# Syntech Programmable Terminal Programmer's Guide

## V1.1

**November 21, 1995**

# Table Of Contents

# 1. Revision History

- V1.1, Sept. 21, 1995

  1) file function *close_all()* is not supported and is deleted from this manual. Upon power-up, all the files are closed during system initialization.
  2) file function *file_length()*, correct name is *filelength()*
  3) file function *get_member()* is added into function list
  4) speaker function *volume()* description is added.
  5) incorrect return values descriptions for : *get_member()*, *has_member()*, *add_member()*, *delete_member()*
  6) new function *clone()* added

# 2. Preface

This programmer's guide provides a step by step description in developing application programs for Syntech Programmable Data Terminals such as 510, 610 and 201. Assumption was made that the programmer has prior knowledge of the C language. Also as the compiler is mostly ANSI compatible. Standard ANSI library routines will be briefly described only, as they can be found in many ANSI C related literature.

This manual is divided into 3 major parts,

- Development process : a step by step procedure from compiling, linking to downloading the program to the target machine flash memory.
- Syntech Library routines : library routines developed by Syntech to access the terminals' specific hardware resources are described in detail.
- Standard library functions : standard library routines accompanying the C compiler are listed.

# 3. Getting Started

The development kit should contain the followings,

1) one full set of target machine including power adapter, cables and so on.
2) Self-test loop back tester for the target machine
3) 2 software diskettes
   - C compiler : compiler, linker, library include files, library codes and so on
   - Sample program source code
4) Hardware maintenance manual of the target machine.
5) This manual

## 3.1 System Requirements

Before installation, please make sure that your system is equipped with the follows,

| | |
|---|---|
| **Machine** | IBM-PC compatible |
| **MS-DOS** | V3.1 or more |
| **CPU** | i386 or more |
| **Minimum RAM** | 4 MB |
| **Minimum available disk space** | 2 MB |

## 3.2 Installation

Insert the C-compiler diskette into the diskette drive and copy all files to the root directory.

**COPY A:*.* \↵**

There are only 2 files in this diskette,

**PKUNZIP.EXE**
**SYNTECH.ZIP**

As these programs and codes are too large to be fitted into one single diskette. They are compressed and stored into one single file "SYNTECH.ZIP". The program "PKUNZIP.EXE" is used for decompression.

**PKUNZIP -d SYNTECH.ZIP↵**

The decompression program does not only decompress the files but also restore their original stored format including directories (the -d option). One directory "SYNTECH" and its 5 sub-directories are created.

1) BIN{xe "BIN"} : This directory contains 21 files.
   - 16 execution files for compilation, linking and so on,

     asm900.exe, cc900.exe, dos4gw.exe, mac900.exe,
     pminfo.exe, privatxm.exe, rminfo.exe, sc900.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
   - mdl.exe : flash download routine
   - cc.bat : batch file for compilation
   - pp.bat : batch file from compilation to download
   - lll.bat : batch file for downloading

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

2) ETC{xe "ETC"} : 11 files, help and version message 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 Syntech Library : synlib.h

4) LIB{xe "LIB"} : Library object code files
   - c900ml.lib : standard library
   - synlib.lib : Syntech own library

5) README{xe "README"} : 4 files, C compiler version update information

Now that the system is successfully installed. Make a sub-directory under SYNTECH, say '510TA' (Cipher-510 Time & Attendance). Then copy all files on the sample program source diskette to this sub-directory.

**XCOPY A:*.* C:\SYNTECH\510TA↵**

Several .C source files are now copied, also a sub-directory INCLUDE is copied which contains all include files for these C programs (not the library routines). If you are to create your own application programs, it is recommended to make a sub-directory under SYNTECH and put all your C source program here. (if you like, create a sub-directory under this directory to store your own include files).

**\ (root)** **SYNTECH** **LIB (library code)**

**ETC (C compiler version information)**

**INCLUDE (include files for library functions)**

**BIN (all executable files)**

**README (C compiler other informations)**

**510TA (Time & Attendance sample program)**

***.C** **(source file)**

**INCLUDE** ***.H(include file)**

**JOB1**

***.C** **(source file)**

**INCLUDE** ***.H(include file)**

**JOB2**

***.C** **(source file)**

**INCLUDE** ***.H(include file)**

## 3.3 Setup

Before using these software, some environmental variables must be added to the autoexec.bat. (assume drive C is installed, if not, change it to the correct drive name)

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).

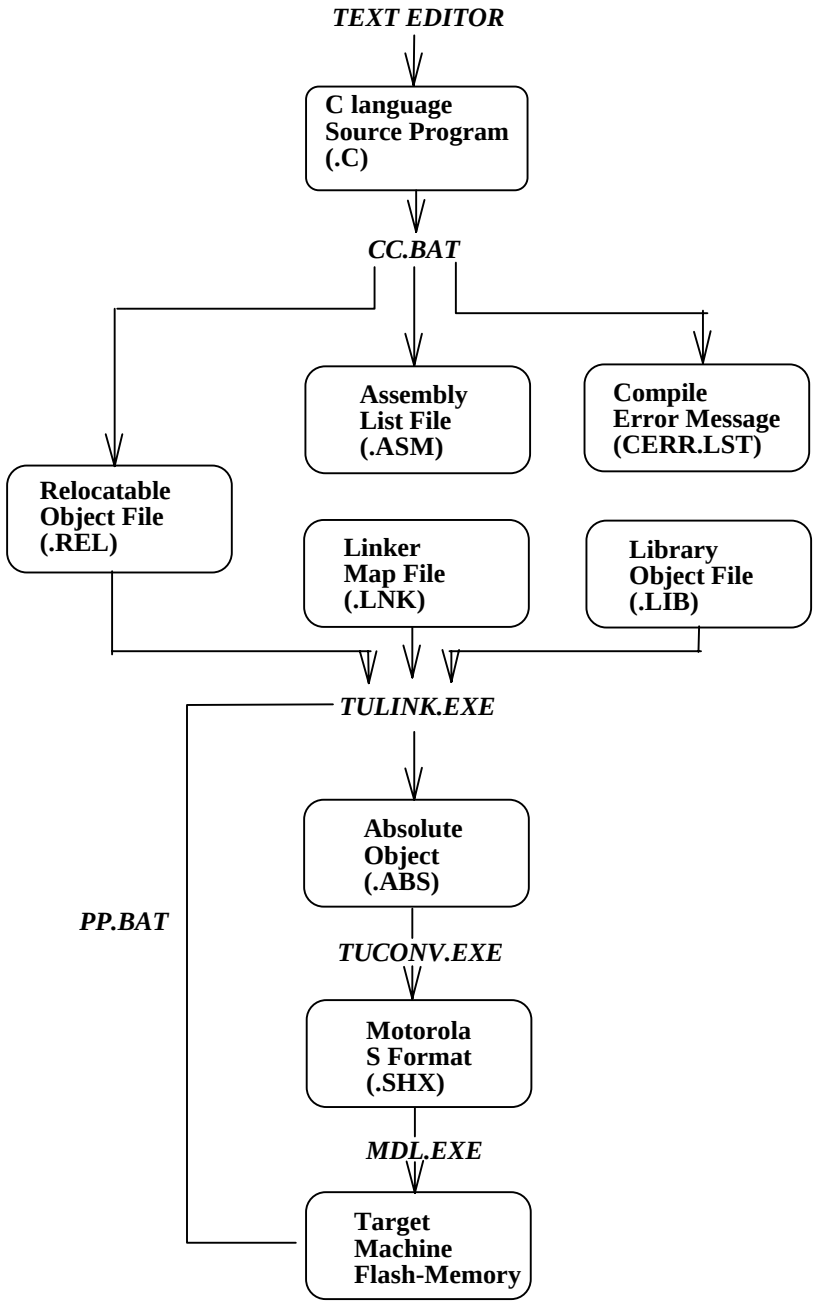
To facilitate efficiency, the compiler invokes a virtual memory manager "DOS4GW". It recognizes and supports various PCs. However, if it does not work on your PC. The program PMINFO can be used to identify the problem. (if you have difficult using the compiler, run the PMINFO, print all messages and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

## 3.4 Development Flow

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




### 3.4.1 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 "C". And then use the "C" 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.

