of the rectangular */
int top;    /* y coordinate of the top most dot of the rectangular */
int width;  /* the width in dots of the rectangular */
int height; /* the height in dots of the rectangular */
unsigned char *pat; /* the buffer where the bit map will be copied to */
```

example call
```
get_image(12,32,60,16, buf);
```

description The *get_image* function copies the bit map pattern of a rectangular area on the LCD display whose top left position and size are specified by *left, top, width,* and *height* to the buffer specified by *pat*. The cursor position is not affected after the operation.

returns none

GetVideoMode

purpose Get current display mode information.

syntax
```
int GetVideoMode( );
```

example call
```
if (GetVideoMode( ) = = VIDEO_NORMAL) puts("Normal Mode");
```

description The *GetVideoMode* function returns the information about the display mode.

returns 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

purpose Move cursor to new position.

syntax
```
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 */
```

example call
```
gotoxy(10,0);       /* move to the 11th column of the first line */
```

description The *gotoxy* function moves the cursor to a new position whose coordinate is specified in the argument *x_position* and *y_position*.

returns Normally the *gotoxy* function will return an integer value of 1 when operation completes. In case of LCD fault, 0 is returned to indicate error.

lcd_backlit

purpose Set LCD backlight

syntax
```
void lcd_backlit(state);

