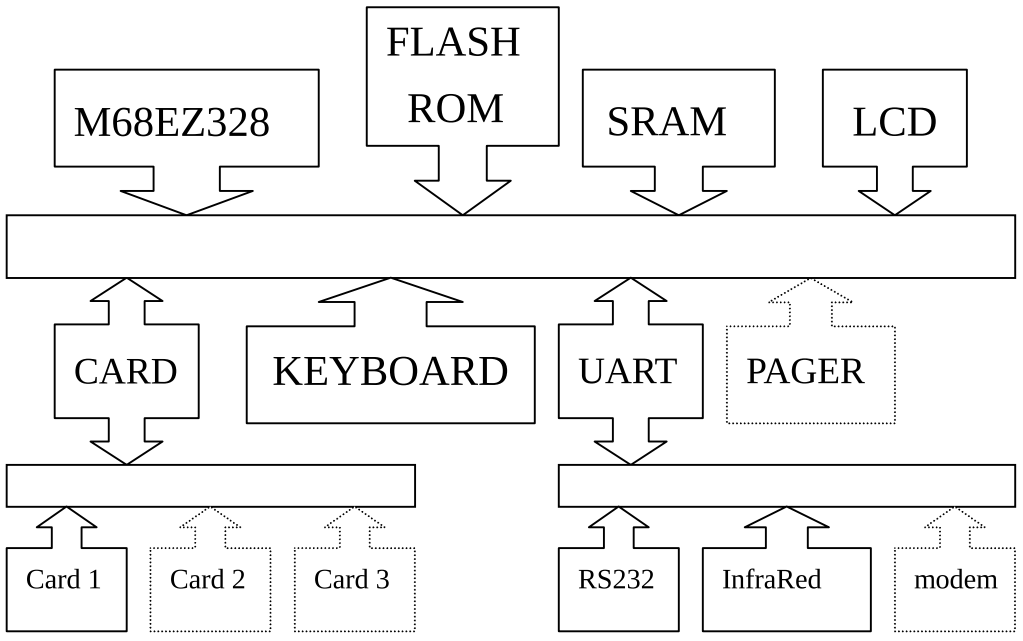
Software Developer's Manual

1.introduction

1.1 MC-998

MC-998 is an advanced handheld battery powered smartcard reader. It features a Motorola Dragonball CPU with 1~4Mbyte of FLASH memory and 128~256kbyte of SRAM. Standard input device consists of a 16-key keyboard and a jog dialer, standard output device is a 128x64 dot matrix LCD panel with several icons. An RS-232 serial port and an IR port are default for serial communications, and an optional modem unit could be used as another serial communication device. Up to 3 card slots could be used (depending on card types). An optional RF unit could be used to receive paging information.



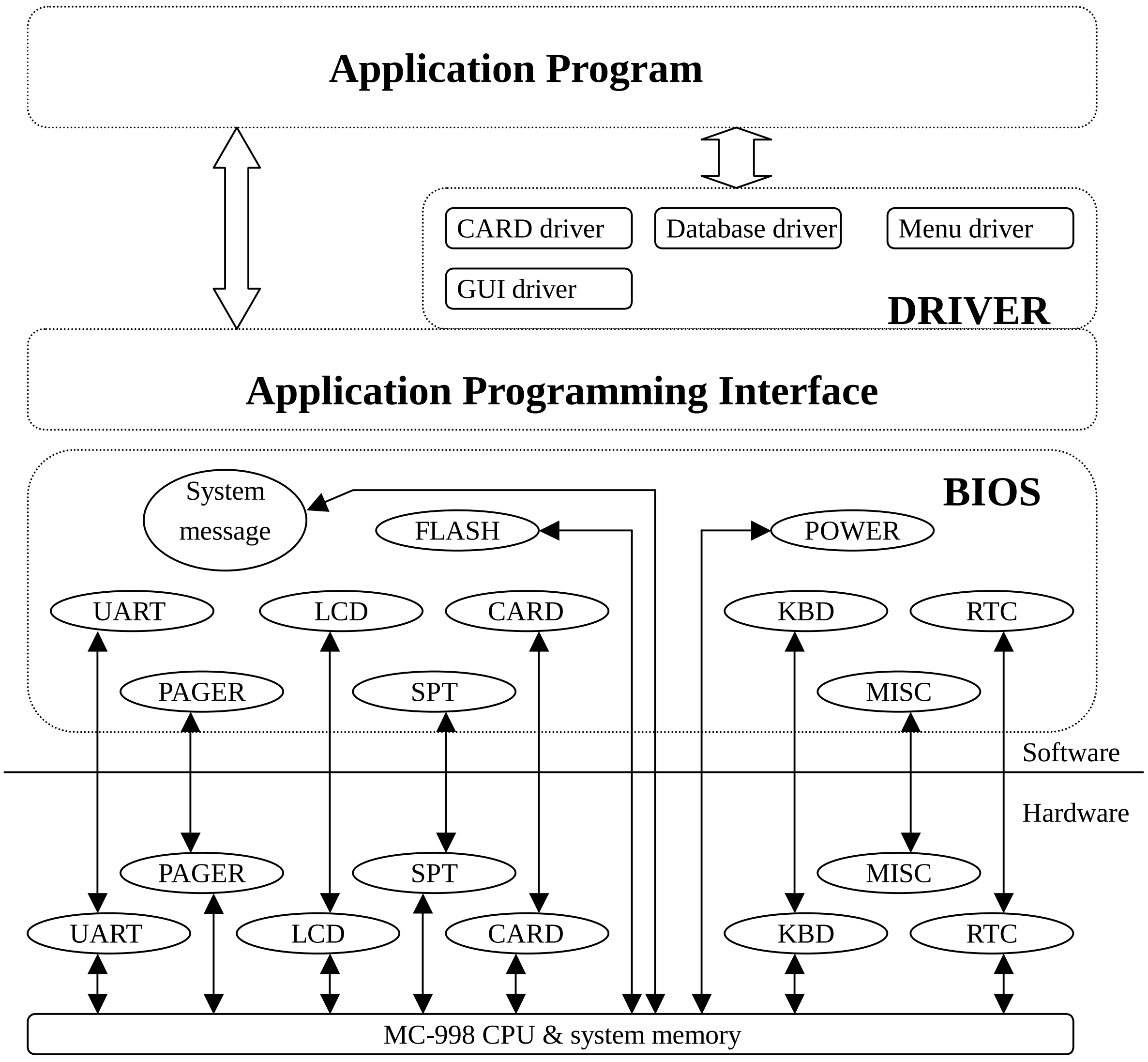
1.2 Software development environment

GNU C is the main programming language, and the GNU m68k tool chain is used for developing. The whole GNU tools chain runs on a PC running MSDOS or an MSDOS window under Win95/98. Debugging tools (debugger, simulator, etc.) is not provided but will be available later.