### 3.4.2 Compile

A batch file "CC.BAT" under the sub-directory "C" (which is supposed to be the working directory) has been created to simplify the compiling process.

*CC program_name,*

".C" is automatically appended to the *program_name*. For example, "*CC 510*" is to compile the C program named *510.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.

### 3.4.3 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 map file listed below, 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\syntech.lib              /* Syntech library */
memory              /* follows are hardware related stuff, don't change */
{
      SFR       : org=0x00,         len=0x80
      SRAM      : org=0x100000,     len=0x0fffff
      IO        : org=0xef0000,     len=0x020000
      ROM       : org=0xfe0000,     len=0x01ffff
}
sections
{
      osec1: {  *(rom1) }           > ROM
      osec12: {       *(f_code) }   > ROM
      osec13: {       *(f_const) }  > ROM
      osec14: {       *(f_data) }   > ROM
      osec15: {       *(romend) }   > ROM        /* end of program code */
      osec21: {       *(ram_stack) }      > SRAM       /* stack area */
      osec22: {       *(ram0) }           > SRAM
      osec23: {       *(ram1) }           > SRAM
      osec31: {       *(f_area) }   > SRAM
      osec4: {  *(ramend) }         > SRAM       /* end of system RAM */
}
```

### 3.4.4 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.

### 3.4.5 Download and program the 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.

**MDL{xe "MDL"} *FILENAME COMPORT BAUDRATE PARITY DATABITS***

5 parameters must be specified and each separated by space/s.

- *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.

As the flash memory cannot be accessed as usual ways during erasing nor downloading. Execution of the loader program from flash memory is not possible. A small portion of code called loader (which can be invoked by calling the library routine "download") will first copy itself to SRAM and then execute the loading process from SRAM. If the downloading process was started, and for whatever reasons not completed. The code is destroyed and the target machine cannot be off and successfully turned on again. Then the flash memory must be taken out for programming again. To prevent this from happening (power off, cable disconnection and so on), if the target machine is equipped

with operation batteries (such as 510, 610), it is recommended to connect this battery before downloading. If the PC side was interrupted during the process, as long as the target machine is not turned off (that is, the loader is still in the SRAM), try run the MDL for several times to recover it.

### 3.4.6 Using ROM emulator

The ROM emulator can speed up development as most of them support Centronic ports downloading which is much faster than the RS232 ports. As the flash memory features the same pin assignments as conventional 128K X 8 ROMs, the ROM emulator can be used. However, the follows must be taken care of

1) Do not activate the target loader as it will activate a +12V power to program the flash memory which might damage the ROM emulator. (please refer to the ROM emulator user's guide)
2) The RESET signal from the ROM emulator must be connected to the target machine for proper operation. However, the RESET generation circuitry of the target machine must be disconnected beforehand. For detail, please refer to Hardware Manual of the target machine.
3) For 510, the self-shutdown circuit should be disconnected. (please refer to the hardware manual)

### 3.4.7 From Link To Download

A batch program called "pp.bat" is created and can be used to facilitate the programming process. Which links, transforms absolute object format and then downloads it to the target machine. This batch file is very simple and you can modify it for your own applications.

# 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,

## 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.

## 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 |

continued on next page

continued from previous page

| Macro Name | Contents |
|---|---|
| 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 |

## 4.3 Floating Types

Float types are supported and conforms to IEEE standards,

| Type | Size in bits |
|---|---|
| float | 32 |
| double | 64 |
| long double | 64 |

## 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.

## 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.

## 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 | | | |

## 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 | |

## 4.8 Bit-Field{xe "Bit-Field"} usage

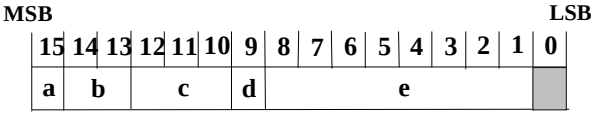
The following types can be used as the bit field base types.

| Type | Bits |
|---|---|
| char, unsigned char | 8 |
| short int, int,<br>unsigned short int, unsigned int | 16 |
| long int, unsigned long int | 32 |

The allocation is made as follows,

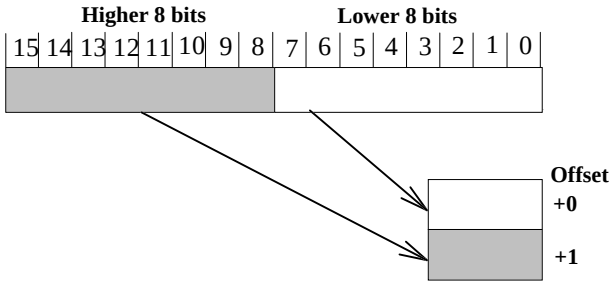
1) Fields are stored from the highest bits

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



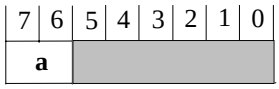
2) Little endien

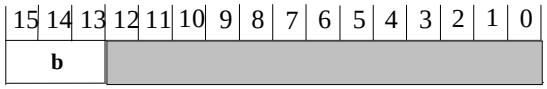
If the base type of a bit field member is a type requiring two bytes or more (e.g. unsigned int), the data is stored in memory after its bytes are turned topside down.




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

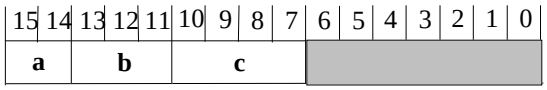
```
struct field {
        unsigned  char      a:2;
        unsigned  short     b:3;
}
```



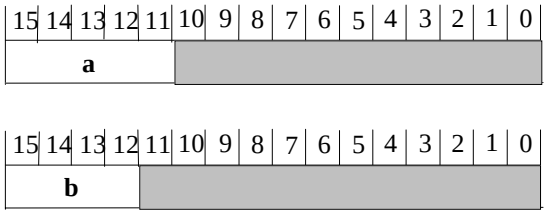

4) Different type (signed/unsigned)

```
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.

# 5. Syntech Library Routines : <synlib.h>

A lot of routines are written for accessing the target system resources. However, some resources are not available for all target systems. For example, there is no membrane keypad for 201. No action is taken when non-existent resources are to be accessed. A variable WHO_I_AM must be declared by the C main program to specify the target machine.

## 5.1 System

### 5.1.1 power on reset

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

- RS232, RS485 ports : all disabled
- reader ports : all disabled
- external AT keyboard : disabled
- keypad scanning (if any) : 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.

### 5.1.2 SRAM

The total SRAM spaces range from 32KB to 1MB depending on the target machine and are separated into 4 parts. Detail allocation map can be found in the file NAME.map, which is an output of the linker. They are allocated sequentially as follows,

1. Stack : **ram_stack**, 4KB stack spaces are allocated. Like other Cs, stack is used to transfer variables and store local variables. Care must be taken, not to overflow the stack or system will crash. It is recommended not to declare a large local variables (such as arrays). Instead, declare it to global which also improves efficiency. 4KB is quite sufficient for layers of function calling.
2. System buffers and parameters : **ram0 & ram1,** which occupies a little more than 6 KB
3. C variables : **f_area**, C global variables, size depends on C programs.
4. File space : **ram_end** to physical SRAM end. This part is managed by Syntech File routines and are not altered by the POR routine (as valuable informations are

stored here). On first power on or after SRAM removed/added, the function **init_free_memory()** must be called to initialize the file space.

### 5.1.3 variables

A constant variable WHO_I_AM **{**xe "WHO_I_AM "**}**must be declared by the C main program to specify the target machine. It must be declared as a global constant variable (that is, outside the { }) such that other programs can get access to it.

- const int WHO_I_AM = 1; /* 510 */
- const int WHO_I_AM = 2; /* 201 */
- const int WHO_I_AM = 3; /* 610 */

Other values are reserved. As this constant is frequently referenced by library routines, it must be correctly specified or severe error might occur (including possible damage to the circuit).