int state;          /* LCD backlight state 0 / 1 (off / on) */
```

example call
```
lcd_backlit(1);     /* turn on LCD backlight */
```

description The *lcd_backlit* fine tunes the LCD backlight on or off depends the value of *state*. The backlight will be on if *state* is 1, off if 0.

returns none.

printf

purpose Write character strings and values of C variables in a specified format to the LCD display.

syntax
```
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 */
```

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 appears 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

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 The *SetCursor* function has no return values.

SetFont

purpose Select the font to be used afterwards.

syntax
```
int SetFont(int font);
int font;          /* integer representing font to be use afterwards */
```

example call
```
SetFont(FONT8X16);
```

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

FONT6X8 : 6X8 font

FONT8X16 : 8X16 font

FONT16X24 : if 16X24 font is used

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 The *SetVideoMode* function has no return values.

show_image

purpose Put a rectangular bit map to the LCD display.

Syntax
```
void show_image(int left, int top, int width, int height, unsigned char
*pat);
int left;       /* x coordinate of the left most dot of the rectangular */
int top;        /* y coordinate of the top most dot of the rectangular */
int width;    /* the width in dots of the rectangular */
int height;  /* the height in dots of the rectangular */
unsigned char *pat; /* the buffer that hold the bit map to be displayed */
```

example call
```
show_image(35, 5, 52, 24, CipherLab_logo[]);
```

description The *showet_image* function displays a rectangular bit map specified by *pat* to the LCD display. The rectangular's top left position and size are specified by *left, top, width,* and *height*. The cursor position is not affected after the operation.

returns none

wherex

purpose Get x-coordinate of the cursor location.

syntax
```
int wherex();
```

example call
```
x_position = wherex();
```

description The *wherex* function determines the current x-coordinate location of the cursor.

returns The *wherex* function returns the x-coordinate of the cursor location.

wherexy

purpose	Get x-coordinate and y-coordinate of the cursor location
syntax	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 */
example call	wherexy(&x_position, &y_position);
description	The *wherexy* function copies the value of x-coordinate and y-coordinate of the cursor location to the variables whose address is specified in the arguments *column* and *row*.
returns	none

wherey

purpose	Get y-coordinate of the cursor location.
syntax	int wherey();
example call	y_position = wherey();
description	The *wherey* function determines the current y-coordinate location of the cursor.
returns	The *wherey* function returns the y-coordinate of the cursor location.

2.11 Power

This section describes the power management functions for 520. The **get_lnpwr**, and **get_bat1** and **get_bat2** functions are used to monitor the voltage level of the external power supply and the two backup batteries.

2.11.1 Backup Batteries

If the optional backup battery is installed, the machine can still be operational by using the power of the backup batteries when there is no external power supplied. The battery life may be shortened or even damaged if the battery was deeply drained. So it is important to constantly monitor the voltage level of the batteries and shut down the system before the voltage level falls too low.

battery_status	
purpose	Check the status of the battery usage.
syntax	int battery_status(void);
example call	bat_status = battery_status();
description	This function can be used to the status of the battery usage.
returns	0, AC power is used. 1, battery is used. 2, battery is used and the backlit is forced off because of low voltage. 3, battery low and system may not function correctly.

get_lnpwr	
purpose	Get voltage level of the external power supply.
syntax	unsigned get_lnpwr( );
example call	if (get_lnpwr( ) < 11000) puts(“Lose External Power”);
description	The *get_lnpwr* function reads the voltage level of the external power in units of mV.
returns	The *get_lnpwr* function returns the voltage level of the external power in units of mV (mili-volt).

get_bat1, get_bat2	
purpose	Get voltage level of the backup batteries.
syntax	unsigned get_bat1( ); unsigned get_bat2( );
example call	bat1 = get_bat1( ); bat2 = get_bat2( );
description	The *get_bat1* function reads the voltage level of the first backup battery in units of mV and the *get_bat2* function reads the voltage level of the second backup battery. If external power is supplied, the reading value is meaningless.
returns	The *get_bat1* and *get_bat2* functions return the voltage level of the respective backup battery in units of mV (mili-volt).

2.12 Communication Ports

There are totally three communication ports on 520, namely COM1, COM2 and COM3. The COM3 is fixed to be RS232. And COM1 and COM2 could be RS232 if the optional RS232 board are installed on 520 for these two ports. 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,

2.12.1 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

2.12.2 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.

2.12.3 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.

2.12.4 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

purpose Clear receive buffer

syntax
```
void clear_com(int port);
int port;      /* port to be opened, from 1 to 3 */
```

example call
```
clear_com(1);/* clear COM1 receive buffer */
```

description 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.

returns none

close_com

purpose Disable specified RS232 port

syntax
```
void close_com(int port);
int port;          /* port to be opened, from 1 to 3 */
```