2.Overview

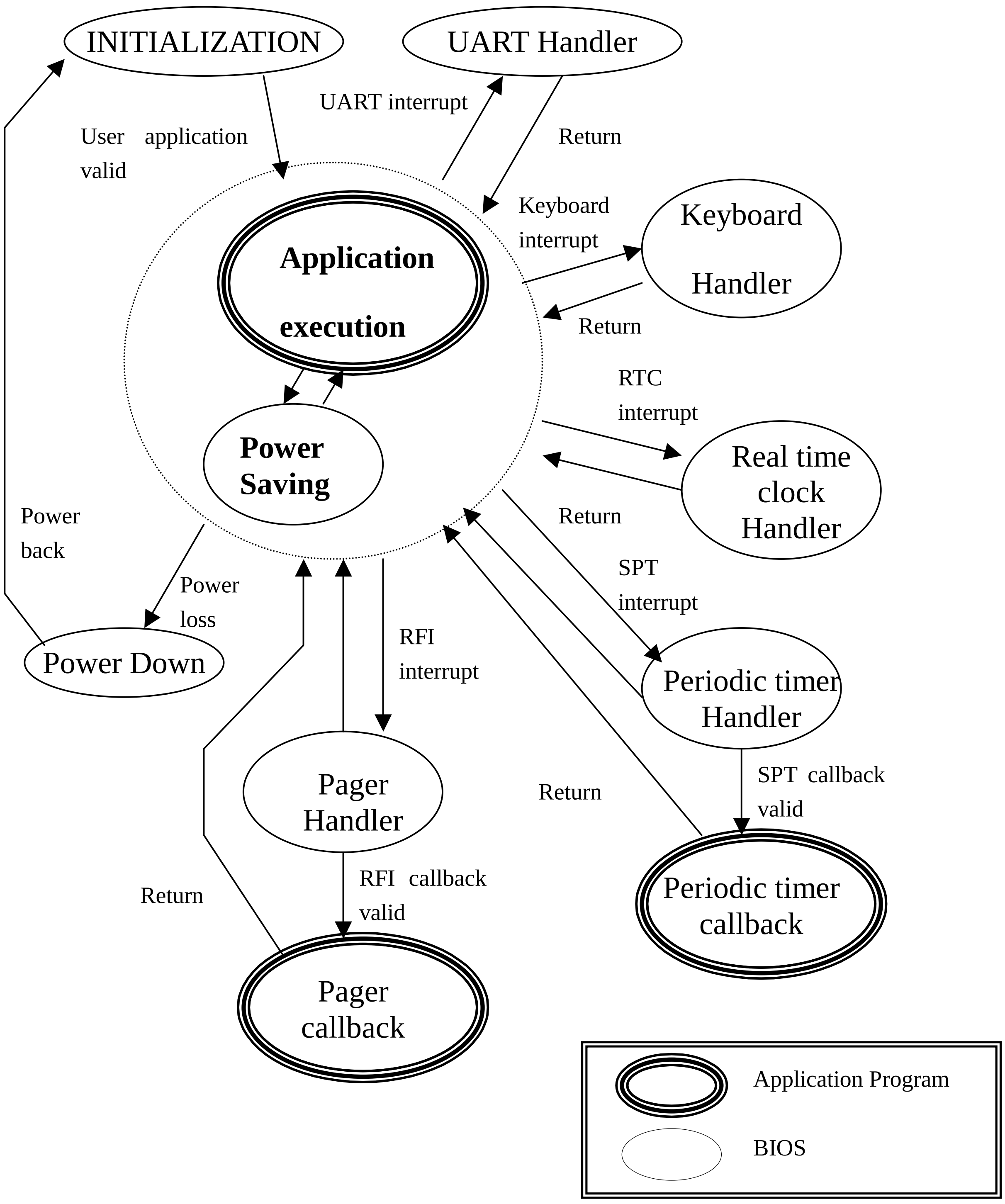
2.1 Architecture

The software of MC-998 consists of 3 layers: BIOS, Driver/API, Application.



BIOS is built in the FLASH of MC-998 when delivered. The APIs are all included in a library called ***libm998.a***; No driver is provided except the card driver for various memory based smartcards.

2.2 Execution states



After initialization, BIOS will check if the user application program is valid. The system will begin the execution of user application program if it's valid. The CPU is in supervisor mode when it’s executing BIOS routines, and user mode when it’s executing user application program.(c.f. m68k manual). The system stays in user application program unless it enters idle mode or interrupts occur. Since MC998 is a battery-powered device, it’s highly recommended that the system stay in the lowest power consuming mode possible most of the time. If no system clock is required for any of the active peripherals (i.e. no UART, no active card session), sleep mode will be the best because the power current could be reduced to as low as 100uA. When system clock is required, idle mode could be used where power current could be reduced to about 10mA for RS-232 peripheral or 3mA for infrared device, while the current will be about 30mA in full speed running mode. The power saving (idle/sleep) mode will not be entered automatically. The only way to enter power saving mode is to call a system message polling routine (c.f. API: system message), which will pull the whole system into power saving mode and returns only when valid events occur.

2.3 Interrupt and exceptions

BIOS will handle all interrupts and exceptions. Any enabled interrupt or exceptions will wake up the CPU if it’s asleep. Any fatal exceptions such as address error or bus error will terminate the application execution immediately.

It’s not possible for user to control these interrupts and exceptions directly. However, several interrupts could be masked if certain message is masked or related device is turned off.

2.4 Callbacks

Two types of user defined interrupt service routine could be implemented via callbacks: periodic timer interrupt, and RFI interrupt for paging information receiving. They are called from BIOS ISR, but it runs in user mode.

Callbacks are only used when timing is critical. Callbacks are ordinary C subroutines, but they should be registered to take effect.

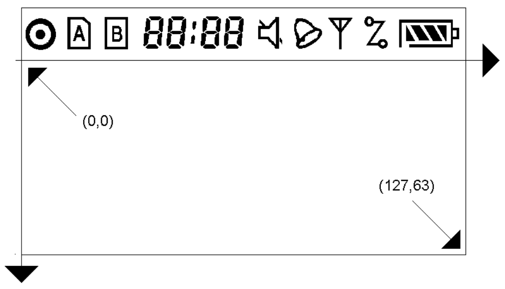
2.5 Subsystems

MC998 consists of several subsystems: display, input, sound & back light, card, serial communication, pager.

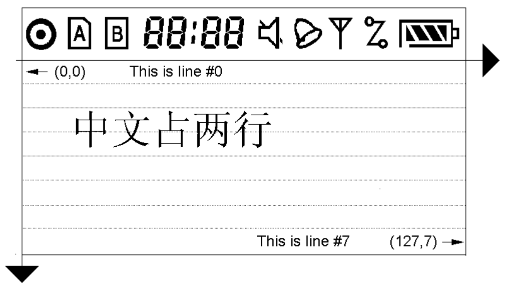
2.5.1 Display

MC998 features a customized 128x64 dot matrix LCD panel with several icons.

There are two sets of coordinates systems for the LCD panel. The graphics routines such as pixel, bitmap uses graphics coordinates, and text routines use text coordinates.

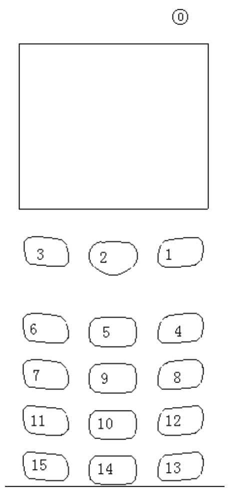


LCD coordinates in graphics mode



LCD coordinates in text mode

2.5.2 Input



Keypad layout

The input device of MC998 consists of a 16-key keypad and a jog dialer. The key on the jog dialer is the same as the Enter key (code 0x0D). The keys on the keypad, including the one on the jog dialer, could be programmed individually to beep and/or turn on EL back-light when pressed.
The numbers on the key in keypad layout figure represents the key scan code returned by key APIs.
The status word of the keypad returned by key API is:

Status word = 0x8000 >> scan code

2.5.3 Sound and back light

MC998 has a buzzer driven by square wave generator built in the Dragonball. The frequency and duty cycle of the wave could be specified via API routines.
The back light system consists of keypad back light, which is LED, and LCD back light, which is an EL panel, and they are connected so it's not possible to control them individually.

2.5.4 Smart Card

There are 2 full-functional smart card interface unit in MC998. One is the main interface, where card insert/gone events are detected. Any kind of smart card could be used in the slot connected to the main interface. The second one is the auxiliary interface, which could be configured as either a full-functional slot where all kinds of cards could be used, or 2 slots with reduced function. When the auxiliary interface is configured as 2 slots, there are 2 hardware options: 2 slots that only one card could be inserted into one of the 2 slots any time, while it's full-functional; or 2 slots that 2 cards could be inserted simultaneously, while only CPU-based smart cards could be used. If it's configured as the later, different card drivers should be used.

2.5.5 Serial communication

3 peripherals are connected to the UART unit of MC998: a standard RS-232 interface, an infrared transceiver, and an optional modem. They cannot be used simultaneously. Various baud rates are supported (up to 115200 for RS-232 and modem, up to 3579MHz for infrared).

The RS-232 interface is used to connect MC998 to another device using RS-232, such as PC, another MC998, a modem, or a printer, using the cable provided.

The infrared transceiver could be used to connect to other IR devices. IrDA support is not a part of API, but it could be implemented using API.

In fact, an external modem would work well using RS-232 interface, but we also provide an option to put a modem inside. A simple 2400bps modem could be mounted so MC-998 could connect to phone line via the RJ11 socket directly, but additional device driver is required.

2.5.6 Pager

A paging information receiver is an option. Only POCSAG page is supported now, but FLEX paging will be available soon.

This option will increase the power consumption significantly.

2.6 Memory and data storage

System memory of MC-998 consists of 128k~256k bytes SRAM and 1M~4M bytes FLASH ROM.

The data stored in the SRAM could retain for several hours after the main battery has gone. Stack, heap, program data is stored here.

The FLASH ROM contains program code (including BIOS, configuration, user applications) and some program data (font data, constant strings, etc). The main difference between FLASH and SRAM is that even though FLASH could be read just like SRAM, the writing is a quite different process, and modification is even more complicated: in the worst case, a block (up to 64k bytes) should be erased before any single bit writing could be performed. Since it’s quite large, it could be used for non-violate data storage. A sample database driver using FLASH as data storage is provided.

2.7 Power management

Power management is vital for a battery-powered device as MC-998. Any high power consuming peripherals could be shut down individually to reduce power consumption: RS-232, IR, modem, card interface, pager, internal devices, etc.

Here are the fundamentals for power management:

Turn off any peripherals not in use;

Always enter a power saving mode;

Do not hang around while a power-consuming device is on. Just hit and run and go back to sleep. It’s especially useful when dealing with smart cards: if a session contains intensive computing, just turn off the card; avoid single byte/word read when *you want to deal with a data block; turn off the card as soon as it's possible.*

2.8 Real time clock and periodic timer

A real time clock (RTC) is built inside MC-998. It's just like an ordinary watch, with date, time and alarm, that keeps on running as long as the battery is there (no matter main or backup). It could be read/set by APIs, and sends a message when alarming.

Periodic timer is a single shot timer which could be set to 1/64~1024 second with 1/64s resolution. It's disabled after the preset time has elapsed. It will send a message when it's over, and a callback could be attached to the ISR so that user program could act immediately.