Two time variables that are already declared 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 int 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.

| **system_start** | |
|---|---|
| **purpose** | Re-initialize the system |
| **syntax** | void system_start(); |
| **example call** | system_start(); |
| **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 |

## 5.2 Reader

The barcode and magnetic card 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 scan port from operating.

### 5.2.1 Barcode and Magnetic Card Decoding

To enable barcode and magnetic card decoding capability in the system, the scanner ports 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 or magnetic 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 5 variables relate to the barcode and magnetic decoding routines : **ScannerDesTbl**, **CodeBuf**, **CodeLen**, **CodeType**, and **ScannerNo**. These variables are declared by the system, the user program need 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.

**ScannerNo** : This character indicates the scanner port being decoded upon successful decoding.

### 5.2.2 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 |

continued on next page

continued from previous page

| Name | Type | Name | Type |
|---|---|---|---|
| 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 | | |

### 5.2.3 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<br>0 : Disable Code 39 |
| 0 | 6 | 1 : Enable Italy Pharma-code<br>0 : Disable Italy Pharma-code |
| 0 | 5 | 1 : Enable CIP 39<br>0 : Disable CIP 39 |
| 0 | 4 | 1 : Enable Industrial 25<br>0 : Disable Industrial 25 |
| 0 | 3 | 1 : Enable Interleave 25<br>0 : Disable Interleave 25 |
| 0 | 2 | 1 : Enable Matrix 25<br>0 : Disable Matrix 25 |
| 0 | 1 | 1 : Enable Codabar (NW7)<br>0 : Disable Codabar (NW7) |
| 0 | 0 | 1 : Enable Code 93<br>0 : Disable Code 93 |
| 1 | 7 | 1 : Enable Code 128<br>0 : Disable Code 128 |
| 1 | 6 | 1 : Enable UPCE no Addon<br>0 : Disable UPCE no Addon |
| 1 | 5 | 1 : Enable UPCE Addon 2<br>0 : Disable UPCE Addon 2 |
| 1 | 4 | 1 : Enable UPCE Addon 5<br>0 : Disable UPCE Addon 5 |
| 1 | 3 | 1 : Enable EAN8 no Addon<br>0 : Disable EAN8 no Addon |
| 1 | 2 | 1 : Enable EAN8 Addon 2<br>0 : Disable EAN8 Addon 2 |

continued on next page

continued from previous page

| Subscriptor | Bit | Description |
|---|---|---|
| 1 | 1 | 1 : Enable EAN8 Addon 5<br>0 : Disable EAN8 Addon 5 |
| 1 | 0 | 1 : Enable EAN13 no Addon<br>0 : Disable EAN13 no Addon |
| 2 | 7 | 1 : Enable EAN13 Addon 2<br>0 : Disable EAN13 Addon 2 |
| 2 | 6 | 1 : Enable EAN13 Addon 5<br>0 : Disable EAN13 Addon 5 |
| 2 | 5 | 1 : Enable MSI<br>0 : Disable MSI |
| 2 | 4 | 1 : Enable Plessey<br>0 : Disable Plessey |
| 2 | 3 | 1 : Enable Code ABC<br>0 : Disable Code ABC |
| 2 | 2 - 0 | reserved |
| 3 | 7 - 0 | reserved |
| 4 | 7 - 0 | reserved |
| 5 | 7 | 1 : Transmitting Code 39 Start/Stop Character<br>0 : No Transmitting Code 39 Start/Stop Character |
| 5 | 6 | 1 : Verifying Code 39 Check Character<br>0 : No Verifying Code 39 Check Character |
| 5 | 5 | 1 : Transmitting Code 39 Check Character<br>0 : No Transmitting Code 39 Check Character |
| 5 | 4 | 1 : Full ASCII Code 39<br>0 : Standard Code 39 |
| 5 | 3 | 1 : Transmitting Italy Pharmacode Check Character<br>0 : No Transmitting Italy Pharmacode Check Character |
| 5 | 2 | 1 : Transmitting CIP39 Check Character<br>0 : No Transmitting CIP39 Check Character |
| 5 | 1 | 1 : Verifying Interleave 25 Check Digit<br>0 : No Verifying Interleave 25 Check Digit |
| 5 | 0 | 1 : Transmitting Interleave 25 Check Digit<br>0 : No Transmitting Interleave 25 Check Digit |
| 6 | 7 | 1 : Verifying Industrial 25 Check Digit<br>0 : No Verifying Industrial 25 Check Digit |
| 6 | 6 | 1 : Transmitting Industrial 25 Check Digit<br>0 : No Transmitting Industrial 25 Check Digit |
| 6 | 5 | 1 : Verifying Matrix 25 Check Digit<br>0 : No Verifying Matrix 25 Check Digit |
| 6 | 4 | 1 : Transmitting Matrix 25 Check Digit<br>0 : No Transmitting Matrix 25 Check Digit |

continued on next page

continued from previous page

| Subscriptor | Bit | Description |
|---|---|---|
| 6 | 3 - 2 | Select Interleave25 Start/Stop Pattern<br>00 : Use Industrial25 Start/Stop Pattern<br>01 : Use Interleave25 Start/Stop Pattern<br>10 : Use Matrix25 Start/Stop Pattern<br>11 : Undefined |
| 6 | 1 - 0 | Select Industrial25 Start/Stop Pattern<br>00 : Use Industrial25 Start/Stop Pattern<br>01 : Use Interleave25 Start/Stop Pattern<br>10 : Use Matrix25 Start/Stop Pattern<br>11 : Undefined |
| 7 | 7 - 6 | Select Industrial25 Start/Stop Pattern<br>00 : Use Industrial25 Start/Stop Pattern<br>01 : Use Interleave25 Start/Stop Pattern<br>10 : Use Matrix25 Start/Stop Pattern<br>11 : Undefined |
| 7 | 5 - 4 | Codabar Start/Stop Character<br>00 : abcd/abcd<br>01 : abcd/tn*e<br>10 : ABCD/ABCD<br>11 : ABCD/TN*E |
| 7 | 3 | 1 : Transmitting Codabar Start/Stop Character<br>0 : No Transmitting Codabar Start/Stop Character |
| 7 | 2 - 0 | reserved |
| 8 | 7 - 0 | reserved |
| 9 | 7 - 6 | MSI Check Digit Verification<br>00 : Single Modulo 10<br>01 : Double Modulo 10<br>10 : Modulo 11 and Modulo 10<br>11 : Undefined |
| 9 | 5 - 4 | MSI Check Digit Transmission<br>00 : the last Check Digit is not transmitted<br>01 : both Check Digits are transmitted<br>10 : both Check Digits are not transmitted |
| 9 | 3 | 1 : Transmitting Plessey Check Characters<br>0 : No Transmitting Plessey Check Characters |
| 9 | 2 | 1 : Converting Standard Plessey to UK Plessey<br>0 : No Converting |
| 9 | 1 | 1 : Converting UPCE to UPCA<br>0 : No Converting |
| 9 | 0 | 1 : Converting UPCA to EAN13<br>0 : No Converting |