example call
```
close_com(1);      /* close com1 */
```

description The close_com disables the RS232 port specified.

returns none

com_cts

purpose Get CTS level

syntax
```
int com_cts(int port);
int port;          /* port to be opened, from 1 to 3 */
```

example call
```
if (com_cts(1) == 0) printf("COM1 CTS is space");
else printf("COM1 CTS is mark");
```

description This routine is used to check current CTS level.

returns
```
1, if CTS is in mark state
0, if CTS is in space state
```

com_eot

purpose See if any COM port transmission in process (End Of Transmission)

syntax
```
int com_eot(int port);
int port;      /* port to be opened, from 1 to 3 */
```

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

CommType

purpose See which type of interface board is installed at the port specified.

syntax
```
int CommType(int port);
int port;      /* port to be checked, 1 or 2. */
```

example call
```
if (CommType(1)==2) {          /* wait till prior transmission completed */
      printf("Star Node interface is installed at COM1.");
}
```

description COM1 and COM2 .

returns 0, RS-232 or 20mA Current Loop
1, RS-485 Half Duplex
2, Star Node
3, RS-485 Full Duplex

com_overrun

purpose See if overrun error occurred

syntax
```
int com_overrun(int port);
int port;      /* port to be opened, from 1 to 3  */
```

example call
```
if (overrun(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 3 */
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 3 */
```

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 3 */
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, 0x0a);
/* open com1 to 9600, 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 3 */
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

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 3 */ 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

2.13 RS485

If RS485 communication is to be used on 520, an optional RS485 board can be installed on COM1 or COM2.

2.13.1 Parameter

The communication parameters are fixed as follows,

- Baud Rate : 76.8 K bps
- Data Bit : 9
- Parity : None

To avoid collision and ensure data integrity, special communication protocol and flow control is utilized and are described below,

2.13.2 Station ID

The RS485 is a multi-drop communication standard which allows up to 30 stations (expandable by use of repeater) to be linked on the same net. Each station must be assigned with an unique station ID for proper communication. This one-byte station ID ranges from 1 to 255. Station ID 0 is reserved for broadcasting purpose only and can not be assigned to any station.

2.13.3 Master/Slave

One and only one of the stations on the link is assigned to be the bus arbitrator (master), while others are listeners (slaves). To avoid collision, the master is the only station that can start a talk and the specified listener can respond to this action by sending an echo back to the master. That is, a talk is always started by the master and ended with an echo from the specified slave. To improve efficiency, echo is done by the interrupt routine.

2.13.4 Packet

Communication is done by transactions of packets. A packet is composed of several characters and is the only meaningful communication unit. That is, a full packet must be transmitted/received to be correctly parsed.

Processing of the packet has been done by background routines and is not really a concern for the C programmer. The materials herein serves as reference only.

Compositions of a packet are listed and explained below,

DLSs..sK

where,

D : destination station ID
L : length of the packet in bytes
S : source station ID
s..s : string to be transferred
K : checksum

2.13.5 Master

- Polling

  Since the communication always starts from the master side. It is the master's responsibility to poll slave stations constantly to see if anything to be taken care of or not. To do so, a null string was sent, that is, the (s..s) in the packet is null. Whereas the slave station echoes (by background routine) back the status word and the message string (if available)to indicate its current status.

  Master send packet DLSK

  where,

D = slave ID
S = master ID

Slave echoes packet DLSWWs..sK

where,
D = master ID
S = slave ID
WW=2 bytes status word
s..s = message if any

- Command/message

  If a command or message is to be sent to a slave station. The slave always echoes with status word only. That is if the (s..s) from the master side is not null, it is treated as a command transaction and only status word is returned.

  Master send packet DLSs.sK

  where,
  D = slave ID
  s..s = command/message string
  S = master ID

  Slave echoes packet DLSWWK

  where,
  D = master ID
  S = slave ID
  WW=2 bytes status word

  The echo from the slave at this time only indicates that the command/message packet is successfully received. To make sure that the command/message is correctly interpreted/processed, the master can then poll the slave station to get the completion result from the slave. Note that if the destination from the master is 0, it is a broadcasting command and is accepted by all slave stations. However, no echo is done since this will cause data collision.

2.13.6 Slave

As the transaction all start from the master side. To improve efficiency, the slave side echoes to the transaction immediately following receipt of a packet by background routines (interrupt). That is, all supported routines for slave **DO NOT** initiate any communication activities, it must wait till the master sends out a valid packet and then echo. All these routines simply modify some internal flags and buffers.

2.13.7 Status Word

A 16-bit RS485 status word was declared by background routines. Bit 0 & 1 are reserved for transaction use and is manipulated by background routines. Whereas others are free to be defined for C program use. This word is initialized to 0 when RS485 port is opened.