Note: Periodic timer will wake the system up every 1/64-second, so it will be more power consuming.

3.Programming

Programming on MC998 is quite like programming on DOS, even simpler: there's only one application that never returns. After the BIOS has done its job initializing the system, the control will be handed over to the application. The application is mainly written in C. In the C runtime routine (CRT0), i.e. start file, all variables are initialized.

After that, is the well-known **main()** subroutine. Please note that main() is a no-return subroutine.

3.1 Initialization

The first thing to do in main() is to initialize all the hardware. BIOS has initialized them all and turned everything off, now it's up to the application to turn on the ones it needs:

The display, keyboard, card interface, perhaps pager, etc. If you have any callbacks, you'd better register them now.

3.2 Typical application flow

After initialization, now that's what's called dead-loop. The application calls sysmsg() continuously to check for any events. After an event comes, the application would try to process it and then go back to sysmsg() for more.

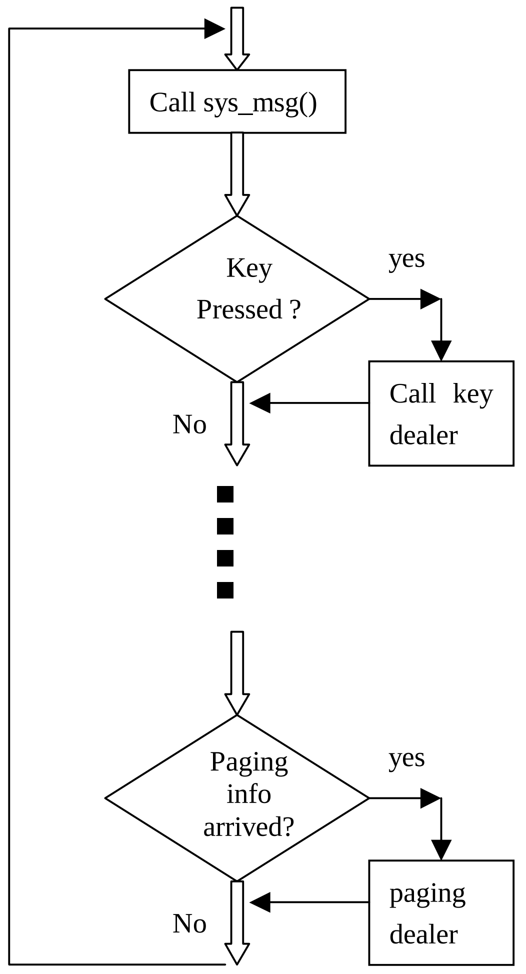
Actually there are two different ways to handle this in a multi-screen menu-based user interface:

One, The dead-loop itself is the message-polling loop (MPL). Each screen is registered with its own event dealers and screen descriptions/refreshers, so the MPL will know what to do with a certain event.

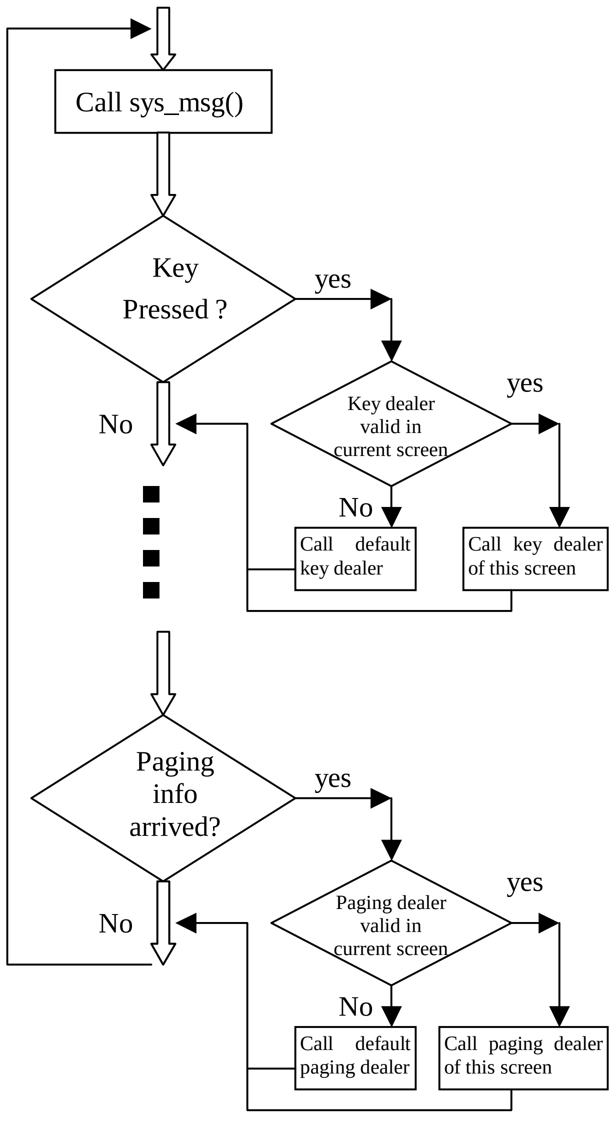
Two, each screen has its own MPL. Now the MPL does not have to be a dead loop.

The second one is simpler but when the menu system gets larger, it will become more like a mess.

The first one is not a very good one for small-scaled projects, but it's highly expandable and it's easy to develop complicated GUIs with visual programming tools. In fact, a menu driver could reside in the driver layer, and does not have to change from application to application at all.



Message Polling Loop
for method 2


Call sys_msg()
Key Pressed ?
yes
Key dealer valid in current screen
yes
No
No
Call default key dealer
Call key dealer of this screen
Paging info arrived?
yes
Paging dealer valid in current screen
yes
No
No
Call default paging dealer
Call paging dealer of this screen

Message Polling Loop
for method 1

4.Interfacing with subsystems

All subsystems could be controlled via calling the BIOS routines.

4.1 Display

Basically, there are two ways to use the display device: write some text, and draw something.

4.1.1 Device control

After initialization by BIOS, the display is ready.

Three modes are available for the display: working, power saving, off.

In working mode, any data in the screen buffer will be displayed; any data change will immediately be shown on the display.

In power-saving mode and off mode, the main display area including dot matrix and most of the icons are turned off, so no data change will be shown. Power consumption is extremely low in these two modes (several microamperes).

Power-saving mode and off mode are almost the same except for the static icon. The static icon is the little bull’s eye on the upper left corner. It could be set to light up or blink in working mode and power-saving mode, but it’s turned off in off mode. It’s especially useful when you want to cut the power consumption to the extreme, and still want to have an indicator. For example, the static icon could indicate power on when lit up, unread message when blinking, etc.

All data sent to display will be stored in the screen buffer on the display device itself, no matter what the mode the device is in. The display data is not visible until the device is working, and will not be lost when the device is turned off or switched into power-saving mode.

So, please make sure the content of screen buffer is what you need before the display is switched into working mode.

API : **Disp_init**

4.1.2 Clearing

It’s always safe to clear the display before screen switching or mode switching if you do not want to see the corpses of last screen. A certain rectangular area could be cleared as well as the whole screen.

APIs: **clr_scr()**
clr_win()

4.1.3 Text and fonts

In text mode the display acts like a notebook that you could write strings row by row.

4.1.3.1 Text

Text is represented in 8-bit (char) for ASCII character, and 16-bit for Chinese character. The string to be displayed should be a NULL terminated string.

4.1.3.2 Cursor

A cursor is where the text should be displayed. Cursor is invisible and it’s write-only. After a string has been displayed, the cursor will automatically set to the end of line

(the position next to the last character displayed). The X coordinate (horizontal position) of the cursor could be 0~127 by pixels, So, you could always put a string in the middle of a row; but for Y coordinate (vertical position), only 8 possible rows are valid (0~7). For example, when a string is to be displayed at (90,7), the upper left corner of the displayed string is (90, 56) in graphics coordinates.
Please always set cursor position unless you are absolutely sure where it stays.
API : **goto_xy()**

4.1.3.3 Fonts and attribute

Font defines how each character should be displayed. A font description should at least include:

Size:	dimension of the character, width by height. Width is fixed for most of the fonts, but variable width is supported
Char set:	range of displayable characters. 0x20~0x7f for most ASCII fonts, e.g.

Single byte of dual byte: shows if a char is 8-bit or 16-bit.

A default font library is included when MC-998 is delivered. It consists of following fonts:
5x7 English character (6x8 actually)
5x7 Cyrillic character (6x8 actually)
7x9 English character (8x16 actually)
16x16 GB-2312 Chinese character (2 rows needed for 1 char)
5x7 graphics symbol for a typical pager. (xX8, variable width)
And several string tables used in paging industry.

Attribute defines if the text should be displayed as inverse.

Attribute and font must be selected before any actual screen text writing.

API : **Disp_set_font_attribute**

4.1.3.4 Writing strings

After the cursor and font/attribute has been set, it’s easy to write a string. Single character could also be written to improve the efficiency.
Please be ware that if the string is too long, the final display will be clipped.
The number of the characters that could be displayed in a row depends on the font selected. And for some fonts, more than one row is needed. E.g., the whole display will be 21x8 characters when 5x7 ASCII fonts is used, and 8x4 characters if 16x16 Chinese character is used.