continued on next page

continued from previous page

| Subscriptor | Bit | Description |
|---|---|---|
| 10 | 7 | 1 : Enable ISBN Conversion<br>0 : No Conversion |
| 10 | 6 | 1 : Enable ISSN Conversion<br>0 : No Conversion |
| 10 | 5 | 1 : Transmitting UPCE Check Digit<br>0 : No Transmitting UPCE Check Digit |
| 10 | 4 | 1 : Transmitting UPCA Check Digit<br>0 : No Transmitting UPCA Check Digit |
| 10 | 3 | 1 : Transmitting EAN8 Check Digit<br>0 : No Transmitting EAN8 Check Digit |
| 10 | 2 | 1 : Transmitting EAN13 Check Digit<br>0 : No Transmitting EAN13 Check Digit |
| 10 | 1 | 1 : Transmitting UPCE System Number<br>0 : No Transmitting UPCE System Number |
| 10 | 0 | 1 : Transmitting UPCA System Number<br>0 : No Transmitting UPCA System Number |
| 11 | 7 | 1 : Converting EAN8 to EAN13<br>0 : No Converting |
| 11 | 6 | 1 : Transmitting Code ABC Concatenation Characters<br>0 : No Transmitting Code ABC Concatenation Characters |
| 11 | 5 | reserved |
| 11 | 4 | 1 : Enable Reversed Barcode<br>0 : Disable Reversed Barcode |
| 11 | 3 - 2 | 00 : No Read Redundancy for Scanner Port 1<br>01 : One Time Read Redundancy for Scanner Port 1<br>10 : Two Times Read Redundancy for Scanner Port 1<br>11 : Three Times Read Redundancy for Scanner Port 1 |
| 11 | 1 - 0 | 00 : No Read Redundancy for Scanner Port 2<br>01 : One Time Read Redundancy for Scanner Port 2<br>10 : Two Times Read Redundancy for Scanner Port 2<br>11 : Three Times Read Redundancy for Scanner Port 2 |
| 12 | 7 | 1 : Industrial 25 Code Length Limitation in Max/Min Length Format<br>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<br>0 : Interleave 25 Code Length Limitation in Fix Length Format |
| 14 | 6 - 0 | Interleave 25 Max Code Length / Fixed Length 1 |

continued on next page

continued from previous page

| Subscriptor | Bit | Description |
|---|---|---|
| 15 | 7 - 0 | Interleave 25 Min Code Length / Fixed Length 2 |
| 16 | 7 | 1 : Matrix 25 Code Length Limitation in Max/Min Length Format<br>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<br>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<br>0000 : Auto Off Mode<br>0001 : Continuous Mode<br>0010 : Auto Power Off Mode<br>0011 : Alternate Mode<br>0100 : Momentary Mode<br>0101 : Repeat Mode<br>0110 : Laser Mode<br>0111 : Test Mode |
| 20 | 3 - 0 | Scan Mode for Scanner Port 2<br>0000 : Auto Off Mode<br>0001 : Continuous Mode<br>0010 : Auto Power Off Mode<br>0011 : Alternate Mode<br>0100 : Momentary Mode<br>0101 : Repeat Mode<br>0110 : Laser Mode<br>0111 : Test Mode |
| 21 | | Scanner Time-out Duration in seconds for Scanner Port 1 |
| 22 | | Scanner Time-out Duration in seconds for Scanner Port 2 |
| 23 - 28 | | reserved |

**Decode**

**purpose** Perform barcode and magnetic card decoding.

**syntax** int Decode( );

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

**description** Once the scanner ports are initialized (by use of *InitScanner1* and *InitScanner2* functions), call this *Decode* function to perform barcode and magnetic card decoding. This function should be called constantly in user's program loops when barcode & magnetic card decoding is required.

If the barcode and magnetic card decoding is not required for a long period of time, it is recommended that the scanner ports should be stopped by use of the *HaltScanner1* & *HaltScanner2* functions.

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. The character variable *ScannerNo* will also indicate the scanner ports being decoded.

**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 respective initialization function (*InitScanner1* and *InitScanner2*) must be called.

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

**returns** These two functions have no return values.

## 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** These two functions have no return values.

## 5.3 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.

### 5.3.1 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.

### 5.3.2 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.

### 5.3.3 Beep Duration

A 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 0.1 second. To get a beep of 1 second, the beep duration should be 10. A 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.

## volume

**purpose** Set buzzer volume

**syntax**
```
void volume(int level);
int level;          /* buzzer volume level from 0 to 3 */
```

**example call**
```
volume(3);                    /* set to loudest */
```

**description** The *volume* function fine tunes the desired buzzer volume according to application needs.

**returns** None

## 5.4 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 turn off.

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

### 5.4.1 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.

### 5.4.2 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

### 5.4.3 Leap Year

The leap-year day is automatically handled by the calendar chip. The **year** field set to the calendar chip must be in AD year to get the correct leap year operation.

| **adjust_timer** | |
|---|---|
| **purpose** | Modify the value of the trimming register of the calendar chip. |
| **syntax** | int adjust_timer(int offset);<br>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.

## 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 13 bytes to accommodate the date, time, and the string terminator. The format of the system date and time is listed below.

**"YYMMDDhhmmss"**

where
- **YY** : year, 2 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.

## 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("940105125800");/* JAN 5, 1994  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,

**"YYMMDDhhmmss"**

where **YY** : year, 2 digits, 0-99
**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.

## 5.5 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 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 also support FIFO data structure manipulation. 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 are easily done by use of the index sequential file routines. There are actually two types of files exist in this file structure. One is the files which store the data records / members, and the other is the associate key /index files. 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 be noted that not all of the routines described in this section apply to both types of file structures. In the paragraph of each routine description, we have listed the target file types that the routine under description applies.

### 5.5.1 File Space

The memories (SRAM) installed are logically divided into three parts : the system space, the user space, and the file space. The system space is used by the system program, consists of system parameters, file allocation table, directory area, and stack area. And the system space takes up about 13K bytes of memory (for 510 T&A sample program). The user space is used by the static variables declared by the application program. And the remaining free memories are all assigned to the file space where file manipulation takes place. For the maximum file space, the static variables declared by the application program must be limited. That is, the smaller the user space the larger the file space.

### 5.5.2 File Name

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

### 5.5.3 File Handle

File handles are used to identify files after files are opened. Most of the file manipulation functions provided use file handles not file names to specify 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.

### 5.5.4 Error Code

All of the file manipulation routines discussed in this section have an error code to indicate operation success or the cause of error encountered when they are called. The error code is an integer value set to the global variable *error_code*. The error code can be fetched by directly referencing the global variable *error_code*, or by making a call to the *read_error_code* function. Error code with a value of 0 indicates no error encountered.

### 5.5.5 Directory

The file system does not support tree like directory structure, which means there is no way to create a sub-directory. And the maximum number of files can exist in the system is limited to 16 files (includes all DAT files, DBF files, and their associate IDX files). The file directory information can be fetched by use of *filelist* routine.

### 5.5.6 DAT Files

The DAT files have a sequential file structure, and all the functions that are needed to manipulate sequential files are included in the system. Besides the ordinary sequential file manipulation routines, we have included some special routines to support FIFO data structure.

The data from the beginning of a DAT file can be removed from the DAT file by calling the *delete_top* and the *delete_topln* functions. The new file top (beginning) position, the file pointer position, and the size of the DAT file will be adjusted accordingly after calling to these two routines. And the *append* and *appendln* functions can write data to a DAT file from 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 the way we usually handle the transaction data input and upload to the host computer.

### 5.5.7 DBF Files and IDX Files

The DBF files and the IDX files together form the platform of the data base system. The DBF files are files which store the actual data records (members), whereas the IDX files are the key (index) files of the DBF files. A DBF file can have at most 8 IDX files. The

IDX files has the same file name as the associate DBF file as you can see in the directory information fetched by calling the *filelist* function, and each IDX files are further identified by the key numbers ranging from 1 to 8. An IDX file can be created by use of the *create_index* function only when the associated DBF is empty. When it is not , use *rebuild_index* function instead to create IDX files.

Data records (members) are not allowed to be accessed directly from the DBF files but rather through their associate IDX files as can be seen from the function description. The file pointers of the IDX files (index pointers) do not represent the address of the data records in the DBF files, but rather the rank number (starting from 1) of the specific data record in the IDX files. The data record sequence in the IDX files are always sorted in the ascending key value order.

**access**

**target file type** DAT DBF

**purpose** Check file existence.

**syntax** int access(char* filename);

| | |
|---|---|
| char* filename; | file name of file being checked |

**example call** if (access("data1") == 0) send_lcds("data1 exist!\n");

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

**returns** If *access* finds the file specified by *filename* exist, it returns an integer value of 0. In case of error or the file does not exist, *access* will return an integer value of -1 and an error code is set to the global variable *error_code* 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. |

**add_member**

**target file type** DBF

**purpose** Add a 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 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.