```
extern  int   RS485_STW;
```

- bit 0 : set to 1 once a complete packet is received and can be used to see if this station is on-lined or not (granted by the master).
- bit 1 : set to 1 if bus contention is encountered. During transmission, the rolled-back character is verified. If fault, the transmission is stopped at once and this bit is set to 1 to indicate this error.

Other bits can be used to show current status, for example, (access control)

- bit 2 : successful ID scanned (to ask master station to verify this card)
- bit 3 : door lock status (locked or not)
- others depend on application need

Once this slave is polled by the master, the status word is returned. And if the bit 1 is set to 1, the master can send another command to read this ID, process the ID. And then send another command to instruct the slave if this is a valid ID or not (open door or not).

2.13.8 RS485 Processing

Unlike RS232 communication, special care must be taken in handling a multi-drop communication like RS485. The recommended flow are as follows,

- Master polling

  1) write_485(ID, ""); /* null string */
  2) read_485();
  3) if string echoed, check ID, if correct then OK, END, else fault, END
  4) if time out then time out error, END
  5) goto step 2 and repeat

- Master command

  Being an arbitrator, it is the master's responsibility to ensure successful transaction. That is, messy retry must be included in the procedure. However, as the slave echoes during the interrupt routine, the recommended time out is only 10-15 ms. Also, to improve performance, the master must try to poll slaves as fast as it can which however overloads the master station. To overcome this, the procedures described above are separated into many small steps as can be seen in the sample program.

  1) write_485(ID, command_string); /* non-null command string */
  2) read_485();
  3) if string echoed, check ID, if correct then go to step 6, else fault, END
  4) if time out then time out error, END
  5) go to step 2 and repeat
  6) Poll slave to get result as the previous one

- Slave

  1) if read_485(s) == 0 then END
  2) parse command string s
  3) prepare result write_485(result);

Close485	
purpose	Close RS485 communication port
syntax	int Close485(int port); int port; /* port to be accessed, 1 or 2 */
example call	Close485(2);
description	this routine disables RS485 port.
returns	

Echo485	
purpose	Enable or disable the RS485 receive interrupt when it's transmitting.
syntax	void Echo485(int port, int state); int port; /* port to be accessed, 1 or 2 */ int state; /* 0/1 to disable/enable echo */
example call	Echo485(0);/* Echo off */
description	If echo is on, the transmitted data will be received and put into the receive buffer of the sender.

returns none

Open485

purpose Open RS485 communication port

syntax
```
int Open485(int port, int master, int ID);
int port;       /* port to be accessed, 1 or 2 */
int master;     /* 1/0, master/slave station */
int ID;         /* station ID from 1 to 255 */
```

example call
```
Open485(2, 0, 10);          /* COM2, slave station #10 */
```

description this routine enables RS485 port and set its station ID and communication attribute (master/slave).

returns

Read485

purpose Read RS485 packet received

syntax
```
int Read485(int port, char *s);
int port;   /* port to be accessed, 1 or 2 */
char *s;    /* string pointer where received packet to be copied */
```

example call
```
char s[50];
if (Read485(2,s) > 0) {
    printf_us("String %s received via COM2", s+3);
}
```

description The background interrupt routines handle receiving of the RS485. That is, to verify the ID (destination), length, checksum and so on. Upon receipt of a successful packet, an internal flag is set and the whole packet is stored in the receiving buffer. This flag disables further receiving operation until the received packet is read by this routine (which in the fact, clear this flag). The whole packet (except checksum) described previously is copied to the string pointer (s). The source ID is also returned and can be used to see if this is a broadcasting command or not.

returns if available, string length of the packet
0, not available

Write485 (for master)

purpose Send a string to slave station

syntax
```
int Write485(int port, int ID, char *s);
int port;   /* port to be accessed, 1 or 2 */
int ID;     /* destination slave ID*/
char *s;    /* string to be sent */
```

example call
```
Write485(2, 5, "READ");    /* send string "READ" to slave #5 via COM2*/
```

description the routine is used by master station to send a string out to designated slave station. The RS485 transmission starts immediately when this routine is called.

returns

Write485 (for slave)

purpose Prepare echo string to master station

syntax
```
int Write485(int port, char *s);
int port;/* port to be accessed, 1 or 2 */
char *s;     /* string to be sent */
```

example call
```
Write485(2, "DONE");    /* echo string "DONE" when polled via COM2 */
```

description Unlike master station, this routine does not initiate any transmission. Instead, it copies the string to an internal buffer and sets a flag. Later, when this station is polled, the stored string is sent back to the master.

This flag acts as follows,

- set, when this routine is called
- clear, on the following conditions,
    1) continuously polled for 4 times, up to 3 retries allowed.
    2) polled and then packet for other station is acknowledged, job for me is completed
    3) a non-null packet for this slave is received, new job for me

returns

2.14 Memory

Flash and SRAM manipulation routines are described in this section.

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

init_free_memory	
purpose	Initialize file space.
syntax	void init_free_memory( );
example call	init_free_memory( );
description	The *init_free_memory* 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.
returns	This function has no return values.

MCDSize	
purpose	Check the memory size of the optional memory card .
syntax	int MCDSize(void);
example call	mcd_size = MCDSize(); printf("The memory card is %d KB.", mcd_size);
description	This routine is used to check the memory size of the optional memory card quickly. The file system will not be destructed by this routine.
returns	The memory card size in units of KB.

test_main_mem	
purpose	Main memory read/write test routine.
syntax	int test_main_mem(void);
example call	if (test_main_mem()==1) printf("Main memory test OK!");
description	This routine is used to test the main memory. The file system will be destructed while it's testing.
returns	1, if test OK. 0, if test fails.

test_MCD	
purpose	Memory card read/write test routine.
syntax	int test_MCD(void);
example call	memory = test_MCD( ); printf("Memory Card Size = %d KB", memory);
description	This routine is used to test the optional memory card. The test takes a few seconds depending on the size of the memory card. The file system will be destructed while it's testing.
returns	The memory card size in units of KB.

2.15 Miscellaneous

DownLoadPage	
purpose	Enter the 'Download' mode
syntax	void DownLoadPage();
example call	open_com(1, 0x08); /* 38400, N, 8 */ DownLoadPage(); /* enter download mode */
description	The *DownLoadPage* function is used to set 520 to the download mode. The Download page will show up and user can select the communication port and the baud rate for program download. And then write the new program to the flash memory. At the end, this routine jumps to the system start point and the system will re-initialize again.
returns	none

prc_menu	
purpose	Process the menu
syntax	void prc_menu(MENU* menu);
example call	prc_menu(&Msystem); /* process Msystem menu */ struct MENU Msystem;
description	The *prc_menu* function is used to process the menu. SMENU and MENU structure are defined in "520lib.h". Users can define their MENU structure and use *prc_menu* function to build a hierarchy menu-driven user interface.
returns	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 sprintf is supported, for formatted output to display, please refer to Syntech Library "LCD".
- Formatted Input: Only sscanf 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 charcater 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) hyprebolic 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<=0, or if x<0 and y is not an integer
- sqrt(x) x, x0

- 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

520 Data Terminal comes with a real-time kernel (μC/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.

μC/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. μC/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 520 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 μC/OS related functions are discussed as follows.

OS_ENTER_CRITICAL

purpose Disable the processor's interrupt

syntax void OS_ENTER_CRITICAL(void);

example call
```
OS_ENTER_CRITICAL();
        … /* user code */
OS_EXIT_CRITICAL();
```

description 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 *OS_ENTER_CRITICAL* 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 *OS_EXIT_CRITICAL* function must be used in pairs.

returns none

OS_EXIT_CRITICAL

purpose Ensable the processor's interrupt

syntax void OS_EXIT_CRITICAL(void);

example call
```
OS_ENTER_CRITICAL();
        … /* user code */
OS_EXIT_CRITICAL();
```

description The function executes in about 5 CPU clock cycles. This function and *OS_ENTER_CRITICAL* function must be used in pairs.

returns none

OSSemCreate

purpose Create and initialize a semaphore

syntax OS_EVENT OSSemCreate(unsigned *value*);

where, OS_EVENT, a data structure to maintain the state of an event called Event Control Block (ECB), is defined as below,