API : **Disp_write_str**
Disp_write_char

4.1.4 Graphics

In graphics mode, the whole display is treated as a dot matrix and all drawing is pixel based.

Pixel based coordinates are used in graphics mode, i.e. (0,0) for upper left corner and (127,63) for lower right corner.

4.1.4.1 Display data updating modes(put modes)

There're four ways to draw pixel or a group of pixel (bitmap) onto display:

Direct drawing	: replace the display data (DD) with the new data (ND), no matter what the original data (OD) is.
OR	: DD = OD or ND
XOR	: DD = OD xor ND
AND	: DD = OD and ND

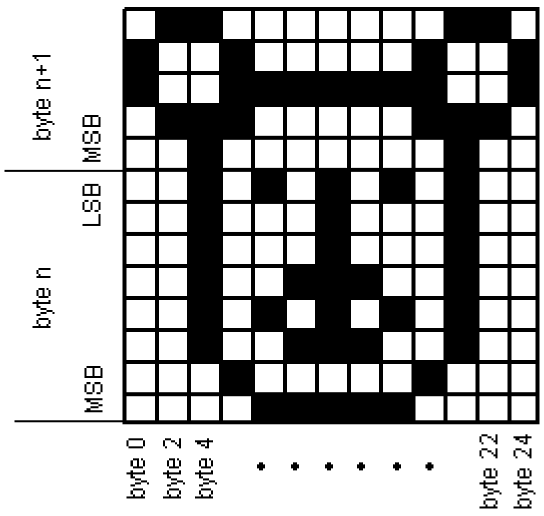
Put mode is a parameter to pass through when drawing pixel.
But it should be set prior to bitmap drawing.
API: **Disp_set_bmp_put_mode**

4.1.4.2 Pixels

In graphics mode, the most direct way to draw something is to draw pixel by pixel. It's less limited but it's quite slow.
The pixels could also be read so that you could know what's on the screen right now.
API: **get_pixel()**
put_pixel()

4.1.4.3 Bitmaps

Bitmap is defined as the data of a rectangular area on the display. It can be read, stored in the memory elsewhere, modified if needed, then put back or put elsewhere.
It's a convenient way to change the screen rapidly, and it is the fundamental method for animation.
Bitmaps is stored as a record shown in API manual (bitmap structure). According to the data organization of the display device, the bitmap data is stored vertically:



The bitmap data of the teddy face shown to the left is:

0x00, 0x30, 0x00, 0x48,
0x3F, 0xC8, 0x40, 0x70,
0x91, 0x20, 0xA8, 0x20,
0xBF, 0x20, 0xA8, 0x20,
0x91, 0x20, 0x40, 0x70,
0x3F, 0xC8, 0x00, 0x48
0x00, 0x30

Note: Clipping is always needed when part of the rectangular is outside the screen. There's a little difference between the clipping of bitmap reading and clipping of bitmap writing. When clipping is needed for bitmap reading, the size of the bitmap read in is adjusted so that only the graphics data on the screen is stored. E.g., if you want to read a 10x10 rectangular bitmap at (120,56), you will only get a 8x8 bitmap. But for clipping in bitmap writing, bitmap data is not effected, the part outside the screen is simply invisible. So, caution should be used while a series of bitmap reading and writing is used for animation: you might loose the data while the object is moving across the screen border.

API: **Disp_get_bmp**
Disp_put_bmp

4.1.5 Icons

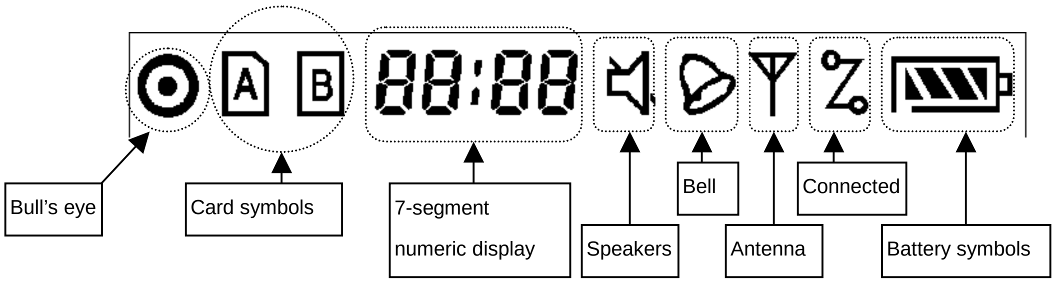
4.1.5.1 Control groups of icons

The icons are controlled by BIOS after reset. There are 8 groups of icons, each containing one or several icon segments which could be controlled individually.

If the application wants to use some of the icons, it should obtain control of them from BIOS first.

Here are the default functions of the icons set by BIOS:

Bull's eye:	lit for LCD on mode and power-saving mode, off for LCD off mode;
Card symbols:	$\boxed{B}$ lit when main slot card inserted, off when gone;
7-segment display:	current time;
Speakers:	n/a
Bell:	n/a
Antenna:	n/a
Connected:	n/a
Battery symbols:	n/a

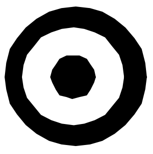


API: **Disp_icon_customize**

4.1.5.2 Icon data

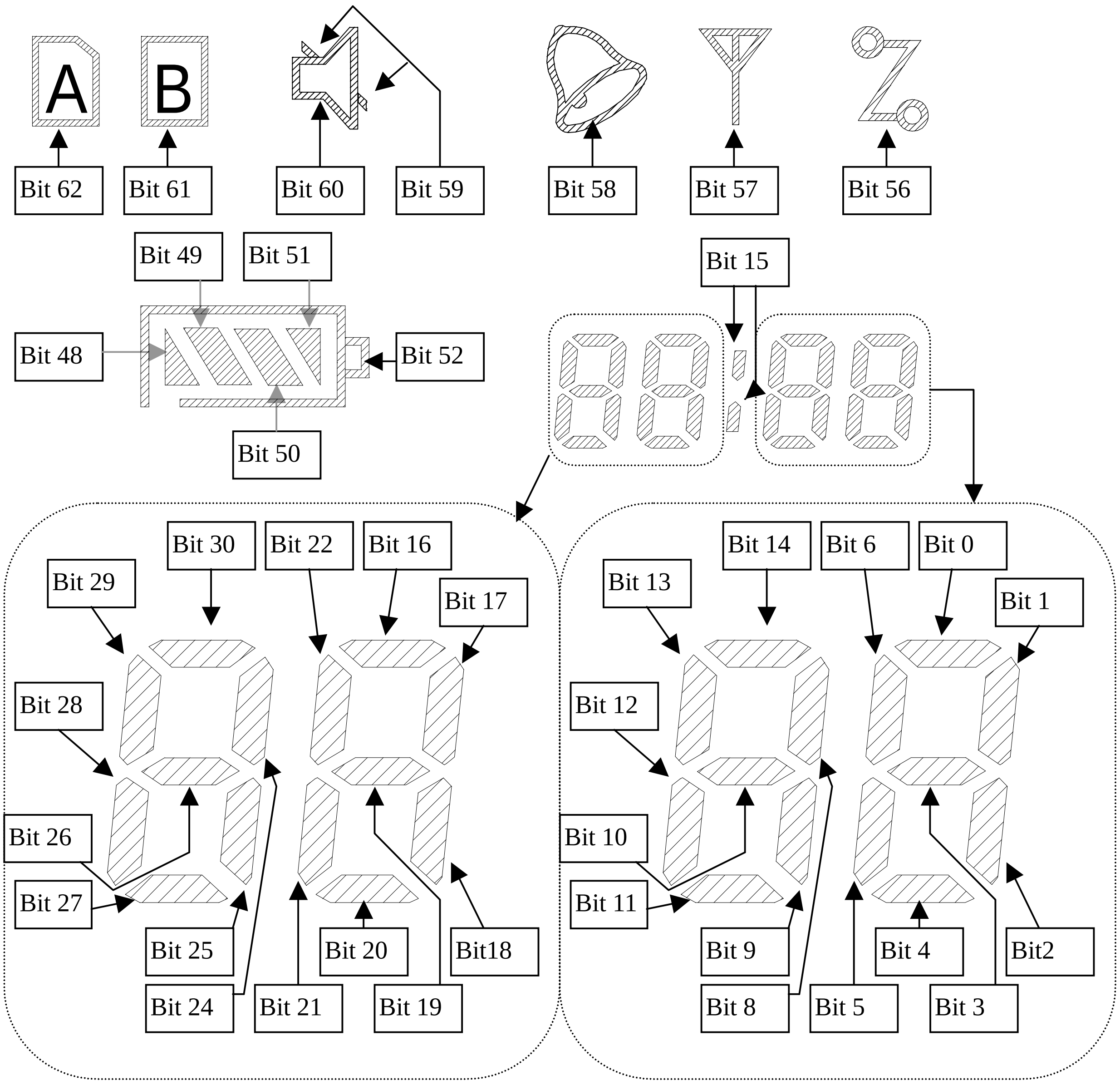
The icon data is mapped into a 64-bit word by bits. It could be read/written.