**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 -1 or 0 and an error code is set to the global variable *error_code* 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. |
| 8 | *DBF_fd* is not a file handle of a previously opened file. |
| 10 | No free file space for adding member. |
| 19 | The member specified by the argument *member* is a null string. |

## 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, 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**
```
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 *error_code* 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. |

**comments** Since *append* returns an signed integer, the return value should be converted to *unsigned int* when writing more than 32,767 bytes of data to a file or the return value will be negative. Because the number of bytes to be written is specified in an unsigned integer argument, you could theoretically write 65,535 bytes at a time. But 65,535 (or FFFFh) also

means -1 in signed representation, so when writing 65,535 bytes the return value indicates an error. The practical maximum then is 65,534.

## appendln

**target file type** DAT

**purpose** Write a line terminated by a null character (\0) 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 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 end-of-file position of the file, and 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 *error_code* 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. |

**comments** Since *appendln* returns an signed integer, the return value should be converted to *unsigned int* when writing more than 32,767 bytes of data to a file or the return value will be negative. You could theoretically write 65,535 bytes at a time. But 65,535 (or FFFFh) also means -1 in signed representation, so when writing 65,535 bytes the return value indicates an error.

## 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)) send_lcds("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 0. In case of error, *chsize* will return an integer value of -1 and an error code is set to the global variable *error_code* 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. |

## 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)) send_lcds("file closed!\n");
```

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

**returns** *close* returns an integer value of 0 to indicate success. In case of error, *close* returns an integer value of -1 and an error code is set to the global variable *error_code* 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. |

## close_DBF

**target file type** DBF

**purpose** Close DBF file together with its 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 0 to indicate success. In case of error, *close_DBF* returns an integer value of -1 and an error code is set to the global variable *error_code* 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. |

## 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 length of the DBF file |

**example call** if (fd = create_DBF("data1",64) > 0) send_lcds("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 long.

**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 *error_code* 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* equals 0. |
| 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 ey_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 0. In case of error, *create_index* will return an integer value of -1 and an error code is set to the global variable *error_code* 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.* |
| 11 | Illegal value in argument *key_number.* |
| 16 | DBF file specified by *DBF_fd* is not empty. |
| 17 | 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
                 points 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 -1 or 0 and an error code is set to the global variable *error_code* 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. |
| 8 | *DBF_fd* is not a file handle of a previously |

| | |
|---|---|
| | opened file. |
| 10 | Internal error in the DBF structure. All index files should be rebuild. |
| 11 | The IDX file specified by *key_number* does not exist. |
| 13 | There are no members in the DBF file. |
| 20 | Internal error in the DBF structure. All index files should be rebuild. |
| 21 | Internal error in the DBF structure. All index files should be rebuild. |

## 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, unsigned count);
int fd;              file handle of the target DAT file
unsigned 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 *error_code* 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 *delete_top* returns an signed integer, the return value should be converted to *unsigned int* when removing more than 32,767 bytes of data from a file or the return value will be negative. Because the number of bytes to be removed is specified in an unsigned integer argument, you could theoretically remove 65,535 bytes at a time. But 65,535 (or FFFFh) also means -1 in signed representation, so when removing 65,535 bytes the return value indicates an error. The practical maximum then is 65,534.

## delete_topln

**target file type** DAT

**purpose** Remove a line terminated by a null character (\0) 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 *error_code* 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 *delete_topln* returns an signed integer, the return value should be converted to *unsigned int* when removing more than 32,767 bytes of data from a file or the return value will be negative. You could theoretically remove 65,535 bytes at a time. But 65,535 (or FFFFh) also means -1 in signed representation, so when removing 65,535 bytes the return value indicates an error.

## 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) == 1) send_lcds("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 *error_code* 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. |

## 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 *error_code* 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. |

## 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.

## 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.

**target file type** DBF

## get_member

**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) send_lcds(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 -1 or 0 and an error code is set to the global variable *error_code* 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. |
| 8 | *DBF_fd* is not a file handle of a previously opened file. |
| 11 | Index file specified by *key_number* does not exist. |
| 13 | There is no members in the DBF file specified. |

## 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") == 0)
        send_lcds("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 *error_code* 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. |
| 8 | *DBF_fd* is not a file handle of a previously opened file. |
| 11 | Index file specified by *key_number* does not exist. |
| 14 | *Key_value* specified is a null string. |

## 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 *error_code* 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 | 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 (in ranks) 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 rank 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 rank 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 *error_code* 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. |
| 8 | *DBF_fd* is not a file handle of a previously opened file. |
| 9 | Illegal *origin* value. |
| 11 | Index 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 *error_code* 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. |
| 8 | *DBF_fd* is not a file handle of a previously opened file. |

## 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) send_lcds("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 *error_code* 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) send_lcds("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 *error_code* 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)
        send_lcds("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 *error_code* 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 *error_code*.

**syntax** int read_error_code( );

**example call** if (read_error_code() == 2) send_lcds("File not exist!\n");

**description** The *read_error_code* function gets the value of the global variable *error_code* 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 *error_code* can be directly accessed without making a call to this function.

**returns** The *read_error_code* function returns the value of the global variable *error_code*.

## 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 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 *error_code* 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 *error_code* 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") == 0) send_lcds("data1 deleted!\n");
```

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

**returns** If *remove* deletes the file successfully, it returns an integer value of 0. In case of error, *remove* will return an integer value of -1 and an error code is set to the global variable *error_code* 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) == 0) send_lcds("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 0 if it successfully deletes the index file. In case of error, *remove_index* returns an integer value of -1 and an error code is set to the global variable *error_code* 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") == 0) send_lcds("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 0. In case of error, *rename* will return an integer value of -and an error code is set to the global variable *error_code* 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 *error_code* 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 *error_code* 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 *error_code* 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. |

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

## 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 *error_code* 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. |

**comments** Since *writeln* returns an signed integer, the return value should be converted to *unsigned int* when writing more than 32,767 bytes of data to a file or the return value will be negative. You could theoretically write 65,535 bytes at a time. But 65,535 (or FFFFh) also means -1 in signed representation, so when writing 65,535 bytes the return value indicates an error.

## 5.6 Digital Input / Output

This section describes the digital Input / Output manipulation routines. Number of digital input/output provided varies from machine to machine.

| Machine | Digital Input | Digital Output | Photo-Isolated |
|---|---|---|---|
| 201 | 2 | 2 | No |
| 510 | 2 | 2 | Yes |
| 610 | 2 | 2 | Yes |

**get_din**

**purpose** Read digital Input.

**syntax**
```
int get_din(int di);
int di;      /* digital input number from 0, depends on machine */
```

**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. else 0

**set_do**

**purpose** Set the digital output

**syntax**
```
void set_do(int do, int mode, int duration);
int do;          /* digital output number starts from 0 */
int mode;        /* output mode */
int duration;    /* duration */
```

**example call**
```
set_io(1,0,10);   /* on digital output for 1 second then off */
```

**description** The *set_do* sets the digital output points specified by do.
The duration specified in the argument *duration* is in units of 0.1 second. That is, if a duration of 1 second is desired, a value of 10 should be assigned to the argument *duration*. A value of 0 in the argument *duration* will keep the I/O stay in the specific state indefinitely.

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

| output mode | interpretation |
|---|---|
| 0 | Turn on the DO immediately for specific duration and then go back off. |
| 1 | Turn off the DO immediately for specific duration and then go back on. |
| 2 | Turn on the DO for exactly the specific duration and then go back off. |
| 3 | Turn off the DO for exactly the specific duration and then go back on. |
| 4 | Flash the I/O with a specific period indefinitely. The flashing period equals to 2 * *duration.* |

**Note** that for mode 0 and 1, the activation is executed at once but the overall duration is longer for at most 0.1 second. Whereas the mode 2 and 3, the activation is not executed at once, but the activation period is exactly (duration X 0.1) seconds.

**returns** none

## 5.7 LED

Number of LEDs provided depends on the target machine as follows,

- 201

| Name | Number |
|---|---|
| good read for reader 1 | 0 |
| good read for reader 2 | 1 |

These 2 LED outputs are directly connected to the corresponding reader ports. Also they are ORed to the built-in GoodRead LED. That is, if one or both of them are turned on, the on-board good read LED will be turned on too.

- 510

| Name | Number |
|---|---|
| good read for reader 1 | 0 |
| good read for reader 2 | 1 |
| ready | 3 |
| good read | 4 |
| battery low | 7 |
| F1 | 9 |
| F2 | 6 |
| F3 | 5 |
| F4 | 2 |
| shift | 8 |

- 610 : to be defined

**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 */
duration;     /* duration in unit of 0.1 seconds */
```

**example call**
```
set_led(3, 2, 10);/* set ready LED to flash for each 1 second */
```

**description** 3 modes are supported,
0 : off for (duration X 0.1) seconds then on
1 : on for (duration X 0.1) seconds then off
2 : flash, on then off each for (duration X 0.1) seconds then repeat

**returns** none

## 5.8 Membrane Keypad

The built-in membrane keypad is available for 510 and 610 but not 201. A scanning circuitry of 8 by 8 matrix is utilized regardless of the keypad used. The background routine constantly scans the membrane keypad and if any key was pressed, stored this scan code into a FIFO (first-in first-out) 32 bytes buffer. However, if the buffer is full, the keys followed will be ignored. The C program must constantly checks to see if any code is available in the buffer.