```
typedef struct os_event {
    unsigned char  OSEventTbl[8];  /* Group corresponding to tasks
                                      waiting for event to occur */
    unsigned char  OSEventGrp;     /* List of tasks waiting for event to
                                      occur */
    long  OSEventCnt;              /* Count of used when event is a
                                      semaphore */
    void  *OSEventPtr;             /* Pointer to message or queue
                                      structure */
} OS_EVENT;
```

value is the initial value of the semaphore. The initial *value* of the semaphore is allowed to be between 0 and 32767.

example call sem_time = OSSemCreate(1); /* create a semaphore sem_time and the initial value of sem_time is set to 1. */

description 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

purpose Create a task

syntax unsigned char OSTaskCreate(void (**task*)(void *pd), void **pdata*, unsigned char **pstk*, unsigned long stk_size, unsigned char *piro*);

where, *task* is a pointer to the task's code.

pdata is a pointer to an optional data area which can be used to pass parameters to the task when it is created.

pstk 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).

prio is the task priority. A unique priority number must be assigned to each task and the lower the number, the higher the priority.

example call 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*);

description This function allows an application to create atask. The task is managed by μC/OS. Tasks can be created prior to the start of multitasking or by a running task.

returns OS_PRIO_EXIST,if the requested priority already exist.
OS_NO_ERR,if the function was successful.

OSTaskDel

purpose Delete a task

syntax unsigned char OSTaskDel(unsigned char *piro*);
where, *prio* is the task priority. A unique priority number must be assigned to each task and the lower the number, the higher the priority.

example call OSTaskDel(10); /* delete a task with priority number 10 */

description 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.

returns

OS_TASK_DEL_IDLE	if the task to delete is an idle task.
OS_TASK_DEL_ERR	if the task to delete does not exist.
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

5 Sample Programs

5.1 User0

5.1.1 Program Description

The sample program in the user0 subdirectory is a data collection program for 520 with a DBF file. User can download a predefined DBF to the 520, scan the ID label to modify the data of the records of the DBF, and/or upload the data to the host computer. This program communicate with the host via RS-232 (19200, no parity, 8 data bit, 1 stop bit, no handshake).

When downloading the DBF, it waits for the download signal "CIPHER\r" and replies "ACK" to start to receive the DBF until it receive "\r\r" from the host. The DBF is consisted by 40-byte-long records. The first 10-byte of each record works as the primary key "ID" of the DBF. The second, third, and fourth 10-byte are the "Name", "Description", and "Quantity" of the record.

When scanning the data, if the scanned data can be found in the pre-downloaded DBF (has the same ID), all the fields of the record will be shown on the display, and the cursor will move to the last field. Otherwise, "Invalid ID!" will show on the display and the scanned ID will be discarded. User can press "Enter" key to modify the value of the fourth (Q'ty) field by clicking the number/alphabet of the keypad, or press "ESC" to scan another ID.

When uploading data, the 520 will keep sending the upload signal "CIPHER\r" until it gets the response "ACK" from the host. If so, it will send the records of the DBF to the host sequentially.

The 520load directory contains a Visual Basic program for a sample back end program to download/upload DBF to/from 520. The source code of the back end program is included as well. This program works with the mentioned user0 program. To expand the functionality, users can modify the sample program to their needs. "test.data", a sample DBF to download to the terminal, is also included in the directory.

The library routines and system variables used in user0.c are listed as follows,

add_member()	close_com()	close_DBF()	clr_eol()	clr_scr()
CodeBuf	creat_DBF()	create_index()	Decode()	delete_member()
en_alpha()	get_member()	getchar()	gotoxy()	has_member()
InitScanner1()	lseek_DBF()	member_in_DBF()		on_beeer()
open_com()	prc_menu()	printf()	pSetting	read_com()
SetCursor()	SetVideoMode()		sys_sec	write_com()

5.1.2 Source Code

5.1.2.1 User0.lnk