Here's the map for these bits, bit 0 is the LSB, and bit 63 is the MSB:



The state of the bull's eye is defined by bit 40 and 41:

Bit 41	Bit 40	State of the bull's eye
0	0	Off
0	1	Blinking at 2Hz
1	0	Blinking at 1Hz
1	1	Always on



API: **Disp_icon_set**

4.2 Keyboard and jog dialer

4.2.1 Initialization

The input subsystem must be initialized. Any input mode change would also invoke initialization. Initialization will also set if the key press will also turn on the EL back light and if the buzzer will be on while a key is held down.

Please note that initialization would clear the key buffer and the key status, so calling the initialization subroutine while a key is held down will make the system quite confused that once again it will send the key pressing message, push the key code, just like a key is pressed.

API: **KEY_init**

4.2.2 Key buffer mode

Key buffer mode is the most commonly used mode. When a key is pressed, a system message is sent, and the key code is pushed into key FIFO. Since the key is buffered (16 keys), the key code won't be lost even a quick hand is typing. Reading the FIFO will pull the key code out of the FIFO as long as the FIFO is not empty. No further system message would be sent as long as the key stay pressed. A key pressed while another key is held down will not be recognized until the other key is released:

		time		
Key 1	pressed		released	
Key 2		pressed		released
FIFO	key 1 pushed		key 2 pushed	
Sysmsg	MSG_KEY_AVAILABLE		MSG_KEY_AVAILABLE	

There might be some side effect for key buffering, so performing a read-till-empty on the FIFO would be a good idea sometimes (especially right after the key unit initialization).

API: **KEY_read**

4.2.3 1-Key mode

In 1-key mode, both key pressing and key releasing will wake the system up and send system message. The key code is not stored in FIFO, it will be lost if it's not processed in time. Just like in the key buffer mode, a key press will block other keys. It will be useful when you want to implement some user interface featuring key hold, e.g., press key 'A' for over 2 seconds to turn on the light.

		time		
Key 1	pressed		released	
Key 2		pressed		released
Key read	key 1	key 1	key 2	key 2
Sysmsg	MSG_KEY_DOWN		MSG_KEY_UP MSG_KEY_DOWN	MSG_KEY_UP

API: **KEY_read_up**
KEY_read_down

4.2.4 Key status

The bit map of the key matrix could be read. It's handy when you want to check if several keys are pressed simultaneously. Only 16 bits are valid, and there is no status bits for the directions of the jog dialer.

API: **KEY_get_status**

4.2.5 Key beep and auto EL on

Key beep and auto EL on could be turn on/off individually, so some special keys like shift and power could be implemented as 'silent keys'. The directions of the jog dialer are always silent.

API: **KEY_mask_set**
KEY_mask_read

4.3 UART: RS-232, Infrared, modem

4.3.1 UART parameters

Several parameters could be specified to configure serial communication:

1) Parity: could be non parity, odd parity or even parity;
2) Stop bits: could be 1 or 2 stop bits;
3) Data bits: could be 7 bits or 8 bits;
4) Baud rate: could be one of the followings:
   115200, 57600, 28800, 14400, 38400, 19200, 9600, 4800, 2400, 1200, 600, 300

Higher baud rate is recommended to increase efficiency when stable connections could be established, such as RS-232 cable connection or close range IR connection (with in 20cm).

Since the UART initialization routine also selects one of the three possible peripherals, so be ware of the parameters to pass through to avoid unwanted peripheral switching when changing UART parameters.

API: **UART_init**

4.3.2 Function control

Several aspect of the serial communication could be adjusted via function controls:
1) TX polarity : could be either positive or negative logic
2) RX polarity : could be either positive or negative logic
3) RTS setting: could be forced high, forced low and automatically set.
4) Some testing controls : loop test, forced parity bit error
5) CTS check : could be disabled
6) Break : break condition could be initiated
7) TX on : TX of the UART could be turned off

The setting of function control is very protocol-specific, please check out the protocol carefully before using any of the controls.

API: **UART_fcntl**

4.3.3 Data accessing

The serial communication could be full duplex: all the data received is stored in a FIFO while transmitting is active. The receiving is in the background so no data would be lost. System message MSG_COMM_DATA will be sent as long as the FIFO is not empty. The FIFO could be read using API routines.
The transmitting is in the foreground. Data could not be thrown into UART too fast or traffic jam occurs. System message MSG_COMM_SB_EMPTY will be sent as soon as the sending buffer is empty and UART is ready to get a new TX data. An alternative way to send a series of data is status polling: wait until the UART send buffer is empty.

API: **UART_get_char**
UART_send_char

4.3.4 Status inquiry

Status inquiry is very important for UART communication. Nearly every aspect of the UART state is returned.
One obvious example for status inquiry is power controlling: since UART and its peripherals are very heavy for the battery, it's always better to turn the entire subsystem off as soon as possible. But unplugging it in the middle of a transmission will result in a broken byte, so status polling is needed to ensure the UART is not busy, i.e. a byte has been finished.

API: **UART_stat**

4.3.5 Peripherals

The peripherals connected to the UART are heavy loads, even when there is no active communication. Do not turn them on unless it's absolutely necessary.

4.4 Smart cards

Smart card subsystem should be initialized first. Once is enough. Card insert/removing detection of the main interface would be available after this routine.

API: **Card_init**

4.4.1 Card ATR

The card interfaces are off when the system enters **main()**. The ATR subroutine turns on the card interface and gets the ATR of the card inside the designated card slot. To save time and energy, it's better to check the designated card type only. Still it's possible for the application to check what card had been inserted by first initiating an ATR session for memory card, then one for asynchronous card. More specific synchronous card ATR subroutines are available in card driver library.

API: **Card_ATR**

4.4.2 Card deactivating

The card interface and the card inserted are such heavy loads that ordinary battery could only drive them for a couple of hours if they are continuously turned on. It's always a good idea to deactivate the card and the corresponding card interface ASAP.

The entire card operation should better be divided into several tightly packed sessions that are separated by intervals where the card is unpowered.

time →

Card operation	ATR	READ	DEACT		ATR	WRITE	DEACT
Interface mode	ON		OFF		ON		OFF

API: **Card_deactivate**

4.4.3 CPU based card (asynchronous card) interface

4.4.3.1 Timing

The timing control of asynchronous cards is implemented in the BIOS. The API routine to adjust the timing parameters should be called right after the ATR routine to make sure the parameters specified in the ATR will take effect.

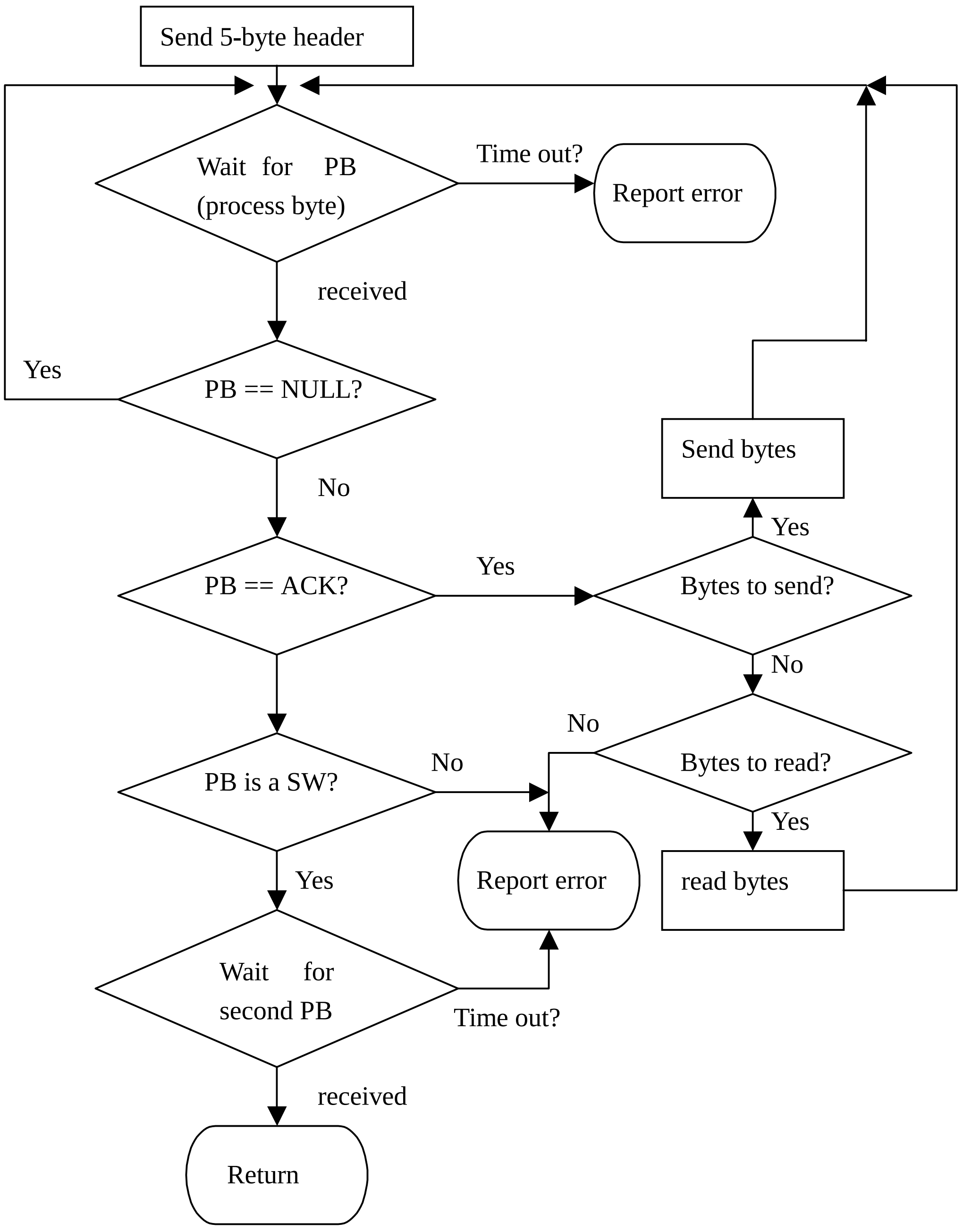
API: **Card_ATR_analysizer**

4.4.3.2 Protocol type

Two types of cards is supported in current version of BIOS: T= 0 and T = 1.