The scanning routine is capable of handling single-key strokes only, combination keys are not supported. That is, if more than one key are pressed at the same time, only the key with smallest scancode is recognized. However, repeat function is supported. To be human friendly, after the key was pressed for 1 seconds, repeat starts with a 0.5 seconds period.

The scan codes for 510 standard keypad are listed below,

| **Key** | Code | **Key** | Code | **Key** | Code | **Key** | Code |
|---|---|---|---|---|---|---|---|
| **F1** | 1b | **0** | 11 | **4** | 08 | **8** | 02 |
| **F2** | 19 | **1** | 12 | **5** | 09 | **9** | 03 |
| **F3** | 1a | **2** | 13 | **6** | 0a | **shift** | 0b |
| **F4** | 1c | **3** | 14 | **7** | 01 | **enter** | 04 |

*The scan code is expressed in hex-decimal form.*

**clr_memkb**

**purpose** Clear the keyboard buffer of the membrane keyboard.

**syntax** void clr_memkb();

**example call** clr_memkb();

**description** The *clr_memkb* function clears the keyboard buffer of the membrane keyboard. This function is automatically called by the system program upon power up

**returns** The *clr_memkb* function has no return values.

**mem_kbhit**

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

**syntax** int mem_kbhit();

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

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

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

**mem_getchar**

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

**syntax** char mem_getchar();

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

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

**returns** The *mem_getchar* function returns the key stroke read from the keyboard buffer. If the keyboard buffer is empty, a null character (0x00) is returned. The key stroke returned is the scan code of the key being pressed. The possible values of the key strokes returned are from 1 to 64 representing each key on the membrane keyboard. The programmer might take some extra effort to translate the key strokes to meaningful characters.

**scan_multi_key**

**purpose** Get multiple key combinations from the membrane keyboard.

**syntax** unsigned long scan_multi_key();

**example call**
```
unsigned long keycode;

keycode=scan_multi_key();
printf_us("Keys %ld pressed", keycode);
```

**description** Unlike the ext_getchar, this routine disables the background scanning routines and directly scans the keypad. At the end, an unsigned long integer is returned to show up to 4 keys that are pressed at the same time. These scancodes are stored in ascending order (higher byte with smaller scan codes). This is used to get the special power-on code for diagnostic and so on and should not be used for normal use.

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

## 5.9 External AT Keyboard

The external AT keyboard is supported by 510 and 201 for handy 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. External keyboard handling routines process the code translation (from scan code to ASCII) and shift, capslock and so on and must be called periodically.

**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 keycode buffer. |
| **returns** | none |

**dis_extkb**

| | |
|---|---|
| **purpose** | Disable external AT keyboard connection |
| **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 |

**clr_extkb**

**purpose** Clear the keyboard buffer of the external AT keyboard.

**syntax** void clr_extkb();

**example call** clr_extkb();

**description** The *clr_extkb* function clears the keyboard buffer of the external AT keyboard. This function is automatically called upon power on.

**returns** The *clr_extkb* function has no return values.

**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.

**ext_kbhit**

**purpose** Check whether the keyboard buffer of the external AT keyboard is empty.

**syntax** int ext_kbhit();

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

**description** The *ext_kbhit* function checks whether there is any character waiting to be read in the keyboard buffer of the external AT keyboard.

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

**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

## 5.10 LCD

This section describes the output routines and the control routines concerning the LCD display. Depending on the target machine, up to 4 types of LCD displays are supported, 20 X 2, 20 X 4, 40 X 2 and 40 X 4. A constant named LCD_TYPE must be defined in the C main program to specify the LCD display utilized.

- const int LCD_TYPE = 1; /* 20 X 2 */
- const int LCD_TYPE = 2; /* 20 X 4 */
- const int LCD_TYPE = 3; /* 40 X 2 */
- const int LCD_TYPE = 4; /* 40 X 4 */

A coordinate system is used in the cursor movement routines to determine the position of the cursor. The coordinate of the top left character position is assigned (0,0) and the bottom right character is assigned with coordinate (column-1, line-1), e.g. (19, 1), (19, 3) and so on. If the cursor is not in the visible area on the LCD display, the cursor is said to fall off the **LCD scope**. To keep these routines function properly, the LCD type must be correctly specified.

### 5.10.1 Scrolling

If the cursor falls off the LCD scope when sending outputs to the LCD by use of the scrolling output routines, a new line character will be automatically inserted which will scroll up the current line (if necessary) and place the cursor to the first character position of the next line. If the unscrolling routines are used instead, no extra output formatting will be made and the characters fall off the LCD scope will be ignored.

### 5.10.2 Customized Fonts

Up to 8 fonts can be customized into LCD display controller and is mapped to hex code 10 to 17. For detail, please refer to set_lcd_cg().

### 5.10.3 Special Characters

The back space character (0x08), the new line (line feed) character (0x0a), and the return character (0x0d) are processed as following.

- **Back Space :** The cursor is backed up one character position and the character at the new cursor position is replaced by a space character.
- **New Line :** The current line will be scrolled up (if necessary) and the cursor will be placed at the first character position of the next line.
- **Return :** The characters from the current cursor position to the end of the line will be cleared (filled with spaces) and the cursor will be placed at the first character position of the line.

**clr_scr**

**purpose** Clear LCD display.

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

**cursor_status**

**purpose** Get current cursor status.

**syntax** int cursor_status();

**example call** if (cursor_status()==0) send_lcds("Cursor Off");

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

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

**lcd_backlit**

**purpose** Set LCD backlight

**syntax** void lcd_backlit(intensity);

unsigned char intensity; /* intensity from 0 to 3 */

**example call** lcd_backlit(3); /* set brightest LCD backlight */

**description** The *lcd_backlit* fine tunes the LCD backlight intensity from fully turned on (3) to fully off (0).

**returns** none.

**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.

## printf_s

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

**syntax** int printf_s(char* format, );

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_s("ID : %s", id_buffer);

**description** The *printf_s* 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*. The *printf_s* function will automatically insert a new line character, if the cursor falls off the scope of LCD during operation. That is, the *printf_s* function will perform scrolling when necessary.

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. |

**returns** The *printf_s* function returns the number characters sent to the LCD display (not including the scrolling new line characters inserted by *printf_s*).

## printf_us

**purpose** Write character strings and values of C variables in a specified format without scrolling to the LCD display.

**syntax** int printf_us(char* format, );

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_us("ID : %s", id_buffer);

**description** The *printf_us* function performs the same task as *printf_s* except that *printf_us* does not perform automatic scrolling if the cursor falls off the scope of LCD during operation.

**returns** The *printf_us* function returns the number characters sent to the LCD display.

## send_lcdc

**purpose** Display a character on the LCD display with scrolling.

**syntax** int send_lcdc(char c);

char c; character sent to the LCD display

**example call** send_lcdc('A');

**description** The *send_lcdc* function sends the character specified in the argument *c* to the LCD display at the current cursor position and moves the cursor accordingly. If the cursor falls off the LCD scope after the operation, a new line character will be automatically sent to the LCD display by the *send_lcdc* function.