```
/*******************************************/
/*    user0.lnk                            */
/*******************************************/
-lm -lg  -ll

user0.rel

..\lib\520lib.lib
..\lib\c900ml.lib

/*******************************************/
/*  User could provide desirable values to */
/*  the following two variables.           */
/*******************************************/
___MainStackSize__ = 0x001000;
HeapSize  = 0x000100;

/*******************************************/
/*  Do not modify anything beyond this line  */
/*******************************************/
memory
{
    RAM       : org = 0x400100, len = 0x01ff00
    ROM       : org = 0xf80000, len = 0x070000
    IO   : org =  0x100000, len =   0x100000
}

sections
{
    code  org =   0xf80000 : {
   *(f_head)
   *(f_code)
    } >  ROM

    sys_area org = 0x400100 : {
   *(f_bcr)
   ..\lib\520lib.lib(f_area)
   ..\lib\c900ml.lib(f_area)
    } >  RAM

    data org=org(code)+sizeof(code) addr=org(sys_area)+sizeof(sys_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

    area org = addr(xcode) + sizeof(xcode) : {
   . += ___MainStackSize__;
   . += HeapSize;
   *(f_unshare)
   *(f_area)
    } >  RAM
}

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

/* End */
```

5.1.2.2 User0.c

```
/*****************************************************************************/
/* program :   user0.c                                                       */
/*                                                                           */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <520lib.h>

struct SMENU MSystem;
struct SMENU_ENTRY EInput;
struct SMENU_ENTRY EDownload;
struct SMENU_ENTRY EUpload;
static void FInput(void);
static void FDownload(void);
static void FUpload(void);

static struct SMENU MSystem={
        3,
        1,
        0,
        "CipherLab 520",
        {
                &EInput,
                &EDownload,
                &EUpload
        }
};

static struct SMENU_ENTRY EInput={
        0,3,
        "1. Input Data",
        FInput,
        0
};

static struct SMENU_ENTRY EDownload={
        0,5,
        "2. Download DBF",
        FDownload,
        0
};

static struct SMENU_ENTRY EUpload={
        0,7,
        "3. Upload Data",
        FUpload,
        0
};

extern void* pSetting;

static const unsigned char OSScannerDefault[23] = {
    0xdf, 0xff, 0xff, 0x00, 0x00,   // enable all symbologies except CIP39
    0x35, 0x54, 0x80, 0x00, 0x10,
    0x3f, 0x00, 0xc0, 0x01, 0xc0,
    0x01, 0xc0, 0x01, 0xc0, 0x01,
    0x00,   // Auto-off mode for Scanner 1 & 2
    0x0a, 0x0a // Time-out: 10 sec
};

#define BYTE unsigned char

static const int KEY_CLICK[] = {19,6,0,0};
static const int OK[] = {16,6,0,0};
static const int ERR[] = {20,10,16,10,20,10,16,10,0,0};
static const int POR[] = {16,10,0,5,16,5,0,5,16,5,0,0};
#define beep_click()    on_beeper(KEY_CLICK)
#define beep_ok()       on_beeper(OK)
#define beep_err()      on_beeper(ERR)
#define beep_por()      on_beeper(POR)
#define SP3          3
#define ID_LENGTH    10     /* DBF Key Length */
#define RECORD_LENGTH   40  /* DBF Record Length */
```

```
int ID_dbf;                 /* DBF File Handle */
char InputBuf[128];
char OutputBuf[128];
int InputCnt;
char Data[4][ID_LENGTH+1];
char Record[RECORD_LENGTH];
static char ACK[] = "ACK\r";
static char NAK[] = "NAK\r";
static char OVER[] = "OVER\r";
static char FAIL[] = "FAIL\r";
static char DownloadCmd[] = "CIPHER";
static char UploadCmd[] = "CIPHER\r";

unsigned long UpDown_sec;

void main(void)
{
    BYTE c;

    beep_por();
    SetFont(FONT8X16);
    clr_scr();
    pSetting = OSScannerDefault;
    InitScanner1();
   InitScanner2();
   ID_dbf=create_DBF("VALID_ID", RECORD_LENGTH);
    create_index(ID_dbf, 1, 0, ID_LENGTH);
    for (c=1;c<17;c++) close_DBF(c);
   if ((ID_dbf = open_DBF("VALID_ID")) <= 0) {
      clr_scr();
      printf("System Fails!");
      while(1);
    }
    en_alpha(1);

    while(1) {
      prc_menu(&MSystem);
    }
}

void FInput(void)
{
   char c,*s;
   int i,j;

loop:
   clr_scr();
   gotoxy(0,1);
   printf("ID  : ");
   gotoxy(0,3);
   printf("Name: ");
   gotoxy(0,5);
   printf("Des.: ");
   gotoxy(0,7);
   printf("Q'ty: ");
   gotoxy(6,1);
   SetCursor(CURSOR_ON);
   while(1) {
      if (Decode()) {
         SetCursor(CURSOR_OFF);
         if (strlen(CodeBuf)<=ID_LENGTH) {
            beep_ok();
            gotoxy(6,1);
            clr_eol();
            printf("%s",CodeBuf);
            s=CodeBuf;
            if (check_member(s)) {
               gotoxy(6,7);
               SetCursor(CURSOR_ON);
               while(1) {
                  if (c=getchar()) {
                     if (c == KEY_ESC) {
                        SetCursor(CURSOR_OFF);
                        goto loop;
                     }
                     else if (c == KEY_CR) {
                        clr_eol();
                        scan_new_data(s);
                     }
```