It's quite easy to use T=1 interface, the API is packet based.

For T = 0, things get a bit more complicated. Only part of the communication session is implemented to provide flexibility: after the command header is sent, only ACK and NULL is processed by BIOS. So, only normal header-ACK-IO-SW will be processed by BIOS, any other SW in the process byte will push the subroutine back to its caller.



Card_exchange_T0_header

API: **Card_exchange_T0_header**
Card_exchange_T1_frame

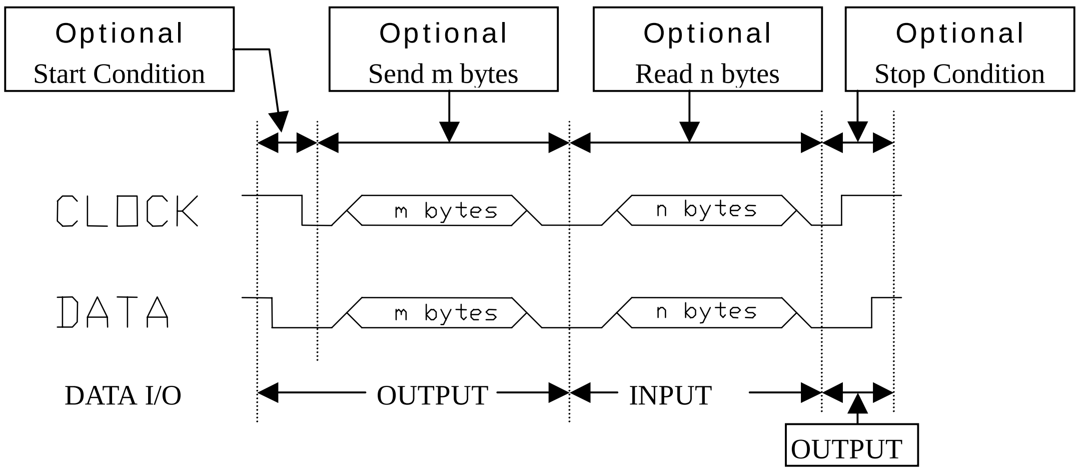
4.4.3.3 Caution

Since the system clock is changed during asynchronous card communication session. UART and buzzer may be out of order. It's recommended that UART and buzzer should be off when asynchronous card is active.

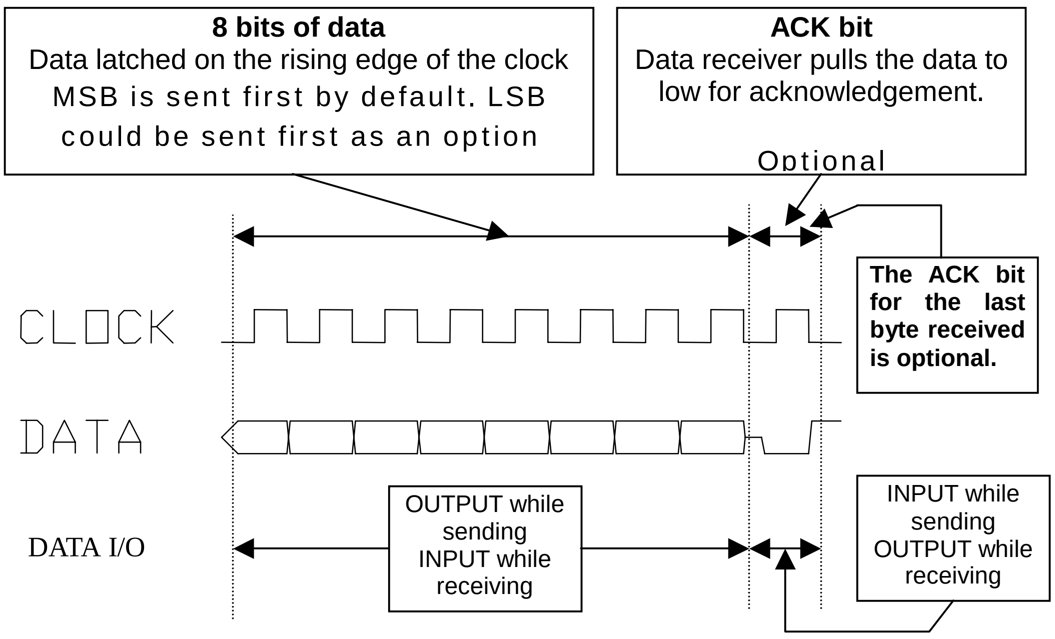
4.4.4 Memory based card (synchronous card) interface

4.4.4.1 I^2C-like serial protocols

Most of the synchronous card communication protocols are derived from I^2C or SPI. So it's reasonable to describe them with a base protocol. Here we define a simplex two-wire serial communication protocol that both I^2C and simplex SPI could be derived from.



I^2C-like serial protocol: packet session


8 bits of data
Data latched on the rising edge of the clock
MSB is sent first by default. LSB could be sent first as an option
ACK bit
Data receiver pulls the data to low for acknowledgement.
Optional
The ACK bit for the last byte received is optional.
CLOCK
DATA
DATA I/O
OUTPUT while sending
INPUT while receiving
INPUT while sending
OUTPUT while receiving

I^2C-like serial protocol: byte session

4.4.4.2 Pin based operations

Special protocols could be implemented using direct pin control API tools. A pin could be set as input or output, could be read when set as input, write as high or low when set as output.

Direct pin control is very slow; so do not use it unless it's absolutely necessary.

API: **Card_pin_session**

4.4.5 Writing card drivers

Card driver is a middle layer between API and user application to simplify the application program. It's especially necessary for synchronous cards. Well-programmed card drivers could improve efficiency significantly.

Here's how to write driver for a synchronous card:

1) Read the data sheet / manual of the card over and over, pay special attention to the timing chart of the serial communication
2) Determine at least how many program interface routines are needed for card operation.
3) Map the protocol to API I^{2}C-like and pin based sessions. Map it with I^{2}C-like sessions as much as possible.
4) Debug each interface routine to make sure it works
5) Pack them up to make a library and write documents for it.

4.5 Real time clock

The API routines for RTC are self-explaining. In order to save space and time, both date and time are packed into 32-bit words. You make need macros to pack them up, and unions to get the fields of the record.

API: **RTC_read_time**
RTC_set_time
RTC_read_date
RTC_set_date
RTC_read_alarm
RTC_set_alarm
RTC_alarm_setting
RTC_get_alarm_setting
MAKE_TIME
MAKE_DATE

4.6 Periodic timer

4.6.1 Single shot

The basic function of the periodic time is a single shot delay timer. Just set the timer and it will tick till it's over. Setting it to zero will disable it right away.

For example, if you want to make a sound for 2 seconds, just turn on the buzzer, set the periodic timer to 2 seconds (128 * 1/64), and disable the buzzer in the time out message dealer.

If accuracy is not the point, another timer setting could be invoked in the time out message dealer, so that a real periodic timer could be implemented. In the example above, if timer is set to 2 seconds again in time out message dealer, and the buzzer is toggled instead of disabled there, the buzzer will beep for 2 seconds and stop for 2 seconds in every 4 seconds.

API: **SPT_set**

4.6.2 Callback

If precisely timing is needed, call back will be the solution.

The callback must be registered first, when no SPI timer is active.

The timer reload could be part of the code of the callback so periodic timer is implemented, and the callback itself acts like a timer ISR. In this case, make sure that the callback returns before the interval ends, otherwise the callbacks will be overlapped.

Reregistering the callback as null will disable callback.

API: **SPT_register_call_back**

4.6.3 Read the timer

The time could also be read. The result is the time left to the time out event, in 1/64 seconds.

API: **SPT_read**

4.7 RF interface

4.7.1 Initialization

The RFI is turned off after reset. It must be initialized first to activate the paging information receiver/decoder. The parameter block must be passed through to the initialization subroutine. The RFI could also be turned off using the same routine with a NULL parameter. The parameter block used is a low level one, so a translation subroutine is necessary.