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

**send_lcds**

**purpose** Display a string on the LCD display with scrolling.

**syntax**
```
char send_lcds(char* string);
char* string;          string to be displayed
```

**example call**
```
send_lcds("Password : ");
```

**description** The *send_lcds* 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. If the cursor falls off the scope of LCD display duration the operation, a new line character will be automatically inserted.

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

**send_lcduc**

**purpose** Display a character on the LCD display without scrolling.

**syntax**
```
char send_lcduc(char c);
char c;          character to be displayed
```

**example call**
```
send_lcduc('A');
```

**description** The *send_lcduc* function sends a character specified in the argument *c* to the LCD display at the current cursor position and moves the cursor accordingly. No new line characters will be inserted should the cursor falls off the scope of the LCD display after the *send_lcduc* function sends the character *c* to the LCD display.

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

**send_lcdus**

**purpose** Display a string on the LCD display without scrolling.

**syntax**
```
char send_lcdus(char* string);
char* string;          string to be displayed
```

**example call**
```
send_lcdus("Password : ");
```

**description** The *send_lcdus* 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. If the cursor falls off the scope of LCD display duration the operation, no new line character will be automatically inserted as opposed to the *send_lcds* function.

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

## set_cursor

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

**syntax**
```
void set_cursor(int status);
int status;        integer representing cursor status to be set
```

**example call**
```
set_cursor(0);                /* invisible the cursor */
```

**description** The *set_cursor* 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 blinking on the LCD display to show the current cursor position. If *status* equals 0, the cursor will be invisible.

**returns** The *set_cursor* function has no return values.

## 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** Normally the *wherexy* function returns an integer value of 1. In case LCD malfunctions, the return value will be 0 to indicate error.

## 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.

## set_lcd_cg

**purpose** Set customized fonts

**syntax** void set_lcd_cg(char *font);
char *font;/* 64 bytes bit patterns */

**example call**
```
char my_font = {
    0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0,
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    .....
};
send_lcdc(0x10);
```
actual image shown on the display

| | Data | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|
| **Byte 1** | 12 | | | | O | | | O | |
| **Byte 2** | 34 | | | O | O | | O | | |
| **Byte 3** | 56 | | O | | O | O | | O | |
| **Byte 4** | 78 | | O | O | O | O | | | |
| **Byte 5** | 9a | O | | | O | O | | O | |
| **Byte 5** | bc | O | | O | O | O | O | | |
| **Byte 6** | de | O | O | | O | O | O | O | |
| **Byte 7** | f0 | O | O | O | O | | | | |

**description** This routine is used to customize up to 8 special fonts for application use.

**returns** none

## 5.11 Power

This section describes the power management functions for 510. The **read_batt** function is used to monitor the voltage level of the main battery and the **shut_down** function is used to halt the operation of the station.

### 5.11.1 Main Battery

If the optional main battery is installed, the machine can still be operational by using the power of the main battery when there is no external power supplied. The battery life of the main battery may be shortened or even damaged if the battery was deeply drained. So it is important to constantly monitor the voltage level of the main battery and shut down the system before the voltage level falls too low. It is recommend that the **battery low LED** be lighted when the voltage level falls below 7.7 volts as a warning, and the system be shut down when the voltage level falls below 7.0 volts.

**read_batt**

| | |
|---|---|
| **purpose** | Get voltage level of the main battery. |
| **syntax** | unsigned read_batt(); |
| **example call** | if ( read_batt() < 7000) set_led(BATTERY_LOW, 2, 1);<br>/* if battery low, flash battery low LED */ |
| **description** | The *read_batt* function reads the voltage level of the main battery in units of mV. If external power is supplied, the read value should be 10000, that is 10 Volts as it is limited by ADC protection circuitry. |
| **returns** | The *read_batt* function returns the voltage level of the main battery in units of mV (mili-volt). |

**shut_down**

| | |
|---|---|
| **purpose** | Shut down system power. |
| **syntax** | void shut_down(); |
| **example call** | shut_down(); |
| **description** | The *shut_down* function turns off the power of the machine immediately. To re-operate the machine, the power switch must be turned off and on again. |
| **returns** | The *shut_down* function has no return value. |

## 5.12 RS232

This section describes the RS232 manipulation routines. There are totally 3 RS232 ports provided , namely COM1 to COM3 and are accessed via the same methods regardless of their hardware deviations. That is, on the software point, they are all the same. 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,

### 5.12.1 Parameters

- Baud rate : One out of 8 baud rates can be selected (38400, 19200, 9600, 4800, 2400, 1200, 600 and 300)
- Data Bits : 7 or 8
- Parity : Even, Odd or none
- Stop bit : 1

### 5.12.2 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.

### 5.12.3 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.

### 5.12.4 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).

**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 = 38400/19200/9600/4800 /2400/1200/600/300 |
| **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

## close_com

**purpose** Disable specified RS232 port

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

**example call**
```
close_com(2);    /* close com2 */
```

**description** The close_com disables the RS232 port specified.

**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 read, 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 read, 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

**clear_com**

**purpose** Clear receive buffer

**syntax**
```
void clear_com(int port);
int port;    /* port to be cleared, 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

**com_eot**

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

**syntax**
```
int com_eot(int port);
int port;    /* port to be accessed, 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** 1, prior transmission still in course
0, transmission completed

**com_overrun**

**purpose** See if overrun error occurred

**syntax**
```
int com_overrun(int port);
int port;    /* port to be accessed, 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 accessed, 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

**com_cts**

**purpose** Get CTS level

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

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

## 5.13 RS485

An RS485 communication port is equipped for multi-station communications. It utilizes the same UART port as RS232 COM2. Only one of them can be enabled at a time. That is, if one of them is enabled, another is disabled.

### 5.13.1 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,

### 5.13.2 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.

### 5.13.3 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.

### 5.13.4 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

### 5.13.5 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.

### 5.13.6 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.

### 5.13.7 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).

### 5.13.8 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);

**open_485**

| | |
|---|---|
| **purpose** | Open RS485 communication port |
| **syntax** | void open_485(int master, int ID);<br>int master; /* 1/0, master/slave station */<br>int ID; /* station ID from 1 to 255 */ |
| **example call** | open_485(0, 10); /* slave station #10 */ |
| **description** | this routine enables RS485 port and set its station ID and communication attribute (master/slave). |
| **returns** | none |

**close_485**

| | |
|---|---|
| **purpose** | Close RS485 communication port |
| **syntax** | void close_485(); |
| **example call** | close_485(0, 10); |
| **description** | this routine disables RS485 port. |

**returns** none

## test_485

**purpose** Test RS485 circuitry

**syntax** int test_485();

**example call** if (test_485() >0) printf_us("RS485 OK");

**description** this routine takes use of the RS485 driver inherent roll-back to see if the RS485 circuitry is working properly. It sends out all possible characters and checks to see if the same characters received.

**Note** that the cable must be disconnected when using this routine.

**returns** 1, roll-back OK
0, fail

## read_485

**purpose** Read RS485 packet received

**syntax**
```
int read_485(char *s);
char *s;    /* string pointer where received packet to be copied */
```

**example call**
```
char s[50];
if (read_485(s) > 0) {
    printf_us("String %s received", 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

## write_485 (for master)

**purpose** Send a string to slave station

**syntax**
```
void write_485(int ID, char *s);
int ID;     /* destination slave ID*/
char *s;    /* string to be sent */
```

**example call**
```
write_485(5, "READ");        /* send string "READ" to slave #5 */
```

**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** none

## write_485 (for slave)

**purpose** Prepare echo string to master station

**syntax**
```
void write_485(char *s);
char *s;    /* string to be sent */
```

**example call**
```
write_485("DONE");          /* echo string "DONE" when polled */
```

**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** none

## 5.14 Memory

Flash and SRAM manipulation routines are described in this section.