```
                    else beep_err();
                 }
              }
           }

           else {
              gotoxy(0,3);
              clr_eol();
              gotoxy(0,5);
              clr_eol();
              gotoxy(0,7);
              clr_eol();
              gotoxy(9,5);
              printf("Invalid ID!");
              while(getchar());
              while(!getchar());
              goto loop;
           }
        }
     }
     if (c=getchar()) {
        if (c == KEY_ESC) break;
     }
   }

   return;
}

/*****************************************************/
/*    download DBF from PC via RS232                 */
/*****************************************************/
void FDownload(void)
{
   int i=0;
   char c;
   char *s;

   clr_scr();
   gotoxy(7,3);
   printf("Waiting for DBF.");
   open_com(SP3,BAUD_19200 | DATA_BIT8 | PARITY_NONE | HANDSHAKE_NONE);
   UpDown_sec = sys_sec;
   while(1) {
      if (read_one_line(SP3,InputBuf,&InputCnt)) {
         if (!strcmp(InputBuf,DownloadCmd)) {
            write_com(SP3,ACK);
            break;
         }
         write_com(SP3,NAK);
         clr_scr();
         gotoxy(8,3);
         printf("Download Fail!");
         while(getchar());
         while(!getchar());
         close_com(SP3);
         return;
      }
      if (abs(UpDown_sec - sys_sec) > 20) {
         clr_scr();
         gotoxy(10,3);
         printf("Time Out!");
         while(getchar());
         while(!getchar());
         close_com(SP3);
         return;
      }
      if (c=getchar()) {
         if (c == KEY_ESC) {
            close_com(SP3);
            return;
         }
      }
   }
   clr_scr();
   gotoxy(8,3);
   printf("Downloading...");
   UpDown_sec = sys_sec;
   while(1) {
      if (read_one_line(SP3,InputBuf,&InputCnt)) {
```

```
         if (strcmp(InputBuf,"")==0) {
            write_com(SP3,ACK);
            beep_ok();
            clr_scr();
            gotoxy(7,3);
            printf("Download Finish!");
            while(getchar());
            while(!getchar());
            break;
         }
         else {
            s=InputBuf;
            if (has_member(ID_dbf,1,s)!=0) {
               delete_member(ID_dbf,1);
               add_member(ID_dbf,s);
               write_com(SP3,ACK);
            }
            else {
               add_member(ID_dbf,s);
               write_com(SP3,ACK);
            }
            UpDown_sec = sys_sec;
         }
         if (abs(UpDown_sec - sys_sec) > 5) {
            write_com(SP3,NAK);
            clr_scr();
            gotoxy(8,3);
            printf("Download Fail!");
            while(getchar());
            while(!getchar());
            close_com(SP3);
            return;
         }
      }
   }
   close_com(SP3);
   return;
}

/*****************************************************/
/*    upload data to PC via RS232                    */
/*****************************************************/
void FUpload(void)
{
   int i;
   int Member;
   unsigned long Up_sec;
   char c;

   clr_scr();
   gotoxy(10,3);
   printf("Upload...");
   open_com(SP3,BAUD_19200 | DATA_BIT8 | PARITY_NONE | HANDSHAKE_NONE);
   write_com(SP3,UploadCmd);
   UpDown_sec = sys_sec;
   Up_sec = sys_sec;
   while(1) {
      if(read_one_line(SP3,InputBuf,&InputCnt)) {
         if (strcmp(InputBuf,"ACK")==0) break;
         if (strcmp(InputBuf,"NAK")==0) {
            clr_scr();
            gotoxy(9,3);
            printf("Upload Fail!");
            while(getchar());
            while(!getchar());
            close_com(SP3);
            return;
         }
         else {
            write_com(SP3,UploadCmd);
            Up_sec = sys_sec;
         }
      }
      if (abs(UpDown_sec - sys_sec) > 20) {
         clr_scr();
         gotoxy(9,3);
         printf("Upload Fail!");
         while(getchar());
         while(!getchar());
```