API: **RFI_init**
RFI_param_translate

4.7.2 Message arriving

After the RFI is initialized and turned on, the receiver will automatically search for the paging information. A system message will be sent if a paging message has been completely received. The message is stored in the message buffer until the next message comes in. So in the system message dealer for RFI new message, it’s suggested that the message be copied to somewhere else in the memory first.

API: **RFI_read_msg**

4.7.3 Data organizer

It's recommended that the paging message received be stored in an organizer so that it could be read later. Other functions of the organizer may include deleting, protecting, auto deleting of the oldest when full, etc. It's up to the application programmer to determine how the organizer should be.

4.8 Miscellaneous

4.8.1 Buzzer

The buzzer is controlled by a single API subroutine setting its frequency and duty cycle. The frequency depends on the system clock so it may be different if the system clock is not the default one.

Note: the buzzer is tuned to be working in a very narrow range of frequencies, so any frequency setting out of the range may result in low audibility. The peak frequency may vary from 2.7 kHz to 3.1kHz depending on the buzzer used.

API: **BEEP_sound**

4.8.2 EL back light

The back light subsystem could be turned on automatically or manually. It will be turned off automatically after a period of time for power saving, no matter how it's lit up. The time could be set by an API subroutine. Each time the back light is turned on (manually or automatically by key pressing), the timer is reloaded.

API: **EL_setting**
EL_set_time_out

4.8.3 Battery level check

Battery check is used when battery gauge is used in user interface. Please read the notes in API manual.

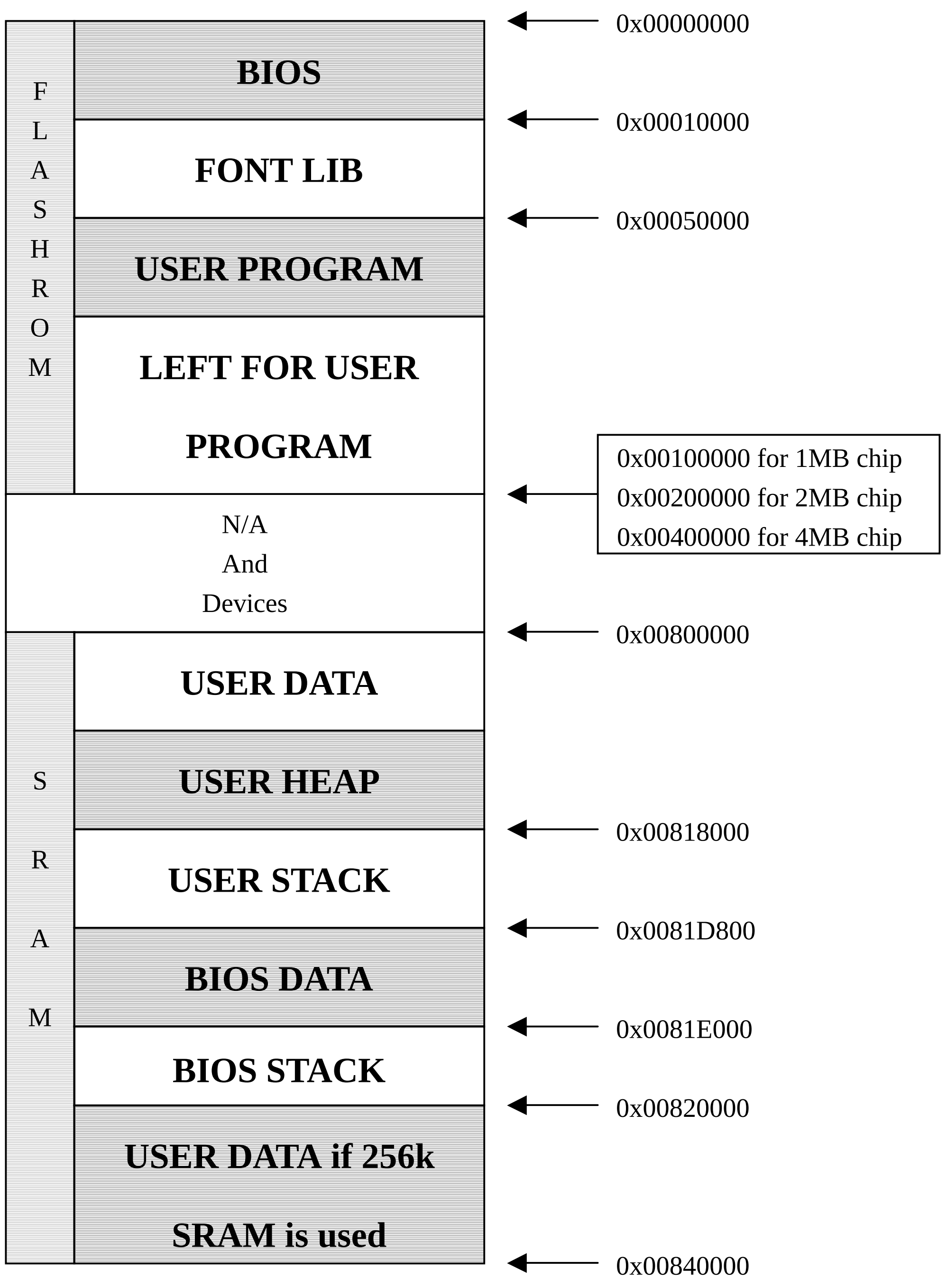
API: **BATT_check_level**

4.9 Memory managing

4.9.1 Memory map

The memory map of MC-998 is showed below. 3 types of FLASH ROM with the same pin layout could be used, with the size of 1Mbytes, 2Mbytes and 4Mbytes. 2 types of SRAM with the same pin layout could be used, with the size of 128kbyte and 256kbyte.

Only the sections marked with 'USER' are available for application programmers (c.f. sample link script for section descriptions).



4.9.2 Program

The program code is stored in CODE section, in FLASH ROM. The CODE section also includes: the initial value of the initialized variables, any global variable defined as ‘const’, constant strings.

4.9.3 Data

Two types of data section make up the USER DATA area: DATA, BSS. Variables declared as global are allocated in this area.

DATA is for the variables with initial values; BSS is for the variables without initial values, which would be filled with 0 in the C runtime routine running before **main()**.

4.9.4 Stack

The return address of subroutines, the parameters passed through, and all the local variables are allocated in stack. Since the stack is quite small comparing with Data and heap, it's recommended that the use of local variables should be limited.

4.9.5 Heap

Heap is the free area where static memory allocations have not occupied. This area could be allocated and freed at runtime using the **malloc** family subroutines.

Runtime allocation is recommended for large data structure (such as data buffer), because it's reusable from function to function and it's well monitored by the heap manager.

4.9.6 Non violate data storage in SRAM

This section (.nvram) is quite similar to the BSS section except that it's left as it was when entering **main()**. This makes it possible to retain the data even after the main power has failed. There is also an API function to indicate if the NVRAM might be corrupted after the power restoration.

Variables must be declared with **NVVAR** or **NVARR** to be non violate. For single data units like ordinary variables and pointers, **NVVAR** is used; for array or structured data units, **NVARR** is used.

API: **NVRAM_invalid**

4.9.7 Example

Here's a little example showing how the memory is used.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <api.h>
#include <string.h>

unsigned char tempbuff[80];       // in BSS
int tempint;                      // in BSS
const unsigned char HexStr[] = "0123456789ABCDEF"; // in CODE
const unsigned char NumStr[] = "0123456789AU -]["; // in CODE
unsigned char x = 0, y = 0;       // in DATA

const unsigned long RFI_param_block[] =            // in CODE
{
 0x400974A9,          /*                  */
 0x611E5AD0,          /* adr 0 : 1234567     */
 0x6121E848,          /* adr 1 : 1000000     */
 0x614B5B4E,          /* adr 2 : 1759858     */
 0x616BD077,          /* adr 3 : 1999802     */
 0x60800000,          /* adr 4 : disabled 0   */
 0x60A00000,          /* adr 5 : disabled 0   */
 0x60C00000,          /* adr 6 : disabled 0   */
 0x60E00000,          /* adr 7 : disabled 0   */
 0x0000FFF0           /* adr 1 as cc, adr 0 as num */
};

NVVAR(int init_data);                     // in NVRAM
NVVAR(int * pt);                          // in NVRAM
NVARR(char datat[80]);                    // in NVRAM
NVARR(typ_UART_stat_word d[10]);          // in NVRAM


void main(void) {
  char * t1, * t2;        // in STACK
  int p1[3] = {3,2,1};    // p1 in STACK, and {3,2,1} in CODE

  t1 = “test line”;       // “test line” in CODE
  if (!(t2 = malloc(20))) {         // *t2 in heap
    int t3 = 5;                    // t3 in stack, and 5 in CODE
    UART_puts(“out of memory”);    // “out of memory” in CODE
    return;
  }
}
```

4.10 System messages

System message is the only way for the BIOS to notify the application of what happened. It’s very important for the application to check them so that no messages will be lost.

4.10.1 Message polling entry

The system message is stored in a 16-bit word. An API for message polling is available. The message polling loop (MPL) might be something like this:

```
while (1) {
  typ_msg_word mw;
 // check msg and goto sleep if none
  mw.s_word = sys_msg(SM_GOTO_SLEEP);
  if (mw.bits.card_insert) {
// put your card insert dealer here
  }
  if (mw.bits.card_gone) {
// put your card gone dealer here
  }
  if (mw.bits.key_available) {
// put your key dealer here
  }
  if (mw.bits.rf_new) {
// put your paging infomation dealer here
  }

//…………
  if (mw.bits.time_out) {
// put your time out dealer here
  }

}
```

API: **sys_msg**

4.10.2 Messages

There are 12 types of system messages:

alarm_clock :	sent if alarm clock event occurred
comm_err:	sent if UART communication error occurred
comm_sb_empty:	sent if UART send buffer empty
comm_data:	sent if new data exists in UART receiving FIFO
RF_on_air:	sent if RF unit has started a receiving process
RF_new:	sent if new data exists in RF unit receiving buffer

time_out:	sent if a preset timer has expired
card_gone:	sent if the IC card in main socket has been removed
card_insert:	sent if a IC card has been inserted into main socket
key_available:	sent if new data exists in keyboard/jogdialer input FIFO.(key buffer mode)
key_up:	set if a key has been released (1-KEY mode)
key_down:	set if a key has been pressed(1-KEY mode)

Each of them could be masked so it would not be sent.

API: **sm_setting**

4.11 Downloading mode portal

There is one API subroutine as the portal for the downloading mode. The system will enter a built in loop for downloading application. This subroutine will never return. The portal is only used when an entry point for downloading the application itself is needed in the user interface.

API: **enter_download_mode**

4.12 Flash ROM

The Flash ROM is available for the applications except for the BIOS and Font library. There should be plenty of FLASH ROM left after the application is downloaded, which could be used as non-violate data storage.

4.12.1 API for Flash ROM

Only two basic API subroutines for FLASH writing and erasing are provided. It's recommended that a FLASH driver be written instead of using the API directly.

API: **FLASH_erase_block**
FLASH_write_record

4.12.2 Notes for writing Flash drivers

The block size of the FLASH ROM used in MC-998 is quite large: 64kbyte. So any algorithm based on SRAM swapping should be reconsidered.

Please always keep these in mind when writing FLASH drivers: the FLASH is not RAM, only the first write after erasing is valid.

So, it's not practical to put any data that's frequently modified from time to time in the FLASH, such as file allocation table, counter, unless a block swap buffer as large as 64kbytes could be found in SRAM.

A sample FLASH driver that supports fixed length records is provided (**database.c**).

4.13 Routines for SIM card operations.

4.13.1 Card interface unit power on/off

These routines make it possible to control the card interface unit power individually. Usually, the card interface unit is not powered for battery saving. The current will increase if the card interface unit is turned on. However, both card sockets are still not powered after the card interface unit power on.

API: **Card_power_on()**
Card_deactivate()

4.13.2 Card socket selecting

Card socket could be selected as ‘current-socket’, where all the following card routines will be performed. The former current-socket is detached from the MCU-card bus but the card there is still powered if it was powered, so that all security status of that card would not be lost after the card socket switching.

API: **Card_select_sock();**

4.13.3 Card socket power on/off

Card socket could be powered on using ATR routine. For some historical reason, the parameter of ATR routine contains socket number, which should be set as exactly the same as the currently select sock.
Power of current-socket could be turned off individually.

API: **Card_ATR();**
Card_deact_curr_sock();

4.13.4 Hopping around

When a card socket is selected as current-socket, the MCU-card bus will be restored as the last time this socket was selected. Since the system clock will change when communicating with the card, the system clock will also be restored. So, any system clock related function such as UART may not be available during card operation, and the program may be much slower. However, it’s possible to deselect all sockets and restore system clock to normal between card sessions. A special card socket number (0) is used to implement this.
Note:

a) It takes about 1ms to switch system clock. If there’s heavy computing load between card sessions, it’s worthy to switch back to normal speed.
b) The current consumption will increase a lot when all the socket are on, may be as high as 70~80mA, So do not get into this state unless it’s absolutely necessary.

4.13.5 SIM authentication

Here’s a typical flow when using SIM authentication:

//

```
// Initialization:
//
Card_power_on();
Card_select_sock(SIM socket number);
Card_ATR(..);
Data exchange with SIM to enable the crypto-machine on SIM;
Card_select_sock(main card number);
Card_ATR(..);
 Data exchange to get the serial number from main card;
…
//
// Authentication:
//
Card_select_sock(main card number);
Data exchange to get challenge from main card;

// back to normal speed so that the next step would be faster.
Card_select_sock(0);
Calculations to convert the challenge and serial number to a hashed key;

Card_select_sock(SIM socket number);
Data exchange to preform cryptographic caculation on SIM card, using hashed key
as input, with the keys stored in SIM card;

Card_select_sock(main card number);
Data exchange to present the result of cryptographic calculation from SIM card to the
main card;
```

4.14 Card reader simulator

Here's a sample for a PC smart-card reader simulator. The card reader takes command from PC via RS-232, and responds after the designated command is carried out. The flow chart showed below is for reference only.

a) Global variables:

```
unsigned short int Current_sock;
```

b) message loop in main():

```
…
Card_power_on();
while (1) {                    // main message dealer loop
       if 1 byte received {
              add it to comm packet buffer;
       if comm packet completed then
```

```
          process the packet;
    }
}
```

c) packet processor:

```
void packet_processor(unsinged char * ptr_packet) {

     if packet invalid then {
        send response packet indicating packet error;
        return
     }
     if command in packet unknown then {
        send response packet indicating command error;
        return
     }
     if parameter in packet unknown then {
        send response packet indicating parameter error;
        return
     }
     switch (command in packet) {
        case card seletion :
           Current_sock = socket number designated by parameter.
           send response packet indicating successful selection.
           break;
        case card power off:
           Card_deact_curr_sock();    // deactivate current socket
           send response packet indicating successful deactivating;
           break;
        case card detect :
           send response packet indicating status of current socket;
           break;
        case card power on and ATR: {
           typ_Card_ATR_param CArec;

           Card_select_sock(Current_sock);              // select current socket
           set CArec as desired;
           Card_ATR(&CArec);                      // perform power on & ATR
           ATR_analysizer(CArec);                 // adjust card interface w/ ATR
           Card_select_sock(0);      // deselect all sockets and back to normal speed;
           if ATR is valid then
              send response packet with ATR
           else
              send response packet indicating error ATR;
```

```
                break;
            }
            case data exchange :
                Card_select_socket(Current_sock)        // select current socket
                Data exchanging;
                Card_select_socket(0)                   // deselect all and set to normal speed.
                if data exchanging OK then
                    send response packet with response data
                else
                    send response packet indicating data error;
        }
    }
```

5.Downloading applications

MC-998 will enter downloading mode if no valid application has been found in the designated area.

If the application is already there, there are two ways to enter downloading mode:

1) Using the portal in the API. Enter the portal by following the user interface specified in the last application (to be overwritten).
2) Hold the Power key (key code 0x00) while resetting. Inserting the battery will reset the system as well as pressing the reset key located in the back of MC-998, by the battery socket. The reset key could be pressed using a pin. A hint message will be displayed on the screen if downloading mode is entered this way.

RS-232 is used for downloading. The parameters for downloading are: 115200bps, no parity check, 8 data bits and 1 stop bits, CTS ignored.

The string “Bootloader OK\n” will be sent via RS-232 as soon as the downloading mode is entered successfully.

A utility program running in WIN95/98 to download applications is also provided.

6.tool chain

The GNU gcc tools chain consists of C/C++ compiler, linker, assembler, and some binary utilities. All the tools have many UNIX-style command line options. It’s suggested that a make file be used to simplify the whole compile-link process.

Please read the GCC manual for details.

6.1 C compiler

The GNU C/C++ compiler is called **gxx**. When you invoke it, it normally does preprocessing, compilation, assembly and linking. It could be stopped at any stage, but when developing MC-998, it’s suggested that it be stopped after assembly, i.e. ‘-c’ option should be specified.

6.2 Linker

The GNU linker is called **ld**. In MC-998 projects, the objects are relocated, linked together with libraries into an executable code module to be downloaded. The relocation addresses are specified in a link script.

6.3 Assembler

The GNU assembler is called **as**. It's invoked by **gxx** internally, and there is no need to invoke it directly, unless assembly source code is part of your project.

6.4 Binary utilities

There are several binary utilities provided by GNU as standard toolbox for development. Not all the tools are used, but some of them are necessary for MC-998 projects:

ar and **ranlib** if you want to make your own libraries;

objcopy and **nm** to convert the output of linker , which is in COFF, to a downloadable module, in binary;

Some useful binary utilities

GNU NAME	68k name	description
ar	68k-ar	Create, modify, and extract from archives
objcopy	68k-objc	Copy and translate object files
objdump	68k-objd	Display information from object files
nm	68k-nm	List symbols from object files
ranlib	68k-ranl	Generate index to archive contents
size	68k-size	List section sizes and total size
strip	68k-stri	Discard symbols
c++filt	68k-cxxf	Filter to demangle encoded C++ symbols