**download**

**purpose** Flash programming routine

**syntax**
```
void download(int port);
int port;          /* COM port to be used, from 1 to 3 */
```

**example call**
```
open_com(1, 0x08);          /* 38400, N, 8 */
download(1);     /* download code from COM1 */
```

**description** The download is used to get the new program code from RS232 port and then write them to the flash memory. The RS232 port baud rate, parity and data bits are not set by this program and should be set prior call to this routine. At the end, this routine jumps to the system start point and the system will re-initialize again.

**returns** none

**clone**

**purpose** Program another machine

**syntax**
```
int clone();
```

**example call**
```
clone();
```

**description** This routine is used to update program codes of another machine and is a complement of the function download(). It automatically searches which RS232 port is connected and then duplicates its own code to the machine attached. The RS232 port is fixed to 38400,n,8. For 210, all 3 RS232 ports (COM1 to COM3) are searched whereas for 510, as COM2 is fixed to RS485 use, it is not tested.

**returns** 1, ok
-1, fail

**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.

**test_memory**

**purpose** SRAM read/write test routine

**syntax**
```
int test_memory(char * saddr, long length, int value, int step);
char * saddr;     SRAM start address to be tested
long length;      size of the SRAM in words to be tested
int value;        starting value to be filled
int step;         increment of the value
```

**example call**
```
if (test_memory(0x420000, 0x4000, 0x55, 0x03) != 0)
        printf_us("SRAM test fail");
```

**description** This routine is used to test SRAM. The SRAM area to be tested were filled with known values and then read back for verification on word basis. Then the SRAM contents are restored. The first location was filled with the *value* while the succeeding location was filled by adding the *step* to the value of the preceding location.

**returns** 0 if test ok, -1 if fail

# 6. Standard Library Routines

The standard library routines supported as categorized and listed below,

## 6.1 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.

## 6.2 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 |

## 6.3 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 |

## 6.4 Mathematical Functions : <math.h>

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

```
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. |

## 6.5 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.

## 6.6 Diagnostics : <assert.h>

Not supported.

## 6.7 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)

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

Not supported.

## 6.9 Signals : <signal.h>

Not supported.

## 6.10 Date and Time Function : <time.h>

Not supported.

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

Please refer to limit.h and float.h.

# 7. Appendix

## 7.1 Syntech Library Functions List

**System**

| | | |
|---|---|---|
| const int WHO_I_AM | specify target machine | 5-2 |
| void system_start() | restart system | 5-2 |

**Reader**

| | | |
|---|---|---|
| unsigned char ScannerDesTbl[28] | reader operation attribute | 5-3 |
| extern char *CodeBuf | decoded data buffer | 5-3 |
| extern int CodeLen | length of decoded data | 5-3 |
| extern char CodeType | type of symbology/standard | 5-3 |
| extern char ScannerNo | reader port of current decoding | 5-3 |
| int Decode() | decode barcode/magnetic card | 5-8 |
| void HaltScanner1() | stop reader 1 | 5-9 |
| void HaltScanner2() | stop reader 2 | 5-9 |
| void InitScanner1() | initialize and enable reader 1 | 5-9 |
| void InitScanner1() | initialize and enable reader 1 | 5-9 |

**Buzzer**

| | | |
|---|---|---|
| void on_beeper(int *sequence) | activate beeper | 5-11 |
| void off_beeper() | de-activate beeper | 5-11 |
| void volume(int vol) | set beeper volume | 5-11 |
| int beeper_status() | check if beeper in sequence | 5-10 |

**Calendar**

| | | |
|---|---|---|
| int set_time(char *new_time) | set calendar time | 5-13 |
| int get_time(char *time_s) | get calendar time | 5-13 |
| int adjust_timer(int offset) | fine tine calendar | 5-12 |

**Digital Input / Output**

| | | |
|---|---|---|
| int get_din(int di) | read digital input | 5-38 |
| void set_do(int do, int mode, int duration) | set digital output | 5-38 |

**LED**

| | | |
|---|---|---|
| void set_led(int led, int mode, int duration) | set LED | 5-38 |

**Keypad**

| | | |
|---|---|---|
| void clr_memkb() | clear keypad input buffer | 5-44 |
| int mem_kbhit() | check if keypad input buffer available | 5-41 |
| int mem_getchar() | read one character from the keypad buffer | 5-42 |
| unsigned long scan_multi_key() | read combination-keys | 5-42 |

**External AT Keyboard**



**LCD**



**Power**



**RS232**

**RS485**

| | | |
|---|---|---|
| unsigned int RS485_STW | RS485 status word | 5-61 |
| void open_485(int master, int id) | enable & setup RS485 | 5-62 |
| void close_485() | disable RS485 | 5-62 |
| int test_485() | RS485 loop back tester | 5-62 |
| int read_485() | read a packet from input buffer | 5-63 |
| void write_485(int id, char *s) for master | send string s to slave station id | 5-63 |
| void write_485(char *s) for slave | prepare echo string to be polled | 5-64 |

**Memory**

| | | |
|---|---|---|
| void download(int port) | download program code from RS232 port | 5-65 |
| int clone() | program another machine | 5-65 |
| void init_free_memory() | initialize file space | 5-65 |
| int test_memory(char *saddr, long length, int start, int step) | SRAM read/write test | 5-66 |

**File (Common)**

| | | |
|---|---|---|
| long free_memory() | get size of free memory | 5-25 |
| int read_error_code() | reader global error code | 5-31 |
| int access(char * filename) | check file existence | 5-17 |
| int rename(char *oname, char*nname) | change file name | 5-34 |
| int remove(char fd) | delete a file | 5-33 |
| int filelist(char *dir) | get file directory information | 5-25 |

**File (DAT)**

| | | |
|---|---|---|
| int open(char *name) | open a DAT file | 5-29 |
| int close(int fd) | close a DAT file | 5-20 |
| int ch_size(int fd, long new_size) | extend or truncate a file | 5-19 |
| long filelength(int fd) | get file length | 5-25 |
| long lseek(int fd, long offset, int origin) | move pointer of a DAT file | 5-27 |
| long tell(int fd) | get file pointer | 5-35 |
| int eof(int fd) | check if point to end of file | 5-24 |
| int read(int fdm char *buf, unsigned count) | read number of bytes from a DAT file | 5-30 |
| int readln(int fd, char *buf, unsigned count) | read one line from a DAT file | 5-31 |
| int write(int fd, char *buf, unsigned count) | write number of bytes to a DAT file | 5-36 |
| int writeln(int fd, char *buf) | write a line to a DAT file | 5-36 |
| int append(int fd, char *buffer, int count) | append number of bytes to file | 5-18 |
| int appendln(int fd, char *buffer) | append a string to a file | 5-19 |
| int delete_top(int fd, unsigned count) | remove number of bytes from top of a DAT file | 5-23 |
| int delete_topln(int fd) | delete one line from top of a DAT file | 5-24 |

**File (DBF)**

| | | |
|---|---|---|
| int open_DBF(char *name) | open a DBF file | 5-29 |
| int close_DBF(int fd) | close a DBF file | 5-20 |
| int add_member(int DBF_fd, char *member) | add a member to file | 5-17 |
| int delete_member(int fd, int keyn) | delete a member of a DBF file | 5-22 |
| long member_in_DBF(int fd) | number of members in a DBF file | 5-28 |
| int has_member(int fd, int keyn, char *key) | check if a member exists in a DBF file | 5-26 |
| int get_member(int fd, int keyn, char *buffer) | read the member pointed by | 5-26 |
| long tell_DBF(int fd, int keyn) | get file pointer | 5-35 |
| long lseek_DBF(int fd, int keyn, long offset, int origin) | move pointer of a DBF file | 5-28 |
| int create_DBF(char *name, unsigned len) | create a DBF file | 5-21 |
| int create_index(int fd, int keyn, int key_offset, int key_len) | create an index file of a DBF file | 5-21 |
| int rebuild_index(int fd, int keyn, int pre_index, int key_offset, int key_leng) | rebuild an IDX of a DBF file | 5-32 |
| int remove_index(int fd, int keyn) | delete an index file | 5-33 |