```
         close_com(SP3);
         return;
      }
      if (abs(Up_sec - sys_sec) > 2) {
         write_com(SP3,UploadCmd);
         Up_sec = sys_sec;
      }
      if (c=getchar()) {
         if (c == KEY_ESC) {
            close_com(SP3);
            return;
         }
      }
   }

   lseek_DBF(ID_dbf,1,0L,1);
   Member= member_in_DBF(ID_dbf);
   get_member(ID_dbf,1,OutputBuf);
   write_com(SP3, OutputBuf);
   while(com_eot(SP3)==0x00);
   write_com(SP3,"\r");
   lseek_DBF(ID_dbf,1,1L,0);
   Up_sec = sys_sec;
   i=1;
   while(i<Member) {
      if(read_one_line(SP3,InputBuf,&InputCnt)) {
         if (strcmp(InputBuf,"ACK")==0) {
            get_member(ID_dbf,1,OutputBuf);
            write_com(SP3, OutputBuf);
            while(com_eot(SP3)==0x00);
            write_com(SP3,"\r");
            lseek_DBF(ID_dbf,1,1L,0);
            Up_sec = sys_sec;
            i++;
            }
         else {
            clr_scr();
            gotoxy(9,3);
            printf("Upload Fail!");
            while(getchar());
            while(!getchar());
            close_com(SP3);
            return;
         }
      }
      if (abs(Up_sec - sys_sec) > 2) {
         clr_scr();
         gotoxy(9,3);
         printf("Upload Fail!");
         while(getchar());
         while(!getchar());
         close_com(SP3);
         return;
      }
   }
   write_com(SP3,OVER);
   beep_ok();
   clr_scr();
   gotoxy(8,3);
   printf("Upload Finish!");
   while(getchar());
   while(!getchar());
   close_com(SP3);
   return;
}

/*****************************************************/
/*    read  one record from RS232                    */
/*****************************************************/
read_one_line(port, s, cnt)
    int      port;   /* RS232 port number */
    char    *s;       /* input buffer */
    int      *cnt;   /* input count */
{
    char c;
    if ( read_com(port, &c) == 0x00) return(0);
    if (c != KEY_CR) {
      if (*cnt < RECORD_LENGTH) {
         *(s+(*cnt)) = c;
```

```
        (*cnt)++;
      }
      return(0);      /* if too long, ignore */
    }
    *(s+(*cnt))   = 0x00;
    *cnt = 0x00;
    write_com(port, "");    /* stop transmission */

    return(1);
}

/*****************************************************/
/*    check if the record existed                    */
/*****************************************************/
check_member(s)
   char *s;
{
   int i,j;

   for (i=strlen(s);i<ID_LENGTH;i++) {
       *(s+i)=' ';
   }

   if (has_member(ID_dbf,1,s)) {
      get_member(ID_dbf,1,s);
      for (i=1;i<4;i++) {
         for (j=0;j<ID_LENGTH;j++) {
            Data[i][j] = *(s+i*ID_LENGTH+j);
         }
         Data[i][ID_LENGTH]=0x00;
      }
      gotoxy(6,3);
      clr_eol();
      printf("%s",Data[1]);
      gotoxy(6,5);
      clr_eol();
      printf("%s",Data[2]);
      gotoxy(6,7);
      clr_eol();
      printf("%s",Data[3]);
      return(1);
   }
   else return(0);
}

/*****************************************************/
/*    scan new Q'ty data                             */
/*****************************************************/
scan_new_data(s)
   char *s;
{
   char c;
   int i=0,j;

   clr_eol();
   while(getchar());
   while(1) {
      while((c=getchar())==0x00);
      switch (c) {
         case KEY_ESC:  return;
         case KEY_BS:   i--;
                        if (i<0) i=0;
                        gotoxy(6+i,7);
                        clr_eol();
                        *(s+3*ID_LENGTH+i) = 0x00;
                        break;
         case KEY_CR:   *(s+3*ID_LENGTH+i) = 0x00;
                        i = ID_LENGTH;
                        break;
         default:    *(s+3*ID_LENGTH+i) = c;
                        gotoxy(6+i,7);
                        printf("%c",c);
                        i++;
      }
      if (i==ID_LENGTH) break;
   }
   gotoxy(6,7);
   beep_ok();
   if (has_member(ID_dbf,1,s)!=0) delete_member(ID_dbf,1);
```

```
   add_member(ID_dbf,s);
   return;
}

/*****************************************************/
/*    Initialize the LCD screen                      */
/*****************************************************/
init_scr()
{
    SetCursor(CURSOR_OFF);
    SetVideoMode(VIDEO_NORMAL);
    clr_scr();
}

/* END */
```